Efficient Monitoring and Display of Thread State in Java

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI 02912-1910
401-863-7641, FAX: 401-863-7657
spr@cs.brown.edu

Abstract

How many times have you asked yourself the question "What is my program doing now?" JIVE is a system that tries to answer this question. It provides insights into the execution of Java programs. It does so with minimal overhead so it can be used on arbitrary systems at any time. A large portion of the complexity of Java systems comes from the behavior and interaction of multiple threads. Thus, JIVE concentrates in part on showing what the various threads are doing. This paper describes how JIVE does this. It first describes our original approach which simply monitored the state of each thread, displaying the time spent in each state on an interval basis. We then describe how we extended this framework to provide additional, detailed information on state changes and on thread interactions. The core of this paper describes the problems that arose here, their solutions, and the resultant visualizations.

1. Introduction

We want to be able to understand the behavior of our software. In particular, we want to be able to understand what the software is doing when performance issues arise, when it undergoes unexpected behavior, and when it interacts with the user or the outside world in a particular way.

To address these issues, we are developing a visualization system, JIVE, that looks both at class-level behavior and at the interaction of threads [16,18]. JIVE demonstrates that it is possible to do dynamic visualization of Java with very low overhead (a factor of two) while producing meaningful and useful views of the software. JIVE summarizes information in terms of intervals of ten milliseconds or more (under user control). Interval-based analysis and display is appropriate here since the system is designed to run concurrently with program execution and to provide immediate insight into what the system is doing.

JIVE actually provides two distinct displays that illustrate program behavior. The first concentrates on class behavior and what portions of the code are executing. For each class (or collection of classes where appropriate) it collects the number of calls of methods of the class, the number of allocations done by the class, the number of allocations of objects of the class, and the number of synchronizations done on objects of the class. This information is displayed dynamically in a compact form that highlights classes with unusual behaviors.

The second display shows thread information. Much of the complexity of today's systems, especially in Java programs, comes from the interactions and behavior of threads. All but the most trivial Java programs are multithreaded. Threaded programs often behave in non-obvious and non-intuitive ways and their behavior is generally time-dependent and hence effectively non-reproducible.

Our initial approach to obtaining and displaying thread-related information was to simply summarize, for each interval, the behavior of each thread. This was done by defining the behavior in terms of the state of the thread. We identified eight different states of interest:

- **NEW** — the thread has been created, but has not started execution.
- **RUN** —the thread is running (or at least is runnable) and is not synchronized.
- **SYNC** — the thread is running inside a synchronized region.
- **WAIT** — the thread is suspended on a call to *Object.wait*().
- **SLEEP** — the thread is suspended on a call to *Thread.sleep*().
- **IO** — the thread is in the process of doing I/O (possibly waiting while doing so).

- **BLOCK** — the thread is blocked on a monitor, waiting to enter a synchronized region.
- **GC** — the thread is actively doing garbage collection.
- **DEAD** — the thread has exited.

The state **NEW** is included here because a thread may be created in the middle of an interval and we needed to note its state before creation. We differentiate between **WAIT** and **SLEEP** since *wait* is generally used when one thread is waiting on the actions of another while *sleep* is used when the thread just wants to hibernate for a while.

Determining the state of each thread and then displaying the resultant summary for each interval was the initial task we took on in the JIVE implementation. This is described in Section 2. After using JIVE for a while, we determined that additional information about each thread, in particular the state transitions and interactions with other threads were important and should also be displayed. Section 3. shows how we extended the framework to obtain and then display this information. We conclude with an evaluation, a synopsis of related work, and a discussion of future work.

2. The Basic Framework

JIVE was designed to provide a view of the user's program in action. As such, its most important goal was to achieve very low overhead data collection for arbitrary Java programs. Low overhead implied that the monitoring code would not cause additional thread synchronizations, would not do additional allocations (which can block or cause garbage collections), and would not make use of expensive library routines, for example any that did allocations. Handling arbitrary programs meant having to deal with the complexities of native code, dynamically loaded code, all the standard Java libraries, and other arbitrary libraries for which we had no source. Finally, we were concerned with keeping the initialization and setup time to a reasonable level so as not to discourage the use of the tool.

The overall structure of JIVE is three-fold. One part is the visualization and user interface. This portion of the system both provides the displays of the program in action and lets the user specify what to run, what to display, and to control the display in various ways. The second part of the system is a setup program which analyses the user's program and all associated libraries and gets it ready for visualization. This system provides information to the front end about what classes are available and what needs to be displayed as well as setting up the monitored program. The final part of the triad is the user's application itself. This is run independently and uses a socket to communicate to the visualizer the data for each interval.

Getting the necessary information efficiently from the application program was the most difficult problem we faced in implementing JIVE. We tried a variety of different approaches, starting with the standard java monitoring tools (JVMPI, JVMDI, JPDA) we had used in our previous work [17,18,20].These proved to be too slow even if nothing was done — just turning them on slowed the program down by a factor of ten. This left us with the alternative of patching the object code to call a monitoring library. This is the approach we took, using JikesBT as the patching library [11]. Even using inserted code, we had to be careful. Many seemingly innocuous method calls (such as *Thread.currentThread*() or *System.currentTimeMillis*()) are prohibitively expensive to call on each method entry and exit, as is any native method. Finally, patching code is an expensive operation itself, and we needed to minimize the amount of code that was patched in order to keep the initialization overhead low.

Within these limits, we wanted to obtain enough information so that we could show the programmer what the program was doing. In particular, we wanted to illustrate thread behavior in terms of thread states. Here we needed to be able to identify when threads change state. Some state changes are relatively easy to detect. For example, the **SLEEP** and **WAIT** states are entered when a thread calls the corresponding method (*Thread.sleep* and *Object.wait* respectively) and are exited when that method returns. Similarly, entering the *run* method for a subclass of *Thread* indicates the start of a thread's execution and hence identifies the states **NEW** and **RUN** (or **SYNC** if *run* is synchronized), while exiting the method denotes entering the **DEAD** state.

Identifying the **IO** state was more difficult since actual input and output is done inside native methods. However, since almost all I/O is done using the standard library, it is possible to identify all the methods in the standard library that represent input or output calls. Then it is a good approximation to assume that a thread enters the **IO** state when it enters one of these methods and exits the **IO** state when the method returns.

To facilitate this, JIVE actually reads in an XML file that identifies all the methods that identify a thread state change. This file lets us easily specify which methods should be associated with the **IO** state, either by identifying specific methods or identifying a set of methods based on overriding. The current XML file includes eighteen methods or sets of methods that are involved with I/O, ranging from the obvious *InputStream.read* to the non-obvious *sun.awt.motif.MTool-*

kit.run which represents native code that reads from the X11 server and then generates events.

The XML file is also useful with the **WAIT** state. It turns out that there are a significant number of different methods in the standard library where waiting is done inside a native method rather than using the library method *Object.wait*. The current XML file identifies seventeen such additional methods.

Detecting the synchronization states is a bit more difficult. First, consider a synchronized region. Obviously, when we start executing code in the region we are in the state **SYNC** and when we exit the region we revert back to whatever state we were in before (either **RUN** or **SYNC**) this particular synchronization. These transitions can be handled by noting a state change at the first instruction of a synchronized region and another state change just before the region is exited.

Just by patching the code, however, it is impossible to detect when the **BLOCK** state occurs. We approximate this by assuming that this state is entered just before we enter a synchronized region. The amount of time in the state, if no blocking occurs, will then be minimal and the result, in terms of the amount of time spent in each state, will be close enough to give the programmer the right information.

A further complication arises in that not only are there synchronized regions, but Java also supports synchronized methods. Synchronized methods are handled inside the JVM rather than by instructions in the byte codes. This makes patching much more difficult. While it is easy to identify the start of synchronized method, it is harder to patch the exit since here one must take into account not just normal returns, but also throws and any uncaught error or exception (which could occur at almost any place in the code). Even more difficult is inserting a state change just before the method is called since the call can be virtual, can come from native code, or can be the result of using reflection. To get around these problems, we change synchronized methods into normal methods with synchronized regions.

We had two alternatives to make this change. We could change the code of every synchronized method so that the synchronization was done in-line rather than using the JVM. This is rather difficult since we would have to generate code for each return and throw and would also have to handle error throws. The simpler approach we use is to generate a new routine with an internal name which is the original method unsynchronized, and then create another routine with the original name that contains a region synchronized on *this* (or the class object for a static method) in which we call the new routine. This greatly simplifies the code that we need to generate.

The final state, **GC** or garbage collection, proved even more elusive. Other than using an external monitor (which was precluded due to efficiency considerations), we could find no way of detecting when garbage collection was done and what thread was doing it. Thus our implementation ignores this state for now.

Finally, we had to make a decision as to what code we patch. Originally we patched all code including libraries so that we could get accurate statistics about when synchronization is performed. This proved too costly in terms of initialization time, taking around a minute even for relatively simple programs. After noting that most of the library synchronizations were irrelevant to most programs, we decided not to patch all the code by default. Instead, we patch all code where routines are identified as special (i.e. are associated with **WAIT**, **IO**, or **SLEEP** states or that identify threads), all the user code, and any other classes that the user identifies as relevant to their particular system. This simplification cuts initialization time to about ten seconds for most applications. We also added the ability to identify additional classes that should always be patched in the XML file. For example *java.lang.Hashtable* is often a problem because it has a synchronized implementation.

The monitoring code is relatively straightforward. If a routine is entered that indicates a state change then the *setState* function is called. This function first identifies the current thread. We don't determine the current thread all the time since calling *Thread.currentThread* is an expensive operation. However, state changes occur infrequently enough so that we can afford to call it when they do occur. Next *setState* optionally pushes the previous state, sets the new state, and adds the time since the previous state change or the beginning of the interval to the statistics for the previous state.

An additional complexity arises because some threads and some transitions are partially hidden, i.e. threads can be started either outside of the user's program (e.g. the main thread) or inside native methods or the JVM itself (e.g. the Motif thread or the finalizer thread). We handle these by identifying the first state change to a thread that was otherwise unknown. Here we set the previous state based on the name of the thread, for example setting the Motif thread to **IO**, the Finalizer, Reference Handler, and AWT-Shutdown threads to **WAIT**, and all other threads to **RUN**. We also detect when a thread enters a **WAIT** state from a **RUN** state and change the prior state to the corrected **SYNC** state.

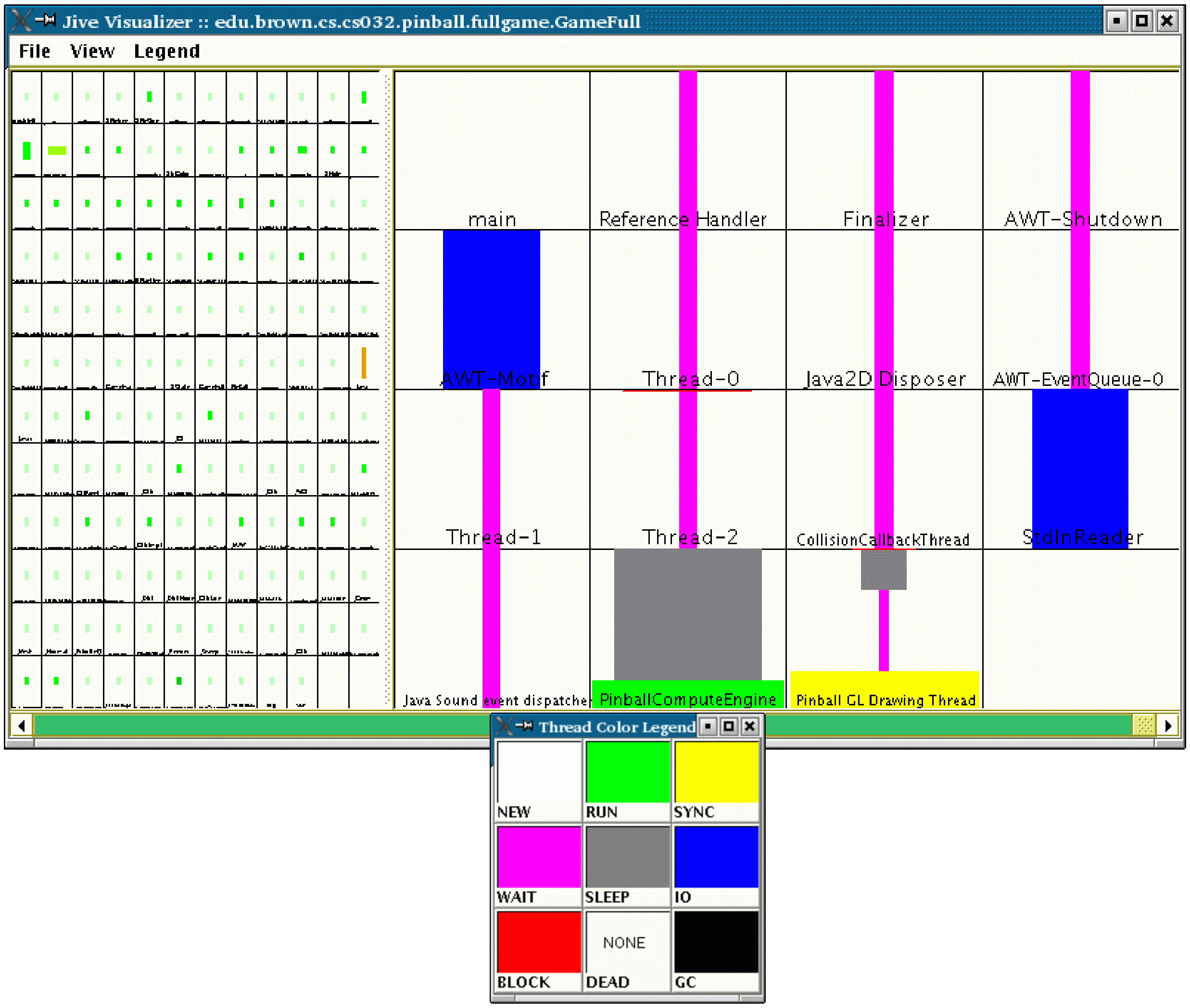


FIGURE 1. JIVE showing the thread visualization on the right for a Pinball program.

Generating statistics for visualization is done by JIVE through triple buffering. At any one time, there is one buffer of statistics that the threads are currently adding information to, one that represents the previous interval, and one that represents the interval that just ended and is being processed. JIVE uses information from the previous interval to determine the state of each thread at the start of the current interval; otherwise it just computes the total time for each state for this interval and generates a corresponding report for the visualizer.

Visualization is done in terms of stacked boxes as seen in Figure 1. Colors are used to represent the different states as shown in the legend window at the bottom of the figure. Each thread is assigned a box left-to-right, top-to-bottom in the order that the threads are created. Within the box, the height of each color represents the fraction of the time the thread spent in the corresponding state. The width of the colored region for a thread represents the fraction of the total time represented by that state that is assigned to the thread. Thus the green and yellow regions, being full width, indicate that the corresponding threads were the only ones in the **RUN** and **SYNC** states respectively, while the narrow cyan regions indicate that many threads were in the **WAIT** state.

In addition to the immediate display for the interval, JIVE provides the ability to replay previous intervals and go backward and forward in time using the scroll bar at the bottom. It also provides the user with the option of displaying either the immediate values or cumulative values.

JIVE also uses the interval information on both threads and classes to determine the current program phase. Programs executes in phases. A simple system first does initialization, then reads input, then processes that input, and finally writes the result out. Actual systems typically go through various phases depending on different input commands and external events, varying processing requirements, and other related factors.

JIVE displays the program phase using the color of the scroll bar. The phase is computed by generating a normalized statistical vector based on the combined interval statistics for the current and previous three intervals. The interval range here was determined experimentally — if it is too small, spurious phases are reported; if it is too large then we miss significant phases. This vector is then compared using the dot product to the normalized saved vector for all previously determined phases. If the vector is close (dot product greater than 0.9) to the current phase, this phase is continued. Otherwise all previously determine phases are compared to the vector and if any of these is close enough it is chosen as the new phase. Finally, if the vector fails to match any previous phase a new phase is started. In any case, the statistics from the vector are merged into those for the determined phase.

3. Extending the Framework

While the information provided to the user by the initial implementation of JIVE was useful, as we were using it we felt that it was incomplete and didn't give us enough information as to what is going on. As developers we wanted to have additional information.

While the display shows when threads are blocking, it does not show why they are blocking, i.e what thread they are waiting on. Moreover, while it gives an overall sense of what each thread is doing, it doesn't tell us how it is doing it. For example, a large synchronized box could be caused by lots of synchronized entries and exits during the interval or by a single time-consuming function that was running synchronized. Similarly, a large amount of I/O wait could indicate a slow input device (such as the keyboard) or lots of reads from a disk. These distinctions are often important in understanding the behavior of one's program.

To extend the initial framework to include this information, we needed to address a number of problems. First, we needed to provide information to the visualizer about all the state transitions within an interval. This is potentially costly both in terms of collecting the information and in terms of transferring it from the application to the visualizer over the socket. Much of the complexity here comes from the need to keep the monitoring efficient, i.e. without any allocations and with minimal use of library routines. Second, we needed to provide timing information as to when the state transitions occurred. Third we needed to detect blocks and to assign blame when a thread does block.

We addressed the problem of recording transitions by preallocating buffers to hold transition events for each interval. This limits the number of events per interval to a preset amount. We experimented with various values here before settling on the current limit of 512. This number is not so large as to make the communication between the application and the visualization overly expensive, while still providing lots of detail about what happened in the interval. Moreover, we found that if the program is complex enough to exceed this limit and transition events are actually discarded, the resultant display still provides a reasonable approximation of the transition sequences. Note that if we allow too many transition events, the messages going back to the visualizer can become quite large and the I/O cost of sending the information can either cause the visualization to become unsynchronized with the execution or can slow down the execution.

Timing transitions is a problem because the clocks available in Java have a relatively coarse resolution of one millisecond or worse. In the original framework, where we are simply accumulating information for an interval, using such a coarse timer is acceptable since the time totals will be statistically correct. However, when we are looking at times within an interval, we found we needed finer resolution.

We achieved a good approximation to a fine clock by using an internal counter. This counter was incremented on each method entry and exit and each allocation. Incrementing was not synchronized so it was possible for multiple threads to report the same time and for the timer to decrease, but the counter would generally increase and provided a fairly accurate low-level clock. Then for each event, we report both the clock time and the local time represented by the value of this counter, and the visualization computes approximate times based on the combination of the two.

The next problem we addressed involved tracking which thread a blocked thread is waiting for. This involved tracking what objects are locked by what threads. To do this we first preallocated a block to hold the locked objects for each thread. Then, when we enter a synchronized state we add the corresponding object to this block and remove it when we exit from synchronization. When we then attempt to enter a **BLOCK** state, we go through all active threads and attempt to identify if any is currently blocked on the blocking object and is not in a **WAIT** state. If so we, indicate that the current **BLOCK** is associated with the identified thread.

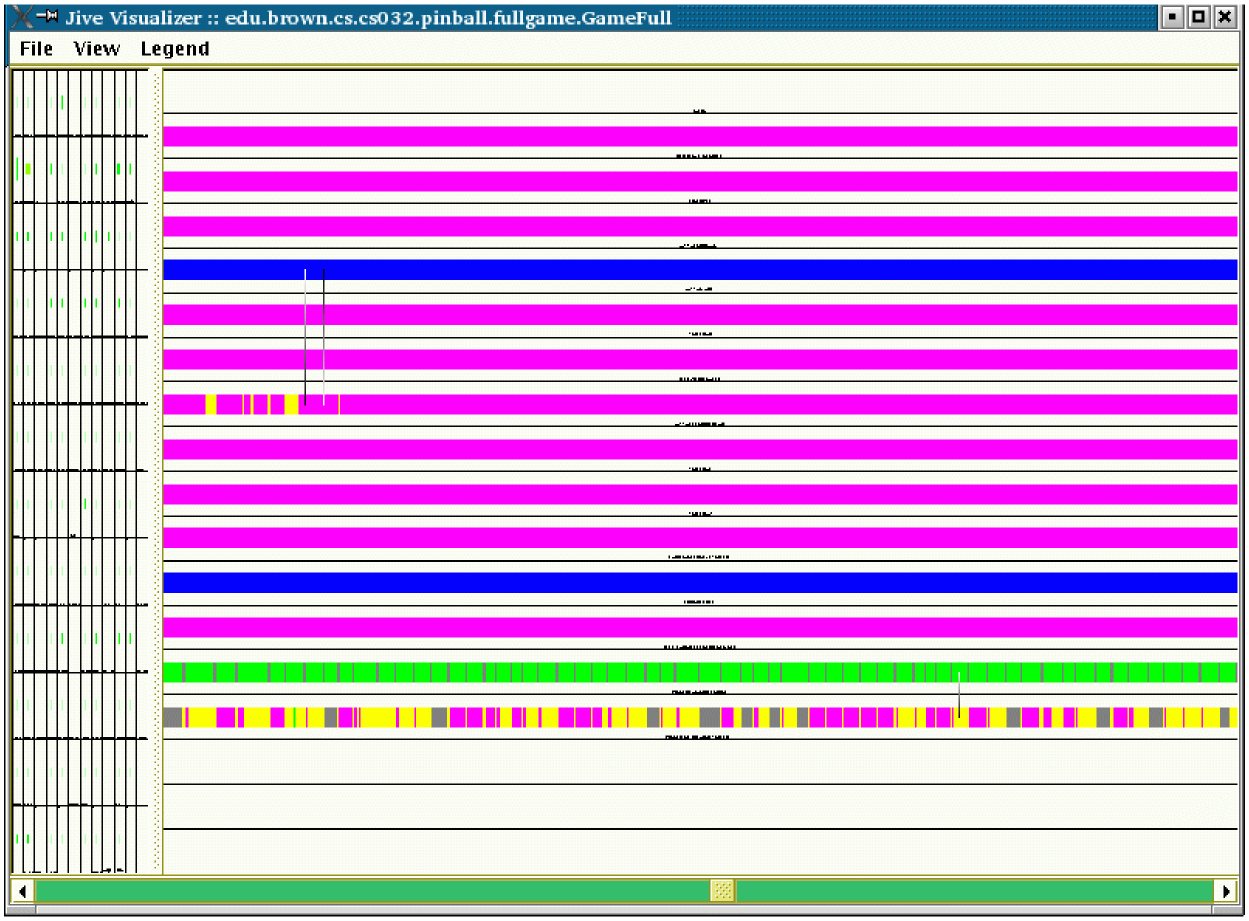

Jive Visualizer :: edu.brown.cs.cs032.pinball.fullgame.GameFull
File
View
Legend

FIGURE 2. Time line display of the Pinball program threads

We note that this is an approximation. If two threads are competing for a lock, we might miss the occurrence; similarly, we might detect a lock when the lock was actually released by the detected thread and picked up by another thread. In practice, however, this works quite well. We have made several close inspections of the visualization and the visualization data looking at the assignment of blocks to threads. These inspections show that we almost always find a blocking thread when we enter a **BLOCK** state that actually blocks and that the thread that is indicated is actually running synchronized at the time.

A more serious problem with our instrumentation strategy in terms of reporting all state transitions is that it reports **BLOCK** transitions that are not actually blocks. A thread goes into the **BLOCK** state before entering a synchronized region and exits that state as soon as the region is entered. If no other thread holds the monitor it is using, then no blocking actually occurs. We detect this when we enter the **SYNC** state. Here we check if the prior state was **BLOCK** and the change in the local time was insignificant. If both these conditions hold, then we remove the **BLOCK** state transition.

The data from the event sequences is used to create a time display as seen in Figure 2. The display shows the state of each thread over the past twenty intervals. The number twenty here is user settable from one to one hundred. The normal state colors are used to show the thread states. While it is not used in the display, the visualizer provides the user with the option of using the height of each state region to reflect the time relative to the state or the interval.

The display is also used to show blocks between threads. A block is displayed as a vertical line from the thread that is blocked to the thread it is blocking on drawn at the start of the block. We tried drawing lines with arrows here, but noted that this was difficult to view in the cases where there were lots of threads or lots of events at one time since the arrow heads either became too small or overlapped. Instead, we currently draw a simple line that uses a gradient from white (the thread that is blocked) to black (the thread blocked on). The result is unobtrusive but easy to understand and interpret. Note that in the figure the actual arrows don't seem to go from a **BLOCK** state to a **SYNC** state as one would expect. The reason for this is that the size of the corresponding states is so small when we are displaying twenty time frames at once, that the regions, while they are there, don't appear on the display. A better view of the arrows can be seen in Figure 3.

4. Experience and Evaluation

We have used JIVE with the extended visualization on a variety of applications ranging from simple illustrative programs to production systems. It has helped us identify a number of problems and to understand our applications.

A simple example where we found the system useful was with a small gas station simulation program that used multiple threads. We were playing around with the program and decided that the application should use a sleep call when it was waiting a specific time before processing the next event rather than a wait since this made more logical sense. Things seemed to work fine, but looking at the JIVE visualization it was immediately clear that the sleep caused several other threads to block which was unexpected. It turned out the sleep occurred in a synchronized region.

A more complex situation occurred in the 3D pinball application developed for a software engineering course that is the subject of Figure 1 and Figure 2. There were two basic questions that arose with the implementation. First, was Java fast enough to handle real 3D graphics (using JOGL), do all the physics calculations, manage sound, and still run on the student machines. Figure 1 shows the program running with 1000 physics computations per second and doing sixty frames of 3D graphics per second. It is clear that neither the physics nor the graphics (represented by the two boxes in the center of the bottom row) take up more than 1/3 of the available CPU, and thus that Java and the implementation are fast enough.

Second, the program was simplified so that all callbacks to the students code (for collisions, time-outs, keyboard events, etc.) were synchronized so that only one could occur at a time. Here we were concerned that their might be too much synchronization since some of these can block the physics computation. We used the extended display to look at where blocks actually occurred. While Figure 2 shows several blocks, looking over the whole execution using the visualizer shows very few blocks and thus that this is not a problem.

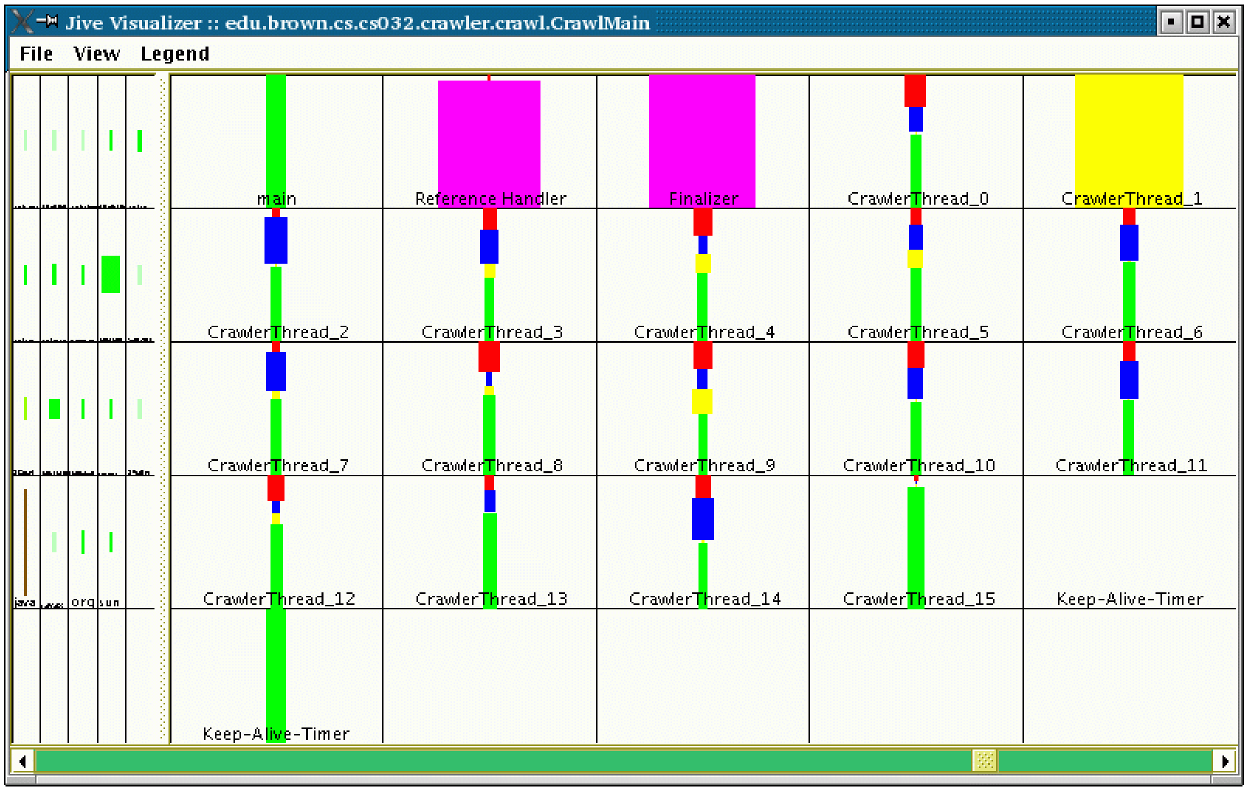
Another instance where we used the visualizer was in a Java-based web crawler. This application uses multiple threads each of which attempts to get a page, parse the corresponding HTML, extract the links and words, and store the links, words, and html information for later processing. Here we were interested in attempting to determine the optimal number of threads that should be used, with the trade-off being that too many threads will cause both synchronization and processor slowdown while too few threads will cause the processor to be idle more than necessary. Figure 3 shows both a sample original and extended view of the result. The original visualization shows that we are spending some time synchronizing, but that the bulk of the time of the various crawler threads is being spent doing actual processing. The more detailed extended visualization shows that the program actually goes through different phases. At times, such as in intervals represented by the left of the visualization, all the threads are actually parsing their web pages. At other times, such as on the right of the visualization, the threads are mainly waiting for I/O and there is some synchronization between them (generally to find the next page to crawl to or to store common information such as additional links that represent crawl sites), but not an excessive amount.

These examples show that the visualization is both practical and useful. Our ability to run it on interactive applications such as pinball and still be able to play the game at real speeds demonstrate that it does have the necessary low overhead. The ability to provide detailed views of the thread interactions in the web crawler shows that the extended visualization if practical and worthwhile and is fast enough to run in a complex system.

The use of sockets to connect the visualizer with the application is also not a problem. The amount of data that needs to be sent each interval is generally between 2,000 and 20,000 bytes, i.e. one to three socket writes, well within the capabilities of today's systems which can do over one hundred writes in ten milliseconds.

Our experience, however, shows that JIVE does not always give a full view of what is happening. There are two particular problems that we still have to deal with. The first is garbage collection. Sun's JVM does garbage collection in whatever thread happens to run out of storage, with the remaining threads secretly blocking while the collection is done. If a program does frequent garbage collection, then this results in a skewed visualization. For example, it is possible that the stable portion on the left of the bottom of Figure 3 is due in part to the fact that garbage collection happened to occur at that point and all the threads remained in their prior state.

A second problem is that transitions that occur in native code can be hidden. One of our concerns regard-



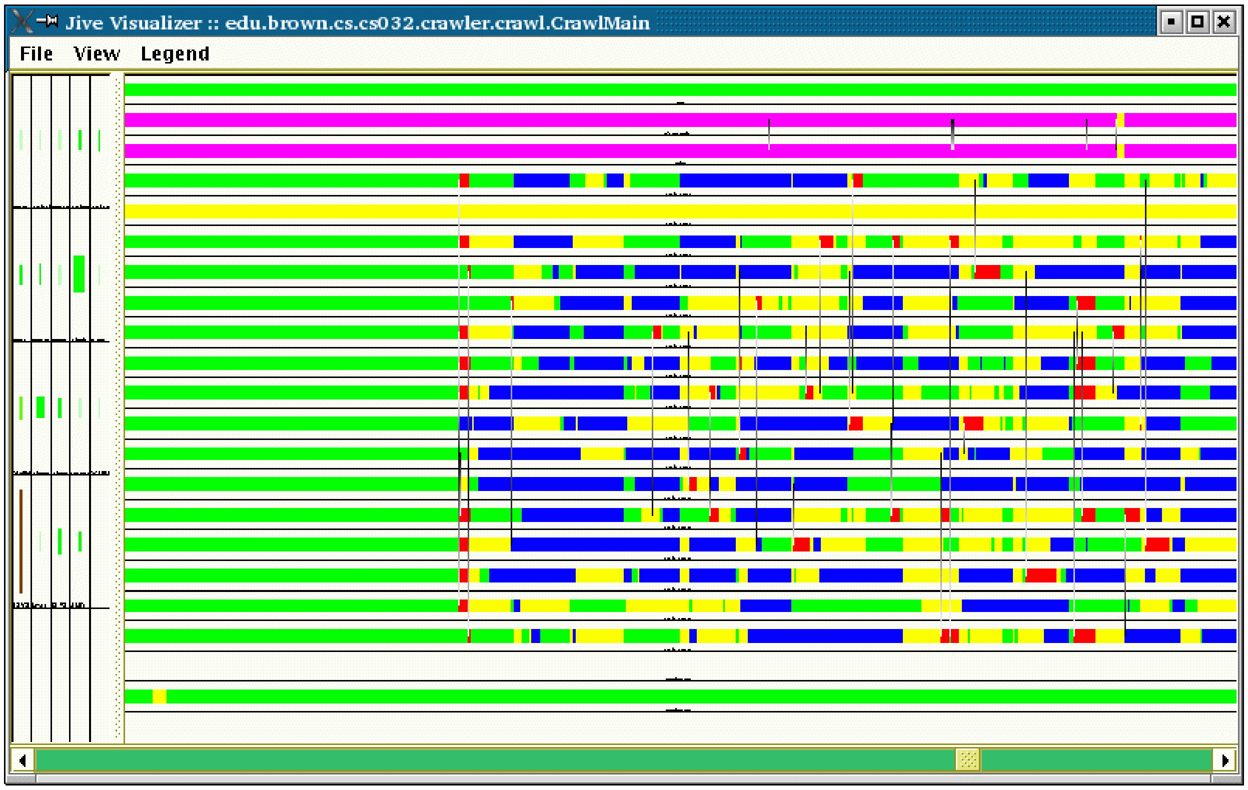

Jive Visualizer :: edu.brown.cs.cs032.crawler.crawl.CrawlMain
File
View
Legend

FIGURE 3. Two visualizations of a Java web crawler in action.

ing the pinball program was with the cost of doing sound processing every twenty milliseconds so that we could have collision noises happen at the time of a collision. Our early visualizations showed that sound processing took up about ten percent of the execution time. However, in Figure 2 it is difficult to see any time spent on sound processing. (The sound thread is the solid blue line toward the bottom — it is almost always in an I/O wait state.) The reason for this is that in the newer version of Java we currently use, almost all the sound processing is done in native code and hence is invisible to the instrumentation and visualization.

5. Related Work

There have been a large number of different systems that provide visualizations of the dynamics of a program. Ours is different in that it attempts to provide high-level program-specific information in real time.

Perhaps the most prominent effort is IBM's Jinsight [12-14]. Jinsight typically runs by collecting detailed trace data as the program executes and then, after execution is complete, letting the programmer understand execution at a very detailed level using a variety of views based on the trace. Trace collection, however, is not that efficient, requires a suitably modified JVM (and the program to work with that particular JVM), and is typically not the type of thing one would use all the time. Recent work on Jinsight has been aimed at letting the programmer identify just those portions of the program for which tracing should be done. This provides for almost immediate visualizations, but assumes that the programmer knows what to look for in advance. Other recent work includes the other JIVE system from the University of Buffalo [4,5] and Sun's JFluid system which uses bytecode instrumentation as we do, but hasn't been used to produce dynamic visualizations [3].

The program visualization group at Georgia Tech has implemented several visualizations that provide insights into program execution using program traces [8,10]. Similar systems include PV from IBM [9], and the dynamic aspects of the Bloom system [19-21]. The problem with these trace-based analyses is that they require the programmer to take the extra effort to run the system with tracing and often are both difficult to use and run too slowly to be practical. Our goal was to get as much of the information that these tools provide as possible without the considerable overhead that they incur.

Another set of relevant tools are performance visualizers that provide insight into what the machine is doing while the program is being run. These range from standard operating-system based performance tools such as those incorporated in Sun's workbench toolkit, IBM's PV system, or Sun's new Dtrace system [2] to viewers that concentrate on some specific aspect of execution. In the later category, one finds dynamic visualizations of thread behavior [1], visualizations of heap, performance and input/output in the FIELD environment [15,16], and the large number of different visualization of the behavior of processors and messages in parallel systems culminating in the various MPI visualization tools such as *upshot* or *xmpi*.

The latest version of Java, Java 5.0, includes two new management interfaces that are designed to replace both the previous monitoring and debugging interfaces, JVMTI as a low-level interface and the Java management extensions. While these offer promise, they can not be easily used to provide the information that JIVE needs. In particular, they have no notion of sleeping or I/O wait, do not report threads running in synchronized regions, and do not report all synchronization calls on objects. What we will be able to do with these interfaces is to get the information about garbage collection and native calls that are currently missing.

The technique of patching code, either object code or byte code, in order to gather performance data has also been widely used. The earliest system that we know that did this was the Pixie analyzer from MIPS [22]. Purify and many related memory checking tools also use the technique [6,7]

Finally, we note that dynamic visualization is nothing really new. Back in the 1960s we (and others) used to try understanding what their program was doing either by looking at the lights or the performance meter of the system (on a GE635) or by placing a radio next to the system and listening to the different types of static that were generated.

6. Future Work

JIVE is a very useful tool, but it is by no means complete or perfect. There are several directions that we are currently pursuing or planning to pursue.

First, we have developed a new visualizer that gathers and displays detailed information at the basic block level, JOVE. Currently JOVE is not an extension of JIVE but is rather a separate system. We plan to merge the two systems so that it will be possible to obtain both detailed thread state visualization and detailed execution information at the same time.

Second, as noted in the evaluation section, there are some problems that JIVE does not currently deal with, notably hidden threads and garbage collection. We are currently looking into ways that we can gather information about these and include it in the display. In particular we are looking into the new management facilities that are included with Java 5.0.

Next, we have realized that the fact that JIVE first instruments the program and then runs it is a limitation in many cases.There are times we are running a long-term production system and suddenly become interested in its behavior. In this case we would like to be able to attach JIVE to a running system, adding the necessary instrumentation dynamically as needed. This seems possible, but is going to require a significant amount of experimentation in order to get it right. Here we will be looking at technology such as JFluid which does dynamic instrumentation [3].

Along similar lines, even though the slowdown of JIVE is small, a factor of two can be significant to some applications. It should be possible to turn on or off instrumentation dynamically, so that there is minimal overhead when instrumentation is turned off, but so that it can be turned on rapidly when desired. Adding this feature to the instrumentation library is relatively easy, but we need to first establish a two-way protocol and determine exactly what we want to do.

Other directions that we are currently pursuing include extending the visualization to other languages and to multiple process systems. Our goal here is to be able to visualize in one view both the back end of a web application including its processing in the web server (say running as a servlet or as a PHP program) and one or more servers that the application uses (the actual server, a database, etc.).

Overall, we have been quite happy with JIVE, its performance, the thread-state-based visualizations, and our ability to use it on real programs. JIVE is available for public use at *http://www.cs.brown.edu/people/spr/research/viz/jive.html*.

7. References

1. Bryan M. Cantrill and Thomas W. Doeppner, Jr., "Threadmon: a tool for monitoring multithreaded program performance," *Proc. 30th Hawaii Intl. Conf. on Systems Sciences*, pp. 253-265 (January 1997).

2. Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal, "Dynamic instrumentation of production systems," *USENIX '04*, (June 2004).

3. Mikhali Dmitriev, "Design of JFluid: A profiling technology and tool based on dynamic bytecode instrumentation," *mddojapt*, (November 2003).

4. P. Gestwicki and B. Jayaraman, "Interactive visualization of Java programs," *IEEE Symposium on Human-Centric COmputing, Languages and Environments*, pp. 226-235 (September 2002).

5. P. Gestwicki and B. Jayaraman, "Jive: Java interactive visualization environment," *OOPSLA 2005 Conference Companion*, pp. 615-616 (2004).

6. Reed Hastings and Bob Joyce, "Purify: fast detection of memory leaks and access errors," *Proc. Winter Usenix Conf*, (January 1992).

7. Pure Software Inc., *Purify 2 User's Guide*, Pure Software Inc. (1993).

8. Dean Jerding, John T. Stasko, and Thomas Ball, "Visualizing interactions in program executions," *Proc 19th Intl. Conf. on Software Engineering*, pp. 360-370 (May 1997).

9. Doug Kimelman, Bryan Rosenburg, and Tova Roth, "Visualization of dynamics in real world software systems," pp. 293-314 in *Software Visualization*: *Programming as a Multimedia Experience*, ed. John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price,MIT Press (1998).

10. Eileen Kraemer, "Visualizing concurrent programs," pp. 237-256 in *Software Visualization*: *Programming as a Multimedia Experience*, ed. John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price,MIT Press (1998).

11. Chris Laffra, Doug Lorch, Dave Streeter, Frank Tip, and John Field, "What is Jikes Bytecode Toolkit," *http://www.alphaworks.ibm.com/tech/jikesbt*, (March 2000).

12. Wim De Pauw, Doug Kimelman, and John Vlissides, "Visualizing object- oriented software execution," pp. 329-346 in *Software Visualization*: *Programming as a Multimedia Experience*, ed. John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price,MIT Press (1998).

13. Wim De Pauw and Gary Sevitsky, "Visualizing reference patterns for solving memory leaks in Java," in *Proceedings of the ECOOP '99 European Conference on Object-oriented Programming*, (1999).

14. Wim De Pauw, Nick Mitchell, Martin Robillard, Gary Sevitsky, and Harini Srinivasan, "Drive-by analysis of running programs," *Proc. ICSE Workshop of Software Visualization*, (May 2001).

15. Steven P. Reiss, *FIELD*: *A Friendly Integrated Environment for Learning and Development*, Kluwer (1994).

16. Steven P. Reiss, "Visualization for software engineering -- programming environments," in *Software Visualization*: *Programming as a Multimedia Experience*, ed. John Stasko, John Domingue, Marc Brown, and Blaine Price,MIT Press (1997).

17. Steven P. Reiss and Manos Renieris, "Generating Java trace data," *Proc Java Grande*, (June 2000).

18. Steven P. Reiss and Manos Renieris, "Encoding program executions," *Proc ICSE 2001*, (May 2001).

19. Steven P. Reiss, "Bee/Hive: a software visualization backend," *IEEE Workshop on Software Visualization*, (May 2001).

20. Steven P. Reiss, "An overview of BLOOM," *PASTE '01*, (June 2001).

21. Manos Renieris and Steven P. Reiss, "ALMOST: exploring program traces," *Proc. 1999 Workshop on New Paradigms in Information Visualization and Manipulation*, (October 1999).

22. MIPS Computer Systems, Inc., *RISCompiler Languages Programmer's Guide*. December 1988.