Tool Demonstration: JIVE and JOVE: Java as it Happens

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

Abstract

Dynamic software visualization is designed to provide programmers with insights as to what the program is doing. Most current visualizations either use program traces to show information about prior runs, slow the program down substantially, show only minimal information, or force the programmer to indicate when to turn visualizations on or off. We have developed a dynamic Java visualizer that provides a statement-level view of a Java program in action with low enough overhead so that it can be used almost all the time by programmers to understand what their program is doing while it is doing it.

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.

Understanding such behavior is difficult at best. Software is large, consisting of tens of thousands of lines of code all of which can interact in arbitrary ways. Software involves high-speed execution. Code executes so fast that it is virtually impossible to look at the fine grain behavior of a large system in any meaningful way, especially as it is occurring. Today's systems are long running. The performance and other issues that arise in a modern server system are temporal, arising only occasionally, and are typically dominated by the sum total of the other behaviors of the system. The performance of a particular event or interaction is difficult to isolate. Finally, today's system are complex. They have to deal with multiple threads of control interacting in non-obvious ways and take for granted such complex entities as garbage collectors and library functions as XML parsing.

We feel that the best way of understanding such software systems is to be able to look at a detailed synopsis of their behavior as it happens in such a way that the types of things that we might be looking for, in particular performance issues, thread interactions, and unusual behavior, stand out. Doing this as the software executes lets us correlate what is going on in the software with the appropriate external events (user interactions or other programs) that trigger the corresponding behavior. Using visualization provides a high-bandwidth channel from the data to the observer, letting us use our visual abilities to quickly find unusual patterns and letting the tool use appropriate visual cures to highlight potential items of interest. Using appropriate synopses compresses the data into something meaningful while letting us isolate what might be of interest.

To this end, we have created two visualization systems. Our first system, JIVE, looked both at class-level behavior and at the interaction of threads [4,5]. It demonstrated that it was possible to do dynamic visualization of Java with very low overhead (a factor of 2) while producing meaningful and useful views of the software. JIVE summarized information in terms of intervals of 10 milliseconds or more (under user control). For each class or collection of classes where appropriate it collected 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. In addition, it tracked the state (running, running synchronized, waiting, blocking, sleeping, doing I/O, or dead) of each thread, the amount of time spent in each state, and any synchronizations between threads. This information was displayed dynamically in a compact form that highlighted classes and threads that had unusual behaviors. The system also kept track of the history of the run so the user could revisit or replay the history to further examine the behavior. Views of JIVE can be seen in Figure 1.

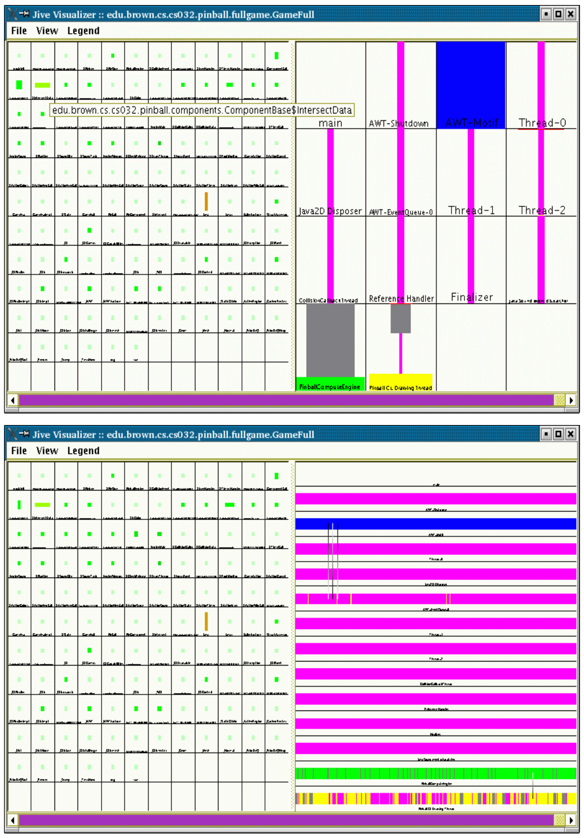
Jive Visualizer :: edu.brown.cs.cs032.pinball.fullgame.GameFull
File
View
Legend
edu.brown.cs.cs032.pinball.components.ComponentBase$IntersectData
main
AWT-Shutdown
AWT-Motif
Thread-0
Java2D Disposer
AWT-EventQueue-0
Thread-1
Thread-2
Reference Handler
Finalizer
Jive Visualizer :: edu.brown.cs.cs032.pinball.fullgame.GameFull
File
View
Legend

FIGURE 1. Views of JIVE

The key aspects to making JIVE successful were to:

- Minimize the overhead so the system could be used on any program at any time.
- Maximize the information gathered and displayed so that complex, interacting patterns could be identified and so it was more likely that the behavior to be understood was represented in the display.
- Provide history information so the user can replay the execution or revisit interesting execution points.
- Key the display so that the types of behavior that are likely to be of interest are highlighted using appropriate visual cues such as color and size.
- Let users adapt the display cues to their particular problems.

While JIVE is very useful for a high-level understanding, it does not provide enough detailed information to address specific problems such as where execution is occurring in the code, why a particular thread is using all the execution time. or even what each thread is actually doing. In particular, we needed:

- Information on where in the source execution is actually occurring so we can determine where the application is spending its time and why.
- Information that relates instruction execution to particular threads of control so we can identify what each thread is doing and not just what state it is in.

This led us to develop an alternative system, JOVE, that gathers data over intervals in terms of basic blocks on a per-thread basis, and then provides a corresponding dynamic display that shows what is going on in the program as it happens [6]. JOVE meets the requirements that made JIVE successful in that it has small overhead (a slowdown of 3-4), lots of available information, a configurable display that highlights unusual information, and a history mechanism to let the user navigate in time over the run.

2. JOVE Overview

In order to obtain more detailed information about the execution of a system at the source level, we needed to look inside methods. In particular, we wanted information about which lines of the program are being executed by each thread and how frequently those lines are executed. While there are several ways of doing this, the standard for counting and performance analysis is to use basic blocks [3,7]. A *basic block* is a segment of straightline code with no internal branches. This means that if the block stars executing, then (in almost all cases) all the instructions in the block will be executed. One can get counts of the number of times an individual instruction is executed by simply determining the number of times the block it is contained in is executed. From this information and compiler information as to what instructions in the executable correspond to what lines in the source, we can use basic block counts to determine the number of times a particular line is executed and the number of machine instructions or byte codes that are executed for each particular line. Moreover, if we know what the block does, for example, that it contains instructions to allocate a new object of a particular type, we can relate this information to the source as well. Thus, the basic tenet of JOVE was to obtain execution information at the basic block level by obtaining basic block counts for each individual thread.

The overall operation of JOVE is based on our previous experience with JIVE. The system consists of four basic components, control, setup, information gathering, and visualization. The control component presents a basic interface to the user. It lets the user define the system that is to be run, provide or change arguments to either the Java interpreter or the user code, identify which classes or packages should be monitored and which should be ignored, and change the settings of how statistical properties are mapped to graphical properties of the visualization.

The setup component is an independent process that takes a set of Java classes and a class path, identifies all the classes that should be monitored, and then patches those class files by inserting appropriate code to do the monitoring. Our current patcher makes use of IBM's JikesBT byte code toolkit [2]. This component produces two outputs. The first is a jar file that contains the modified class files. This is used to replace the original class files when the application is actually run. The second output is a descriptive file that itemizes information about each basic block in the program, including the containing method and class, the source lines, the number of instructions in the block, the number of allocations in the block, and the types of objects allocated by each of these allocations. This second file is used by the visualizer to translate raw counts from the basic blocks into more meaningful information for the user.

The information gathering component is a small library that is loaded with the user's program. This library is called initially and by the instrumented code. It is responsible for keeping track of counts and providing the appropriate data dynamically to the visualizer.

Finally, the visualization component provides two displays. One is a transcript of the program's input and output. The second is the main display of the collected information. This component is in charge of storing and accessing the dynamic information, of creating an appropriate display based on the user's preferences, of dynamically updating that display as the program runs, and of providing time-based access to the data. The displays are fully customizable through a series of dialog boxes.

3. Detailed Dynamic Visualization

Information gathering yields basic block counts for each interval for each thread or context. For display purposes, we accumulate this information by context and globally for each source file, accumulating all blocks for a given file and mapping the basic blocks to the corresponding file lines.

Since our data is organized by file and line, the basic display we chose is a variant of SeeSoft [1]. As seen in the appendix, the actual display is split into a number of vertical regions, each of which represents a file. If there are too many files, then multiple rows are used, with the number of rows chosen to maintain an average height-to-width ratio of 8 to 1. Each file region is split into two parts. On the top, we have the context region. This contains a circular display showing what threads are active in the file. The bottom of the file region is used to display information about the basic blocks from the file, organized by lines.

The overall file region uses size and color to show cumulative information about the file. The width of the file display is by default proportional to the total number of instructions executed in that file. While this could be tied to any statistic, we found that associating it with immediate, rather than total, values caused the display to change too much and became annoying and difficult to view. The color of the file display shows three different statistics. The hue is associated with the number of instructions executed in the current interval; the saturation with the number of threads executing in that file during the interval; and the brightness with the number of allocations done by blocks in that file during the interval. If no code in the file was executed during the interval, the color defaults to gray. Finally, we actually use two colors in the file region, the color computed above and a lighter version of that color. This lets us separate the file region into two parts and display a further statistic, the total number of allocations, using the height of the darker region within the file display.

The context display is shown as a circle at the top of the file display, with the size of the circle being dependent on the size of the file region. The circle contains a pie chart that shows what fraction of the number of instructions executed within the file during the interval were done by each thread. Different threads are shown using different colors, with the colors fixed for each thread and chosen to maximize the differences between adjacent threads.

The basic block region can be used to show up to five statistics for each block. First, the size of the block in terms of width and height can be used. The default display uses the number of lines in the block as the height and the number of instructions executed during the interval as the width. The color of the block then shows the remaining statistics. By default, the hue shows the thread or threads executing in the block in the proportion they executed, the saturation shows the number of threads, and the brightness shows the number of allocations.

While the display provides a lot of information, the programmer needs to correlate it back to the source program. Rather than attempting to label everything in place (where the text would often be too small to read), we use tooltips to provide detailed information to the user.

Figure 2 contains pictures of the visualization of a Pinball program. From this visualization we can detect several things about the program and its execution at the particular point in time. First, from the relative widths of the different file blocks, we can see that most of the execution time is spent in the large block on the left (ComponentBase.java) which represents a generic

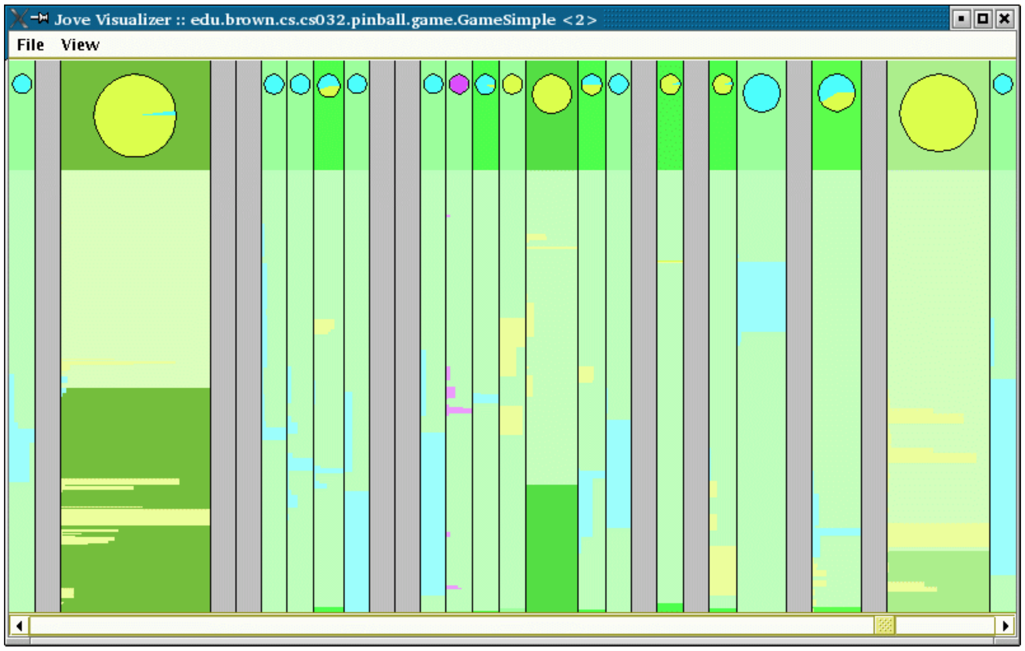
Jove Visualizer :: edu.brown.cs.cs032.pinball.game.GameSimple <2>
File
View

FIGURE 2. JOVE view of Pinball program.

object on the pinball board. Within this file, most of the time is spend in the area designated by the yellow lines at the bottom, notably within the *intersectLline* method. Most of this execution time derives from the computation thread (yellow). Note that a small fraction of the execution of this file is in light blue, representing the drawing thread. The particular methods here set the color and material based on the component.

There are several files where execution is split roughly evenly between the drawing and computation threads. Most notably, the file panel toward the right. This file implements wall components. The blue here represents the routines for drawing the wall; the yellow represents the routines for computing intersections. We can tell from a glance that there is no overlap. Moreover, we can also tell from the small highlighting of this panel, that the code here does few allocations.

Finally, while most of the execution is dominated by the computation and drawing thread, there is one small file containing purple. This represents the sound thread and is an indication that sound uses few computational resources in the program.

4. Conclusion

While not perfect, our efforts show that detailed dynamic visualization of real applications is possible and may be practical as a default way of running the application. The program runs with a slowdown of a factor typically between 3 and 4 depending on the structure of the application. Given the wide performance range of today's machines, this seems to be quite acceptable.

JIVE and JOVE are available from our website at http://www.cs.brown.edu/people/spr/research/bloom.html as part of the BLOOM package.

5. Acknowledgements

This work was done with support from the National Science Foundation through grants CCR021897 and ACI9982266.

6. References

1. Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, Jr., "Seesoft - a tool for visualizing software," AT&T Bell Laboratories (1991).

2. 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).

3. James R. Larus, "Abstract execution: a technique for efficiently tracing programs," U. Wisc.-Madison Computer Sci. Dept. TR 912 (February 1990).

4. Steven P. Reiss, "Visualizing Java in action," *Proc. IEEE International Conference on Software Visualization*, pp. 123-132 (2003).

5. Steven P. Reiss, "JIVE: visualizing Java in action," *Proc. ICSE 2003*, pp. 820-821 (May 2003).

6. Steven P. Reiss and Manos Renieris, "JOVE: Java as it happens," *Proc. SoftVis '05*, (May 2005).

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