JIVE: Visualizing Java in Action

Demonstration Description

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

Dynamic software visualization should provide a programmer with insights as to what the program is doing. Most current dynamic 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 view of a program in action with low enough overhead 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

Software visualization has not been particularly successful for program understanding. Visualizations that look at the static aspects of a software system are only able to provide limited insights and say nothing about the important and more complex dynamic behavior of the system. Dynamic visualizations have been expensive to use because they require the programmer to run the program in an environment that produces the appropriate trace data, generally slowing program execution by an order of magnitude or worse. The result is that programmers generally don't bother using visualizations even if they would be helpful.

We wanted to provide a dynamic visualization environment that could actually be used for real running programs. Such an environment would provide programmers with the information they needed to understand what their program was doing as it was doing it. The environment had to be simple to use, had to minimize the overhead involved with the visualization, had to work with arbitrary programs, and had to provide immediate feedback to the programmer. Moreover, the resultant system had to be not only informative but also entertaining — we wanted programers to use visualization just because it was fun.

A system meeting these requirements would provide a first step toward making visualizations both useful and used. Moreover, it would demonstrate that software visualization could be an everyday thing rather than something only to be used when problems were so severe that nothing else worked.

2. Getting Java Trace Data

The key to a successful real-time dynamic visualization system is obtaining appropriate trace data with minimal overhead.

Rather than attempt to show everything that the program was doing, we break the execution into intervals and then display a summary of what the program did during each interval. This let us cut down substantially on the amount of data that had to be conveyed from the application to the visualization tool and made sense since the visualization tool would have to report summary information in any case.

The information we provide for each interval includes:

- What classes were executing.
- The number of calls to or within each class.
- The number of synchronization calls for each class.
- What was being allocated.
- What was being deallocated.
- What threads are in the program.
- The state of each thread.
- The number of blocks caused by each thread.

This information is obtained by patching the user's program and associated libraries and system files using IBM's JikesBT byte code package.

The patching is kept to a minimum by dividing the application's classes into three categories. Detailed classes are those directly in the user's code. For these we provide information that considers all methods and details any nested classes for separate visualization. Library classes, on the other hand, are grouped into packages and we only generate events for the initial entry into the library. Finally, classes that are neither detailed nor library are treated at an intermediate level of granularity where nested classes are merged with their parent and we only consider public methods.

3. Box Display Visualization

Once the data is available, we needed to have a visualization for the data. In particular we wanted a visualization that could show a large number of objects (e.g. all the relevant classes and packages or all the application's threads) and several pieces of information about each object (e.g.

for a class, the number of entries, the number of synchronization calls, the number of allocations, and the number of allocations by methods in this class; for a thread, the time spent in each of the possible states) in a small display area. We also needed something that was simple and fast since we wanted the visualization to run in real time.

We settled on what we call a box display. Here each class or thread is represented as a box on the display. Within the box we can display one or more colored rectangles. The various statistics can be reflected visually in the vertical and horizontal sizes of the colored display, in the color (hue, saturation and intensity as separate items) of the displayed region, or through textures where the density of the texture is used to represent the corresponding statistic.

For the class display, we typically display five simultaneous values. First, the height of the rectangle is used to indicate the number of calls. Second the width of the rectangle is used to represent the number of allocations by methods of the class. Third, the hue of the rectangle is used to represent the number of allocations of objects of the given class. Fourth, the saturation of the rectangle is used as a binary indicator as to whether the class was used at all during the interval. Finally, the brightness of the box is used to represent the number of synchronization events on objects of this class.

For threads, we create a stack of color rectangles inside each box. Here we vary the height of each rectangle based on the percent of time within the interval the thread is in the corresponding state and the width of each rectangle as the percent of time in this state represented by the given thread. The hue then is used to denote the actual thread state and saturation is used to indicate the number of blocks on this thread.

All of these parameters can be changed dynamically by the user through a dialog box. Moreover, the user can choose, for each of the count statistics, whether to use a linear or a log scale.

4. Running the Visualizer

The actual visualization window is divided into two panes as shown in Figure 1. The left half of the figure show class and package usage information while the right half shows the thread data.

Figure 1 shows the visualizer on an application that uses Java's Swing package. Here one can see the state of the various threads in the application and Swing, see what is being allocated (red), seeing what is doing the allocations (the wide rectangles), see what is actively running. In addition, one can quickly detect which threads are running (green for normal, yellow for synchronized), and what the remaining threads are waiting or blocking for.

The dynamic visualizer works in two different modes. It defaults to showing the current execution as it is happening. Alternatively, the visualizer can be used as a browser over the execution, either as it is happening, or more commonly, after the fact. The scroll bar on the bottom of the window is used to let the user scroll over time for the execution. The display updates dynamically as the user scrolls. This facility is useful for going back and viewing transient events and getting a better understanding of the application's execution. The dynamic display also can be viewed either incrementally or by looking at totals.

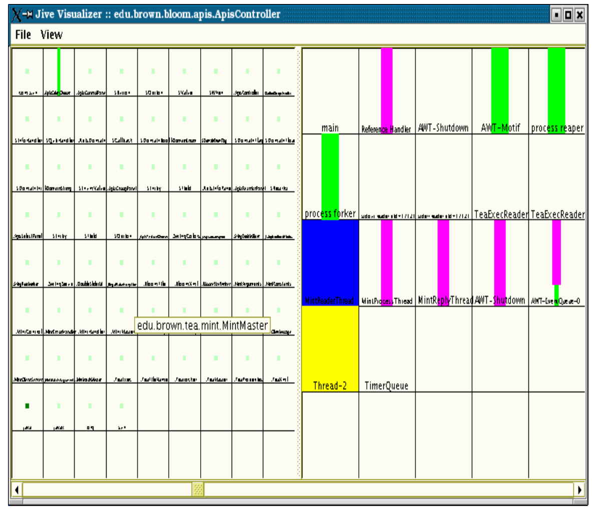

FIGURE 1. Dynamic view of a Java program using Swing.

5. Experience

While not perfect, our efforts show that 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 2 and 3 depending on the structure of the application. Given the wide performance range of today's machines, this seems to be quite acceptable.

We have used the visualization tool on a wide variety of Java programs ranging from simple student programs, to complex single threaded applications (a static checker), to user-interface based applications, to a multithreaded web crawler. We have found that it provides useful information and that watching it makes one think more about what the application is actually doing. For example, we have noted that the visualization makes it obvious when the application goes through various phases. Each phase has its own distinctive set of active nodes, it is easy to tell the phases apart, and the phase transitions are obvious.

6. Demonstration

We will demonstrate the various capabilities of this system. The demonstration would involve running a variety of applications and viewing and interpreting the displays as the systems were running. During the demonstration we would illustrate how the system can be used effectively for software understanding and would illustrate its various features and capabilities.