Demonstration of JIVE and JOVE: Java as it Happens

Steven P. Reiss, 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.

Categories and Subject Descriptors

D.2.6 Programming Enviroments - graphical environments.

General Terms

Performance, Measurement, Design.

Keywords

Software visualization, program comprehension.

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.

ICSE'05, May 15-21, 2005, St. Louis, Missouri, USA.
ACM 1-58113-963-2/05/0005.

2. JIVE OVERVIEW

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 [2,3]. 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). It let the user either display what was currently happening (effectively as a movie) or to go back and forth over the history to look at the visualization in more detail.

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. It displayed this information in a box with a central colored rectangle. The height of the rectangle reflected the number of calls and the width the number of allocations done by the class. Thus, the overall size of the rectangle quickly tells the user where execution is occurring. The hue of the rectangle is used to indicate the number of allocations of the class and the saturation the number of synchronizations. A red or nearly red rectangle then indicates lots of allocations of that class while a dark rectangle indicates lots of synchronizations.

For threads, JIVE 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 for each interval. Each thread was displayed in a box with stack of rectangles each representing a possible thread state. The color of each rectangle was keyed to the thread state, while its height indicated the amount of time the thread spent in that state in the interval. Saturation was used to indicate synchronizations blocks caused by the thread. From the resultant display it is easy to see which threads are running and which are blocked and why they are blocked. Moreover, synchronization

problems often become obvious either due to blockages or by seeing threads that cause others to block.

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.

3. JOVE OVERVIEW

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 [4]. 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.

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]. 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.

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. This is displayed as a pie chart with colors used to indicate the different threads. The user can tell at a glance either what threads are active in what files or what code is being used simultaneously by multiple threads.

The bottom of the file region is used first to display information about the overall code for the file. The width of the file display is by default proportional to the total number of instructions executed in that file. 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.

Basic block information is overlaid on the file display in the file region. Here JOVE shows up to five statistics for each basic 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.

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.

JOVE is available from our website at http://www.cs.brown.edu/people/spr/research/bloom.html as part of the BLOOM package. This work was done with support from the National Science Foundation through grants CCR021897 and ACI9982266.

5. REFERENCES

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

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

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

[4] 4. Steven P. Reiss and Manos Renieris, "JOVE: Java as it happens," *SOFTVIS '05*, (May 2005).