Dynamic Detection and Visualization of Software Phases

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

Software executes in phases. JIVE is a software visualization tool that provides a high-level view of what is occurring in a Java system as it happens, offering information about both what classes are executing, what classes are being allocated, synchronizations, and what are the threads and what state each thread is in. This paper describes how we used the information available to JIVE to detect and then display the current phase of execution.

Categories and Subject Descriptors

D.2.5 Testing and Debugging - Monitors, debugging aids.

General Terms

Program monitoring, software visualization, software phases.

1. INTRODUCTION

Software 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.

We are developing systems that use visualization to provide users with an understanding of the behavior of their software. Understanding software 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 large library routines.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Workshop on Dynamic Analysis (WODA 2005), 17 May ,2005, St. Louis, MO, USA.
Copyright 2005 ACM 1-59593-126-0 17/05/05 … $5.00.

We feel that the best way of understanding such software systems is to look at a 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, phases, 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 cues 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 created a visualization system, JIVE, that looked both at class-level behavior and at the interaction of threads [7]. JIVE demonstrated that it was possible to do dynamic visualization of many aspects of a Java program with very low overhead (a factor of 2) that included 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 as can be seen in Figure 1. 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.

Figure 1 shows an image of JIVE in action. The window is divided into two halves. The left side shows information about classes. Each box here represents either a single class or a set of classes (i.e. all classes in a package or a set of packages). Within the box is a colored rectangle. The height of the rectangle reflects the number of calls made on methods of the class(es) during the interval and its width represents the number of allocations done by those methods. The hue of the rectangle reflects the number of allocations of objects of the given class(es), while the brightness reflects the number of synchronizations done on objects of the class (where darker means more synchronizations). The right side of the figure contains information about threads. Each box here represents some thread in the program. Within this box is a stack of rectangles

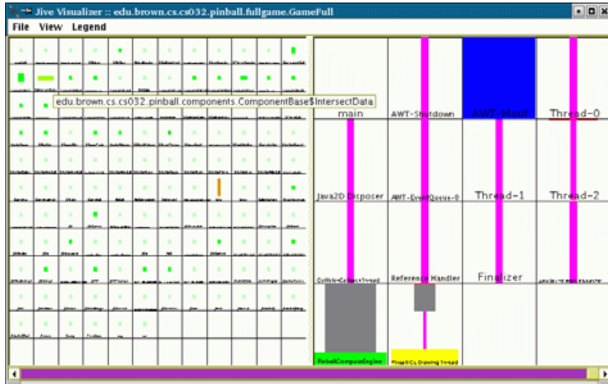


Figure 1. Jive visualization of a Pinball program.

that indicate through their height the amount of time the thread spent in various states (running, running synchronized, doing I/O, waiting, blocked, sleeping, or dead) during the interval. The width of the rectangles indicate the proportion of the particular state allocated to the given thread. In Figure 1 we are looking an interval in the middle of a pinball program. From the class view, we can see that most of the run time is spent in the Component class and the java.* library and that most allocations are for the class *IntersectData*. From the thread view, we can see the AWT thread waiting for I/O while the gravity and drawing threads are each using about 20% of the CPU for their computations and either waiting or sleeping the remainder of the time.

Once we started using JIVE, it became clear that the displays showed information about the phases of the program. On our own programs, where we intuitively knew what the phases were, we could readily distinguish the patterns of class and thread behavior that were occurring during the different parts of the execution. Since understanding the program phase is a central part of understanding the dynamic behavior of a software system, we felt that this was valuable information and was a useful side effect of the JIVE display. As a result of this, we decided that we should attempt to determine automatically the program phase from the available information and then provide phase information as part of the overall display.

This paper describes the particular data that JIVE made available, how we use that data for phase determination, and how we display the phase so that it is readily visible yet doesn't get in the way of the remainder of the JIVE visualization.

2. PROGRAM PHASES

The first problem we faced in attempting to display the program phase was determining exactly what we meant by a phase of the program. Intuitively, for our own programs, we understand what the program should be doing and often think in terms of program phases. However, the visualizer didn't write the code and doesn't understand the program.

A program phase can be defined as a portion of the program's execution that shares a set of common properties or traits. From the programmer designer's perspective, these properties relate to what the program is doing at a high level, e.g. reading input, processing a command, accessing a database, waiting for a connection, or computing some set of values. From the hardware perspective, these properties relate to how the program is using the hardware, what memory is being accessed, and page or disk access patterns. We are interested in phases based on properties that lie between these two. For our perspective, a program phase is a portion of the program that exhibits common behavior at a level the programmer would recognize, e.g. in terms of procedures, threads, processes, or classes.

Hardware program phase detection is used today to facilitate dynamic optimization [5,8,9]. The idea here is to identify portions of the program that look the same to the underlying processor so that appropriate optimizations (e.g. compilation based on branch prediction or memory compression) can be done effectively. The phases that are identified here are likely to correspond to higher-level program phases, but need not. There are several possible reasons for differences here. The first involves the time scale. If we are visualizing execution, then we are really only interested in phases that take a seeable amount of time, i.e. at least on the order of seconds. From the hardware point of view, the time scales can be significantly smaller. Second, hardware phases are interested in the lowest level behavior of the entire program. The software phases we want to visualize define the high level behavior and are restricted to the user's code. Library behavior, for example, while relevant to hardware phases, is generally irrelevant to software phases.

However, the problem of phase detection in the two cases is clearly related, and the techniques that are applicable to the hardware case are also relevant to the software case we are interested in. A variety of techniques have been used for hardware [1-4]. The techniques basically look at some accessible statistic such as working set signature, hardware counters, or branch counters. They check the values of these statistics over time, either periodically or, in the case of [2] on method entry/exit, and test statistically if the pattern of values changes significantly. This is the obvious overall strategy and the one we use. Our approach differs in that we use higher-level statistics and bias those statistics to look primarily at the user's code as opposed to library code. A second difference from these techniques is that we are doing our computation in software rather than in the hardware and at larger granularities and thus can afford more detailed computations.

Software phase detection has also been tackled in postmortem visualization as extensions to the EVolve system [10,11]. This work lets the programmer define phases manually by indicating particular methods that start the phase. It also does automatic detection based on various time visualizations such as method or stack usage over the complete execution, looking primarily for steady-state behavior. Our work differs in that we are attempting to do the phase detection on the fly and want to concentrate on the user's application rather than the whole program.

3. JIVE DATA

JIVE operates by continually gathering data from the application and periodically sending the data along to the visualizer. The data gathering is done by inserting code into the binary (Java class files) just before execution. The inserted code maintains counters for the various classes and packages, and keeps track of the thread states. Periodically (every 20 milliseconds by default, but can be set from 10 milliseconds to minutes), the counters and thread state information is sent over a socket to the visualizer and the counters are cleared for the next period.

JIVE's counters are computed on method entry. JIVE divides the application's classes into three categories, *Detail*, *Normal*, and *Library*. *Detail* classes are meant to represent the users' application per se. For these classes, JIVE counts every method call. *Library* classes are represented by packages or sets of packages. For example, the library class denoted by *java.util.* represents the set of all classes and packages under java.util, and the count for this class group represents the number of times a public method in one of the underlying classes is called from the user code. *Normal* classes are handled at the class level as are detailed classes, but only public methods are counted. JIVE also keeps counters of the number of allocations done for each class or group of classes by the code in the *Detail* classes, both in terms of allocations of a class and allocations by a class. By default, JIVE automatically computes the set of library, detail, and normal classes based on the class name of the main program and knowledge of the standard libraries.

The thread state data gathered by JIVE provides the amount of time each thread in the application spent in each state during the current interval. This is gathered by identifying the library routines that indicate thread changes and processing the entry and exit of those routines.

Let $C = c_1 \ldots c_n$ be the set of classes or groups of classes that are being counted at run time. Then the data that the visualizer obtains for each period of execution is a set of vectors indexed by C:

V_c^e = number of entries into methods of c

V_c^a = number of allocations of objects of class c

V_c^b = number of allocations by methods of class c

In addition, for each thread it obtains a tuple *T* with the amount of time that the thread spent in each possible thread state. When we want to note that a vector is associated with an interval i, we will write this as ${}_iV_c^e$.

This data is the basis for the visualization and is substantially more than is used in hardware phase detection. Moreover, it is biased toward the actual application by the use of different categories of classes and hence should provide a strong basis for determining the program phases we are interested in.

The class data available in JIVE is biased toward the user program. The data covers a fixed time interval of tens of milliseconds. For this interval it includes all the calls in the detail classes, all public calls to methods in the normal classes, and all calls to methods of the library classes from the detailed or normal classes. It ignores counts for calls within the library (other than those that change thread states) and private methods of normal classes. JIVE can safely ignore the other calls and still provide an accurate view of execution because the interval granularity is coarse and we are looking at counts rather than the amount of time spent in the routines.

4. PHASE DETERMINATION

The basic approach to computing the current phase is to construct a vector *W* for each interval from the available V^e, V^a, V^b, and *T* data. The vector for the interval is then compared to that of the previous interval. If the two vectors are statistically "close" then we assume that the phase of the previous interval persists into the current one; if the two vectors are significantly different, then we assume that we are starting a new interval.

There are two issues, however, that make this more complicated. First, we want to avoid spurious, very-short lived phases. We actually want to look at the counts not over a single interval, but over a range of intervals, and see if the phase has actually changed. Second, we don't just want to detect phase changes, we want to detect common phases. That is, if the values in the new interval are significantly different, we want to see if they match some previous known phase and, if so, note that we are going back to that one.

We first needed to decide what data we wanted to use for phase determination. We wanted to keep this simple while still including whatever might be relevant. We noted that the allocation counts are generally going to be dependent on what is currently being called, both because routines tend to allocate the same type and number each time they are called and because each allocation yields a call to at least a constructor. This meant that we could safely restrict ourselves to looking at V^e, the set of entry counts, to cover the class data. We also decided that the thread data could and often should be ignored. It could be ignored because each thread or set of threads generally execute in their own sets of routines and hence the activity of the threads would be reflected in the entry counts. It should be ignored in the relatively frequent case where there are pools of threads that share a common purpose. The phase of the program is not dependent on which particular threads are active, but rather on the fact that some threads in the pool are active. The thread data we have is very thread-specific and would falsely differentiate between these cases.

Next we wanted to make the time span for phase determination larger than a single interval. We did this by computing *W* from V^e for a window of intervals ending with the current interval. The size of the window ω, we left as a parameter to be determined experimentally. Given this, we defined W_i, the phase counts for interval i, as

$$W_i = \left\| \sum_{i - \omega < j \le i} {}_jV^e \right\|$$

where $\| \ \|$ denotes normalizing the vector, that is either W_i is zero or is normalized such that its size is 1, $W_i \bullet W_i = 1$.

We assume we have a vector P_s of the same size and coefficients as W_i that represents the current phase, s. As a simplification, P_s can be thought as W_{i-1}, the characterization of the previous interval. To compare the current interval with the current phase, we then compute $P_s \bullet W_i$. This dot product will be close to 1 if and only if the two vectors are similar. We then assume that W_i is part of the phase represented by P_s if and only if $P_s \bullet W_i > \varepsilon$ where we determine the value for ε experimentally. This provides us with the mechanism for determining if the current interval is part of the current phase.

Using W_{i-1} for P_s is an over simplification. What we really want to have for each phase is a vector that truly models the whole phase and not just the last interval. We do this by actually letting P_s be the normalized vector that results from looking at the counts from all the intervals considered part of that phase, i.e.

$$P_s = \left\| \sum_{j \in \text{Phase s}} {}_jV^e \right\|$$

We maintain the various P_s incrementally, adding the V^e for the current interval to the total counts for the phase once the phase is determined and then normalizing.

The last step involves determining what happens when W_i does not match the current phase. In this case we look at all previously known phases, P_t and determine the phase that maximizes $W_i \bullet P_t$. If $W_i \bullet P_t > \delta$ for some experimentally determined constant δ, we assume that W_i is actually part of phase P_t and set the current phase to P_t. Otherwise, we start a new phase for this interval and set its initial values to the current V^e.

The computations that are needed here are relatively simple. The size of the vectors is the number of classes and groups of classes that are part of this display, generally between 25 and 200. We keep a windowed count for each interval, so that the computation of the counts for a new interval can be done with one addition and one subtraction for each vector element. The extra overhead of computing phases is then insignificant in the overall JIVE framework.

5. DISPLAYING THE PHASE

Once we had computed the phase we needed to include it as part of the JIVE display. We wanted the phase display to be distinctive and obvious while at the same time not obtruding into the existing class and thread displays.

To accomplish this we decided to color the scroll bar at the bottom of the window with a color that represents the phase. If we could make the different phase colors be distinct enough then phase changes would be obvious to those looking for them but would not standout if you wanted to ignore them. There were two basic problems, however. First, Java on the Macintosh does not support colored background on scroll bars. Here we changed the background color of the menu bar instead of the scroll bar. The real difficulty, however, is that we do not know a priori when we run an application how many phases there are going to be or which phases are going to be adjacent.

To handle this we map the color space into a 0-1 interval from red to violet. The first phase is assigned to 0 (red) and the second phase is assigned to 1 (violet). Beyond this, we bisect the intervals and assigned phases alternately. Thus p2 = 0.5, p3 = 0.25, p4 = 0.75, p5 = 0.125, p6 = 0.625, p7 = 0.375, p8 = 0.875, p9 = 0.0625, etc. This maximizes the differences between phases when there are a relatively small number of phases and provides as much contrast as possible between adjacent phases.

6. DETERMINING PHASE PARAMETERS

The above left us with three parameters that needed to be determined experimentally, the size of the window ω, the cutoff value for matching the current phase, ε, and the cutoff value for matching a prior phase, δ. To do this we one of our own systems (the program described in the next section) where we could determine a priori what the phases should be and experimented with different parameter settings so that JIVE correctly chose the proper phases. This various settings were then double checked by looking briefly at the phase changes of other programs to ensure the values produced meaningful results. The latter checks showed that the original values were sufficient, i.e. that the parameter settings were global and not dependent on the particular program.

For the window size, we tried values ranging from 1 to 10 intervals. As expected, a value of 1 caused spurious changes that we felt shouldn't be there. Higher values, typically above 5, caused noticeable delays in the phase changes. For example, the program asked for input and then a second or so later, the phase changed. Values between 2 and 4 seemed to work well and we settled on a value of 3.

The cutoff value for the current phase needed to be high enough so that new phases could be detected rapidly, but not too high that minor differences cause spurious phase changes. Here we experimented with values between 0.8 and 0.99. It turned out that the actual differences between the phases was significant, i.e. a vector for one phase and a vector for another typically had a dot product that was 0.75 or below. This is probably due, in large part, to the fact that the vectors concentrate on the user's program and merge most library routines. Much of the execution time is spent in the library routines which are common between phases, but our analysis elides this commonality. The large difference between phases meant that the actual setting of the cutoff value did not make that much difference. We did find that values above 0.95 tended to yield occasional spurious phases while values below 0.85 tended to occasionally overlook phase transitions. We thus use a value of 0.90 which is in the middle of the range of "working" values.

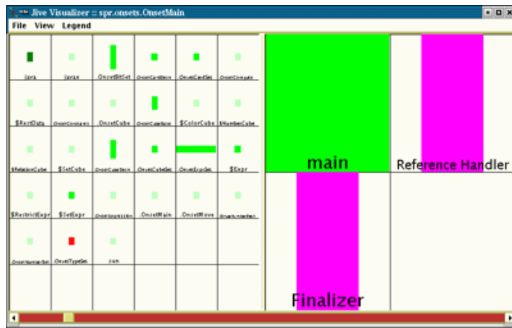
The final parameter, δ, determines whether to form a new phase or restart an old phase. We originally made this independent of ε so that we could prefer reusing phases to starting new phases and because the vectors at a phase transition tend to be further from the phase norms than the vectors in the middle of a phase. This constraint said that δ should be less than ε. However, we found that putting δ less than ε sometimes identified the previous current phase as the new phase, which wasn't what we wanted. Because of the high correlation among phase vectors, the actual setting of δ again made little difference in the 0.85 to 0.95 range, so we ended up setting δ to the same value as ε, 0.90.

7. EXAMPLE

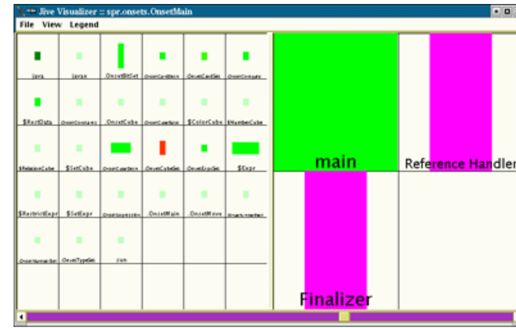
To demonstrate the how the different phases are reflected in the JIVE display, we use our implementation of the set theory game OnSets manufactured by WFF N' PROOF and ran it under JIVE. The different phases selected by JIVE are shown in Figure 2.

The first two phases deal with initialization. The first phase occurs when the program is taking the configuration that was randomly generated and constructing the set of all possible final configurations. The second phase represents the last portion of initialization where the set of possible restrictions is computed. These show up in this particular run because the random case is complex and the phases are long lived; when the program is run on an easy case, these two phases are either combined or disappear entirely. Note that these phases differ in both where execution is occurring (as noted by the height of the various class rectangles), in where allocations are occurring (as noted by the different widths of the rectangles), and in what is being allocated (as noted by the different colors).

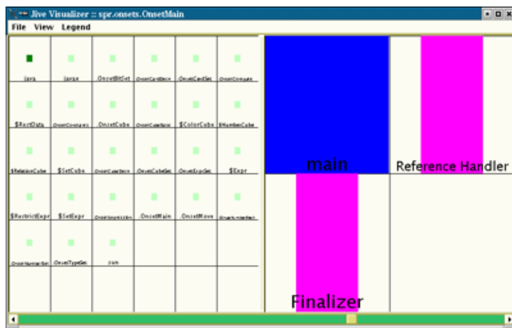
The third phase represents the time spent waiting for input. Note here that there is no activity in the class views and the threads are all in a wait or I/O wait state. This phase actually represents a vector of all zeros since nothing is happening in the program. We treat this empty phase as special because it requires special matching since its dot product with any exist-



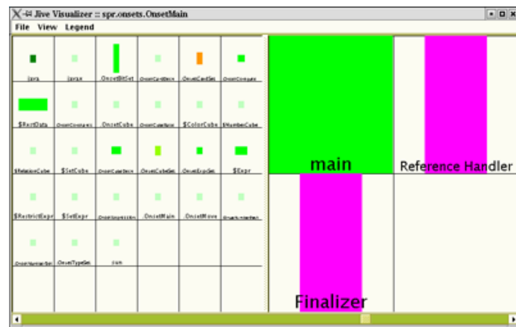
a) Initialization of possible moves.



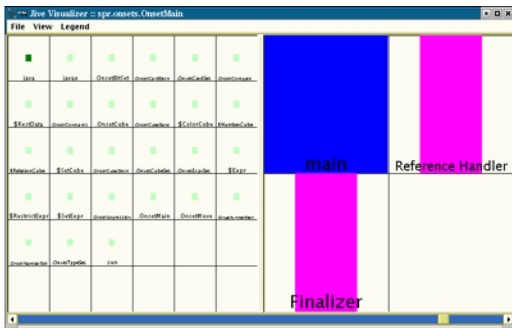
b) Initialization of restrictions.



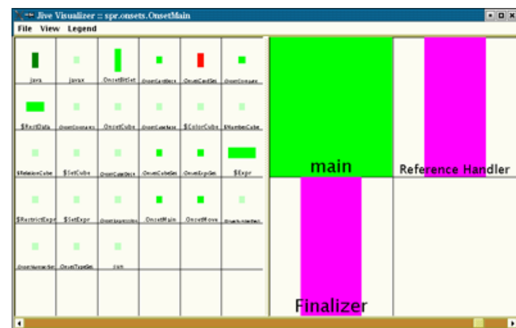
c) Waiting for input



d) Computing next move possibilities



e) Outputting move possibilities



f) Computing final configuration possibilities

Figure 2. Phases from the OnSets program

ing phase (even itself) is zero. The first time we find an empty vector, we create an explicit zero phase and remember it. When we find a subsequent empty vector, we automatically choose this saved phase.

The fourth phase is the normal computation of the next move once the user has given input. The fifth phase then represents the output of possible next moves to the user. This phase does not always occur between move computation and input, and when it does it is generally short lived because of the small amount of program time spent doing this output. Note that the display looks identical to the input wait display. This is an artifact of using a sliding window to compute that phase. The differences actually occurred in the previous interval.

The last phase represents the computation that the program does when there is a winning move. This is a very different computation than computing the next move and is identified correctly as a different phase. Again the differences can also be seen visually in the various class rectangles.

8. EXPERIENCE

We have run JIVE on a variety of our own systems for which we have some idea of what the phases should be and then compared the phase transitions marked by JIVE to what we would expect. These are in addition to the example of the previous section which was used for determining the phase parameters. These include a pinball program, a tool for finding inconsis-

tencies among software artifacts, a software specification checker, a web crawler, a visual query language for visualization, and a front end for handling interfaces for Internet-scale programming.

For the most part, the phases selected by JIVE matched our intuitive notion of what the program is doing. There are some places where we would expect a phase and didn't see one. For example, in the Pinball program, we only have 3 phases: initialization, running, and paused. There is no separate phase for when the ball is not moving, for when the ball collides and a sound is played, or when keys are pressed causing flippers or the shooter to move. Analyzing why shows that the program is dominated by the physics and graphics computations, which proceed whenever the program is running, and the sound computations are done mainly in native routines and don't really show up in the count vectors.

When we run JIVE on the multithreaded web crawler, it typically identifies four phases. Three are clear, initialization, running, and finalization (when it is writing out the set of new links that it didn't crawl to). The fourth phase was a surprise. It occurred periodically in the execution. Further investigation showed that this phase actually represented garbage collection.

When we run JIVE on our constraint-based program evolution environment, CLIME [6], the system identifies seven phases which we loosely correlate as initialization, reading data files, waiting for external input, processing constraint data, setting up the constraint display, handling mouse input, and setting up the file display. This is reasonably close to what we would expect. One major difference is that we would view the initial set up of constraint data and future setups based on clicks to be different phases. However, in the actual program these share much of the same code and do the same operations, so a single phase makes sense. Similarly, we would expect that reading the original data files that define the project and reading the data files that define the constraints would represent different phases. Here we noted, however, that in both cases the program is essentially idle and the phase determination code can't distinguish between them.

Most of the other programs we have looked at exhibit similar behavior, with the phases that are selected by JIVE indicating actual program phases, but with some differences that one might intuitively expect.

9. FUTURE WORK

Our experience with JIVE shows, it should be possible to dynamically detect and convey software phases as the program runs. There are several potential pitfalls that can be addressed through future work.

First, the current window size is fixed, but actually should vary with the size of the chosen interval. Since we rarely use other than the default interval with JIVE this currently isn't much of a problem, but is logical next step. The size also might be put under user control, especially if we can provide data at smaller intervals (right now, there are system limitations that make 10 milliseconds the smallest interval), since the length of phases does vary from program to another and can be based on what the programmer is currently interested in.

Second, there should be a means for the user to label the various phases of the program. The current use of color works nicely as long as the number of phases is small. What is needed is a display showing the phases and letting the user see where they occur over the run. This could be done, for example, by providing a graphical region above the scroll bar that showed the phase history of the program in terms of intervals and colors and a separate legend window where the user could actually label the phases.

Third, JIVE provides phase indications for the current run. The programmer might want to keep track and use consistent phase colors or labels across multiple runs of the same program. This ability is provided, for example, in EVolve [11]. This could be done by providing a mechanism for saving and reloading the phase vectors for the program along with the user's phase labels if they were available.

JIVE is available for download as part of the Bloom software visualization project at

http://www.cs.brown.edu/people/spr/research/vizjive.html.

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

10. REFERENCES

1. Ashutosh S. Dhodapkar and James E. Smith, "Comparing program phase detection techniques," *36th IEEE/ACM Intl Symp. on Microarchitecture*, pp. 217-227 (2003).
2. Andy Georges, Dries Buytaert, Lieven Eeckhout, and Koen De Bosschere, "Method-level phase behavior in Java workloads," *Proc OOPLSA 04*, pp. 270-287 (October 2004).
3. Jeremy Lau, "Structures for phase classification," *IEEE Intl. Symp. on Performance Analysis of Systems and Software*, (March 2004).
4. Jeremy Lau, "Transition phase classification and prediction," *HPCA '05*, (February 2005).
5. Doron Nakar and Shlomo Weiss, "Selective main memory compression by identifying program phase changes," *Third Workshop on Memory Performance Issues*, (2004).
6. Steven P. Reiss, "Constraining software evolution," *International Conference on Software Management*, pp. 162-171 (October 2002).
7. Steven P. Reiss, "Visualizing Java in action," *Proc. IEEE International Conference on Software Visualization*, pp. 123-132 (2003).
8. Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder, "Automatically characterizing large scale program behavior," *ASPLOS '02*, (October 2002).
9. Timothy Sherwood, Suleyman Sair, and Brad Calder, "Phase tracking and prediction," *30th Intl Conf on Computer Architecture*, pp. 338-349 (2003).
10. Qin Wang, Wei Wang, Rhodes Brown, Karel Driesen, Bruno Dufour, Laurie Hendren, and Clark Verbrugge, "EVolve: an open extensible software visualization framework," *Proc of SoftVis 2003*, (June 2003).
11. Wei Wang, "EVolve: An extensible software visualization framework," *McGill University School of Computer Science*, (2004).