Generating Java Trace Data

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

ABSTRACT

We describe a system for gathering and analyzing Java trace data. The system provides relatively complete data collection from large Java systems. It also provides a variety of different analyses of that data for use with a software visualization system.

1 INTRODUCTION

As Java programs get larger and more complex, they become more difficult to understand. We have embarked on a project that attempts to use software visualization to provide this understanding. This paper describes the first part of that effort, a package that uses the JVMPI interface to collect and then analyze Java traces.

Java was originally used for small scale applications such as applets or small clients for a more complex, non-Java server. Today, the language is being used for constructing large-scale systems including scientific and data-intensive applications. As the applications have increased in scale, understanding their structure and behavior has become both more important and more difficult. The authors of a large Java system need to understand what is going on inside the system as it is running. They need to see how the various threads interact, whether the system is over or under synchronized, where the performance bottlenecks are, how much memory is actually required, where objects are being allocated, where the memory leaks are, which classes are active when, and how the various classes in a complex system interact.

Our approach to addressing such specific understanding problems is to provide programmers with a system that makes it easy to define software visualizations that can provide the answers quickly and efficiently [8-11]. There are several aspects to such a system. The first is to provide a wide variety of relevant data that can be used to drive the visualization. Here we let the programmer combine both structural data describing the system with several different analyses of the trace data that is obtained by running the system. The second aspect of our visualization approach is to provide a framework to make it easy for programmers to specify what data they need to visualize to understand the problem at hand. Here we use a visual query language based on the universal relation assumption that lets programmers select the data of interest without having to know the structure of the underlying databases or how the different analyses are done. The final aspect of the visualization approach uses a graphics framework to provide a range of high-quality 3D visualizations of the selected data. Here we have emphasized visualizations that can display large quantities of data in a small space using various abstractions rather than more traditional box-and-line drawings.

This previous work worked with C++ programs and demonstrated the utility of the approach. Our current efforts involve extending this work to support Java. This is more than simply porting the previous code to handle Java rather than C++. Understanding Java involves a somewhat different set of issues than we had been addressing. In particular, our previous efforts had not tackled the problem of multithreaded computations (the lack of a standard threading model for C++ made this difficult), and they paid a lot of attention to memory allocations; our work for Java has emphasized threading and takes a very different view of how memory is utilized.

Looking at trace data for visualization is not something novel. Performance visualizations have been around for 20 years in various forms. (Even the old UNIX *prof* command had a graphics mode to produce histograms.) The FIELD environment, among others, provided dynamic views of the heap, file I/O and performance while the program was running [7]. Various efforts at Georgia Tech and elsewhere have developed a number of different dynamic visualizations including the notion of call dags that we are using [2,12]. More recently, the IBM Jinsight efforts use relatively complete call trace data to provide the user with insights into how the program is running and to address such questions as Java memory leaks [1,6].

Our approach differs from previous efforts in its ability to combine trace data with structural data about the program, in its ability to let the user define what information should be visualized and how it should be displayed rather than only providing a small set of fixed displays, in its emphasis on high-density abstract visualizations, on its ability to handle C++ as well as Java (and thus to display information about multi-lingual systems), and on providing a general framework that can be used for a wide variety of different analysis of the same trace data.

In this paper we look at our on-going efforts at generating the appropriate data for Java visualization. The next section looks at structural data. Section 3 looks at the problem of generating trace data. This is followed by a description of the various analyses that we are doing with the trace data in order to get data to visualize. We next consider some of the resultant visualizations and conclude by describing our continuing and future efforts.

2 STRUCTURAL DATA

Most of the early work on software visualization dealt with visualizing the structure and organization of large systems, for example showing the call graph or the class hierarchy. Our experience was that such diagrams were helpful for navigation, but, except for some examples from reverse engineering, were not too helpful for

File:
- File name
- Date last modified
- Dependencies

Dependency:
- Depends on file
- Dependency type

Definition:
- Name
- Definition scope
- Symbol type
- Storage type
- Flags
- Parameters
- Type
- New scope
- Start and end location

Reference:
- Name
- Scope
- Definition
- Flags
- Start and end location

Scope:
- Name
- Parent scope
- Include scopes
- Scope type
- Associated symbol
- Associated type

Type:
- Name
- Type style
- Base type
- Definition
- Flags
- Parameters
- Super type
- Interfaces
- Scope
- Primitive type

Figure 1. Overview of the information in the structural database.

software understanding. However, our experiences with trace visualization showed that one often wanted to combine such structural data with the trace data in order to create more meaningful visualizations. As examples, one can look at trace data more compactly by collapsing methods into classes and classes through their class hierarchy, and one can get a better understanding of the execution of a program by seeing how different class hierarchies actually interact at run time.

The first step in providing a comprehensive visualization strategy for Java applications was to ensure that the necessary structural information was available. Our previous efforts provided such information in the form of a program database that was generated using information provided by the compiler (.sb files from Sun's C++ compiler for example) [4,7]. This program database was organized as an in-memory relational database containing relations for references, definitions, scopes, files, calls, the class hierarchy, and class elements.

In moving to Java, we wanted to both simplify and extend this approach. We moved to a more object-oriented database from a purely relational one. This let us store the definition corresponding to each reference as a link rather than attempting to compute it each time. We eliminated the separate relation for class elements since in Java all definitions are essentially class elements and the two relations can be easily combined. (In C++ the information for class members, i.e. protection, visibility, and properties such as virtual, static, and friend do not apply to general definitions.) Finally, we replaced the class hierarchy relation with one that describes types in the language in general, providing information about the underlying type algebra. The result is shown in Figure 1.

Our previous work had demonstrated that it was important to gather this structural information without bothering the programmer. That is, the information should be gathered automatically and with a minimum of overhead. Our current approach has the programmer define the set of directories and files that constitute the system in question using either an interactive visual tool or a simple text editor. Once this is done, our program database will automatically gather and maintain the database as the specified files or any file in the specified directories changes.

To gather this information accurately and quickly, we took the IBM Jikes compiler and modified it to dump the necessary information from the abstract syntax trees that it produces in the front end. The information is produced on a file-by-file basis and is stored as XML files. These file are then read by our specialized database system that is capable of quickly discarding all previous information from a file and inserting all the new information. The database actually runs the Jikes compiler in its incremental mode so that the compiler shares some of the load of determining what files need to be rescanned and so that updates beyond the first are extremely quick. Our experience with Jikes has been that it is extremely fast (especially since we are not doing any code generation). For example, it takes Jikes less than ten seconds to generate the data for about 30,000 lines of Java comprising our Java tool set. (It currently takes our database under three minutes to process the resultant data and build and store the resultant 12M database initially; updates after that are almost immediate.)

3 TRACE DATA

The best visualizations are created by correlating a variety of different data from a single trace. For example, one interesting visualization we have worked with in our previous system involved correlating allocations of objects along with the dynamic call graph. This allowed us to get an overview of how segmented memory was for each portion of a system's execution. Our goal in collecting Java trace data was to generate the data so that any number of different analyses could be done after the fact for visualization.

This goal required that our trace collection meet certain criteria. First, it should be as complete as possible. One cannot analyze data that one does not have. Ideally, we wanted to know everything that was possible, including a full call trace of all the active threads, locking information, memory utilization, and performance information. Second, one needs to have a consistent way of referring to the program elements from the trace. For example, one needs to

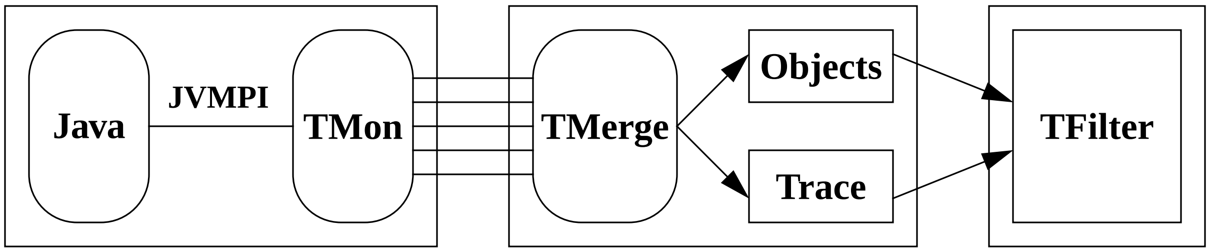


Figure 2. Overview of the Java trace data generation architecture.

have consistent object, thread, class and method identifiers throughout the trace. Third, the data needs to be collected with a minimal impact on the performance of the underlying program. This requires that the execution of the program not be slowed down so much that any interactive portions of it are unusable and that the tracing does not introduce side effects such as synchronizing multiple threads. Finally, we wanted to do this in a portable way with a minimum number of changes to the JVM or the user's program.

3.1 JVMPI

Java provides an interface for generating performance information, the JVMPI, that meets most of these criteria. It provides hooks into the JVM that can be used without modifying the user program or the JVM itself. It provides almost all the data that we needed for visualization. It is relatively fast as well, with most of the hooks being integrated directly into the JVM and with little extra overhead required other than the call to the JVMPI code.

The JVMPI however, does not meet all the requirements. It is fast because it reports addresses rather than true object identifiers for objects, classes, threads, etc. Because of garbage collection, such addresses will change over the course of a run. Thus it also generates appropriate events indicating such changes and requires that the application track them. It also makes the calls to the JVMPI from within the context of the current Java threads. Thus multiple calls can be occurring simultaneously to the JVMPI application. Since we needed to combine the multiple threads so as to provide a single trace as output, our code had to merge all these calls. However, it had to do so without synchronizing the Java threads.

We addressed these problems by generating multiple data streams from the JVMPI calls, one per thread, and then merging these data streams and, at the same time, mapping the identifiers from those that the JVMPI provided to our own consistent and unique ones. The result of the trace is than stored in two parts, one that contains information about the identifiers (for example, the class associated with an object identifier or the methods and file associated with a class), and one that contains a sequential description of what the Java program is doing. The later contains events for each call and return, each object allocation, as well as locking and synchronization information. In it, all the threads are interspersed to illustrate as closely as possible how the execution actually occurred.

3.2 TMon

The architecture for generating this data can be seen in Figure 2. Here the user's Java program is interpreted (or compiled and executed using the JIT compiler[1]) and the JVMPI is used to provide the appropriate information to our front end trace package, TMon. TMon then creates an output stream for each thread. It does this by looking at what the current thread is when it gets control, and checking if that thread is known. If it is a new thread, then a new output stream is created. This requires some synchronization to ensure that the streams get unique names, but only happens once and should have a minimal effect on the application. Otherwise, data from the current JVMPI entry is recorded into the current thread's output stream. Finally, if the entry was for a thread exit, then the output stream is closed.

The trace data that is generated by TMon is designed to reflect the parameters of the JVMPI events. Thus, little or no processing is performed on the data. One simple exception involves the need to have performance information included in the trace. This can be obtained by using the JVMPI to get the run time of the individual thread each time an entry occurs. The corresponding call, however, can be quite expensive. (On the Sun's, it takes several microseconds to get the value of the nanosecond timer.) Instead, we obtain this information only every 4096 actual events and approximate for those events that come in between. A second modification involves saving a sequence number with each trace record. The sequence number is global and is used to intersperse the traces from the different threads. Here TMon maintains a global counter which is incremented for each record that is output. No synchronization is kept for the counter so that multiple threads might read (and store) the same value if they happen to be executing at the same time. This is not really a problem since the effective serialization of such coexecuting threads is ambiguous in any case.

The output streams used by TMon can take one of two forms. They can first be files. Here TMon is given a directory (through an environment variable at start up) and creates new raw trace files as needed. Alternatively, we have implemented a shared-memory communication mechanism. Here TMon uses a shared memory buffer to communicate with the back end to set up a new shared memory buffer for each new thread. Once the buffer is set up, TMon simply adds information to that buffer as needed.

1. We did make minor modifications to the JVM to allow simultaneous use of JIT and JVMPI.

Thread:
- Thread id
- Total execution time of thread
- Thread name
- Thread group name
- Thread parent name

Object:
- Object id
- Class id
- Array type
- Object size

Class:
- Class id
- Class name
- Source file name
- Number of interfaces
- Number of methods
- Number of static fields
- Number of instance fields

Method:
- Method id
- Class id
- Method name
- Method signature
- Start and end line

Field:
- Field id
- Class id
- Field name.
- Field signature

Monitor:
- Monitor id.
- Monitor name.

Figure 3. Information stored for trace objects.

In order to let the application run as fast as possible, the back end reads the shared memory buffer as often as possible. Rather than processing this information immediately (which would restrict the front end to run at the speed of such processing), it stores the message for latter use. This ensures that TMon will almost always have room in the buffer for its information and will have to wait a minimal amount of time. We take advantage of this by having TMon do a busy (unsynchronized) wait if the buffer is full most of the time. Only if the back end has stored over some large number (currently a million) messages, will it tell TMon to actually stop processing that thread until at least half the stored messages have been processed. This tends to work well, especially with interactive applications since the interactive threads are rarely stopped and can work at a reasonable speed.

3.3 TMerge

The output from TMon, either from the set of files generated or from the set of shared memory buffers, is processed by the next phase system, TMerge. The purposes of TMerge are to merge the various output streams into a single stream containing the results of each thread, to accumulate from the trace information about the various objects, classes, methods, and threads into a random-access database, and to provide a consistent set of unique identifiers for all such items.

TMerge works by setting up a priority queue of all the available data streams ordering these by the sequence numbers inserted in the trace. This ensures that the trace records are read in approximately the correct order. It maintains a hash table that maps the “current” JVMPI ids into the unique trace identifiers that are used as part of the trace output. Each trace entry is then processed for three things.

First, TMerge processes each entry to determine if it adds to or updates the mapping from JVMPI ids to trace identifiers. If the identifier is new, a new logical identifier of the appropriate type is created and a new mapping is set up. (The system knows from the context what type of identifier is expected in each situation.) Second, a trace record indicating that an object has moved is used to map the corresponding objects from their old JVMPI id to the new one. Note that we say objects here because a class is represented both as the corresponding class object (which can be used as an object for calls, etc.) and as the identifier describing the class.

Second, each entry is processed to add or update information in the database of information about identifiers. Here information from the trace records is stored with the proper type of item in the database. For example, an object allocation entry causes a new object to be defined with the appropriate class pointer, while a class load entry will add not only the class object and the class type to the database, but will also add objects describing each field and method of the class using their names and signatures. The particular information stored with the different types of objects is shown in Figure 3.

The final processing of each trace entry is used to generate the resultant trace file. Each trace entry contains the event type and the thread id. The remaining information is type dependent and can be seen in Figure 4. Note that the thread execution time is not stored with each entry, but is stored as occasional trace records indicating additional run time. This follows from the scheme whereby this information is only recorded periodically. It can be used by the program processing the trace data to generate approximated execution times for each call or other events.

3.4 Results

The current system is able to trace execution rather efficiently. We took a commercial program written as a server for processing search queries consisting of about 30,000 lines of Java. A sample run with about a thousand client requests takes about 25 seconds with JIT and 2:39 minutes without JIT. This run generates 1.68 gigabytes of trace data and 167 megabytes of object data as the result from TMerge. Generating this trace data takes about 17 minutes (a factor of 6-40). (We are still working on attempting to improve this further through additional optimizations and simplifications.) Moreover, compaction such as provided by gzip or the UNIX compress command, can shrink the raw data by a factor of five without any loss of information.

4 ANALYZING THE TRACE DATA

While we have only begun to work on a full range of trace analyses for visualization, we already have developed a number of visual-

- Class Load and Unload
  - Class id
- GC Start and Stop
  - Number of used objects
  - Total used object space
  - Total object space
- JVM Initialize and ShutDown
- Method Entry:
  - Method id
  - Object id
- Method Exit:
  - Method id
- Monitor Enter and Exit:
  - Object id
- Monitor Wait:
  - Object id
  - Timeout
- Object Allocate and Free
  - Object id
- Thread Start and End
- Run time:
  - Thread execution time.

Figure 4. Summary of trace file entries.

ization using a variety of different analyses of the trace data, both for C++ and now for Java. The different analyses can be divided into three basic groups. The first include information obtained by looking at the sequence of calls. These are used to provide detailed dynamic views of the execution of generally small programs (although they can be used for small portions of large programs as well). The second provide views of the use of memory throughout the application. The final set provide useful summary views gleamed from the trace data.

In order to facilitate visualization, each of these analyses is output as a sequence of tuples for a corresponding relation. The visualization package treats these trace files as a single-relation database when collecting and setting up the visualization. The general package that does the analysis, TFilter for Java or Vark for C++, can cache its results for future visualization as well as returning the results directly.

4.1 Call Trace Analysis

The basic call trace visualization is done from a sequence of call (and optionally return) records in a tuple. These are ordered by sequence number. Each record shows where the call was from, what the call was to, the time of the call, the active thread, the level of the call, and the first argument (the object for a method call). A return here is indicated as an empty to routine. An alternative form of trace outputs the entries in completion order. The advantage of this is that each entry is extended with the time spent in the routine locally and the time spent in the routine and everything it called.

A second form of call trace tries to shrink the size of the resultant data by only providing information about the top of the stack. Entries here indicate a call to a specific function or a return. This is used both for some simple visualizations, but we also plan to use it as input to data mining techniques.

4.2 Memory Trace Analysis

In both C++ and Java, the behavior of memory is important to understanding the performance and actions of the underlying code. We provide a simple analysis that offers the information about memory allocations during the run. The output here consists of one trace record per object. This record contains the routine that allocated the block, the type of the block, the size of the block, when in the execution the block was allocated and freed. For C++ it also indicates the start and end address of the block. The times and routines given here can be used to correlate the memory information with the call trace information in a visualization.

4.3 Path Analysis

Driven by the fact that the amount of data we collect is extremely large, we looked into ways of compressing them. Gzip gives us a compression factor of four to five, which is not enough. We use the Sequitur algorithm [5], which has also been used successfully for compressing traces at the basic block level [3]. Sequitur takes as input a string of symbols and outputs a context free grammar, whose start symbol, when fully expanded, gives us back the original string. This is combined with the conversion of the dynamic call tree into a call dag [2] and finding repeated sequences of calls.

Our algorithm maps each dynamic call into an identifier that represents that call and all the calls it makes. This is done bottom up, so that when we are analyzing a particular call, we already have identifiers for each of the calls that it makes. We use the Sequitur algorithm to build a compact representation of the sequence of calls in the form of a grammar. We convert this grammar representation into a string. Combining the string with the name of the calling function yields the identifier for the particular call. In doing so, we find common nonterminals between this grammar and grammars for other calls and note any recurring sequences of calls and encode them using run-length encoding. Here we have the option of saying “N or more” calls rather than using the exact number. The identifier that is constructed in this way is then matched to see if the call sequence had previously occurred and, if so, the two are combined, thus forming the call dag. We keep statistics (run time, real time, and allocation information) with each identifier so that we can provide the mean and standard deviation for each path.

This whole process gives us a compression factor on the trace described in Section 3.4 of about 800, two orders of magnitude better than gzip (but the Lempel-Ziv compression that gzip uses is lossless). More importantly, repetitions and patterns in the dynamic call tree are discovered and stored.

4.4 Summary Views

There are several ways of summarizing the information contained in the call trace. One of the simplest that we currently do is to gather performance information from the call trace and yield a corresponding output relation. This relation contains one tuple for each method or routine called during the execution. Each such tuple identifies the routine and then gives the number of times it was called, the time spent executing code in the routine, and the time spent executing code in the routine and any other routines it called.

Because the call traces can be extremely large, we are interested in developing ways of shrinking the trace data accordingly. One approach we have taken is to replace the call tree with a call dag. Here each unique call subtree only occurs in the output once. Processing such data is complex because much of the trace summary must be kept in memory while processing the trace data. However, our experience has shown that such views will typically cut the trace data to about 40% of its original size (much more with repetitive applications) with only a minor loss in the accuracy of the reported trace information.

To provide high level views of long-running programs, however, a different approach has to be taken. We have started such an approach using the notion of interval summaries. Here the overall execution is broken down into a small number (we are currently using 1024) of intervals and the trace file is used to generate summary information for each interval. The corresponding visualizations can then provide a high level view showing at a gross level what was occurring over the whole program. Where additional detail is needed, such a view can be linked to a detailed call trace (or other) view restricted to the interval(s) in question.

We currently provide two distinct interval summaries of Java traces. The first is a call interval trace. This not only divides the run time into 1024 discrete blocks, but also summarizes all the methods of a given class within the class. For each class and interval pair it outputs information for each thread. Here it provides the amount of time that the thread spent executing a method of that class during the interval, the average depth of the call stack for the thread during the interval for methods of this class, and the average time spent waiting by the thread for monitors related to objects of this class. The second interval summary concentrates on memory utilization. For each interval and class pair it outputs the average number of objects of that type that are allocated.

5 VISUALIZING THE RESULTS

We used the trace generation and analysis facilities described here along with our existing software visualization tools as a starting point for the full scale visualization of the dynamics of Java programs. Three of these visualizations are shown in Figure 5.

The visualization in the upper left is an interval visualization of a simple Swing-based java program. Here time (in terms of intervals) runs along the X axis while class (sorted alphabetically by name) runs along the Y access. The visualization fills each interval-class block using a box colored by the threads active for that time and class. The saturation of the color indicates the depth of the call stack at that point. Blocking threads are shown using black. The visualization that results clearly shows the initialization and execution phases of the program as well as where some of the other processing is being done.

The second and third pictures in Figure 5 show views of the call stack over time. The one in the upper right represents the stack using a spiral starting at the center and growing out. Here color is used to indicate the routine, the width of an entry its total run time, and the height above the floor of the spiral represents the stack depth. The program being traced here does a knight's tour and the increasing depth of the call stack results from the recursive search that it uses. The picture at the bottom of the figure shows a simpler, but often easier to read, zoom, and scroll, linear view of another program.

6 FUTURE WORK

The efforts reported here represent a solid first step toward our goal of producing a wide variety of different program analyses from a single comprehensive call trace with a minimum of effort on the part of the programmer. The architecture that we use is robust and efficient. It imposes a significant but not overwhelming overhead while collecting trace information. Programs being traced can be used interactively without the user being aware that the tracing is occurring. The various analysis both show that the architecture is general enough to support different analyses and that different analyses are interesting in their own right.

We are continuing this work in a number of different directions. The first is to add interactive control to the trace collection. Here we are developing a user control panel that will let the user interactively enable and disable various types of tracing as well as define mark events that can be used to correlate user interactions with the resultant visualizations.

A second direction involves doing some of the analysis as part of TMerge. While some of the analysis techniques need to have the whole trace available in order to run (e.g. performance summaries or interval analysis), others can be done on the fly. We are experimenting with such techniques in order to provide instantaneous visualizations that convey at a glance a variety of properties about systems as they are executing.

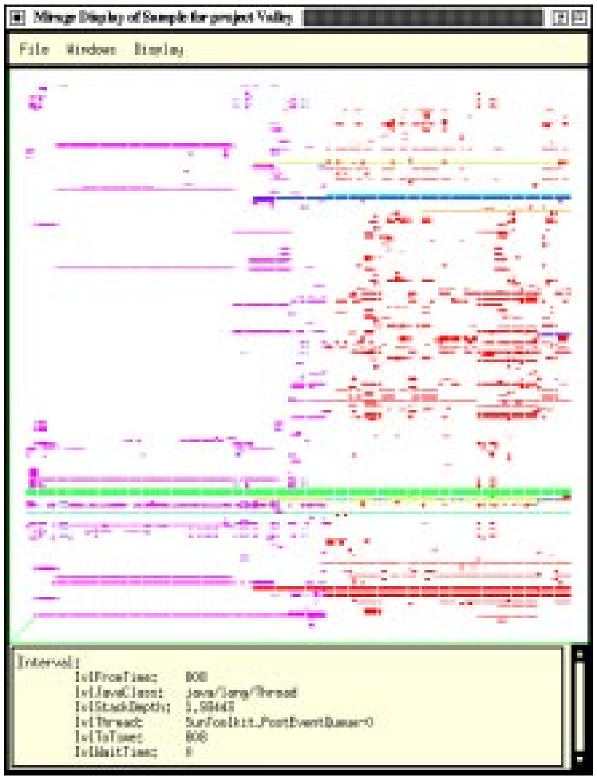
A third direction involves developing compact trace representations that provide a high degree of compaction without a minimal loss of information. Here we are looking at dynamic interval analysis, ways of excluding system and other irrelevant classes from the traces, and even doing dynamic call dag analysis. Another aspect we are looking at here is developing abstract models of the computation for the purpose of visualization. These can be used, for example, to model the behavior of message passing libraries to show such behavior in the context of the overall trace.

The fourth direction that our current research is exploring involves creating and using visualizations to provide better understandings of the inner workings of software. This work attempts both to develop new visualization techniques and to provide the necessary browsing and linkage mechanisms to let the user synergistically use multiple visualizations simultaneously.

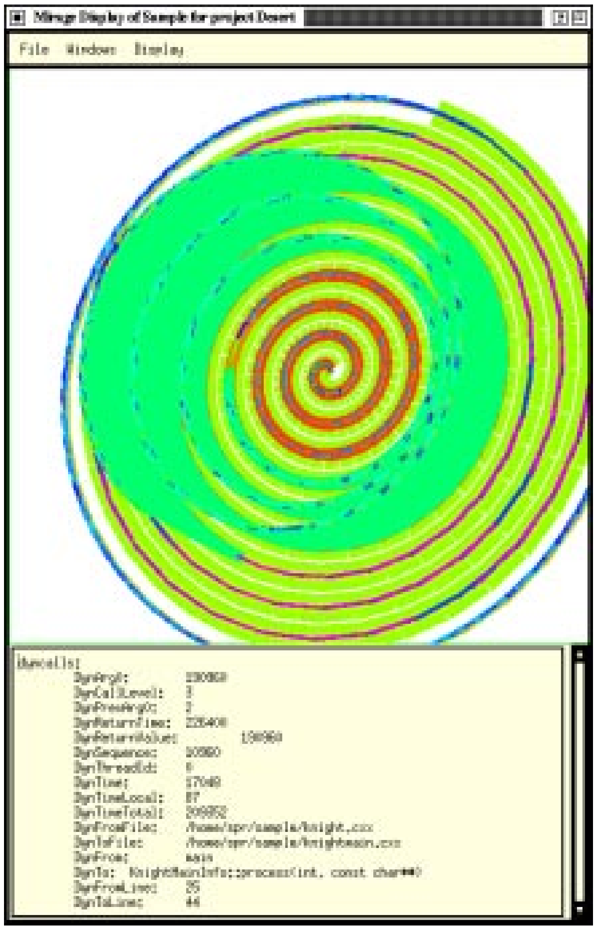
Overall, we feel that visualization, and particularly visualization of behavior, will become increasingly important in understanding the large, complex, multithreaded and multiple-process Java systems that are being developed. Our efforts provide one starting point along these lines.

7 REFERENCES

1. Jong-Deok Choi and Harini Srinivasan, "Deterministic replay of Java multithreaded applications," pp. 48-59 in *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, (1998).



```
Interval:
        IvlFromTime:    808
        IvlJavaClass:   java/lang/Thread
        IvlStackDepth:  1,90443
        IvlThread:      SunToolkit.PostEventQueue-0
        IvlToTime:      808
        IvlWaitTime:    0
```



```
dyncalls:
        DynArg0:           190960
        DynCallLevel:      3
        DynPrevArgC:       2
        DynReturnTime:     226400
        DynReturnValue:            190960
        DynSequence:       10880
        DynThreadId:       0
        DynTime:           17040
        DynTimeLocal:      87
        DynTimeTotal:      209352
        DynFromFile:       /home/spr/sample/knight.cxx
        DynToFile:         /home/spr/sample/knightmain.cxx
        DynFrom:           main
        DynTo:  KnightMainInfo::process(int, const char**)
        DynFromLine:       35
        DynToLine:         44
```

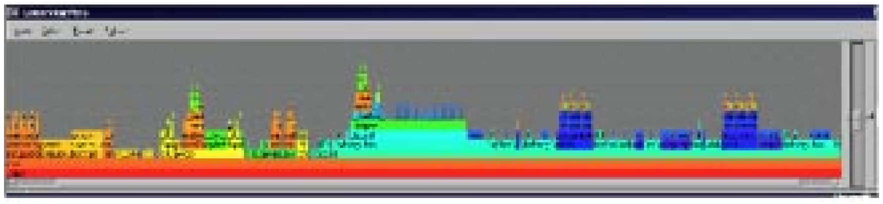


Figure 5. Visualizations done using trace data.

2. 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).

3. James R. Larus, "Whole Program Paths," *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pp. 259-269 (1999).

4. Moises Lejter, Scott Meyers, and Steven P. Reiss, "Support for maintaining object-oriented programs," *IEEE Trans. on Software Engineering* Vol. **18**(12) pp. 1045-1052 (December 1992).

5. Craig G. Nevill-Manning and Ian H. Witten, "Indentifying hierarchical structure in sequences: a linear-time algorithm," *Journal of Artificial Intelligence Research* Vol. **7** pp. 67-82 (September 1997).

6. 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).

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

8. Steven P. Reiss, "An engine for the 3D visualization of program information," *Journal of Visual Languages*, (December, 1995).

9. Steven P. Reiss, "Cacti: a front end for program visualization," *IEEE Symp. on Information Visualization*, pp. 46-50 (October 1997).

10. Steven P. Reiss, "Software visualization in the Desert environment," *Proc. PASTE '98*, pp. 59-66 (June 1998).

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

12. John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, *Software Visualization: Programming as a Multimedia Experience*, MIT Press (1998).