An Information Exploration Tool for Performance Analysis of Java Programs

Gary Sevitsky, Wim De Pauw, Ravi Konuru
IBM T. J. Watson Research Center
30 Saw Mill River Road
Hawthorne, NY 10532
+1 914 784 7619
{sevitsky, wim, rkonuru}@us.ibm.com

Abstract

The diagnosis of performance and memory problems can require the analysis of large and complex data sets describing a program's execution. An analysis tool must help the user both find the right organization of the data to uncover useful information, and work with the data through a lengthy and unpredicatable discovery process. In this paper we present Jinsight EX, a tool for analyzing Java performance, that adopts techniques that have been successfully used to explore large data sets in other application domains, and adapts them specifically to the needs of program execution analysis. We introduce execution slices, a high-level organizing abstraction that the user may define and then easily reuse in various settings. We illustrate techniques that allow the user to perform a range of common analysis tasks and to structure a longer analysis process, using this abstraction. We present the tool, its implementation and initial experience of its use.

1. Introduction

The diagnosis of performance and memory problems in Java programs can require the analysis of large and complex data sets that describe the dynamic behavior of a program. Tools are often ill-suited to both the complexity of this information and the ways in which programmers need to work with this information. Given the richly connected nature of dynamic program information, meaningful information can easily remain hidden if the data is organized along the wrong lines. Choosing an appropriate organization at each stage of analysis is a difficult and crucial component of the analysis process. The user may have valuable knowledge that could help structure the analysis, based on the specific question under consideration, an understanding of the informal structure of the code, or general knowledge about analyzing certain classes of programs.

Various classes of tools exist that allow users to explore large, complex sets of data in other application domains. Tools such as on-line analytic processing (OLAP) systems [22], geographic information (GIS) systems [1], and information visualization systems (for example, Diamond [16]) have been successfully used to analyze marketing, environmental, and other types of data. These systems allow the user flexibility in organizing and presenting information, and they typically provide a highly interactive style of work to support an incremental discovery process. The user can filter or interactively drill down to focus on an

area of interest, group or classify the information along various lines, and summarize and compare attributes of the data along a number of dimensions using a variety of presentation techniques. We believe this general approach can be helpful, and we specialize ideas and techniques from these fields into a domain-specific tool for analyzing program behavior.

In this paper we present Jinsight EX, a tool for the analysis of Java program performance, which we define broadly to include diagnosis of time, memory, and other resource usage problems. The tool gives the user the flexibility to organize the execution information along lines that are useful for the analysis task at hand. We propose *execution slices*[1] for this purpose, which the user may define based on both static and dynamic criteria. Once defined, an execution slice may be used as a first-class object to consistently represent an aspect of interest throughout the analysis in various views. A number of execution slices taken together may in addition be treated as a user-defined dimension, independent of the predefined structure of the database, enabling the computation of multidimensional performance measures. Execution slices, while based in part on queries, may be specified using a range of domain-specific techniques that hide much of the complexity typically found in pure query-based approaches. Moreover, to facilitate the larger analysis process the tool allows slices to be organized into *workspaces*. A workspace encapsulates the experimental context at a particular stage, allowing incremental discovery and experimentation with alternative hypotheses.

We have built our tool by extending Jinsight [4], a visual tool for analyzing Java program behavior. Jinsight has been successfully used to diagnose performance and memory problems in many industrial applications. The target program is run under a specially instrumented Java virtual machine that generates traces of method calls and returns, lifecycle events of objects, classes, and threads, and user-initiated snapshots of the object population and object references. The user analyzes this information after it has been collected, using the Jinsight visualizer. The visualizer allows the user to explore the information through various graphical views, each designed to bring out a different aspect of the execution.

The rest of this paper focuses on describing and illustrating the various features of Jinsight EX via a running example of using the tool to study a real-world Java application. In Section 2 we introduce the concept of execution slice and give a high-level tour of the ways it can be used to analyze programs. In Section 3 we describe ways to structure the working environment using execution slices, to support the larger analysis process. In Section 4 we give a more precise definition of exection slices, and present the various slice specification techniques. In Section 5 we outline how we exploit the slice abstraction and assumptions about common user tasks to implement the features efficiently. Related work is addressed in Section 6. Preliminary experiences using the tool, conclusions, and future work are discussed in Section 7.

2. Analysis using execution slices

2.1 Slices as a user-defined unit

A central problem when analyzing program behavior is that information of interest is often intertwined with unrelated information. The static units of a program, such as classes, methods, and packages, are not always the most suitable organization for analyzing the more complex and richly connected data describing a program's dynamic behavior. The problem may manifest itself as visual clutter in detailed graphical views, or as numerical summaries skewed

---

[1] Execution slices are not to be confused with program slices [23]. Throughout the paper we use the term *slice* to mean execution slice.

by the inclusion of irrelevant information. In either case, key information needed for analysis can remain hidden without the right organization.

The user may be aware of more useful ways to organize the information, based on a hypothesis about the source of a problem, knowledge of the informal structure of the code, or discoveries made at an earlier stage of analysis. We introduce the concept of execution slice as a means for the user to carve out a subset of the execution information that corresponds to an aspect of interest, and treat it as a higher-level unit for analysis.

We begin with a transaction processing application as an example. This three-tier application consists of a client, a Java server and a relational database (DB2 in this example). The server accesses the database via the Java Database Connectivity (JDBC) interface. We would like to optimize the use of the database by the server. Based on experience analyzing other Java database applications, we would like to group the database activity into three broad categories:

1. administrative overhead, such as establishing database connections and compiling queries
2. query execution
3. results processing

Breaking the problem down along these lines can help us decide whether to put effort into reducing overhead (e.g. by reusing connections or precompiling queries), tuning the database, or optimizing the Java code that is processing the results of queries.

The class structure of JDBC does not line up exactly with these categories, as we show in Figure 1. For example, we would like to group the factory method *DriverManager.getConnection()* with some methods from the class *Connection* in the overhead category, but move the *Connection.commit()* and *Connection.rollback()* methods to the query execution category.

DriverManager	Connection	PreparedStatement	ResultSet
getConnection	createStatement	<init>	close
	prepareStatement	execute	next
	close	executeQuery	getString
	rollback	executeUpdate	getFloat
	commit	setBoolean	get ...
		set ...	<init>

overhead *execute queries* *process results*

Figure 1. JDBC class structure vs. areas of interest

We can define three execution slices that correspond to our desired breakdown of activity. In Figure 2 we show how the user can create a slice by selecting methods from a table view. The user can assign a name and a color to each slice, and then use the slices as first-class objects throughout the analysis. Later on in the paper we describe techniques for defining more complex slices, based on dynamic as well as static information.

Figure 2. Creating the "overhead" slice by selecting methods

To begin our analysis we would like to see an overview of the program using this categorization. We can compare the total execution time and number of calls made in each slice, as in Figure 3. In this view, the first slice, labeled "Workspace", is the base slice representing the entire program execution.

slice	cumulative time	number of calls
Workspace	383753605	565454
Overhead	7084444	3780
Execute queries	76839158	81521
Process results	9743123	46054

Figure 3. Execution time and number of calls per slice

2.2 Slices as an analysis dimension

Once we have identified each type of activity, we would now like to see when it is occurring. The Execution View [4], depicted at two zoom levels in Figures 4 and 5, shows the sequence of method calls in each thread. The vertical axis represents the passage of time, with the start of the program at the top. Each vertical lane represents a thread, and each vertical bar represents a method call in progress. The stack depth increases to the right within each thread. Zoomed all the way out as in Figure 4, we can see the overall "shape" of an execution in time. In this example we can see that there are five threads where most of the activity is occurring; these are the threads responsible for transaction processing. In Figure 5, we have zoomed in to study the first portion of the run in two of the transaction processing threads. In this figure, the system has colored each bar according to the slice that it falls into. We can see that overhead activity is occurring only at the beginning, and so we may rule out this category as a problem during actual transaction processing. This view can also show more detail at higher zoom levels [4].

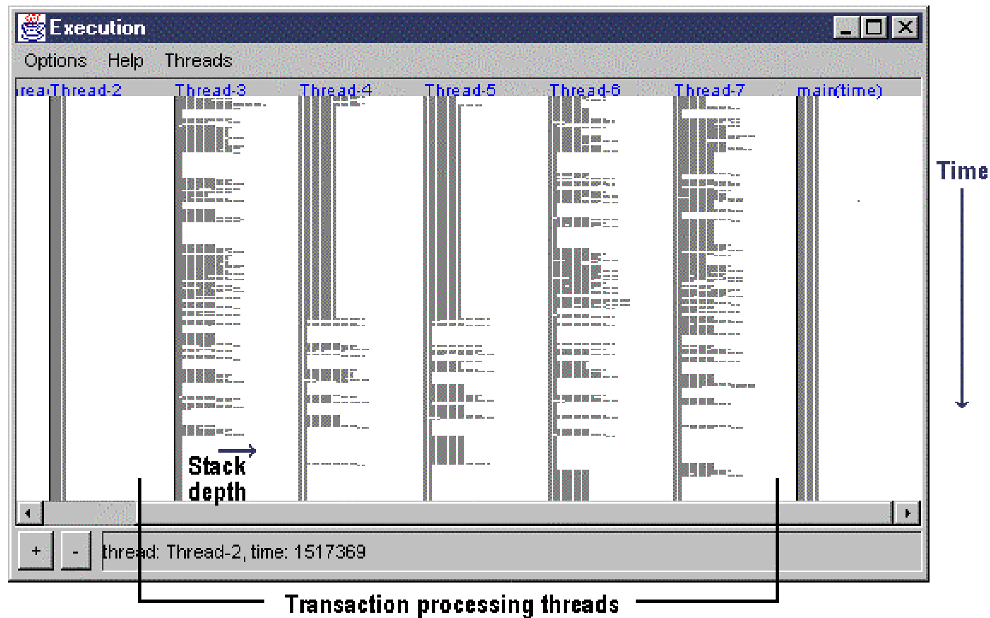

Figure 4. Execution view showing the entire run

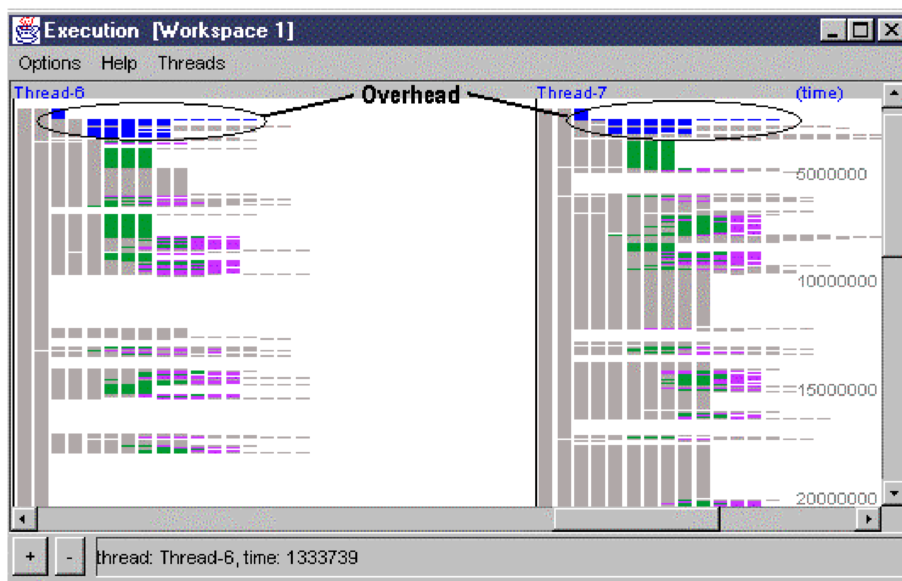

Figure 5: Detail of two transaction processing threads, with activity colored by slice

This example illustrates an important feature of execution slices: slicing criteria are automatically recast to cover whatever type of execution data we are viewing. In this example, we originally expressed our slicing criteria in terms of methods; in Figure 5, the criteria were automatically extended to apply to all activity generated by these methods (i.e. all calls to these methods, plus all of their callees). This feature allows us to study a program from many angles using a consistent organizational scheme.

Now that we have ruled out the overhead category, our next step is to measure the remaining types of activity, to determine where to best focus our optimization effort. It might

be worthwhile to look at activity for each thread, since only some of the threads are used for transaction processing. In Figure 6 we show the time per thread, broken down by slice. In this figure, each thread appears in a row, and each column labeled "cumulative time" represents the activity in a slice. The first of these columns represents the entire run, and the next two columns represent the remaining two categories of database activity we have defined. We have chosen to show the comparison as a percent; each cell displays the time spent in a slice as a percent of the total time for the thread. Looking at the rightmost column we can see that the five transaction processing threads spend only a small percentage of their time processing results. This preliminary result suggests that we focus our analysis in the database.

Threads [Workspace 1]

File Rows Columns Selected Help

name	cumulative time Workspace	cumulative time Execute queries	cumulative time Process results
main	49390099	0.0%	0.0%
Finalizer_thread	46305259	0.0%	0.0%
Cmd_Thread	0	0.0%	0.0%
Thread-2	49265729	0.0%	0.0%
Thread-3	47791666	15.3%	5.6%
Thread-4	47750355	54.6%	1.5%
Thread-5	47750250	55.4%	1.6%
Thread-6	47750164	18.3%	6.1%
Thread-7	47750083	17.3%	5.7%

Figure 6. Percentage of time per thread in each slice

In general we may treat a set of execution slices as a user-defined dimension, orthogonal to the predefined database structure of classes, threads, etc. This allows us to measure resource usage in finer detail against user-defined aspects of a program run. In our application example, each transaction processing thread may process more than one type of transaction. It may be worth looking at the performance of each transaction type individually, since each may use the database differently. In Figure 7 we show the time in each of the transaction processing methods, broken down by type of database activity. The difference in performance characteristics among the transaction types gives us a clue as to which specific queries or results-processing code to focus on.

Methods [Workspace 1]

File Rows Columns Selected Help

class name	method name	cumulative time Workspace	cumulative time Execute queries	cumulative time Process results
StockLevelTransaction	process	4799839	98.8%	1.1%
DeliveryTransaction	process	887937	58.8%	10.3%
OrderStatusTransaction	process	550388	56.0%	24.3%
PaymentTransaction	process	53079678	94.9%	3.2%
NewOrderTransaction	process	29879221	70.0%	26.0%

Figure 7. Percentage of time per transaction type in each slice

We could continue this process at a finer level of detail if we suspect that there is variation among the individual invocations of one of these transacation processing methods. We could select the method and drill down to see a similar view (not shown) showing the time of each individual invocation of that method broken down by slice.

2.3 Filtering with slices

We often want to focus on one particular area of interest, and exclude unrelated information. In our example, we are concerned with optimizing the processing of transactions, so it would help to exclude all other activity, such as idle time waiting for client requests, so that we could perform an accurate analysis. The base for our study so far has been the entire run. Our earlier

computations in Figure 6, of time per thread, are actually misleading, since the rightmost columns are measured against a base that includes activity unrelated to transaction processing. In our tool this base is itself an execution slice, and we may define criteria for it that will serve as a filter for the entire study.

Figures 8 and 9 show the same two views as Figures 5 and 6, after we have limited the base slice to cover only transaction processing activity. In Figure 8, the areas completely outside of our study are now barely visible. Figure 9 is identical to Figure 6, but the numbers in each column now reflect this narrowed focus. We can see in the rightmost column in Figure 9 that the time spent processing results, relative only to the transaction processing time, is higher for each of the threads than we had originally thought from Figure 6. If our goal is to optimize all transaction processing activity, then these more accurate numbers tell us that results processing may warrant some optimization effort.

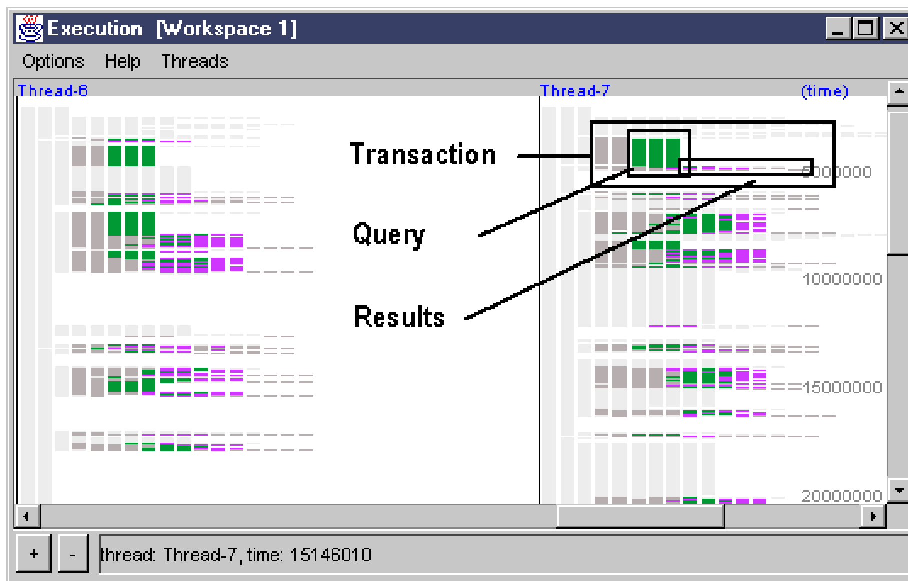

Figure 8. Query vs. results processing only during transaction processing

Threads [Workspace 1]

File Rows Columns Selected Help

name	cumulative time Processing tra...	cumulative time Execute queries	cumulative time Process results
main	0	0.0%	0.0%
Finalizer_thread	0	0.0%	0.0%
Cmd_Thread	0	0.0%	0.0%
Thread-2	0	0.0%	0.0%
Thread-3	10670072	68.4%	24.9%
Thread-4	27025613	96.5%	2.7%
Thread-5	27392489	96.7%	2.8%
Thread-6	12380457	70.4%	23.4%
Thread-7	11728432	70.6%	23.1%

Figure 9. Transaction processing time per thread, broken down into query vs. results processing

3. Supporting the analysis process

Program analysis can be an iterative, experimental process. We introduce *workspaces* to explicitly address the needs of this larger process. A workspace allows the user to set up a filtering and categorization framework for a given stage of analysis, as we have done in the previous example. This framework may then be used to explore many aspects of the program execution, without having to restate the framework in each view. A workspace is composed of a base slice establishing the scope of the study, a set of slices which are subsets of this base, and then any number of views which may make use of these slices.

The system allows multiple workspaces to be active concurrently, supporting a number of common analysis scenarios. One such scenario is progressive refinement, where the user is iteratively narrowing a problem down to a smaller and smaller area that appears to be the likely cause. This can be an experimental process in which the user may need to backtrack. Workspaces may be used to save the state at each stage of refinement.

An important special case of progressive refinement is where the user has grouped the information according to alternative hypotheses about where a problem might lie. In our example so far, we have been trying to determine if the problem is in one of the three categories of database activity we have defined. To study one of these areas in more depth, we can open a new workspace whose base is one of these slices. In Figure 10 we show an Execution view as it would appear in a workspace whose base is the results-processing slice. Everything other than the results- processing activity is almost invisible in this view. We can now continue to break this activity down along some other lines by defining slices relative to this new base.

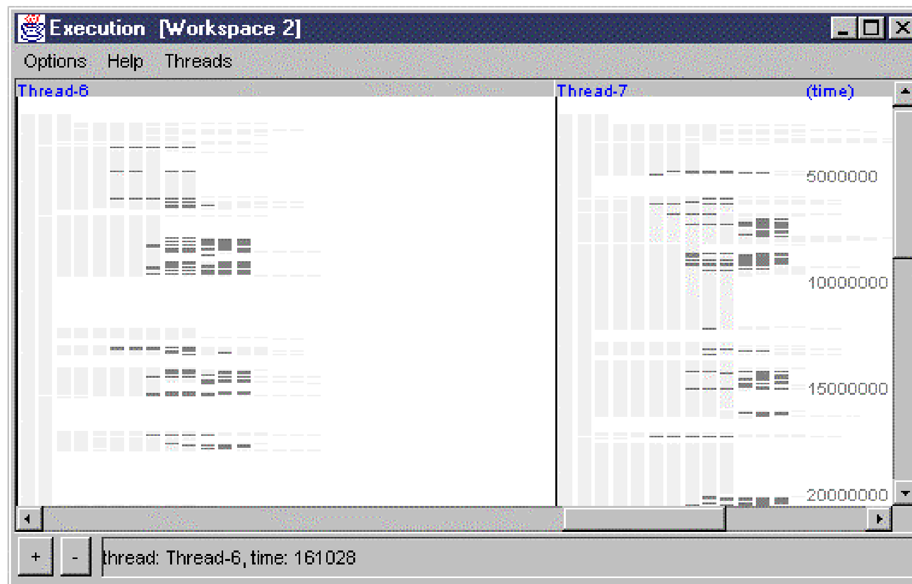

Figure 10. View showing new workspace base in dark gray

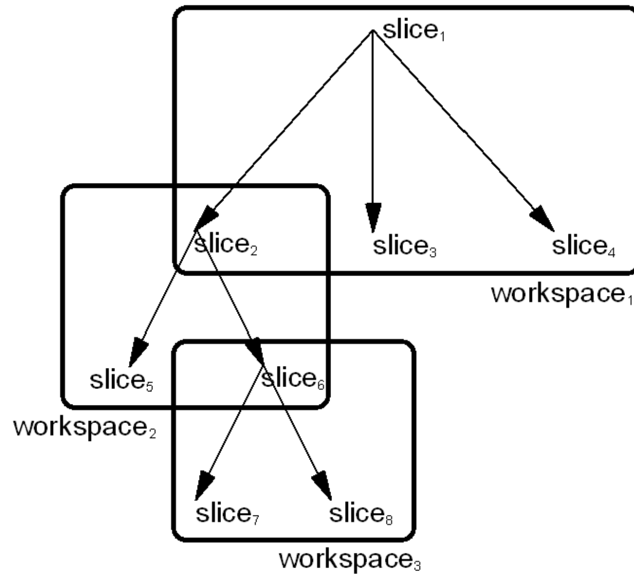

Figure 11. Slice hierarchy and workspaces

In another scenario, the user may not know in advance what the right organization of information is that will illuminate the cause of a problem. He or she may want to explore multiple hypotheses simultaneously, and compare results of the same information analyzed along different lines, using views from two different workspaces.

Underpinning the workspaces is a hierarchy of slices, each a subset of its parent, as shown in Figure 11. The system maintains a live connection from each slice to its parent, so that if the user decides to go back and adjust the filtering from an earlier stage, this will automatically be reflected in any dependent workspaces.

4. Execution slices

Our overall goal is to provide the user a large degree of flexibility in organizing the information for analysis, while at the same time giving the user access to this power in ways that are easy to use for common tasks. Our approach is to provide a variety of ways to specify slices, with defaults for common cases, so that complex features are progressively disclosed only as needed. In this way the user does not have to become an expert on the intricacies of our data model or a general purpose query language in order to accomplish most tasks. In this section we briefly give a more precise definition of execution slices and then give an overview of the specification techniques. For a more detailed discussion of these topics see [18].

4.1 Execution slices

A set of execution slices is a classification of the entire database of trace information, in the sense that each element of information in the database (describing each thread, class, method invocation, etc.) may be classified as being a member of one or more slices. A single slice spans all types of information in the database, and may be used as a lens through which to view the entire database. This mediated view of the database retains the database structure of classes,

threads, etc. and their relationships, but contains only those individual elements that are members of the given slice. Values of certain attributes, such as summary measures, are evaluated relative to the given slice by taking into account the slice filtering on the entire database.

The membership criteria for each slice is determined by a combination of filtering queries and *recasting rules* (described below).

A given slice has a filtering query for each type of information, such as threads, classes, objects, etc. The query language is designed for program analysis, and has both features and restrictions tailored to this application domain. Filtering criteria may be based on both static and dynamic attributes of the execution information. The query language supports subqueries and transitive closure operations, useful for working with calling and object reference relationships.

4.2 Rules for recasting

Studying a single aspect of a program can require working with many different types of information in the database. We simplify this process by requiring the user to specify the criteria just once when defining an execution slice, and the system will automatically recast these criteria to cover related information. This is achieved by using recasting rules, which determine how each filtering query will be extended to include or exclude related elements. A set of predefined recasting rules is designed to address what we believe are the most common cases. When a slice is defined some of these rules are enabled by default. The user may enable or disable individual rules to fine tune the recasting behavior for that slice. As we gain more usage experience we will continue to refine these rules.

We give an example where we would like to analyze activity related to *Vectors*, since we have discovered that there are more *Vectors* than we had expected in our program. We can create a slice by selecting the class *Vector* from a table of classes, as in Figure 12, and view the activity using the Execution View, as in Figure 13.

Figure 12. Creating a slice for *Vector* activity

The following default rules were automatically applied to this slice:

1. Filter methods based on class
2. Filter calls based on method
3. Include callees of any calls

This slice represents all activity generated by calls to *Vector* methods. If instead we would like to study why *Vector* methods are called, we could disable rule 3, and enable rule 4:

4. Include callers of any calls

Figure 14 shows the result of this change. Note that summary computations of information about this slice will now reflect this change as well. For example, measuring the number of calls will now yield the number of calls leading to, rather than from, methods of *Vector*.

While these rules are simple, if the user had been required to state them directly using a general purpose query language, it would have required an understanding of the structure of the data model and the use of subqueries to recast the criteria.

In the following sections we describe three techniques for the user to specify slice criteria. The recasting rules are in effect no matter which of these techniques is used to define a slice.

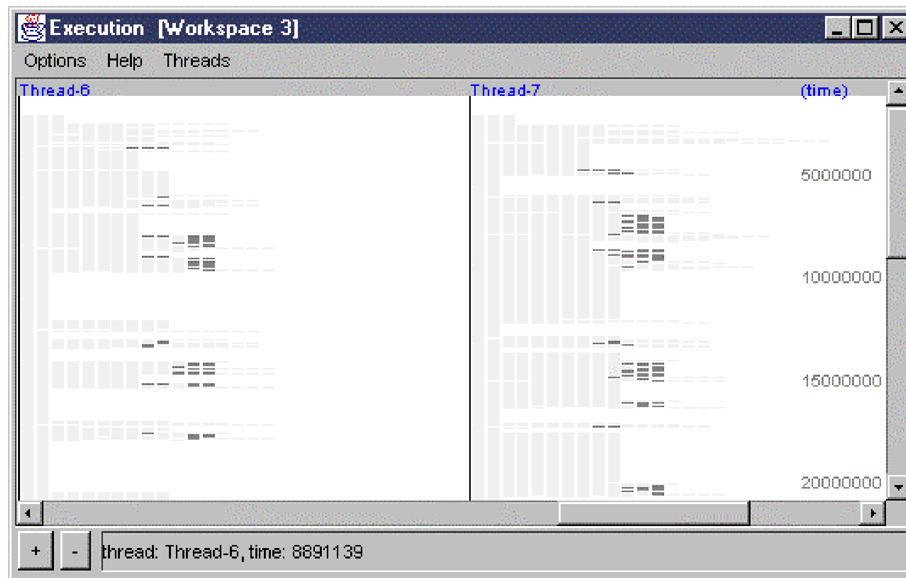

Figure 13. Activity caused by *Vector* methods

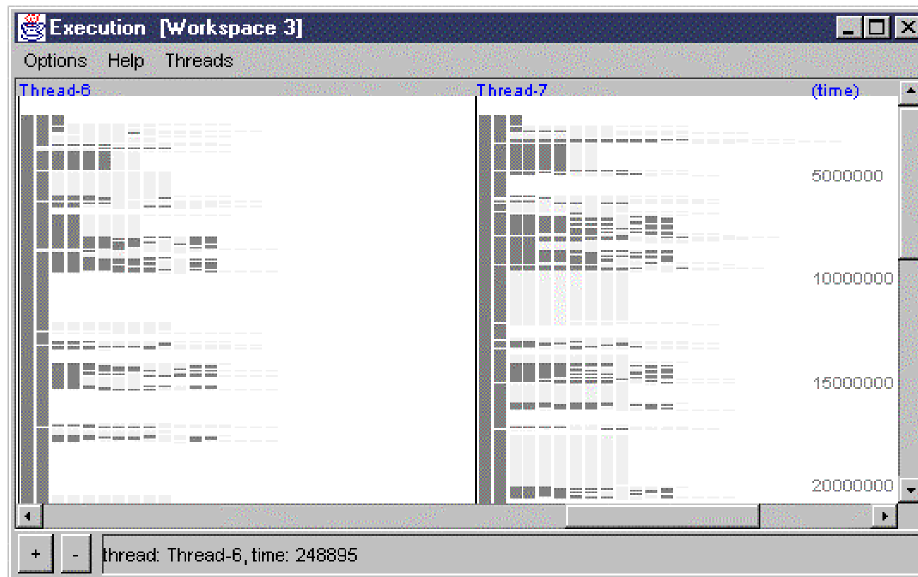

Figure 14. Activity leading to *Vector* methods being called

4.3 Specifying slices based on elements

As we have shown in the examples so far, an easy way to define a slice is to interactively select elements of interest from any of the views. Static structural elements such as classes and methods, as well as dynamic elements such as individual instances or method invocations may be chosen as the basis for a slice.

Elements may also be selected for exclusion from a slice. This is useful for filtering out cases known to be outliers. For example, if we are studying a method that is called many times, we may want to exclude the first call if it is much more expensive due to initialization costs.

4.4 Specifying slices based on attribute values

Values of attributes of the execution information, both static and dynamic, can be useful dimensions for categorizing information. For example, to analyze a method with many repeated invocations, we can categorize each invocation based on its CPU time as being slow, medium, or fast. If we create a slice for each of these categories we can then compare the slow and fast cases from various angles to understand what is causing the slow cases. Similarly, to diagnose a complex memory leak in a long-running application, we can classify the objects into a number of distinct periods by creation time. We could then measure the memory used per class during each time period.

To define a set of slices based on an attribute, the user selects the attribute, and the system will group the values of the attribute into buckets. The user is presented with a number of options to control how the buckets are determined. A slice is then created for each bucket.

4.5 Specifying slices using the query editor

To specify more complex criteria we provide an interactive query editor where the user may access the query language directly. The user may create new slices from scratch using this editor, or use it to fine tune the filtering criteria of slices created in other ways. The query editor guides the user somewhat, progressively disclosing only those features that make sense at a given point in a query.

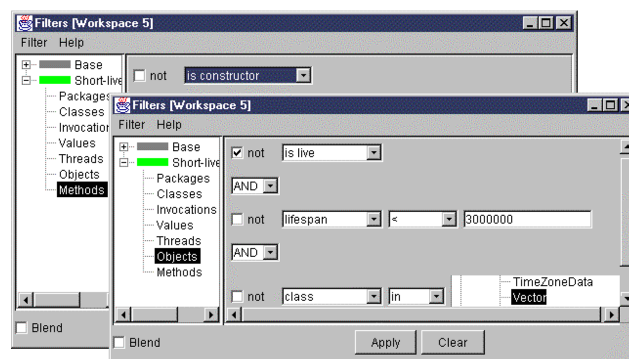

Figure 15. Query editor: construction of short-lived *Vectors*

In this example, we may have determined that our program is creating a lot of *Vectors*, and suspect that many of them are used only briefly and then garbage collected. To see if it is worth creating a pool of reusable *Vectors*, we would like to know if constructing these objects is a

significant expense. In Figure 15 we show how a slice could be specified to study the construction activity of short-lived *Vector*s. We could then proceed to measure this slice in various ways as we have shown in other examples.

Note that the query editor does requires more knowledge of the data model and some database concepts than the other specification techniques. However, since the recasting rules still apply, the specification is still simpler than it would have been with a more general purpose database.

5. Implementation

The system is built completely in Java. At its core is the trace information, which we store as a custom in-memory database of Java objects and references. In this section we give a brief overview of the implementation of this database as it relates to supporting the features we have described.

We represent slices using slice membership information distributed throughout the database of program elements. Each program element has a bitmap describing which slices it belongs to. This element-centric rather than slice-centric representation matches the usage of the system, where we are typically asking about a specific element as seen through the lens of various slices. This enables fast slice membership computations, needed for computing summary values, coloring views based on slice membership, and supporting the hierarchy of slice filtering in workspaces. This approach is space-efficient as well, since we expect the number of slices in a given session to be relatively small, on the order of tens, or perhaps hundreds in the worst case.

We compute all slice membership and most summary computations relative to a particular slice at the time that slice is defined, or when it is invalidated due to a change in its filtering criteria or that of its parent slice. Precomputing many of the summary computations is desirable since some of these computations require traversing all of the calling hierarchies in their entirety. This also matches our expectation of usage, where users specify filtering and grouping criteria first and then interactively explore the data from many angles. We save space and avoid unnecessary computations by postponing some detailed summary computations, such as summaries by individual invocations, until these values are actually needed in a specific view. Computations such as these which are not saved with the data are marked so they will be cached at a higher level by the views.

We compute slice membership first, followed by summary computations in a second pass. We take advantage of the recasting rules to optimize the query evaluation for common cases. For example, if the user has specified inclusion of callers or callees using these rules, we are able to incorporate this into a single pass traversal of the calling hierarchy.

Some measures such as cumulative time (defined as time in a method call plus everything it calls) require care when aggregating over larger units, to make sure we are not counting the same time more than once. For example, to aggregate cumulative time in all calls to all methods of a class, we must make sure we adjust for time spent in different methods of the same class that directly or indirectly call each other. A further complication is that these computations are done relative to the filtering criteria of a specific slice. While it is functionally possible to do these computations with a general purpose relational database, we are able to highly optimize them in our implementation which is specialized for this purpose. We are able to compute cumulative time relative to a slice, aggregated by every object, method, class, package and thread, together in $O(n)$ time, where n is the total number of method calls in the program [*18*].

The performance of this implementation approach appears promising, based on a number of applications we have analyzed so far. The examples shown in the paper were based on a single trace of 1.3 million events. When analyzing the trace on a 266 MHz Pentium II machine, most slice-definition operations shown took on the order of a few seconds, with no single operation taking more than a minute. Each of the interactive operations on the views (painting, sorting, drill down) took under two seconds.

6. Related work

In the area of visualizing program behavior, the need to support the exploration of large amounts of information has been addressed with a number of approaches. Stasko [20] has done extensive research on algorithm animation and software visualization, and addresses the problem of handling large data sets using "semantic zooming" and the Information Mural [9]. The main focus here is on how to visualize large data sets as a whole, rather than on extracting semantic slices as we propose.

Various organizing abstractions have been used for filtering or for grouping execution information into larger units. Many systems for performance analysis, such as work by De Pauw et al. in [7], use the object-oriented paradigm. Sefika et al. [17] use larger architectural units, and Walker et al. [24] introduce additional structural units as organizing principles. Some of these systems allow the user to define their own groupings to some extent. Dynamic relationships are also commonly used for organizing information, for example into call trees as in OptimizeIt [13] or as views showing patterns in Jinsight [5,6]. All of these techniques provide useful abstractions, though none with the degree of flexibility that a full query-based approach provides to precisely define an aspect of interest.

A number of systems use queries of execution information to let the user filter out extraneous information and to group together an area of interest. The Desert system [14] provides a powerful query capability against static and dynamic information for various types of program understanding applications, as does the Hy+/GraphLog system [3] for analyzing distributed and parallel programs. Snodgrass [19] allows queries to be used for analyzing operating system behavior, and Lencenvicius [12] uses queries of dynamic program information for the debugging of live programs.

Our approach differs significantly from pure query-based systems in a number of ways. In query-based approaches, each query is generally used to filter or group together a set of elements of a single type. Execution slices, however, introduce an independent analysis dimension, with each slice mediating access to the *entire* database. This allows a given slice to be automatically reused in different views for different purposes (such as filtering, visual classification, and computation of measurements), and across different types of information, without requiring the user to restate criteria. Because slices maintain the structure but alter the apparent content of the database, they allow the entire database to be studied as a unit under different filtering and grouping scenarios, using workspaces.

In general, pure query-based approaches provide a great deal of flexibility but leave the user exposed to many details in order to apply this power in visualizations and summary computations. By introducing a higher-level, domain-specific analysis unit, we are able to introduce rules, task-specific specification techniques, and summary computation algorithms that hide complex query language details from the user.

Ball and Eick [2] present a visualization technique using program slices [23]. This work is similar in that it addresses how to visually represent those parts of a program that are relevant to a given task. The goals and techniques of this work are substantially different from ours, however, since it is providing techniques for exploring source code rather than dynamic

execution information. In addition, for performance analysis, static program slicing techniques may not be adequate by themselves to address the complexity of the dynamic behavior of object-oriented programs.

Program Explorer [11] implements Jacobson's interaction diagrams [8] to visualize object interactions. This system employs simple filtering techniques to help manage a large number of objects. Lange's technique starts from a few classes; our system starts from a larger context, from which we can carve out interesting slices.

A number of other systems have combined static and dynamic analysis techniques to enable program understanding. In ISVis [10] the user can define various units to highlight events and interactions in the execution of the program. The Information Mural is used to depict the results. The system is primarily designed for understanding program architecture. Richner and Ducasse [15] present a query-based approach, and allow the user to create high-level abstractions. The main purpose of this tool is to recover design information. Systa's work [21] also addresses reverse engineering, using a methodology that combines static and dynamic approaches. These systems differ from our work in their focus on extracting design information rather than performance information. Our system allows a selection of elements based on dynamic attributes important for performance analysis, such as object lifespan or memory usage. Moreover, our visualizations can also render a program's execution in terms of resource usage.

Some of the presentation and analysis techniques we use are similar to techniques found in more general purpose systems to explore large databases. Filtering and drilling down to narrow a problem space are common features of OLAP [22] systems. Our table views are related to OLAP multidimensional reports, allowing the user to study summary values at multiple levels of aggregation. We provide the ability to pivot along a user-defined dimension of the trace information; typically OLAP systems do not provide this degree of flexibility in letting the user define new dimensions. OLAP systems are also not capable of performing the type of computations we need over hierarchical data such as call trees. We have adapted from Diamond [16] the idea of using bright, contrasting colors to highlight user-defined classifications across various views, and as an aid in seeing relationships in the data. The techniques we use for classifying information by attribute value are similar to those in the ArcView geographic information system [1]. We have applied these techniques within a single integrated framework designed specifically for performance analysis.

7. Conclusions and future work

We have presented a system for analyzing the performance of Java programs, with the following novel characteristics:

- We have introduced execution slices as a user-defined unit for focusing and grouping execution information based on knowledge of both the program and the particular problem at hand. Execution slices provide the following benefits for performance analysis:
  - They provide the flexibility of a query-based approach, while hiding much of the complexity from the user, by the use of recasting rules and a range of specification techniques.
  - They give the user a single handle for an aspect of a program execution that can be easily reused to analyze that aspect from many angles.
  - They may be used as the basis of multidimensional summary computations, enabling precise resource measurements to be studied at multiple levels of aggregation.
  - They enable optimizations of query processing and computations.

- We have provided explicit support for the iterative and often experimental analysis process, introducing workspaces, based on a hierarchy of execution slices, as a means for the system to maintain an experimental context for the user.
- We have demonstrated a highly interactive and versatile environment, learning from techniques found in information exploration systems in other application domains. With our system the user has great freedom to isolate, summarize, compare, and classify information about a program's execution at many levels of granularity, using a variety of techniques.

We have successfully used the system for tuning performance and diagnosing memory leaks on a number of real world applications, including some large web-based applications at customer sites. From our usage so far the results are promising, both in the usefulness of the system and in the technical feasibility of its internal design.

Our next step is to continue gathering experience by applying the tool to more industrial applications. During this process we will be identifying the nature and flow of analysis tasks, in order to further tune the recasting rules and the user interface in general. We believe identifying these tasks is also a necessary precursor to further optimizing the performance of the query evaluation mechanism.

One direction for further research is to incorporate information from additional sources to define execution slices. Experience analyzing programs that use common Java libraries and frameworks may allow us to build some of these specifications manually, and then provide them to the user as a gallery of standard slices or workspaces for analyzing Java programs. Some of the approaches in the static reverse engineering tools cited in the related work section may be worth integrating as well. High-level design information from software design tools may be another useful source of slice definitions.

Another important direction of research is to address the issue of large trace sizes, particularly when analyzing long-running applications. We may want to do more filtering and preprocessing of the trace information before bringing it over to the visualizer. Execution slices may be a useful unit in this context, for communicating filtering criteria and aggregate measurements between the trace collector and the visualizer.

We believe the conceptual framework and general approach are applicable to a number of other areas. Regression testing, debugging, and program characterization are all areas which we believe could benefit from the application of these techniques in some form.

8. Acknowledgements

We would like to thank Olivier Gruber, Erik Jensen, Robert Johnson, Nick Mitchell and Mark Wegman for encouragement, support, and many helpful discussions, and Harold Ossher, Sriram Padmanabhan, Harini Srinivasan, and John Vlissides for valuable critique on this paper. We would also like to thank the users of Jinsight for their suggestions.

9. References

[1] ArcView Documentation from ESRI Web Site at <http://www.esri.com/library/literature.html>.

[2] Ball, T., Eick, S. G. Visualizing Program Slices. Proceedings of the IEEE Symposium on Visual Languages. 1994.

[3] Consens, M. P., Hasan, M. Z., Mendelzon, A. O. "Visualizing and Querying Distributed Event Traces with Hy+", Lecture Notes in Computer Science, Vol. 819, Springer Verlag, 1994, 123-141.

[4] De Pauw, W., Jensen, E., Konuru, R., Gruber, O., Sevitsky, G., Vlissides, J. Jinsight, A Visual Tool for Optimizing and Understanding Java Programs. IBM Corporation,Research Division. Information and tool at Web Site <http://www.alphaWorks.ibm.com/ tech/jinsight> (1998).

[5] De Pauw, W., Lorenz, D., Vlissides, J., and Wegman, M. Execution Patterns in Object-Oriented Visualization. In Proceedings of the Fourth Conference on Object-oriented Technologies and Systems (COOTS), Santa Fe, New Mexico (1998), 219-234.

[6] De Pauw, W., Sevitsky, G. Visualizing Reference Patterns for Solving Memory Leaks in Java, ECOOP '99, June 1999, Lisbon, Portugal, in Lecture Notes in Computer Science Vol. 1628, Springer Verlag, 116-134.

[7] De Pauw, W., Kimelman, D., Vlissides, J. Modeling Object-Oriented Program Execution, ECOOP '94, July 1994, Bologna, Italy, in Lecture Notes in Computer Science Vol. 821, Springer Verlag, 163-182.

[8] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison Wesley, Reading, Massachussets, 1992.

[9] Jerding, D., Stasko, J., The Information Mural: A Technique for Displaying and Navigating Large Information Spaces, IEEE Transactions on Visualization and Computer Graphics, Vol. 4, No. 3, July-Sept. 1998, 257-271.

[10] Jerding D., Rugaber S., Using Visualization for Architectural Localization and Extraction, in Proc. of WCRE '97, pp. 56-65.

[11] Lange, D.B., Nakamura, Y. Interactive visualization of design patterns can help in framework understanding. In Proceedings of the 10th Annual Conference on Object-oriented Programming Systems, Languages, and Applications, pp. 342-357, Austin, Texas, USA, Oct. 1995. OOPSLA '95, ACM SIGPLAN Notices 30(10) Oct. 1995.

[12] Lencenvicius R., Hoelzle, U., Singh, A. K. Dynamic Query-Based Debugging, ECOOP '99, June 1999, Lisbon, Portugal, in Lecture Notes in Computer Science Vol. 1628, Springer Verlag, 135-160.

[13] OptimizeIt Web Site. Available at <http://www.optimizeit.com/oproductinfo.html>.

[14] Reiss, B. S. Software Visualization in the Desert Environment, ACM PASTE '98, Montreal, Quebec, Canada, 1998. 59-66.

[15] Richner, T., Ducasse, S., Recovering High-Level Views of Object-Oriented Applications form Static and Dynamic Information, In Proceedings of the International Conference on Software Maintenance (ICSM99), Oxford, England, Sept. 1999, pp. 13-22.

[16] Rogowitz, B. E., Rabenhorst, D. A., Gerth, J.A., Kalin, E.B. Visual Cues for Data Mining. Proceedings of the SPIE/SPSE Symposium on Electronic Imaging, 2657, February 1996, 275-301.

[17] Sefika, M., Sane, A., Campbell, R.H. Architecture-Oriented Visualization, In Proceedings of ACM OOPSLA '96. San Jose, CA, Oct 96, Published as SIGPLAN Notices 31(10), 389-405.

[18] Sevitsky, G., De Pauw, W., Konuru, R. Execution Slices for Analyzing Program Behavior. IBM Research Technical Report in preparation.

[19] Snodgrass, R. A Relational Approach to Monitoring Complex Systems, ACM Transactions on Computer Systems, Vol. 6 No. 2, May 1988, 157-196.

[20] Stasko, J. and Muthukumarasamy, J., Visualizing Program Executions on Large Data Sets, Proceedings of the IEEE Symposium on Visual Languages, Boulder CO, Sept. 1996, pp. 166-173.

[21] Systa T., On the Relationships between Static and Dynamic Models in Reverse Engineering Java Software, In Proc. of the 6th Working Conference on Reverse Engineering, Altanta, GA, USA, 1999, pp.304--313.

[22] Thomsen,E. OLAP Solutions: Building Multidimensional Information Systems. Wiley Computer Publishing, ISBN 0-471-14391-4.

[23] Tip, F. A Survey of Program Slicing Techniques. Journal of Programming Languages, Vol. 3 No. 3, Sept. 1995, pp. 121-189.

[24] Walker, R. J., Murphy, G. C., Freeman-Benson, B., Wright, D., Swanson, D., Isaak, J. Visualizing Dynamic Software System Information through High-level Models, OOPSLA '98, Vancouver, Oct 1998, Published as SIGPLAN Notices 33(10), pp. 271-283.