Scaling an Object-oriented System Execution Visualizer through Sampling

Andrew Chan, Reid Holmes, Gail C. Murphy and Annie T.T. Ying

Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver BC Canada V6T 1Z4
{chana, rtholmes, murphy, aying}@cs.ubc.ca

UBC CS Tech Report TR-2002-05
July 26, 2002

Abstract

Increasingly, applications are being built by combining existing software components. For the most part, a software developer can treat components as black-boxes. However, for some tasks, such as performance tuning tasks, a developer must consider how the components are implemented and how they interact. In these cases, a developer may be able to perform the task more effectively if dynamic information about how the system executes is available. To deal with the voluminous amount of dynamic information that can be generated, tool support is typically needed. In previous work, we demonstrated the utility of a tool, called AVID (Architectural VIsualization of Dynamics), that animates dynamic information in terms of developer-chosen architectural views. One limitation of this earlier work was that AVID relied on detailed—trace-based—information collected about the system's execution, limiting the duration of execution that could be considered. To enable AVID to scale to larger, longer-running systems, we have been investigating the visualization and animation of sampled dynamic information. In this paper, we discuss the addition of sampling support to AVID, and we present two case studies in which we experimented with animating sampled dynamic information to help with performance tuning tasks on the Eclipse integrated development environment.

1 Introduction

Increasingly, applications are being built by instantiating, combining, and extending existing software components. A developer integrating a specialized Java editing tool into the Eclipse integrated development environment (IDE) [2], for instance, will make use of a number of existing components, such as GUI, windowing, and file navigation components. This approach to development can provide many benefits, including reducing the time and effort needed to develop and deploy complex applications.

These development benefits are realized when a developer can treat the components being used as black-boxes. A developer accesses the functionality of the components through a set of programmatic interfaces.[1] For most development and evolution tasks on a system, this view of a component is sufficient. However, for some tasks, a developer needs to "open up" the component and consider how the component is implemented. As an example, the developer integrating the Java editing tool described above may use the Eclipse Java package view as a black-box component when implementing a system, but may need to consider the

[1] We use the term *software component* in this paper to refer to any piece of software that the developer treats largely as a black-box, and with which the developer interacts through some set of programmatic interfaces. We include in this definition libraries and object-oriented frameworks.

implementation of the package view if switching between packages leads to a performance problem.

In cases where opening up a component is possible and desirable, a developer can benefit from tool support to analyze the source and execution of the system. In this paper, we focus on cases where a developer can benefit from analyzing the execution; we refer to information collected from a system's execution as *dynamic information*.

Dynamic information is voluminous. One approach to dealing with the volume is to summarize the data collected and to present the developer with the summary. Profiling tools, such as JProbe Profiler [10], are examples of this approach. For some tasks, summary information is sufficient. A developer may be able to tune the performance by knowing which methods consumed the most execution time. As another example, knowing which methods executed may be sufficient to compute a coverage metric for guiding testing activities.

At other times, a developer requires more detailed information about the order in which execution events occurred, the frequency of certain patterns of calls, or other similar information [12]. In these cases, a developer can use a detailed visualization tool, such as Jinsight [9], that allows a developer to track and analyze various information including interactions between classes, and the contents of the heap. One of the major assets of these tools—their support for detailed investigation of execution—is also a liability for some tasks. A developer must typically have narrowed down the problem to a small part of the execution of the system for the tool to handle the volume of information, and the developer must consider the system largely at a uniformly low-level of detail, such as classes, making it difficult to correlate the information to a component view the developer may be using when building the system.

To help a developer in cases where a coarser-grained, component, view of the execution is useful, we introduced the AVID tool (Architectural VIsualization of Dynamics) [18]. AVID supports the off-line visualization of dynamic information collected from the execution of a Java system in terms of user-defined architectural views. One limitation of this earlier work was that AVID abstracted detailed dynamic information, working from essentially the same information as the detailed visualization tools described above. To enable AVID to visualize longer durations of large system, we have been investigating the visualization and animation of *sampled* dynamic information.

In this paper, we describe how we added sampling support to AVID, and the effect that support has on the size of the dynamic information. We describe two case studies in which we used AVID with sampling support to investigate two performance tuning tasks on the Eclipse IDE development. This paper makes two contributions:

- it demonstrates the utility of visualizing and animated sampled execution traces, and
- it discusses the tradeoffs of different sampling options.

We begin by describing the AVID tool (Section 2) and the support we have added to AVID for sampling. Next, we describe the case studies in which we applied AVID to two tasks on Eclipse (Section 3). We then present a discussion of issues involved with sampling (Section 4), and compare with related efforts (Section 5) before summarizing the paper (Section 6).

2 AVID

To provide context for our description of sampling support, we begin with a brief overview of the basic capabilities of the AVID tool. In-depth descriptions and discussions of AVID's capabilities are available elsewhere [18, 17, 1].[2]

2.1 Basic Features

AVID is an off-line visualizer for Java applications. A developer collects information—a trace—about the calls between methods and about the instantiation and destruction of objects in an execution of a Java application of interest. The developer then specifies, through a *mapping* file, a view to use to present the dynamic information. The view consists of a set of *entities*: Each entity represents a collection of classes in the application. The developer chooses a view that is relevant to the task at hand. Given the trace and the mapping, AVID presents a user-controllable animation

[2]The first version of AVID supported visualizing the execution of Smalltalk applications [18]. Although the current tool supports visualizing the execution of Java applications, the basic features are unchanged from those described in the earlier publication.

that allows the developer to traverse the trace and to view the execution in terms of the described entities. The user can, at any time, change the definition of entities in the animation to refine the view as desired.

To make this abstract description of AVID concrete, consider the following example. A developer is asked to fix a bug on the Java Petri Net editor (JARP) that involves a problem with handlers not appearing on nodes added dynamically [6]. JARP is built on the JHotDraw framework, which supports the creation of structured drawing editors [8].[3] As a first step, if the developer is not intimately familiar with JARP or JHotDraw, the developer could use AVID to investigate the interactions between parts of the framework and parts of the application when the bug occurs.

To proceed, the developer collects a trace of the execution of the system when the problem occurs. The AVID toolset uses the Jinsight tracer to collect dynamic information; a Jinsight trace is then postprocessed using AVID tools into the AVID format [17], which enables fast abstraction of the information in terms of user-defined entities. The developer must then define the entities of interest for investigating this bug. The developer chooses to focus on major framework and application components, specifying the mapping shown in Figure 1. This mapping lists five entities: four entities associated with major parts of the JHotDraw framework (prefixed with JHD), and one entity representing the JARP application. Associated with each entity is a regular expression describing a pattern for the names of classes associated with the entity. For example, the first line specifies that all classes starting with CH.ifa.draw.framework in their fully-qualified name are associated with the JHD-framework entity.

Given the processed trace and the mapping file, AVID displays the window shown in Figure 2. This window shows the *cel mode* of AVID in which the execution is broken into a sequence of *cel*s. Each cel displays both incremental and summary dynamic information about the dynamic information collected to that point. The incremental information consists of a hyperarc (in grey) showing the current call stack. The summary information consists of arcs showing the cumulative number of calls between different entities, and bars in each entity showing the number of object allocations and deallocations corresponding to the classes associated with the entity. For instance, in Figure 2, to this point in the collected dynamic information, 128 calls have occurred between objects associated with the JHD-standard and the JHD-framework entities, and 88 objects have been instantiated that are associated with the JHD-framework entity. A histogram for each entity can also be displayed that shows when objects have been allocated and deallocated: these histograms are controlled by a user preference set in the AVID menus and are not shown in Figure 2.

In the cel mode, buttons are active that allow a user to animate the execution. A user can choose to play the animation forward, can choose to step, forward or backward, through the animation, or can move the navigation bar to any point they desire to see in the animation. The position of the slider in Figure 2 indicates that the animation is about $3/4$ of the way through the dynamic information. The summary button at the top of the AVID window switches the window to a *summary mode* view. The summary mode allows a user to see cumulative call and object activity information in terms of the defined entities. As this mode is not used in the case studies, we do not discuss it further. The *reload* button allows a developer to change the definition of entities to use in the view during an AVID session.

When viewing a cel, a developer may wish to view more detailed information about the calls that have occurred, or the objects that are being allocated or deallocated. To determine this detailed information, a developer may click on a summary arc, or on an entity, and the appropriate information will be loaded into a slice definition in Jinsight. The slice definition allows a developer to use the Jinsight views to investigate just that piece of the execution. For example, a developer may use the table view in Jinsight to determine how many objects of each class, to that point in the execution, have been allocated or freed.

2.2 Sampling Support

The trace file collected to investigate the bug reported for JHotDraw described above is over 37Mb in size. This trace file represents only a small part of the execution, specifically starting the JARP application and placing a few figures into the editor.

[3] This bug is #477918 reported on the JHotDraw framework.

```
category JHD-framework class CH.ifa.draw.framework.*
category JHD-standard class CH.ifa.draw.standard.*
category JHD-figures class CH.ifa.draw.figures.*
category JHD-util class CH.ifa.draw.util.*
category JARP class edu.lcmi.petri.*
```

Figure 1: AVID Mapping File for JHotDraw Task

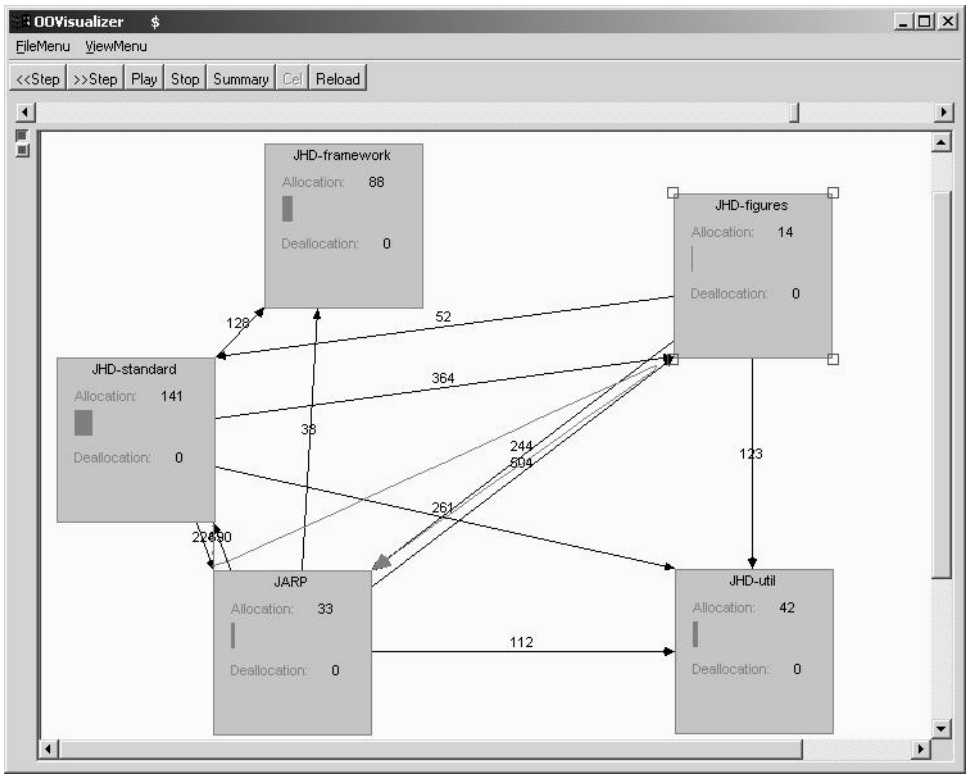

Figure 2: AVID with JARP Example Loaded

When the piece of the execution that needs to be analyzed is short and can be determined, the size of the trace may not be an issue. After all, disk space is plentiful, and the growing amounts of memory commonly found in computers can accommodate the use of detailed visualizers for traces in the hundreds of megabytes in size.

When a software developer is trying to isolate a piece of execution representative for the task at hand, or when the system is large, the size of the trace can quickly become Gigabytes in size. Reiss and Renieries reported, for instance, that a tracer they built for Java produces approximately one gigabyte of data for every ten seconds of JITed Java execution [13]. There are a number of approaches possible for reducing the size of the trace; we discuss these approaches in Section 5. Given the style of visualization in AVID, we decided to investigate the use of sampling to reduce the trace size.

The problem we faced was deciding what to sample. The input consisted of a discrete stream of events, including method entry, method exit, object allocation, object deallocation, thread start, and thread stop events. To provide flexibility in our investigations, we chose to support separate configuration of memory and control-flow event sampling. For memory events, a developer can choose to:

M-1 take every xth memory (object allocation or deallocation) event,

M-2 take the first memory event that occurs during or after xth timestamp, or

M-3 do M-1 or M-2 with a snapshot of the call stack output before every memory event that is sampled.

The call stack snapshot option provides context for allocation or deallocation.

For control-flow events, a developer can choose to:

C-1 take every xth control-flow (method entry or method exit) event,

C-2 take a snapshot of the call stack every xth event, AVID will determine which methods are entered or exited by comparing two consecutive snapshots, or

C-3 C-2 except that the snapshots are taken every xth timestamps.

```
<sampling>
  <range>
    <sample object="0"
            method="0"
            stack="50"
            stackonobject="no"
            type="event"
            start="10000"
            end="200000"/>
  </range>
  <default>
     <trace/>
  </default>
</sampling>
```

Figure 3: Control File

In addition to being able to control the kind of sample taken, the developer can also choose when sampling occurs, and can choose to intersperse sampling and trace information. Currently, a developer specifies the interspersion of sampling in the trace through a control file that is input to the trace post-processing step. The control file supports the definition of sampling ranges in the trace; for each range, different sampling parameters can be set. We chose to support sampling in the post-processing step to support the investigation of different sampling approaches over the same execution trace. We discuss issues associated with collecting sampling information in Section 4.

For example, a developer might choose to sample control-flow events using approach C-2 every 50 events between timestamps 10000 and 200000.[4] The portion of an XML-based control file to specify this choice is shown in Figure 3.

The visualization supported by AVID changes slightly when displaying sampled information. The word "sampling" appears near the timestamp shown at the top of the window for any cel displaying sampled information, and any memory event histograms, if displayed by the developer, are suppressed for the duration of the sampled information.

[4]The timestamps refer to tics recorded during the Jinsight tracing.

3 Case Studies

To investigate if there is utility in visualizing and animating sampled execution traces from an architectural view, and if so, to investigate different sampling options, we performed two case studies. Each case study focused on a performance tuning tasks on the Eclipse IDE for which the problem had been identified, and solved. Eclipse is a large system, consisting of approximately 775,000 lines of Java source code. Using completed performance tuning tasks for our case studies allows us to focus on the sampling capabilities of AVID. For each study, we describe the performance problem, the features of the problem that are evident when running AVID over trace dynamic information, the effect of sampling options on the visualization and animation of those features, and the results we synthesize from each study.

3.1 Case Study #1

This case study focused on bug #10216 in the Eclipse bugzilla problem reporting system, which is described as "filesystem is accessed too often". Specifically, when an Eclipse workspace was located on a slow(er) network connection, the performance of navigating in the package view and other parts of Eclipse degraded. This problem was noted against Eclipse 2.0 (build 20020214).

With AVID, we investigated two versions of Eclipse: build 20020125 in which the problem existed, and build 20020521 in which the problem was fixed. We refer to the former as the *unoptimized* version, and the latter as the *optimized* version.

3.1.1 AVID View

We used five entities in AVID to investigate the performance problem:

- JavaProject, representing a specific class in the Eclipse implementation that provides access to the files comprising a Java project,
- JDT-CORE, representing the classes involved in providing the non-UI parts of the Java programming environment support,
- JDT-UI, representing the classes involved in providing the UI parts of the Java programming environment support,
- CORE, representing the classes involved in supporting Eclipse plug-ins and the plug-in registry, and
- JDK, representing the classes comprising the Java development kit.

The mapping file is shown in Figure 4. From the viewpoint of a developer performing the task, this map includes seemingly omniscient information. While a developer might reasonably be expected to posit architectural entities corresponding to major components in Eclipse, how would the developer know to separate out the JavaProject class? We separated it out because it was mentioned in the description of the bug report. This might be a reason a developer assigned to the project would choose it as an entity. Alternatively, a developer might find, through a coarser AVID view, or through the use of another tool such as a profiler, that the class was heavily involved in the functionality of interest, and might choose to separate it out. In our uses of AVID, we have frequently started with a coarse-grained view, and then have subsequently refined the view based on investigating the arcs between entities, amongst other features of the visualization. The `reload` feature of AVID makes it straightforward to refine the view.

To investigate the problem, we collected a trace from each of the optimized and unoptimized versions. Each trace is representative of the same use of Eclipse: We focused on the behaviour of the system when a user adds an external jar (org.eclipse.core.boot/boot.jar) into a Java project, which contains only two other external jars (org.eclipse.jdt.core/jdtcore.jar and org.eclipse.jdt.ui/jdt.jar).

We then used AVID to view each of the traces. We were interested in how JavaProject interacted with the other entities. Figure 5 shows AVID positioned to a point near the end of the trace.[5] Viewing

[5] In this view, the developer set a parameter called the *step size* to a number greater than the default of one. The step size determines how many events are shown in a cel. A step size greater than one allows faster playing of the animation, and makes it easier to step across the execution. The step size can be set at any time through the user interface. As the view shows, when the step size is greater than one, an additional value appears on the summary call arcs representing the incremental change in the number of calls between the previous and the current cel. For example, five calls between JDT-CORE and JDK have occurred in the current step. These incremental values were not used in the case studies.

```
category JavaProject class org.eclipse.jdt.internal.core.JavaProject
category JDT-CORE class org.eclipse.jdt.core.*
category JDT-CORE class org.eclipse.internal.jdt.core.*
category JDT-UI class org.eclipse.jdt.ui.*
category JDT-UI class org.eclipse.internal.jdt.ui.*
category CORE class org.eclipse.core.*
category CORE class org.eclipse.pde.*
category JDK class java.*
```

Figure 4: Mapping File for Filesystem Access Study

cels in the trace from the unoptimized version, we found the following features of the problem:

F-1 20 calls occur from JDT-CORE to JavaProject before any call from JDT-UI. These calls surprised us because we had assumed that JavaProject was not used prior to adding the external jar in our usage scenario: We believed we had collected a trace from the point when the behaviour was triggered from the user interface. A developer assigned the performance tuning task would likely want to investigate these calls.

F-2 51 calls occur from JDT-CORE to JavaProject after a call from JDT-UI to JDT-CORE. The developer might choose to investigate why 31 additional calls are needed to JavaProject for a simple external jar addition to a simple project.

F-3 a second call occurs from JDT-UI to JDT-CORE before the end of the trace. After this call, there are no further calls to JavaProject. We used Jinsight to investigate the two calls from JDT-UI to JDT-CORE and found that the JDT-UI operations invoked were `processDelta` on org.eclipse.jdt.ui.JavaElementContentProvider: This method processes changes to the Java model.

To verify that these features were of likely interest in the performance tuning task, we also viewed the trace from the optimized version with AVID. We found:

- about the same number of calls from JDT-CORE to JavaProject before any call to JDT-UI.
- less calls—23 instead of 51—from JDT-CORE to JavaProject after a call from JDT-UI to JDT-CORE.
- no second call from JDT-UI to JDT-CORE.

3.1.2 AVID With Sampling

We sampled the unoptimized trace in a number of different ways and viewed the resulting animations to see if the features described above were evident. Since none of the features involved memory, we considered seven control-flow event samplings:

- C-1 with x set to 1000 (sample every 1000th control-flow event),
- C-1 with x set to 100,
- C-1 with x set to 10,
- C-2 with x set to 1000 (snap call stack every 1000 events),
- C-2 with x set to 100,
- C-3 with x set to 10000 (snap call stack every 10000 timestamps). and
- C-3 with x set to 1000.

Our current dynamic information format when the call stack is snapped results in a larger trace than the original when x is set to less than 100 for C-2 sampling and less than 1000 for C-3 sampling.

3.1.3 Results

Table 1 summarizes the results. The first column describes the sampling parameters. The second column reports the total number of bytes required to represent the sampled information in AVID format. The third column reports the percentage,

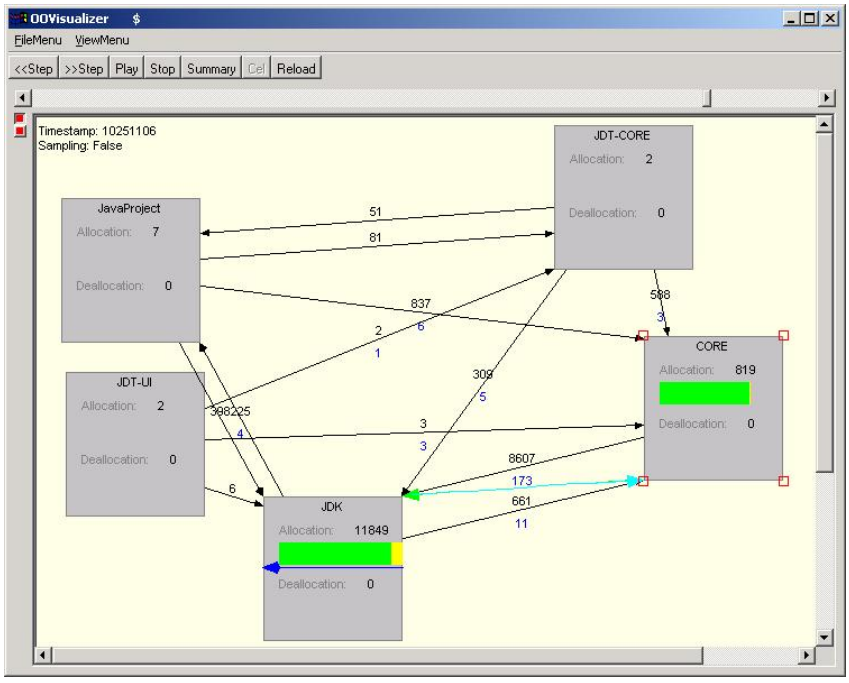

Figure 5: AVID with Trace from Unoptimized Filesystem Problem

based on size, of the sampled dynamic information compared to the trace dynamic information, which was over 14.4Mb. The fourth column describes whether the features were evident when viewing the sampled information with AVID.

The table shows that we were not able to find any evidence of usefulness for C-1 sampling, which involves taking every xth method entry or method exit event. The difficulty is that this form of sampling does not retain sufficient context about the individual events. C-2 and C-3 sampling, which involve snapshots of the call stack at every xth event or timestamp, show more promise. Neither is able to fully detect the specific features we identified for the performance problem because these features were all dependent upon the identification of two calls from JDT-UI, which were apparently not sampled. However, these kinds of sampling were able to detect numerous calls from JDT-CORE to JavaProject, which might lead a developer in the right direction for solving the problem. These kinds of sampling required significantly less data; the sampled data was 7 to 63% the size of the original trace.

One might argue that simply seeing a "large" number of calls, without the context of the JDT-UI calls of interest, could be achieved by using a profiler. This criticism is valid with our current sampling parameters. If a developer determined that a particular kind of event was important to solving a problem, such as a developer positing that a call from JDT-UI to JDT-CORE is of interest, it might be helpful to state certain kinds of events that are to be included in the sampled information, whether or not they appear at a sample point. For instance, a developer could state that any calls between JDT-UI and JDT-CORE be included. We can currently simulate the result of this kind of sampling directive by using AVID's ability to intersperse sample and trace data. We knew at what point the calls to JDT-UI occurred. Thus, we set the sampling to default to C-3 with x=1000, but included trace data around the JDT-UI calls. Prior to the first call from JDT-UI to JDT-CORE, 18 calls were sampled from JDT-CORE to JavaProject which is similar to Feature F-1.

Table 1: Sampling Results for FileSystem Problem

Sampling Parameters	File size	% of Original Size	Results
C-1, x=1000	35K	.3%	No features are present
C-1, x=100	205K	1.8%	No features are present
C-1, x=10		12%	No features are present.
C-2, x=1000	793K	7%	Partial support of F-2: No calls from JDT-UI are shown, but 17 calls are present from JDT-CORE to JavaProject at the end of the trace.
C-2, x=100	7.1M	63%	Partial support of F-2: No calls from JDT-UI are shown, but 47 calls are present from JDT-CORE to JavaProject at the end of the trace.
C-3, x=10000	436K	3.8%	Partial support of F-2: No calls from JDT-UI are shown, but 17 calls are present from JDT-CORE to JavaProject at the end of the trace.
C-3, x=1000	2.7M	24%	Partial support of F-2: No calls from JDT-UI are shown, but 17 calls are present from JDT-CORE to JavaProject at the end of the trace.

3.2 Case Study #2

This case study focused on the "import from files" operation. This operation adds files to an existing Eclipse Java project. The files are copied from the source location into the location of the Eclipse project workspace. This study considers Eclipse versions 0.107 and 0.137. The former is the *unoptimized* version, and the latter is the *optimized* version. Both versions use a `Path` class, which represents and gets segments from a filesystem path. In the unoptimized version, the implementation of `Path` stored the resource location as one `String` object: This object was parsed on the fly to retrieve the segments. This implementation was costly, both in terms of objects allocated and objects garbage collected. In the optimized version, the implementation of `Path` was changed to store the segments in memory. Although more `String` objects are held in memory, fewer strings overall need to be created and garbage collected, improving performance. The problem and the versions were identified with the help of an expert Eclipse developer.

3.2.1 AVID View

We used five entities in AVID to investigate the performance problem:

- UI, representing the basic UI operations in Eclipse,
- ImportWizard, representing the triggering of the import operation,
- Path, representing the Path class of interest,
- Runtime, representing the Eclipse runtime other than Path, and
- JDK, representing the classes comprising the Java development kit.

The mapping file is shown in Figure 6. As in the previous case study, this mapping is not the first that a developer might specify. We separated out the ImportWizard entity from the UI entity after realizing that there were a number of operations happening involving the UI: We wanted an entity, ImportWizard, that would allow us to determine when the behaviour of the import operation began. We separated the `Path` entity based on our knowledge of the problem. As with the previous case study, a developer might iterate towards separating out this entity based on their previous knowledge, or as they investigated the problem with AVID and other tools.

As before, we collected a trace from each of the unoptimized and optimized versions that focused on importing 60 files into a project. We then used AVID to view the traces, and we found the following features in the unoptimized trace which indicated the problem:

```
category ImportWizard class org.eclipse.ui.wizards.datatransfer.*
category UI class org.eclipse.ui.*
category Path class org.eclipse.core.runtime.Path
category Runtime class org.eclipse.core.runtime.*
category Runtime class org.eclipse.internal.runtime.*
category JDK class java.*
```

Figure 6: Mapping File for Import Case Study

F-1 there are 4 calls to Path from ImportWizard, and 62 calls from Path to the JDK, when the ImportWizard is called.

F-2 roughly one-third of the way through the trace, there are still 4 calls to Path from ImportWizard, and 159440 calls from Path to the JDK, with over 21000 objects allocated in the JDK.

F-3 At the end of the trace, there are 1881 calls to Path from the ImportWizard, and 253368 calls from Path to the JDK, with over 114000 objects allocated in the JDK.

We verified these features by viewing the optimized trace. These views indicated:

- When the ImportWizard is called, there are no calls to Path from ImportWizard.
- Roughly 1/3 of the way through the trace, there are 12 calls to ImportWizard, 30 calls from ImportWizard to Path, but far fewer calls from Path to the JDK, only 435, and only 131 JDK objects allocated,
- At the end of the trace, there are 18 calls from UI to ImportWizard (compared to 1 in the unoptimized version), 1150 calls (many more!) from ImportWizard to Path, but far fewer calls from Path to JDK (137518) and far fewer JDK objects allocated (22941).

3.2.2 AVID With Sampling

As before, we sampled the unoptimized trace in a number of different ways and viewed the resulting animations to see if the features described above were evident. In this study, we considered both control-flow and memory event samplings:

- C-1 with x set to 100 (take every 100th control-flow event) and M-1 with x set to 100,
- M-3 with M-1 and x set to 1000 (take every 1000 memory event and snap the callstack),
- M-3 with M-1 and x set to 100, and
- M-3 with M-2 and x set to 1000 (take a memory event every 1000th timestamp and snap the callstack).

3.2.3 Results

Table 2 summarizes the results. The format of the table is the same as used for Table 1.

Since the features of interest in this case study were largely based on the magnitude of calls or objects allocated, it was more difficult to determine when a feature was present when viewing the sampled data. We subjectively determined when the number of calls or objects allocated would have triggered further investigation, and used the terms "partially evident" if it was possible that the numbers would have triggered action on the part of a developer, and "somewhat evident" if it was possible, but less likely that the numbers would have triggered a developer to act.

Table 2 shows that we again required context information to find the features of the problem. Thus, we were successful when both control-flow and memory events were sampled (C-1 and M-1) at a relatively fine-granularity (i.e., every 100 events), and when information from the call stack was included in when sampling based on timestamps (M-3 with M-2). In all of these cases, the sampled data was significantly smaller than the original data, ranging from 1% to 13% the size of the original trace file.

4 Discussion

Based on our case studies, is it useful to software developers to visualize and animate sampled data?

Table 2: Sampling Results for Import Problem

Sampling Parameters	File size	% of Original Size	Results
C-1, x=100 & M-1, x=100	875K	1%	F-2 is partially evident with 247 calls from Path to JDK and 174 JDK objects allocated. F-3 is somewhat evident with 490 calls from Path to JDK and 1140 JDK objects allocated.
M-3 with M-1, x=1000	245K	0.3%	No features are evident.
M-3 with M-1, x=100	2.2M	2.6%	F-1 is partially evident with 3 calls to Path from ImportWizard when ImportWizard is called.
M-3 with M-2, x=1000	10.1M	12.5%	F-2 is partially evident with 661 calls to JDK from Path and 222 JDK objects allocated. F-3 is partially evident with 145 calls from ImportWizard to Path, 984 calls from Path to JDK, and 1053 JDK objects allocated.

Is sampling the only way to deal with visualizing and animating systems as they grow in size and execution time? If visualizing and animating sampled dynamic information may be useful, where should we go from here? We discuss each of these questions in turn below.

4.1 Usefulness

Our case studies show that there exist some kinds of sampling that, when the data is visualized and animated, do retain some of the features of the performance problem being studied. In these cases, the sampled data is often much less than half, and sometimes is just 10%, of the size of the original trace. Such reductions could enable the collection and subsequent analysis of data from longer running systems.

Our case studies also indicate that the *animation* of the sampled data was an important characteristic, leading to helpful lines of questioning about the sequencing of behaviour. For example, in the first case study, the existence of unexpected calls between the JavaProject and the UI architectural entities *before* the trigger call to the UI entity suggests that a developer may need to investigate how JavaProject is used in more detail. As another example, recognizing linked growth patterns over time in calls or allocations can be beneficial in identifying a performance problem; for example, in the second case study, we noted the calls to JDK rising with the calls to the Path entity. Questions of this form are less likely to arise if only summarized sample data, such as produced by a profiler, are viewed. The fact that animations of some kinds of sampled data retained these features is encouraging. Since our focus in these investigations was to determine if the features could exist in animations of sampled data, the question of whether a developer would notice such features in the sampled data without prior knowledge of the animations of the trace data is still open.

4.2 Scale

An alternative way to enable developers to more effectively analyze dynamic information from long-running, large systems is to rely on on-line approaches rather than the off-line approach taken by AVID. In an on-line approach, the visualization is displayed as the data is produced from the executing system. When an on-line approach is used, sampling is not necessary as a means of reducing the amount of information collected. However, on-line approaches can limit the kinds of analyses that can be conducted on the information collected. For instance, it may be difficult in an on-line approach for a developer to investigate the sequence of behaviour without rerunning the system many times; for some systems, it may be costly to rerun the system. In these cases, it may be preferable to use an off-line approach.

Sampling may also have a useful role to play with on-line approaches if sampling, rather than tracing, would perturb the system less when the dynamic information was collected, making it possible for a developer to investigate the behaviour of the system of interest, rather than the system plus tracing.

4.3 Next Steps

The work described in this paper provides a first step towards understanding how the visualization and animation of sampled dynamic information could aid software developers. The next step in evaluating the approach would be to have software developers, likely in a controlled setting, attempt to use AVID with sampling to solve performance tuning and other tasks, without knowing the "solution" to the task in advance. The next step in advancing the technology would be to collect sampled dynamic information from the execution rather than sampling collected trace information.

5 Related Work

This paper has discussed the feasibility and utility of visualizing and animating sampled dynamic information in the context of user-defined architectural views of a software system. The intent of this work is to enable the use of animated visualizations for longer-running, larger systems. We thus focus our comparisons to earlier work on tools aimed at providing coarse-grained visualizations of software system execution, tools that visualize sampled system execution information, and approaches aimed at reducing the size of traces.

5.1 Coarse-Grained Visualization

Two approaches have been taken to visualize larger amounts of the execution of larger systems.

One approach is to display as much detailed information on the screen as possible. The information mural work by Jeerding and colleagues takes this approach [7]. Their work on execution murals places classes on an axis and uses coloured single pixel bars to indicate calls between the classes. The result, an execution mural, is able to display thousands of interactions on one screen. The authors also describe applying this basic idea in a pattern mural that displays sequences of calls (patterns) that are automatically detected from the execution. The view provided in such a mural does not support the investigation of the execution in terms of architectural components.

The second approach is to display, in terms of an architectural view, information about the execution. This approach is the one taken in AVID. This approach was also taken by Sefika and colleagues [15]. Their approach differs from AVID in two ways. First, they visualize coarse-grained information collected from the system rather than abstracting fine-grained dynamic information as is done in AVID. Using their approach, a developer must a priori determine the abstractions of interest, and must instrument the system to produce that information. Second, their system executes on-line. As we discussed earlier, an on-line approach bypasses problems associated with storing the data, but it may limit investigations by a developer. As an example, a developer who wishes to view a portion of the execution from a different coarse-grained view, must reinstrument and re-run the system. When running the system to the point of interest is expensive in terms of set-up time or equipment, an off-line approach may provide more flexibility.

Grundy and Hosking's SoftArch system also uses architectural views to display dynamic information [4]. With SoftArch, a developer creates an architectural model of interest and describes the refinement of that model to classes using UML diagrams and refinement links [14]. As the system executes, events associated with classes are forwarded through the refinement links to the architectural views. A developer can access a variety of visualizations, including colors to denote object allocations associated with architectural entities, bar graphs of method invocations, and dialogs of cached method invocations that have occurred. No performance information about SoftArch is available. Similar to AVID, SoftArch maps fine-grained execution information to the architectural view and may thus have similar issues regarding the visualization of larger, longer-running systems.

5.2 Visualizing Sampled Data

The `gprof` tool is perhaps the most common tool that software developers use that involves the visualization of sampled data to aid software engi-

neering tasks. This profiling tool primarily produces information about the time spent in parts of the program in terms of the call graph of the program [3]. The tool samples the program counter, and infers execution time from the samples in the program. The tool displays the summarized execution time in the context of the call graph. The use of sampling in AVID differs in two fundamental ways. First, the sampling is not intended to be used as a means of estimating the time spent in a piece of the program, and thus, AVID supports a number of different kinds of sampling, both event and timestamp based. Second, AVID supports the animation of the sample data; `gprof` presents a summary of the sampled data at the end of execution. The `gprof` tool was developed for programs written in C; tools, such as JProbe, provide support similar to `gprof` for Java.

A number of tools that are intended to help improve or steer the performance of parallel or distributed programs use sampling as a means of reducing the amount of data considered. An example of such a tool is the PVaniM system that supports on-line and post-mortem visualization of network computing environments [16]. The on-line visualizations rely on sampled data; the post-mortem visualizations rely on trace data. The on-line visualizations include host views in which the average number of jobs in the run queue of each host is displayed, and a communication matrix view showing aggregate and interval statistics regarding message communication. These views are updated according to a sampling rate set by the user. The most similar view to AVID is the communication matrix view, which is categorized as a debugging view. The authors note that "[a]lthough the level of detail is reduced compared to its postmortem counterpart, in many cases the view is still able to provide some initial indication of anomalous behavior" [16, p. 9].

5.3 Trace Compression

A number of techniques have been developed to collect and store trace information [11]. These approaches have largely focused on the efficient collection and representation of detailed execution information, such as which data locations are referenced. These techniques often use static analysis of the program text to determine the appropriate points to use to create a minimal amount of trace information. These techniques were developed to assist in the design of memory systems, and to guide the behaviour of parallelizing compilers; less detailed traces are needed for the software engineering tasks we are supporting.

One approach to reducing the size of traces to support software engineering activities was mentioned above. Sefika and colleagues reduced the size of trace information visualized by having the developer build architectural instrumentation into the system. As described above, this approach limits the views a developer can use to view the system. It is unclear if the amount of information produced is sufficiently reduced to support the visualization of long-running systems.

Reiss and Renieris take a two-phased approach to reducing the size of traces: they select subsets of the data and compact it, and then they encode the data in a way that allows the structure of the data to be inferred [13]. An example of a first phase approach is limiting the collection of dynamic information to a certain set of classes in the system. This approach is also supported by AVID: A developer can specify parts of a system for which dynamic information should not be retained. An example of a second phase approach is to use run-length encoding or to build a finite state automaton that is representative of the trace. The approaches Reiss and Renieris use in the second phase tend to focus on one kind of event, specifically calls, and focus on the aggregation of statistics, such as number of calls, into the encoded representation. These encodings are not well suited to an animation style visualization.

Hollingsworth and colleagues describe a hybrid approach to instrumenting a large-scale parallel or distributed application that is detailed, frugal and scalable [5]. In their approach, detailed, exact metrics are collected about resource usage, such as the time spent in a procedure. These exact metrics are then sampled. This approach permits accurate reporting of a metric at some chosen frequency. This approach is well suited to cases where an aggregate statistic is to be reported against some structure, such as procedures. To be applicable to animated visualizations such as AVID, the approach would need to be extended to provide some temporal ordering of the information, such as *x* calls happened between these two entities and then *y* calls happened between another two entities, and so on.

6 Summary

AVID supports the off-line visualization and animation of the execution of a Java-implemented application in the context of an architectural view defined by the user. This paper describes the addition of sampling support to AVID, and our initial investigations into the utility of this sampling support. Our intent in adding support for visualizing and animating sampled dynamic information to AVID was to allow AVID to scale to larger, longer-running systems.

We found that visualizing and animating sampled dynamic information can be potentially useful to a software developer. We found that any dynamic information that is sampled must include sufficient contextual information to support interpretation of the animation. Specifically, we found utility when we sampled every xth event or timestamp, and when we, at that point, also took and reported a snapshot of the call stack: the call stacks can be compared to add contextual information into the animation. In our two case studies, these kinds of sampling led to dynamic information that was significantly smaller, often only 20%, the size of the original trace-based dynamic information

Acknowledgments

This research was funded by the Consortium for Software Engineering Research (CSER) in cooperation with Object Technology International. Andrew Catton, Thad Heinrichs, Robert Walker, and Albert Wong contributed to the implementation of AVID. “Java” is a trademark of Sun Microsystems.

About the Authors

Andrew Chan is a M.Sc. student in the Department of Computer Science at the University of British Columbia. His research interests are human-computer interaction and software engineering. He may be contacted at chana@cs.ubc.ca.

Reid Holmes is a M.Sc. student in the Department of Computer Science at the University of British Columbia. His research interests are in refactoring and extreme programming. He may be contacted at rtholmes@cs.ubc.ca.

Gail Murphy is an associate professor in the Department of Computer Science at the University of British Columbia. She is interested in creating and evaluating methods and tools to help software developers manage and evolve software systems both at design time and in source code. She may be contacted at murphy@cs.ubc.ca.

Annie Ying is a M.Sc. student in the Department of Computer Science at the University of British Columbia. She is interested in data mining and software engineering. She may be contacted at aying@cs.ubc.ca.

References

[1] http://www.cs.ubc.ca/~Murphy/AVID.

[2] http://www.eclipse.org.

[3] S.L. Graham, P.B. Kessler, and M.K. Mckusick. Gprof: a call graph execution profiler. In *Proc. of SIGPLAN ’82 Symposium on Compiler Construction*, pages 120–126, 1982.

[4] J.C. Grundy and J.G. Hosking. High-level static and dynamic visualization of software architectures. In *Proc. of IEEE Symposium on Visual Languages*, pages 5–12, 2000.

[5] J.K. Hollingsworth, B.P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *Proc. of Scalable High-Performance Computing Conf. (SHPCC)*, pages 841–850, 1994.

[6] http://www.sourceforge.net/projects/jarp.

[7] D. Jerding, J.T. Stasko, and T. Ball. Visualizing interactions in program executions. In *Prof. of the Int’l Conf. on Software Engineering (ICSE)*, pages 360–370, 1997.

[8] http://www.sourceforge.net/projects/jhotdraw.

[9] http://www.research.ibm.com/jinsight.

[10] http://www.sitraka.com/software/jprobe/jprobeprofiler.html.

[11] J.R. Larus. Efficient program tracing. *Computer*, 26(5):52–61, 1993.

[12] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proc. of the ACM*

Conf. on Object-oriented Programming Systems, Languages, and Applications (OOPSLA), pages 326–337. ACM Press, 1993.

[13] S.P. Reiss and M. Renieris. Encoding program executions. In *Proc. of the 23rd Int'l Conf. on Software Engineering (ICSE)*, pages 221–230. ACM Press, 2001.

[14] J. Rumbaugh, I. Jacobson, and G. Booch. *UML Reference Manual*. Addison Wesley, 1998.

[15] M. Sefika, A. Sane, and R.H. Campbell. Architecture-oriented visualization. In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 389–405, 1996.

[16] B. Topol, J.T. Stasko, and V. Sunderam. Pvanim: a tool for visualization in network computing environments. *Concurrency: Practice and Experience*, 10(14):1197–1222, 1998.

[17] R.J. Walker, Gail C. Murphy, Jeffrey Steinbok, and Martin P. Robillard. Efficient mapping of software system traces to architectural views. In *Proc. of CASCON 2000*, pages 31–40, 2000.

[18] R.J. Walker, G.C. Murphy, B.N. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *Proc. of the ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 271–283. ACM Press, 1998.