# 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

## Abstract

*Increasingly, applications are being built by combining existing software components. For the most part, a software developer can treat the components as black-boxes. However, for some tasks, such as when performance tuning, 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 by using dynamic information about how the system executes. 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 trace information collected about the system's execution; traces for even small parts of a system's execution can be enormous, limiting the duration of execution that can 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.*

## 1. Introduction

Increasingly, applications are being built by instantiating, combining, and extending existing software 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, accessing the functionality of the components through programmatic interfaces. For most development and evolution tasks, this view of a component is sufficient. However, for some tasks, such as when performance tuning, a developer needs to "open up" the component and consider how the component is implemented.

When a component is opened up, a developer can benefit from tool support to analyze the source and execution of the system. In this paper, we focus on the analysis of *dynamic information* collected from a system's execution. Dynamic information is voluminous. One approach to dealing with the volume is to present a summary of collected data to the developer. Profiling tools, such as JProbe Profiler [6], are examples of this approach. For some tasks, summary information is sufficient. For example, a developer may be able to tune the performance by knowing which methods consumed the most execution time.

At other times, a developer requires more detailed information about the order of execution events, the frequency of certain patterns of calls, or other similar information [8]. In these cases, a developer can use a detailed visualization tool, such as Jinsight [5], that allows a developer to track and analyze such information as interactions between classes and the contents of the heap. A major asset of these tools—their support for detailed investigation of execution—can also be a liability. A developer must typically have narrowed the problem down to a small piece of the execution for the tool to handle the volume of information, and the developer must typically view the system at a low-level of detail, such as classes, placing the onus on the developer to correlate the information to a component view.

To help a developer in cases where a coarser-grained, component-type view of the execution is useful, we introduced the AVID tool (Architectural VIsualization of Dynamics) [12]. 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 relied on es-

Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)
1092-8138/03 $17.00 © 2003 **IEEE**
IEEE COMPUTER SOCIETY

sentially 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 the results of our initial explorations in visualizing and animating sampled execution traces. We describe how we have added sampling support to AVID, and describe two case studies in which we used AVID with sampling support to investigate performance tuning tasks on the Eclipse IDE. This paper demonstrates that visualizing and animating sampled execution traces shows sufficient promise to warrant future investigation, and provides an initial discussion of 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

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 that allows the developer to traverse the trace and to view the execution in terms of the described entities.

We focus here on features of AVID relevant to this paper; in-depth descriptions and discussions of AVID's capabilities are available elsewhere [12, 13, 1].[1]

To make this abstract description of AVID concrete, we consider an example. A developer working on the Eclipse open-source IDE [2] is asked to investigate a bug where the "filesystem is accessed too often".[2] As a first step, if the developer is not intimately familiar with Eclipse, the developer could use AVID to investigate interactions between the framework and the application when the bug occurs.

[1] The first version of AVID supported visualizing the execution of Smalltalk applications [12]. Although the current tool supports visualizing the execution of Java applications, the basic features are unchanged from those described in the earlier publication.

[2] This bug is #10216 in the Eclipse Bugzilla problem reporting system.

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 [13], 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 below.[3]

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

Each entity in the file is a regular expression describing the names of classes to associate with the entity. For example, classes starting with `org.eclipse.jdt.ui` or `org.eclipse.internal.jdt.ui` are to be associated with the JDT-UI entity.

Given the AVID trace and the mapping file, AVID displays the window shown in Figure 1. This window shows the *cel mode* 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 1, to this point in the collected dynamic information, 837 calls have occurred between objects associated with the `JavaProject` and the `CORE` entities, and 819 objects have been instantiated that are associated with the `CORE` entity.

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 1 indicates that the animation is about three-quarters of the way through the dynamic information. 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

[3] We are not showing the full mapping syntax, and sometimes elide ([...]) the class names, for lack of space.


IEEE COMPUTER SOCIETY

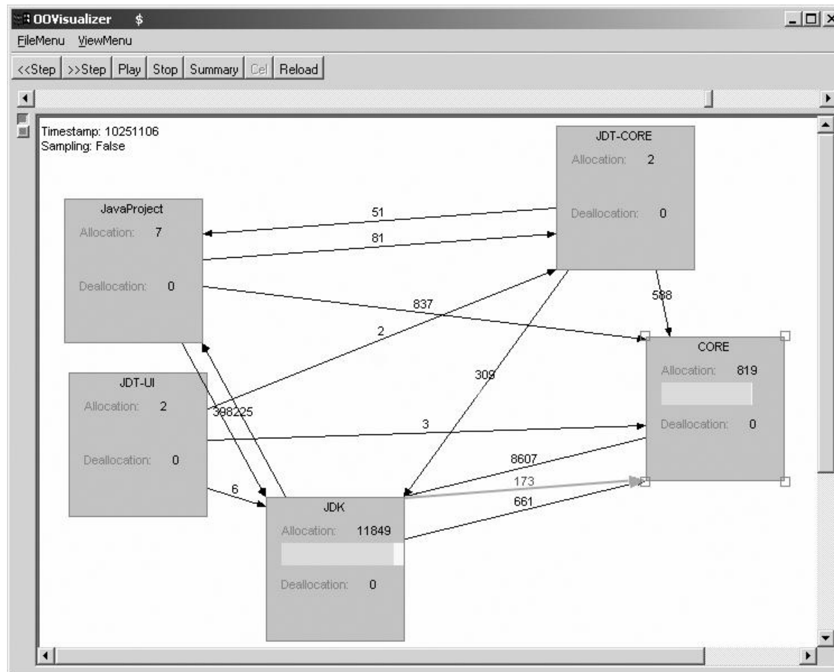


**Figure 1. AVID with Trace from Unoptimized Filesystem Problem**

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.

### 2.1. Sampling Support

The AVID trace file collected to investigate the Eclipse bug is over 14.4 MB. This trace represents only a small part of the execution: adding an external jar file into a Java project. When the relevant piece of the execution can be determined and is short, the size of the trace may not be an issue.

However, when a software developer is trying to isolate a relevant piece of execution for the task at hand, or when the system is large, the size of the trace can become an issue. As an example, a tracer Reiss and Renieries built for Java produces approximately one gigabyte of data for every ten seconds of Java execution with JIT enabled [9]. A number of approaches to reduce the size of the trace are possible (see Section 5). Given the style of visualization in AVID, we decided to investigate the use of sampling to reduce the size of the trace.

We had to decide 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. Since AVID supports animation of the data, we did not need to limit ourselves to investigating statistically significant samples. Rather, we wanted to explore whether sparse samples would retain enough features of the execution to help a developer reason about a task. To provide flexibility in this exploration, we chose to support separate configuration of memory and control-flow event sampling. For memory events, a developer can choose from three options:

M-1 take every $x$th memory (object allocation or deallocation) event,

M-2 take the first memory event that occurs during or after $x$th timestamp, or

M-3 do M-1 or M-2, and snapshot the call stack before each sampled memory event; consecutive snapshots are compared to determine which methods have been entered or exited, thereby providing control-flow context for the memory events.

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

C-1 take every $x$th control-flow (method entry or method exit) event,

C-2 take a snapshot of the call stack every $x$th event; consecutive snapshots are compared to determine which methods have been entered or exited, or

C-3 C-2, except that the snapshots are taken every $x$th timestamps.

Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC'03)
1092-8138/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY

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 sampled and traced data in an AVID session. We discuss issues associated with collecting sampling information in Section 4.

## 3. Case Studies

We performed two case studies to investigate if there was any utility in visualizing and animating sampled execution traces from an architectural view. Each case study focused on a previously identified and solved performance tuning task on Eclipse. Eclipse is a large system, consisting of over 775,000 lines of Java source code. Using completed performance tuning tasks for our case studies allowed 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 dynamic trace 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 the bug described in Section 2. 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 the mapping described in Section 2 to investigate the performance problem in AVID. The JavaProject entity represented a specific class in Eclipse providing access to the files comprising a Java project. The JDT-CORE entity represented the classes providing the non-UI parts of the Java programming environment support. The JDT-UI entity represented the classes providing the UI parts of the Java programming environment support. The CORE entity represented the classes supporting Eclipse plug-ins and the plug-in registry, and the JDK entity represented the classes comprising the Java development kit.

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. 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.

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. Viewing 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 the call from JDT-UI to JDT-CORE. The developer might choose to investigate why these additional calls are needed 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 from JDT-CORE.

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.
- fewer 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 the six control-flow event samplings shown in Table 1.


IEEE COMPUTER SOCIETY

#### 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, based on size, of the sampled information compared to the AVID trace file, which was over 14.4 MB. 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 $x$th 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 $x$th event or timestamp, show more promise because they enable tracking of longer-running methods. 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. If a developer determined that a particular kind of event was important to solving a problem, such as a call from JDT-UI to JDT-CORE, it might be helpful to state that those kinds of events must be included in the sampled information whether or not they appear at a sample point. A developer could then animate the now contextualized sampled information. More work is needed to understand the kinds of software engineering tasks for which a developer can benefit from animating summarized sampled information.

### 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, which 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. The UI entity represented basic UI operations in Eclipse. The ImportWizard entity represented the triggering of the import operation. The Path entity represented the `Path` class of interest. The Runtime entity represented the Eclipse runtime other than Path. The JDK entity represented the classes comprising the Java development kit. The mapping file is shown below.

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

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 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 three features in the unoptimized trace that indicated the problem:

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 and found the following.

**Table 1. Sampling Results for FileSystem Problem**

| Parameters | File size | % Size | Results |
|---|---|---|---|
| C-1, $x$=1000 | 35K | 0.3% | No features are present |
| C-1, $x$=100 | 205K | 1.8% | 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. |

- 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, studying the settings shown in Table 2.

#### 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? Is sampling the only way to deal with visualizing and animating systems as they grow in size and execution time? 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



IEEE COMPUTER SOCIETY

Table 2. Sampling Results for Import Problem

| Parameters | File size | % 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. |

fact that animations of some kinds of sampled data retained these features is encouraging.

On the other hand, the success of some sampling parameters in one case and not the other, such as the success of C-1 sampling in the second case but not the first, indicate the sensitivity of the sampling parameters to the task and the structure of the system. Our work to date has focused on *whether* animating samples of traces shows any value; more work is needed to understand how to pick appropriate sampling parameters for a given system and task.

Also, since our focus in these investigations was to determine if the features could exist in animations of sampled data, a large open question is whether a developer would notice such features in the sampled data without prior knowledge of the animations of the trace data.

### 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 AVID's off-line approach. In an on-line approach, the data is visualized as the system executes, eliminating the need to collect the data, and possibly limiting the kinds of analyses that can be conducted. 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 in on-line approaches if sampling, as compared to tracing, would perturb the system less during data collection.

## 5. Related Work

### 5.1. 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 engineering tasks. This profiling tool displays a summary of the execution time spent in each part of the call graph of the program [3]. 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.

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 [11]. 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" [11, p. 9].

### 5.2. Trace Compression

A number of techniques have been developed to collect and store trace information [7]. These approaches have


IEEE COMPUTER SOCIETY

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.

Sefika and colleagues reduced the size of trace information visualized by having the developer build architectural instrumentation into the system of interest [10]. This approach limits the views a developer can use to examine 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 [9]. 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. 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 [4]. 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 may 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 potential utility when we sampled every $x$th 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.

## 7. Acknowledgments

This research was funded by CSER in conjunction with IBM (Ottawa Software Labs). A. Catton, T. Heinrichs, R. Walker, and A. Wong contributed to the implementation of AVID. “Java” is a trademark of Sun Microsystems.

## References

[1] `http://www.cs.ubc.ca/~murphy/AVID`.
[2] `http://www.eclipse.org`.
[3] S. Graham, P. Kessler, and M. Mckusick. Gprof: a call graph execution profiler. In *Proc. of SIGPLAN ’82 Symp. on Compiler Construction*, pages 120–126, 1982.
[4] J. Hollingsworth, B. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *Proc. of SHPCC*, pages 841–850, 1994.
[5] `http://www.research.ibm.com/jinsight`.
[6] `http://www.sitraka.com/software/jprobe/jprobeprofiler.html`.
[7] J. Larus. Efficient program tracing. *Computer*, 26(5):52–61, 1993.
[8] W. D. Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proc. of OOPSLA*, pages 326–337. ACM Press, 1993.
[9] S. Reiss and M. Renieris. Encoding program executions. In *Proc. of ICSE*, pages 221–230. ACM Press, 2001.
[10] M. Sefika, A. Sane, and R. Campbell. Architecture-oriented visualization. In *Proc. of OOPSLA*, pages 389–405, 1996.
[11] B. Topol, J. Stasko, and V. Sunderam. Pvanim: a tool for visualization in network computing environments. *Concurrency: Practice and Experience*, 10(14):1197–1222, 1998.
[12] R. Walker, G. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *Proc. of OOPSLA*, pages 271–283. ACM Press, 1998.
[13] R. Walker, G. C. Murphy, J. Steinbok, and M. P. Robillard. Efficient mapping of software system traces to architectural views. In *Proc. of CASCON*, pages 31–40, 2000.



IEEE COMPUTER SOCIETY