SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions[*]

Shrinivas Joshi
Advanced Micro Devices
Shrinivas.Joshi@amd.com

Alessandro Orso
Georgia Institute of Technology
orso@cc.gatech.edu

Abstract

Because of software's increasing dynamism and the heterogeneity of execution environments, the results of in-house testing and maintenance are often not representative of the way the software behaves in the field. To alleviate this problem, we present a technique for capturing and replaying partial executions of deployed software. Our technique can be used for various applications, including generation of test cases from user executions and post-mortem dynamic analysis. We present our technique and tool, some possible applications, and a preliminary empirical evaluation that provides initial evidence of the feasibility of our approach.

1 Introduction

Today's software is increasingly dynamic, and its complexity is growing, together with the complexity of the environments in which it executes. Because modern software can behave quite differently in different environments and configurations, it is difficult to assess its performance and reliability outside the actual time and context in which it executes. Therefore, the results of traditional testing and maintenance activities—performed in-house, on a limited number of configurations, and using developer-created inputs—are often not representative of how the software will behave in the field [4, 9, 11].

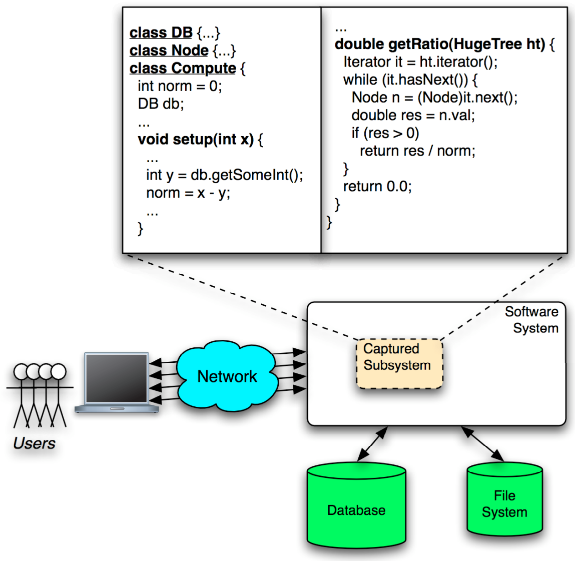
To help address the limitations of purely in-house approaches, we present a technique for capturing and replaying executions of deployed software. Our technique works by (1) letting users specify a subsystem of interest, (2) automatically identifying the boundary between such subsystem and the rest of the application, (3) efficiently capturing at runtime all interactions across this boundary, and (4) replaying the recorded interactions on the subsystem in isolation.

The technique can be useful for many maintenance tasks. In testing, for instance, the ability to capture and replay executions can allow for automatically getting test cases from users. Given a deployed program, we could capture user executions, collect and group them into test suites, and then use such test suites for validating the program in the way it is used. For another example, capture and replay would also allow for performing dynamic analyses that impose a high overhead on the execution time. In this case, we could capture executions of the uninstrumented program and then perform the expensive analyses off-line, while replaying.

State of the Art. Most existing capture-replay techniques and tools (*e.g.*, WinRunner [17]) are defined to be used in-house, typically during testing. These techniques cannot be used in the field, where many additional constraints apply. First, traditional techniques capture complete executions, which may require to record and transfer a huge volume of data for each execution. Second, existing techniques typically capture inputs to an application, which can be difficult and may require ad-hoc mechanisms, depending on the way the application interacts with its environment (*e.g.*, via a network, through a GUI). Third, there are issues related to side effects. If a captured execution has side effects on the system on which it runs, which is typically the case, replaying it may corrupt the system. Furthermore, the environment may have changed between capture and replay time, and there is no guarantee that the system will behave in the same way during replay. Finally, existing techniques are not concerned with efficiency because they are not designed to be used online or on any kind of deployed software. Therefore, these techniques typically impose a huge overhead during capture.

To better illustrate the above issues, Figure 1 shows a software system that will be used as an example in the rest of the paper. The system is a networked, multi-user application that receives inputs from users and performs read and write accesses to both a database and the filesystem. The example is representative of situations in which capturing all of the information required to replay a whole execution would involve technical challenges (*e.g.*, collecting the data that flows from the users to the application and vice versa), storage problems (*e.g.*, the technique may have to record consistent portions of the database), and privacy issues (*e.g.*, the information provided by the users may be confidential). Using this kind of application as an example lets us stress that many systems are complex and operate in a varied and complicated environment. However, the above issues would arise, to different extents, for most applications (*e.g.*, mail clients, word processors, web servers).

[*]We presented an early version of this work at WODA 2005 [10]. In this paper, we extend the technique, describe the SCARPE tool, discuss possible applications of the approach, and present a more extensive empirical study.



```
class DB {...}
class Node {...}
class Compute {
  int norm = 0;
  DB db;
  ...
  void setup(int x) {
    ...
    int y = db.getSomeInt();
    norm = x - y;
    ...
  }
```

```
  ...
  double getRatio(HugeTree ht) {
    Iterator it = ht.iterator();
    while (it.hasNext()) {
      Node n = (Node)it.next();
      double res = n.val;
      if (res > 0)
        return res / norm;
    }
    return 0.0;
  }
}
```

Figure 1. Example application.

Advantages of our Approach. Defining a technique for capture-replay of field executions that accounts for practicality, safety, and (to some extent) privacy issues involves many challenges. Our technique is based on *selective capture and replay of executions*, is specifically designed to be used on deployed software, and addresses the issues with existing capture-replay techniques through a combination of novel technical solutions and careful engineering.

When *practicality* is concerned, our technique allows for limiting the volume of data that we need to record because it lets users select a specific subset of the application for which to capture information. Moreover, it disregards data that, although flowing through the boundary of the subsystem of interest, does not affect its execution, further reducing the amount of data collected. (Intuitively, we capture only the subset of the application's state and environment required to replay the execution on the selected subsystem.) Finally, our technique addresses the problems represented by complex execution environments because it always captures (and replays) at the boundary between parts of the application. Therefore, no custom capture mechanism is required when different types of applications are considered.

When *safety* is concerned, our technique eliminates all side effects because it replays the subsystem in a sandbox—all interactions with the rest of the application and with the environment are only simulated during replay.

When *privacy* is concerned, the use of our technique can help in two ways. First, it allows for excluding from the subsystem of interest parts of the application that may handle confidential information. Second, when this is not possible, our technique can be used to perform the replay on the users' machines instead of retrieving the captured execution and replaying it in-house. For example, if the technique is used to perform expensive dynamic analyses on some part of an application, it could capture executions for that part while users are running the application, replay them on an instrumented version when free cycles are available, and collect only sanitized results of the analysis.

The main contributions of this paper are:

- a detailed definition of our technique (Section 2),
- a discussion of its possible applications (Section 3),
- a description of SCARPE, our publicly available tool that implements the technique (Section 4), and
- an empirical study that provides initial evidence of the feasibility of our approach (Section 5).

2 Capture and Replay Technique

In this section, we define our technique. For space considerations, we limit the discussion to the main technical characteristics of the work. We discuss some additional details separately, in Section 2.4, and provide complete details in a technical report [5].

Because the characteristics of the programming language targeted by the technique considerably affect its definition, we define our technique for a specific language: Java. However, the technique should be generally applicable to other object-oriented languages with similar features.

Terminology. We refer to the selected application subsystem as the *observed set* and to the classes in the observed set as the *observed classes* (or code). *Observed methods* and *observed fields* are methods and fields of observed classes. We define analogously *unobserved set*, *unobserved classes*, *unobserved code*, *unobserved methods*, and *unobserved fields*. The term *external code* indicates unobserved and library code together. The term *unmodifiable classes* denotes classes whose code cannot be modified (*e.g.*, `java.lang.Class`) due to constraints imposed by Java Virtual Machines, and the term *modifiable classes* refers to all other classes.

2.1 Overview of the Approach

Our technique consists of two main phases: capture and replay. Figure 2 informally depicts the two phases. The capture phase takes place while the application is running. Before the application starts, based on the user-provided list of observed classes, the technique identifies the boundaries of the observed set and suitably modifies the application to be able to capture interactions between the observed set and the rest of the system. The application is modified by inserting probes into the code through instrumentation.

When the modified application runs, the probes in the code suitably generate events for the interactions between the observed classes and the external code. The events, together with their attributes, are recorded in an *event log*.

In the replay phase, the technique automatically provides a *replay scaffolding* that inputs the event log produced during capture and replays each event in the log by acting as both a driver and a stub. Replaying an event corresponds to either performing an action on the observed set (*e.g.*, writing

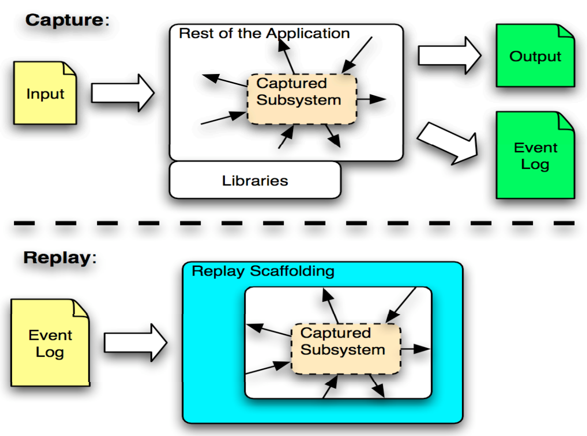


Figure 2. Overview of the technique.

an observed field) or consuming an action from the observed set (*e.g.*, receiving a method invocation originally targeted to external code). We now discuss the two phases in detail.

2.2 Capture Phase

Before discussing the details of this phase, we need to introduce the concept of object ID. In our technique, an *object ID* is a positive numeric ID that uniquely identifies a class instance. To generate such IDs, our technique uses a numeric *global ID* that is initialized to zero when capture starts. For modifiable classes, the object ID is generated by adding a numeric field to the classes and by adding a probe to the classes' constructors. The probe increments the global ID and stores the resulting value in the numeric field of the object being created. Therefore, given an instance of a modifiable class, the technique can retrieve its object ID by simply accessing the ID field in the instance.

For unmodifiable classes, we associate IDs to instances using a *reference map*, which contains information about how to map an object to its ID and is populated incrementally. Every time the technique needs an object ID for an instance of an unmodifiable class, it checks whether there is an entry in the reference map for that instance. If so, it gets from the map the corresponding ID. Otherwise, it increments the global ID and creates a new entry in the map for that instance with the current value of the global ID.

2.2.1 Capturing Partial Information

When capturing data flowing through the boundary of a subsystem (*e.g.*, values assigned to a field), the types of such data range from simple scalar values to complex objects. Whereas capturing scalar values can be done relatively inexpensively, collecting object values is computationally and space expensive. A straightforward approach that captures all values through the system (*e.g.*, by serializing objects passed as parameters) would incur in a tremendous overhead and would render the approach impractical. In preliminary work, we measured several orders of magnitude increases in the execution time for a technique based on object serialization. Our key intuition to address this problem is that (1) we only need to capture the subsets of those objects that affect the computation, and (2) we can conservatively approximate such subset by capturing it incrementally and on demand, without the need for sophisticated static analyses.

Consider a call to method `getRatio` in Figure 1, such that the first node whose value is greater than zero is the fifth node returned by the iterator. For that call, even if `ht` contains millions of nodes, we only need to store the five nodes accessed within the loop. We can push this approach even further: in general, we do not need to capture objects at all. Ultimately, what affects the computation are the scalar values stored in those objects or returned by methods of those objects. Therefore, as long as we can automatically identify and intercept accesses to those values, we can disregard the objects' state. For instance, in the example considered, the only data we need to store to replay the considered call are the boolean values returned by the calls to the iterator's method `hasNext`, which determine the value of the `while` predicate, and the `double` values associated with the five nodes accessed.

Although it is in general not possible to identify in advance which subset of the information being passed to a method is relevant for a given call, we can conservatively approximate such subset by collecting it incrementally. To this end, we leverage our object-ID mechanism to record only minimal information about the objects involved in the computation. When logging data that cross the boundaries of the observed set (*e.g.*, parameters and exceptions), we record the actual value of the data only for scalar values. For objects, we only record their object ID and type. (We need to record the type to be able to recreate the object during replay, as explained in Section 2.3.) With this approach, object IDs, types, and scalar values are the only information required to replay executions, which can dramatically reduce the cost of the capture phase.

2.2.2 Interactions Observed–External Code

Method Calls. The most common way for two parts of an application to interact is through method calls. In our case, we must account for both calls from the unobserved to the observed code (*incall*s) and calls from the observed to the unobserved code (*outcall*s). Note that the technique does not need to record calls among observed methods because such calls occur naturally during replay. Our technique records four kinds of events related to method calls:

OUTCALL events, for calls from observed to unobserved code.
INCALL events, for calls from unobserved to observed code.
OUTCALLRET events, for returns from outcalls.
INCALLRET events, for returns from incalls.

OUTCALL/INCALL events have the following attributes:

Receiver: Fully qualified type and object ID of the receiver object. For static calls, the object ID is set to -1.
Method called: Signature of the method being called.

Parameters: A list of elements, one for each parameter. For scalar parameters, the list contains the actual value of the parameters, whereas for object parameters, the list contains the type of the parameter and the corresponding object ID (or a zero value, if the parameter is `null`).

OUTCALLRET and INCALLRET events have only one attribute: the value returned. Like call parameters, the attribute is the actual value in the case of scalar values, whereas it is the type of the value and the corresponding object ID if an object is returned. To capture OUTCALL events, our technique modifies each observed method by adding a probe before each call to an external method. The signature of the method called is known statically, whereas the receiver's type and object ID and the information about the parameters is generally gathered at runtime.

To capture INCALL and INCALLRET events, our technique performs two steps. *First*, it replaces each public observed method `m` with a proxy method and an actual method. The *actual method* has the same body as `m` (modulo some instrumentation), but has a different signature that takes an additional parameter of a special type. The *proxy method*, conversely, has exactly the same signature as `m`, but a different implementation. The proxy method (1) creates and logs an appropriate *INCALL* event, (2) calls the actual method by specifying the same parameters it received plus the parameter of the special type, (3) collects the value returned by the actual method (if any) and logs an INCALLRET event, and (4) returns to its caller the collected value (if any). In this case, all the information needed to log the events, except for the object ID and the return value, can be computed statically.

Second, it modifies all calls from observed methods to public observed methods by adding the additional parameter of the special type mentioned above. In this way, we are guaranteed that calls that do not cross the boundaries of the observed code invoke the actual (and not the proxy) method and do not log any spurious INCALL or INCALLRET event (these calls and returns occur naturally during replay).

Finally, to capture OUTCALLRET events, our technique again modifies the observed methods: it instruments each call to an external method by adding a probe that stores the value returned by the call (if any) and logs it.

Access to Fields. Interactions between different parts of an application also occur through field accesses. To account for these interactions, our technique records accesses to observed fields from unobserved code and accesses from observed code to unobserved fields and fields of library classes. For accesses from unobserved code to observed fields, we only record write accesses—read accesses do not affect the behavior of the observed classes and are irrelevant for replay. We record three kinds of events for accesses to fields:

OUTREAD events, for read accesses from observed code to unobserved or library fields.

OUTWRITE events, for write accesses from observed code to unobserved or library fields.

INWRITE events, for modifications to an observed field performed by external code.

These three events have the following attributes:

Receiver: Fully qualified type and object ID of the object whose field is being read or modified. As before, value -1 is used in the case of access to a static field.

Field Name: Name of the field being accessed.

Value: Value being either read from or assigned to the field. Also in this case, the value corresponds to the actual values for scalar fields and to an object ID or zero (for `null`) otherwise.

To capture OUTREAD and OUTWRITE events, the technique first analyzes the observed code and identifies all the accesses to fields of external classes. Then, it adds a probe to each identified access: for read accesses, the probe logs an OUTREAD event with the value being read; for write accesses, the probe logs an OUTWRITE event with the value being written. The information about the field name is computed statically and added to the probes, whereas the information about the type and object ID is computed dynamically.

INWRITE events are captured similarly to OUTWRITE events. The only difference is that the technique analyzes the modifiable external classes, instead of the observed ones, and instruments accesses to observed fields.

Exceptions. Exceptions too can cause interactions between different parts of an application. For example, for the code in Figure 1, if the call to `ht.iterator()` in method `getRatio` terminated with an exception, the rest of the code in the method would not be executed. Not reproducing the exception during replay would result in a complete execution of the method, which does not correctly reproduce the recorded behavior. However, there is no point in `getRatio`'s code in which the fact that an exception has occurred is explicit.

To capture interactions that occur due to exceptions, our technique records two types of events: (1) *EXCIN*, for exceptions that propagate from external to observed code; and (2) *EXCOUT*, for exceptions that propagate from observed to external code. EXCIN and EXCOUT events have only one attribute that consists of the type and object ID of the corresponding exception. The mechanism to capture exceptions is fairly complex [5]. Intuitively, our techniques captures exceptions by wrapping relevant methods in `try-catch` block that suitably log exceptions and re-throw them.

2.3 Replay Phase

In the replay phase, our technique first performs two steps analogous in nature to the first two steps of the capture phase: it (1) identifies all the interactions between observed and external code, and (2) suitably instruments the application code. Then, it inputs an event log generated during capture and, for each event, either performs some action on the observed code or consumes some action coming from the observed code.

2.3.1 Object Creation

In Section 2.2, we discussed how our technique associates object IDs to objects during capture. Although we use a global ID and a reference map also during replay, the handling of IDs is different in this case. Unlike the capture phase, which associates IDs to objects flowing across the subsystem boundaries, the replay phase extracts object IDs from the events' attributes and retrieves or creates the corresponding objects. Another difference between the two phases is that, during replay, all object IDs are stored in a reference map.

Instances of External Classes. Every time the technique processes an event whose attributes contain an object ID, it looks for a corresponding entry in the reference map. (The only exception is the case of object IDs with values zero or -1, which correspond to `null` values and static accesses, respectively.) If it finds an entry, it retrieves the object associated with that entry and uses it to reproduce the event. Otherwise, the technique increments the global counter, creates a placeholder object of the appropriate type, and creates a new entry in the map for that instance with the current value of the global ID. A *placeholder object* is an object whose type and identity are meaningful, but whose state (*i.e.*, the actual value of its fields) is irrelevant. We need to preserve objects' identity and type during replay for the execution to be type safe and to support some forms of reflection (*e.g.*, `instanceof`). Our technique uses *placeholder constructors* to build placeholder objects. For modifiable classes, the placeholder constructor is a new constructor added by our technique. The constructor takes a parameter of a special type, to make sure that its signature does not clash with any existing constructor, and contains only one statement—a call to its superclass's placeholder constructor.

For unmodifiable classes, our technique searches for a suitable constructor among the existing constructors for the class. In our current implementation, for simplicity, we hard-code the constructor to be used in these special cases (*e.g.*, `java.lang.Class`), but other approaches could be used.

Instances of Observed Classes. The case of observed classes is simpler. When replaying the incall to a constructor, the technique retrieves the object ID associated with the INCALL event, creates the object by calling the constructor (see Section 2.3.2), and adds an entry to the reference map for that instance and object ID. Note that, because of the way in which we replay events, instances will always be created in the same order. Therefore, we can use object IDs to correctly identify corresponding instances in the capture and replay phases and correctly reproduce events during replay.

2.3.2 Events Replaying

During replay, our technique acts as both a driver and a stub. It provides the scaffolding that mimics the behavior of the external code for executing the observed code in isolation. The replay scaffolding processes the events in the event log and passes the control to the observed code for INCALL, OUTCALLRET, and EXCIN events. When control returns to the scaffolding (*e.g.*, because an incall returns or an exception is thrown), the scaffolding checks whether the event received from the code matches the next event in the log. If so, it reads the following event and continues the replay. Otherwise, it reports the problem and waits for a decision from the user, who can either stop the execution or skip the unmatched event and continue. The case of events that do not match (*out-of-sync events*) can occur only when replaying events on a different version of the observed code than the one used during capture (*e.g.*, if the technique is used for regression testing).

Note that, whereas recording INCALL, INWRITE, OUTCALLRET, and EXCIN events (*incoming events*) is necessary to replay executions, the need for recording the events generated in the observed code(*outgoing events*) depends on the specific use of our technique. For example, if we use the technique to generate unit or subsystem test cases for regression testing, outgoing events are useful because they can be used as oracles. For a different example, if we use the technique to compute def-use coverage off-line, we can disregard outgoing events. For space reason, instead of discussing how our technique replays the various events, we only illustrate the replay of one type of events, OUTCALL events, using the example code in Figure 1. The mechanisms used to replay all other types of events are discussed in detail in [5].

OUTCALL events are consumed by the replay scaffolding. Our technique instruments all observed classes so that each call to external classes is divided into two parts: the invocation of a specific method of the scaffolding (`consumeCall`), whose parameters contain information about the call, and an assignment that stores the value returned by `consumeCall`, if any, in the right variable in the observed code. For example, for the code in Figure 1, statement "`Iterator it = ht.iterator();`" would be replaced by the code (assuming that classes `HugeTree` and `Iterator` are defined in package `foo`):

```
Object tmp = scaffolding.consumeCall(``foo/HugeTree'',
                                     < object ID for ht >,
                                     ``iterator:()Lfoo/Iterator'',
                                     < empty array of paramters >);
Iterator it = (Iterator)tmp;
```

Method `consumeCall` retrieves the next event from the event log and checks whether the event is of type OUTCALL and the parameters match the attributes of the event. If so, the replay continues with the next event. Otherwise, if either the event is of the wrong type or the parameters do not match (*e.g.*, the target of the call differs between capture and replay or a parameter of the outcall does not match the corresponding captured parameter), an error is reported to the user.

2.4 Additional Technical Details

To simplify the presentation, we purposely glossed over some of the technical details of the approach. In this section, we concisely discuss the most relevant ones.

2.4.1 Assumptions.

Our technique works under some assumptions. *First*, we assume that there is no direct access from an unmodifiable class to a field of an observed class. Unmodifiable classes are typically in system libraries, so we expect this assumption to hold in most cases—libraries do not typically know the structure of the application classes. *Second*, because our current implementation does not instrument native code, we also assume that there is no direct access from native code to an observed field. Except for this case, our technique can handle native methods in external code like any other method. *Finally*, we assume that the interleaving due to multi-threading does not affect the behavior of the observed code because our technique does not order "internal events" (*e.g.*, calls between observed methods), which occur naturally during replay.

2.4.2 Special handling of specific language features.

Reflection. Our technique can handle most uses of reflection. However, in some cases (*e.g.*, when reflection is used in external code to modify fields of observed classes), additional instrumentation is required. For instance, to capture reflection-based field access events, the additional instrumentation intercepts calls to "getter" (*e.g.*, `getByte(java.-lang.Object)`) and "setter" (*e.g.*, `setBoolean(java.lang.Object)`) methods of class `java.lang.reflect.Field`.

Arrays. To correctly handle all accesses to arrays, some additional instrumentation is also required. In particular, our technique needs to intercept Java's special instructions for array operations performed on the observed code, such as `BALOAD`, `CALOAD`, and `arraylength`, and capture them as pairs of OUTCALL and OUTCALLRET events.

Garbage Collection. To account for garbage collection, our technique must ensure that it does not keep references to objects that would be otherwise garbage collected. For example, the reference map must use weak references (*i.e.*, references that do not prevent the referenced objects from being garbage collected) to avoid memory leaks.

Finalize. Because calls to `finalize` are nondeterministic in Java, they can generate out-of-sync events during replay. Therefore, we treat calls to `finalize` (and method calls originated within a `finalize` method) in a special way: by not ordering them like the other events and matching them even if out of sync.

3 Possible Applications of our Technique

We discuss three possible applications of our technique: post-mortem dynamic analysis of user executions, debugging of deployed applications, and user-based regression testing.

3.1 Post-mortem Dynamic Analysis of User Executions

This first application involves the use of the technique for selectively capturing user executions and performing various dynamic analyses while replaying these executions. Being able to perform dynamic analysis on the users platforms would provide software producers with unprecedented insight on the way their software is used in the field. A perfect example is the identification of memory-related problems performed by tools like Valgrind [14]. These tools have been used very successfully in-house to identify such problems, but may miss problems that occur only in some specific configuration or for some specific runs. Unfortunately, the overhead imposed by these runtime memory-checking tools is too high for them to be usable on deployed software.

Our technique could be used to capture user executions, store them on the user machines, and rerun them while performing dynamic analysis when free cycles are available (*e.g.*, at night). The only data collected in-house would be the results of the analysis, possibly further sanitized, which would help address privacy issues and also limit the amount of data to be transferred over the network. Additionally, if capturing all executions would still result in too much data being collected, some criterion could be used to decide which executions to collect. There are several criteria that could be used to this end, some of which may depend on the specific kind of analysis the software producer is interested in performing. For example, one criterion could be to collect only executions that terminate with an exception and discard normally-terminating runs.

3.2 Debugging of Deployed Applications

Consider again the example in Figure 1, which contains the following fault: if (1) the integer passed to `setup` has the same value as the integer returned by the call to `db.getSomeInt` within `setup`, (2) the value of field `norm` is not redefined, (3) method `getRatio` is called, and (4) predicate "`res > 0`" evaluates to true at least once, then the application generates a division by zero and fails. An execution that terminates with such failure could be arbitrarily long and involve a number of interactions between users, application, and database/filesystem. However, capturing only the execution of class `Compute` would provide enough information to locate and remove the fault.

In situations like this one, our technique could be applied to capture executions that can reproduce a failure in the field and send them to the software developer. As a possible scenario for this application, consider a program that is deployed in a number of instances, and for each instance a different subsystem is captured (*e.g.*, by partitioning the program in subsystems and assigning the capturing of each subsystem to one or more user sites). When a failure occurs at a given site, in a subsystem that is being captured at that site, the corresponding execution is saved for later debugging.

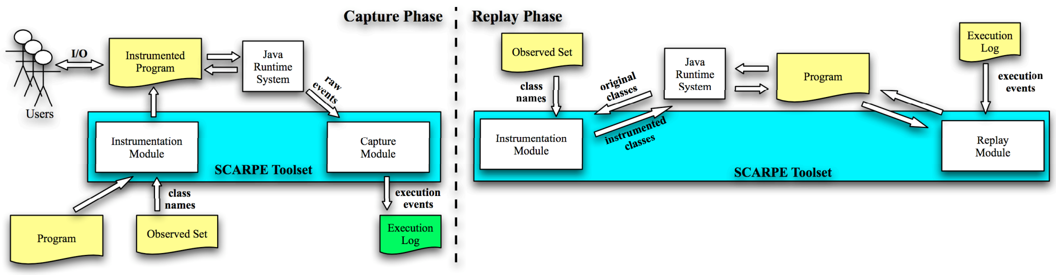


Figure 3. SCARPE **during capture (left) and replay (right).**

At this point, one option is to send the captured execution to the software producer, who could use it for traditional debugging. Being able to debug failing executions collected in the field would be extremely useful in itself. Alternatively, we could perform (semi)automated debugging remotely, on the site(s) where the application fail. To this end, our capture-replay technique could apply an automated debugging technique to executions captured in the field and send back to the software producer minimized executions, as we did in [2].

3.3 User-based Regression Testing

Regression testing is performed on a modified version of a program to provide confidence that the changed parts behave as intended and the unchanged parts are not adversely affected by the modifications. A typical way to perform regression testing is to keep a *regression test suite* and to rerun all or part of it on the changed program. The effectiveness of regression testing highly depends on how well the regression test suite represents the way the program is used in the field. The problem of unrepresentativeness of a test suite can actually be more serious for regression testing than for testing of new applications: the release of a new software version is typically targeted to users who are already familiar with the application. Differences in behavior between the old and the new versions are likely to generate more user dissatisfaction than, for instance, the lack of a feature in a new product.

Unfortunately, regression test suites often exercise the application in a very different way than the actual users, as shown by some of our previous work [9]. Our capture-replay technique could address this problem by generating regression subsystem and unit test cases from complete, real user executions captured in the field. These test cases would have the advantage of testing the software exactly in the way it is used on the field. These subsystem and unit test cases could be collected, for example, in the form of JUnit test cases, and could be used to test new versions of such subsystems and units. Test cases could be captured in the field and sent back to the software producer, where they are collected and used as a regular regression test suite. Test cases could also be stored on the users' machine and be run on the new version of the program remotely, with only their outcome collected at the software producer's site. Like for the previous application, this second approach would allow for (1) eliminating privacy issues, and (2) reducing the amount of information to be transfered over the network.

4 The Tool: SCARPE

We implemented our technique in a tool called SCARPE (Selective Capture And Replay of Program Executions), which is written in Java and consists of three main modules. The **Instrumentation Module** adds probes to the program being captured or replayed. It instruments at the bytecode level, using the Byte Code Engineering Library (BCEL – `http://jakarta.apache.org/bcel/`). The **Capture Module** is invoked at runtime by the probes added by the instrumentation module. It uses the information provided by the probes and suitably generates execution events. The **Replay Module** produces events for the observed code according to the execution events in the execution log. It also consumes events produced by the observed code.

Figure 3 provides a high-level view of how the tool works during capture and replay. During capture, SCARPE's instrumentation module inputs the program to be captured and the observed set, instruments the classes in the observed set and the classes that access fields of such classes, and produces an instrumented version of the program that is ready to be deployed. (SCARPE can also instrument classes on the fly; we do not discuss this functionality here for brevity.) While users interact with the program, the instrumentation probes send *raw events* to the capture module. Raw events contain enough information for the module to build the actual *execution events* that are then recorded into an execution log.

During replay, the replay module inputs an execution log and suitably reproduces and consumes execution events, as described in Section 2.1. It is worth noting that SCARPE needs to instrument the program also during replay, to ensure that the program operates in a sandbox. To this end, all interactions between the observed and the external code are transformed, through bytecode rewriting, into interactions between the observed code and the replay module. The replay module acts as the replay scaffolding, that is, it mimics the behavior of the external code.

5 Empirical Evaluation

To assess the feasibility and the efficiency of our approach, we performed two preliminary empirical studies using SCARPE on two software subjects. In the studies, we investigated two research questions:

RQ1 (feasibility): Can our technique correctly capture and replay different subsets of an application?

RQ2 (efficiency): Can our technique capture executions without imposing too much overhead on the executions?

5.1 Study 1: RQ1 – Feasibility

The goal of this study is to assess how reliable is our technique in capturing and replaying partial executions. To achieve this goal, we selected a software subject and used SCARPE to capture and replay a large number of executions for different subsets of the application. As a subject for this study, we used NANOXML, an XML parser that consists of about 3,500 LOC and 19 classes. We obtained NANOXML, along with a test suite of 216 test cases, from the Subject Infrastructure Repository (`http://cse.unl.edu/~galileo/php/sir/`). We performed the study in two parts. In the first part, we captured executions for subsystems containing only one class. For each class c in the application, we defined an observed sets consisting of c only and ran all test cases in the test suite using SCARPE. In this way, we recorded 216 event logs (one for each test case in the test suite) for each of the 19 classes in the application, for a total of more than 4,000 logs. We then replayed, for each class, all of the recorded executions for that class.

In the second part, we captured executions for observed sets of sizes two, three, four, and five. We randomly created 25 observed sets for each of the sizes considered, so as to have a total number of 100 observed sets. Then, analogously to the first part of the study, we ran all of NANOXML's test cases for each of the observed sets and later replayed the so collected execution logs. Therefore, overall, we recorded and replayed more than 20,000 partial executions.

The study was successful, in that all executions were correctly captured and replayed. We checked the correctness of the replay by both making sure that all of the events generated by the observed set were matching the logged events and spot checking some of the executions. Although this is just a feasibility study, the successful capture and replay of about 25,000 executions is a promising result.

5.2 Study 2: RQ2 – Efficiency

The goal of Study 2 is to assess the efficiency of our approach. Although NANOXML is an appropriate subject for Study 1, because its size let us check and verify SCARPE's results, for this second study we used a larger and more realistic subject: JABA. JABA (Java Architecture for Bytecode Analysis) is a framework for analyzing Java bytecode that performs complex control-flow and data-flow analyses and consists of about 60,000 lines of code and 400 classes. JABA has an extensive regression test suite that was created and used over the last several years of the system's evolution. Because JABA is an analysis library, each test case consists of a driver that uses JABA to perform one or more analyses on an input program. The test suite that we used for the study contains 4 drivers and 100 input programs, for a total of 400 test cases.

To measure the efficiency of our technique, we proceeded as follows. *First*, we ran all 400 test cases and collected the normal execution time for each test case. Because SCARPE's initialization code takes around 1.5 seconds to run, we only considered executions that take more than a second to run (all but 60 executions). We could have considered also shorter executions, but we believe that an overhead dominated by a fixed cost, when the cost is in the order of a second, does not provide useful information. *Second*, for each driver, we identified nine classes in JABA that were covered by the test cases involving that driver. *Third*, for each driver and each class c considered, we defined c as the observed set and ran all 340 $(400 - 60)$ test cases using SCARPE. In this way we captured 3060 $(340 * 9)$ executions of JABA test drivers. `Fourth`, we measured the overhead imposed by SCARPE in terms of percentage increase between the execution times computed with and without SCARPE.

Table 1. Overhead results for the four drivers.

	ACDGDriver	CFGDriver	DefUseDriver	ICFGDrover
Min Ovh	6%	**3**%	4%	4%
Max Ovh	494%	306%	**877**%	501%
Avg Ovh	104%	72%	145%	69%

Together with the timing data, we also collected the number of events captured for each execution, which is simply the number of entries in the execution log for that execution. We performed all experiments on a dedicated Pentium III, with 2GB of memory, running the GNU/Linux Operating System (2.6.16). We repeated all measures 10 times and averaged the results to limit the risk of imprecision introduced by external effects, such as caching.

Results and Discussion. Table 1 shows a summary of the overhead measured in the study. For each driver (ACDGDriver, CFGDriver, DefUseDriver, and ICFGDriver) the table shows the minimum, maximum, and average percentage overhead (computed over all of the test cases for that driver). The absolute minimum and maximum are highlighted using a boldface font.

To provide a better idea of the distribution of the results we also report, in Figure 4, the overhead information for each captured class (for each of the four drivers). Also in this case, the figure shows average, maximum, and minimum overhead (measured, this time, over the executions involving a specific captured class and driver). As far as space overhead is concerned, the average size of the execution logs collected, in uncompressed and unoptimized format, is in the order of 50KB per 1000 events (i.e., about 60MB for our largest log, consisting of 114,953,200 events).

As the results show, SCARPE's overhead varies considerably across drivers and observed classes, ranging from 3% to 877%. A careful examination of the raw data collected in the study revealed that the cost of collecting a single event is similar across different types of events, and that the overhead is by and large proportional to the number of events collected per unit of time. For example, for driver ACDGDriver, the execution with 6% overhead generates 470 events, takes 22

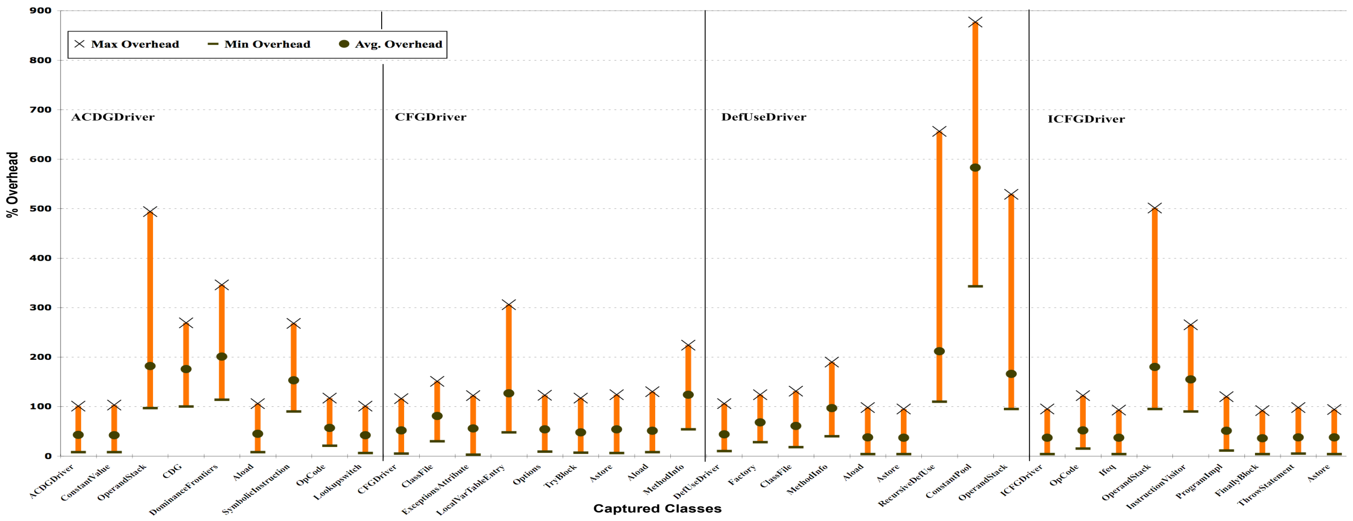


Figure 4. Performance results for SCARPE.

seconds for the uninstrumented code, and takes about 23 seconds for the instrumented code. Considering the 1.5s fixed cost imposed by SCARPE, we can see that the overhead due to the 470 events is mostly negligible. For another example, for the same driver, the execution with the highest overhead (494%) produces 3,528,210 events, takes about 9 seconds during normal execution, and takes about 53 seconds when the code is instrumented. In this case, the high number of events per second is the cause of the considerable overhead. (The execution that causes 877% overhead has similar characteristics to this one). For yet another example, the lowest overhead for driver CFGDriver (3%) corresponds to an execution that produces 114,458 events, which is a fairly large number of events. However, in this case, the execution takes about 87 seconds, that is, the number of events per second is two orders of magnitude lower than for the previous example.

Looking at the results in Figure 4, we can also observe that there are many cases in which the overhead imposed by SCARPE is on average between 30% and 50%, in the worst case around 100%, and in the best case in the single digits. Although 100% overhead is likely to be problematic in many cases, we must consider that JABA is a processing intensive applications with no interaction with the user. There may be cases in which even a 100% overhead is not relevant for an application that runs completely in batch mode (*e.g.*, overnight). More importantly, we hypothesize that the same overhead may become acceptable for interactive applications, such as word processors or Web browsers. For these applications, the user"thinking time" is likely to decrease considerably the amount of events per second produced and also provide free cycles that SCARPE could leverage.

To get some initial evidence that could support our hypothesis, we performed an informal study in which both authors used a graph-drawing application while the execution of different subsets of the drawing classes was being captured. Although we cannot claim any generality of the result, and our assessment may be biased, we can report that the slowdown in the drawing operations that involved the instrumented class was barely noticeable, despite the fact that a large number of events was being captured.

In summary, we believe that our results are encouraging and show that the approach we propose can be feasible, especially considering that the optimization of SCARPE is still ongoing, and the performance of the tool can be further improved. Nevertheless, it is likely that there will be cases where the current technique is too expensive to be used in the field. To account for these cases, we are currently investigating variation of the approach in which the capture stops after a given number of events or when the frequency of events produced (measured using a buffer that is flushed at regular intervals) is too high. We are also investigating ways in which preliminary information, collected in-house, could be used to identify problematic part of an application (in terms of overhead) and exclude them from the capture.

6 Related Work

The technique that is most related to ours is JRAPTURE, by Steven and colleagues [15], which captures and replays executions of Java programs. Although able to completely capture and replay executions, this technique incurs in many of the problems that we mention in the Introduction because it captures complete information for each execution. Moreover, JRAPTURE requires two customized versions of the Java API for each Java version targeted.

More recently, Saff and Ernst [13] and Elbaum and colleagues [3] presented two techniques for deriving unit tests from system tests to speed up regression testing. These techniques, designed to be used in-house, would impose unnecessary overhead if used in the field. The first technique instruments all classes and, for each method call, stores all parameters and return values. The second technique captures executions at the method level, and each run results in a large number of independent method-level test cases, each including a partial dump of the program state. Although effective in-house, these techniques would be difficult to use for our goals due to the time and space overhead they impose.

Other related techniques perform record and replay for testing and debugging (*e.g.*, [6, 8, 17]). Also in this case, these techniques are designed to be used in-house, and the overhead they impose (on space, time, or infrastructure required) is reasonable for their intended use, but would make them impractical for use on deployed software. In particular, commercial capture-replay tools typically focus on a specific domain (*e.g.*, GUI-based applications) and require a complex infrastructure and setup to be used. The few record and replay techniques that may be efficient enough to be used in the field (*e.g.*, [6]) require a specialized operating-system or hardware support, which considerably limits their applicability in the short term.

Other related techniques aim to reproduce the concurrent behavior of applications (*e.g.*, [1, 7, 12, 16]). Although related, these techniques have different goals and, thus, a different set of constraints and tradeoffs. They focus on reproducing, giving the same inputs, the same application behavior in terms of concurrency-related events, have no need to store input and output values, and do not have efficiency constraints (being targeted to debugging). Our technique is mostly concerned with automatically capturing and replaying subsystems and has efficiency as a priority because we want the technique to be usable also on deployed software.

A general difference between our technique and many other existing techniques is that it works at the application level, through bytecode rewriting. Because our technique does not require any runtime-system or library support, it can be used on any platform that provides a standard implementation of the Java Virtual Machine, which improves portability and enables the use of the technique on users' platforms.

7 Conclusion

We have described a technique for partial capture-replay of user executions that allows for capturing executions for a given subsystem and replaying them in a sandbox. We have also discussed three possible applications of the technique and presented a tool, SCARPE, that implements the technique and is freely available (`http://www.cc.gatech.edu/~orso/software.html`). Finally, we have presented an initial empirical evaluation showing that, although there is room for improvement, the approach is feasible.

Our immediate goal in future work is to improve the implementation of SCARPE in terms of performance. This improvement will be performed in parallel with additional experimentation on more subjects and executions, to further refine and extend the approach.

Other future work involves investigating the three applications proposed in the paper. We have already started investigating the use of capture-replay techniques for debugging of deployed applications and got promising results [2].

Acknowledgments

This work was supported in part by NSF awards CCF-0541080 and CCR-0205422 to GA Tech. Bryan Kennedy helped implementing a first prototype of SCARPE. Gregg Rothermel provided NANOXML and associated test cases.

References

[1] B. Alpern, T. Ngo, J.-D. Choi, and M. Sridharan. Dejavu: Deterministic java replay debugger for jalapeño java virtual machine. In *Proc. of the Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 165–166, 2000.

[2] J. Clause and A. Orso. A Technique for Enabling and Supporting Debugging of Field Failures. In *Proc. of the 29th Intl. Conf. on Software Engineering*, pages 261–270, May 2007.

[3] S. Elbaum, H. N. Chin, M. Dwyer, and J. Dokulil. Carving Differential Unit Test Cases from System Test Cases. In *Proc. of the 14th Symposium on the Foundations of Software Engineering*, Nov. 2006.

[4] S. Elbaum and M. Diep. Profiling Deployed Software: Assessing Strategies and Testing Opportunities. *IEEE Transactions on Software Engineering*, 31(4):312–327, 2005.

[5] S. Joshi and A. Orso. Capture and Replay of User Executions and its Applications. Technical Report 2006-04-14, Georgia Tech – College of Computing, Apr. 2006. `http://www.cc.gatech.edu/~orso/papers/abstracts.html#joshi06apr-tr`.

[6] S. King, G. Dunlap, and P. Chen. Debugging operating systems with time-traveling virtual machines. In *Proc. of the Usenix Annual Technical Conf.*, pages 1–15, Apr. 2005.

[7] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed java applications. In *In Proc. of the 14th Intl. Parallel & Distributed Processing Symposium*, pages 219–228, 2000.

[8] Mercury LoadRunner, 2006. `http://www.mercury.com/us/products/performance-center/loadrunner/`.

[9] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proc. of the 9th European Software Engineering Conf. and 10th Symposium on the Foundations of Software Engineering*, pages 128–137, Sep. 2003.

[10] A. Orso and B. Kennedy. Selective Capture and Replay of Program Executions. In *Proc. of the Third Intl. ICSE Workshop on Dynamic Analysis*, pages 29–35, May 2005.

[11] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proc. of the 21st Intl. Conf. on Software Engineering*, pages 277–284, May 1999.

[12] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proc. of Conf. on Programming Languages and Implementation*, pages 258–266, 1996.

[13] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic Test Factoring for Java. In *Proc. of the 20th Intl. Conf. on Automated Software Engineering*, pages 114–123, Nov. 2005.

[14] J. Seward. Valgrind, an Open-Source Memory Debugger for x86-Gnu/Linux, 2002. `http://valgrind.kde.org/`.

[15] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jrapture: A capture/replay tool for observation-based testing. In *Proc. of the Intl. Symposium on Software Testing and Analysis*, pages 158–167, Aug. 2000.

[16] K. C. Tai, R. H. Carver, and E. E. Obaid. Debugging concurrent ada programs by deterministic execution. *IEEE Transactions on Software Engineering*, 17(1):45–63, 1991.

[17] Mercury WinRunner, 2006. `http://www.mercury.com/us/products/quality-center/functional-testing/winrunner`.