Efficient Checkpointing of Java Software Using Context-Sensitive Capture and Replay[*]

Guoqing Xu Atanas Rountev Yan Tang Feng Qin
Department of Computer Science and Engineering
Ohio State University
{xug,rountev,tangya,qin}@cse.ohio-state.edu

ABSTRACT

Checkpointing and replaying is an attractive technique that has been used widely at the operating/runtime system level to provide fault tolerance. Applying such a technique at the application level can benefit a range of software engineering tasks such as testing of long-running programs, automated debugging, and dynamic slicing. We propose a checkpointing/replaying technique for Java that operates purely at the language level, without the need for JVM-level or OS-level support. At the core of our approach are static analyses that select, at certain program points, a safe subset of the program state to capture and replay. Irrelevant statements before the checkpoint are eliminated using control-dependence-based slicing; the remaining statements together with the captured run-time values are used to indirectly recreate the call stack of the original program at the checkpoint. At the checkpoint itself and at certain subsequent program points, the replaying version restores parts of the program state that are necessary for execution of the surrounding method. Our experimental studies indicate that the proposed static and dynamic analyses have the potential to reduce significantly the execution time for replaying, with low run-time overhead for checkpointing.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—*Program Analysis*

General Terms

Algorithms, Languages

Keywords

Checkpoint, replay

[*]This material is based upon work supported by the National Science Foundation under grant CCF-0546040.

1. INTRODUCTION

Checkpointing/replaying is a well-known technique which can replay a program execution from an intermediate point which was captured at a *checkpoint*. Originally developed to support fault tolerance in distributed computing, this approach has also been used to facilitate debugging of operating systems (e.g., [9, 13]) and of parallel and distributed software (e.g., [21]).

In this paper we are interested in replaying a previous execution of a program. In this earlier *capturing* execution, program state is recorded at the checkpoint. During the replaying execution, starting from the checkpoint, the run-time behavior is the same as the behavior of the capturing execution. Such functionality can benefit a number of software engineering tasks. For example, in *software testing*, one could perform checkpointing at the boundaries of components of interest during a system test, and use the results for defining unit tests and for testing of evolving software; several authors have proposed techniques based on this idea [20, 27, 19, 10]. As another example, checkpointing and replaying can reduce the cost of *dynamic slicing* [14] of long-running programs. Existing evidence [35] indicates that a fault is usually located within a short dependence distance from the point where it manifests itself. With regular checkpointing for a long-running program, once a failure occurs, one could roll back to the latest checkpoint, instrument only between the checkpoint and the manifestation point, and slice this part of the execution. If the fault is not found, the process could move back to the previous checkpoint, and slice only the interval between the two checkpoints, etc.

In addition to replaying precisely the captured execution, our technique can be easily adapted for producing variations of the captured behavior, for the purposes of automated debugging or regression testing. For example, *delta debugging* [33, 34, 7] requires comparing program states of a passing run and a failing run, and reexecuting the passing run numerous times, with values of some variables replaced with the corresponding values in the failing run, in order to automatically locate the infected transitions. For large programs, however, it may be prohibitively expensive for the delta-debugging algorithm to work on the entire execution. One could functionally partition the execution of the program, and take a checkpoint at the end of each partition. Delta-debugging could then be applied partition-by-partition. When a partition is rerun, program state will be restored at its closest preceding checkpoint. Therefore, the partition of interest can be executed efficiently multiple times without having to run the prior partitions.

Open problems. Mainstream research on checkpointing is typically focused on operating/runtime system support for C programs [22, 30, 28, 18]. Due to the emergence of extremely large Java programs (such as web and application servers), applying such techniques to Java can provide important benefits for various software engineering tasks. One possible checkpointing technique is to record all live objects inside the JVM to disk, and then load them back to memory during the replaying phase. This implementation can be unacceptably expensive — recording and loading could be even more costly than running the program. Furthermore, this approach requires modifications to the JVM, which creates numerous obstacles for real-world deployment of this technology. User-driven language-level checkpointing for Java has been proposed in [15], requiring classes to implement a *Checkpointable* interface, and to provide a *record* method to perform the actual recording. However, in practice, programmers could be reluctant to write such a non-trivial method for each class. Furthermore, this approach does not work for preexisting library classes.

Capture and replay techniques (e.g., [20, 19]) can be used for recording and restoring a set of interactions between a subsystem and the rest of the application. This work considers the interactions between components (i.e., *spatial partitions* of the program execution). The focus of our work are interactions between the program states before and after the checkpoint (i.e., *temporal partitions* of program execution).

Summary of our approach. We propose a checkpointing/replaying technique for Java that operates purely at the language level, without the need for any JVM-level or OS-level support. Unlike system-level checkpointing which records the entire program state (including, for example, program counter, call stack, etc.) and then restores this state during replay, our technique achieves checkpointing and replaying entirely through instrumentation of the program code. Given a single-threaded Java program and a checkpoint specified by the user, we employ several static analyses to compute a set of *control-decision-making points* (CDMPs), insert instrumentation code at these points, and output the bytecode for two versions of the program: a *checkpointing version* and a *replaying version*. The instrumentation in the checkpointing version records relevant run-time information, while the instrumentation in the replaying version uses this information to replay the execution.

At each CDMP before the checkpoint, the checkpointing version captures a minimum set of run-time values that contribute to making the control-flow decision at that CDMP. At the checkpoint itself, the instrumentation captures a set of local variables, static fields, and heap object graphs that could potentially affect the subsequent execution, similarly to [10]. After the checkpoint, our approach also captures parts of the state at certain call sites whose earlier execution affected the flow of control leading to the checkpoint.

The capture operations at CDMPs before the checkpoint produce run-time values that are used in the replaying version to recreate, indirectly, the run-time call stack of the original program at the checkpoint, without directly manipulating the call stack or the program counter. To achieve this, we perform backward slicing in the original program solely based on control-flow dependencies. The code generation for the replaying version eliminates all statements except for CDMPs in the slice. As a result, when the replaying version runs, the checkpoint can be reached quickly, as if the execution directly started from it. The replaying algorithm restores the relevant captured values at each CDMP to force the execution to take the correct control flow.

At the checkpoint, the replaying version restores the subset of the program state that could affect the subsequent execution in the method containing the checkpoint. After the checkpoint, additional "restore" operations are performed after returns at call sites that (directly or transitively) call the method containing the checkpoint, in order to recover additional parts of the program state that are necessary for execution of the surrounding method.

To achieve efficiency, we employ static analyses that identify, at each instrumentation point, the subset of the program state that should be captured/replayed for the execution of the rest of the method that contains the point. The technique is safe because when the execution reaches the checkpoint and any subsequent instrumentation point, all values that will be used afterward are correctly restored.

Our approach is *context-sensitive* because it allows a user to specify an "interesting" call chain that leads to the method that contains the checkpoint; only the corresponding run-time instances of the checkpoint are used for capture and replay. We define a pattern language for describing the call chain; in this manner, the user can define calling-context-sensitive checkpoints. Methods in the call chain are replicated to guarantee that the instrumentation in these methods does not influence their invocations from other contexts (i.e., from methods not in the chain).

We generalize the approach to support taking checkpoints for *multiple execution regions* in the program. The approach has been implemented in our JCP (*J*ava *C*heck*p*ointing) framework, based on the Soot analysis toolset [31]. We performed an experimental evaluation of the technique on a set of Java programs. Our preliminary results indicate that (1) the analyses efficiently generate instrumentation and select a small subset of the state, and can scale to large Java applications such as Soot itself; (2) the checkpointing version introduces low run-time overhead (e.g., on average 1.8% for six different runs of Soot); and (3) the replaying version has the potential to significantly reduce the execution time of long-running programs.

Contributions. The main contributions of this work are:

- A checkpointing/replaying technique based on static analyses that determine program points at which capture/replay should occur, identify a safe subset of the program state at these points, and generate the checkpointing version and replaying version by slicing and instrumenting the original program.
- A generalization to support checkpoints at multiple execution regions, and an optimization technique based on call chain merging for these regions.
- A checkpointing/replaying framework JCP.
- An experimental study of the static analyses and the running times of the checkpointing version and the replaying version. These initial results indicate that our approach should be investigated further as a promising candidate for efficient checkpointing and replaying.

2. EXAMPLE AND DEFINITIONS

Running example. We will use the Soot analysis framework [31] to illustrate our technique. The code in Figure 1 is extracted from classes *soot.Main* and *soot.PackManager*.

```
1  class G { ...
2     static G instance = new G();
3     Options op;
4     static G v() { return instance; }
5     Options soot_options_Options() {
6        if (op == null) op = new Options();
7        return op;
8     }
9  }
10 class Options { ...
11    static Options v() {
12       G g = G.v();
13       return g.soot_options_Options();
14    }
15    boolean parse(String[] args) { ... }
16 }
17 class Main { ...
18    void processCmdLine(String[] args) {
19       ... Options.v().parse(args) ...
20    }
21    void run(String[] args) {
22       processCmdLine(args);                        // phase 1
23       loadNecessaryClasses();                      // phase 2
24       Set wp_packs = getWpacks();
25       Set body_packs = getBpacks();
26       if (Options.v().whole_jimple()) {            // phase 3
27           getPack("cg").apply();
28           // --- checkpoint ---
29           getPack("wjtp").apply();
30           getPack("wjop").apply();
31           getPack("wjap").apply();
32       }
33       retrieveAllBodies();                         // phase 4
34       for (Iterator i = body_packs.iterator();
35            i.hasNext();) {                         // phase 5
36            String s = (String)body_packs.next();
37            getPack(s).apply();
38       }
39       ...
40    }
41    static void main(String[] args) {
42       Main m = new Main();
43       if (args.length !=0)
44           m.run(args);
45    }
46 }
```

Figure 1: Soot startup example.

Soot is a popular program analysis toolset for Java that contains a large number of static analyses which can be used for a variety of compiler optimization and software engineering tasks. Starting from *main*, Soot parses the command line (phase 1), resolves the necessary classes loaded during JVM bootstrapping (phase 2), optionally runs whole-program packs (phase 3), retrieves all method bodies (phase 4), and runs body packs, each one of which performs a specific intraprocedural analysis on the body of every loaded method (phase 5). Whole-program analyses are usually time consuming, especially in the presence of large libraries. For example, running a points-to analysis in the call graph pack, invoked by line 27, typically takes more than half an hour for large programs (including the time to read bytecode from disk and to build the intermediate representation). If user-defined body packs invoked at line 37 need the results of this analysis (e.g., points-to sets or a precise call graph), they have to wait until all whole-program packs finish.

Because user-defined body packs are dependent on the results produced by time-consuming preceding computations, the complexity of testing and debugging of these packs dramatically increases. Suppose one would like to take a checkpoint immediately after the execution of the call graph pack (at line 28), so that when the program is rerun, the execution can skip the points-to analysis and quickly flow to the pack of interest. Using this specific example, we will show how such checkpointing and replaying can be achieved.

Definition 1. A *crosscutting call chain* (CC-chain) is a user-specified call chain that leads to the method that contains the checkpoint; this method will be referred to as the *checkpoint container* (CP container). A CC-chain can be specified by users using a pattern language (described later).

▶ *Example.* Given the checkpoint in Figure 1, the corresponding CC-chain is *main(44)*→ *run(28)*, where *(44)* specifies the call site in *main* that calls *run*, and *(28)* specifies the checkpoint. The CP container is method *run*. ◀

Definition 2. A *pre-X region*, where X is either a call site in the CC-chain or the checkpoint itself, includes all statements in the method containing X that could potentially be executed before X during one run of the method. This region contains all and only control-flow graph (CFG) nodes n such that X is reachable from n in the method's CFG. Similarly, the *post-X region* includes all statements in the method containing X that could potentially be executed after X during one run of the method. The post-X region contains all and only CFG nodes reachable from X.

▶ *Example.* The pre-44 region in *main* includes the statements at lines 42 and 43. The pre-28 region (i.e., pre-checkpoint region) in *run* includes all statements before line 28. The post-44 region is empty, and the post-28 region includes all statements after line 28. If the checkpoint were between lines 36 and 37, the pre-checkpoint region would include all statements before line 38, and the post-checkpoint region would include all statements after line 34. ◀

Definition 3. A *control decision making point* (CDMP) is either a call site in the CC-chain or a predicate. A predicate CDMP is such that either the checkpoint or some call site on the CC-chain is (directly or transitively) control-dependent on that predicate. Intuitively, CDMPs are the only program points that can affect the control flow leading to the checkpoint under the calling context specified by the CC-chain. For example, the CDMPs for Figure 1 are {26, 43, 44}.

3. STATIC ANALYSES

The complication in performing checkpointing/replaying at the language level lies in the inability to manipulate the complete program state at the checkpoint (e.g., the program counter and the call stack). This makes it impossible to resume the execution directly from the checkpoint during the replaying phase. An alternative approach is to generate a replaying version of the program by removing statements before the checkpoint, so that when one runs this version, the execution can quickly reach the checkpoint as if the execution was directly resumed from the checkpoint. Thus, the first problem we face is how to prune the execution *before the checkpoint* in order to reach that checkpoint correctly and efficiently. The second problem, of course, is how to recover enough state *at the checkpoint* so that the subsequent execution proceeds correctly.

To solve the first problem from above, we remove all computation before the checkpoint, while keeping the control flow unchanged. Thus, we preserve only the CDMPs, due to the following reasons. First, if a call site in CC-chain is removed, the CP container will not be called. Essentially, we need to preserve the call sites in the CC-chain in order to recreate the run-time call stack. Second, if a predicate that directly or transitively guards a call site in the CC-chain (or guards the checkpoint itself) is removed, the con-

```
main(String[] args) {
 //@replay
 if(args.length != 0) {
     //@replay
     m.run(args);
}
```

```
run(String[] args) {
  //@replay
  if(Options.v().whole_jimple()) {
      //@replay
      //everything unchanged after here
       getPack("wjtp").apply();
       getPack("wjop").apply();
       getPack("wjap").apply();
  }
  retrieveAllBodies();
  for (...) { ... }
}
```

Figure 2: Replaying version for Figure 1.

trol flow could be changed. For example, if the checkpoint is contained in a loop, and we were to remove the loop predicate, the loop would iterate only once. By preserving the loop predicate code in the replaying version, and using the sequence of run-time predicate values recorded during the capturing execution, we can reproduce precisely all run-time instances of the checkpoint.[1] Strictly speaking, predicates outside of loops do not need to be captured and replayed; however, we chose to preserve them because this simplifies code generation for complex CFGs.

Given a program and a user-defined CC-chain, our static analysis computes the CDMPs using a reverse dominance frontier algorithm [8] and generates the replaying version by essentially performing interprocedural control-dependence-based backward slicing from the checkpoint, and preserving only CDMPs from the slice. Figure 2 shows the resulting code, with all irrelevant statements removed. However, running this code will fail because of uninitialized variables, such as *args* and *m*. Relevant values from the capturing run should be recovered at CDMPs, shown by *//@replay* in the figure, in order to make the appropriate control decisions.

Our tool generates a checkpointing version of the program by instrumenting the original program at the CDMPs to capture the values of variables that are required to be restored during the replaying phase. A key question becomes *what variables have to be recorded/restored at each CDMP and the checkpoint so that the replaying version can be correctly executed?* The checkpointing version can write these variable values to disk at a CDMP, so that the replaying version can read them at the same execution point in the order that they are recorded. Our goals are (1) to select a small subset of variables to record and restore in order to reduce the disk I/O overhead, similarly to [10], and (2) to make the execution of the replaying version behave the same as the execution of the original version after the checkpoint.

3.1 Environments for Instrumentation Points

We will refer to the variables that need to be captured or replayed at a certain program point as the *environment* for that point. For a predicate, the environment contains only the boolean condition variable. The execution can correctly make a control decision when the variable is restored. For a call site in the CC-chain, the environment contains the run-time type of the receiver object if this is an instance method call, or nothing if this is a static method call. During the replaying version, before the call site is reached, we instantiate the recorded type (by using *sun.misc.Unsafe*), and pass default values as actual parameters at the call site (e.g., null for reference types, and 0 for primitive types), so that the correct target method can be invoked.

The most complex case occurs when computing the environment for the checkpoint itself. Because all statements after the checkpoint in the original version are also in the replaying version, we have to record and restore a sufficent subset of the program state at the checkpoint to ensure that every subsequent statement can be executed correctly. There are three kinds of memory locations that constitute the environment for the checkpoint: local variables (including formal parameters), the static fields in all classes currently loaded in the JVM, and objects allocated on the heap. The naive approach of recording all these memory locations is infeasible. Instead, we impose the following restrictions in the selection, in order to reduce the size of the recorded state.

First, we select a local variable only if it is written in the pre-checkpoint region and read in the post-checkpoint region. Formal parameters are considered to be written at the beginning of the method. Capturing a local variable of primitive type simply records the variable's value. For variables of reference types, the entire object graph reachable from the variable is captured, as discussed in Section 3.2.

► *Example.* For the checkpoint in Figure 1 there are thirteen local variables: formal *args*, declared locals *wp_packs*, *body_packs*, *i*, *s*, as well as compiler-generated locals for intermediate results. However, only *body_packs* should be recorded because this is the only variable that is written in the pre-checkpoint region and will still be used in the post-checkpoint region. ◄

We use the same idea to select static fields. However, it is necessary to interprocedurally inspect all methods in the CC-chain to determine which static fields should be selected. The algorithm in Figure 3 is used for this selection. Before running the algorithm, a conservative whole-program Mod/Use analysis is executed to compute, for each method, all static fields that it could potentially read and write. For computing the Mod effects, we classify a static field as "written" if either its value is directly changed, or any heap object that is (directly or transitively) reachable from it is mutated. For Use effects, a static field is "read" only if its value is directly read. These results are contained in maps *use_map* and *def_map* respectively.

The analysis for computing *use_map* and *def_map* is built on top of a points-to analysis and an escape analysis, and is context-sensitive and flow-insensitive. It considers the strongly-connected components (SCC) in the call graph and performs a bottom-up traversal of the SCC-DAG. For each SCC, a fixed-point computation is used to handle recursion. The analysis employes the call graph and the points-to relationships generated by Spark [16], which is a points-to analysis engine available as part of Soot.

For each method m, the analysis maintains a set C_m of all objects from which a mutated object can be reached. If the points-to set of a static field f contains any object from C_m, f should be included in set $def\_map.get(m)$. When a caller of m is processed, we only propagate the objects from C_m that can directly escape m and be referenced by m's caller; this approach resembles escape analysis, but we need to track only escaping through formal parameters and return values. The analysis is based on the observation that at the level of m, it is not necessary to consider *all* static fields: it is enough to focus only on the static fields whose objects are referenced in m, and check if these objects can be found in C_m. Due to space limitations, we do not provide

[1]When the checkpoint is within a loop, non-CDMP statements located between the loop predicate and the checkpoint are removed, because their execution is unnecessary for recreating the program states at successive run-time instances of the checkpoint.

```
$Select\_Static\_Fields$(Map $use\_map$, Map $def\_map$)
Set $pre = \emptyset$ /* set of statements */
Set $post = \emptyset$ /* set of statements */
/* step 1: compute statement set */
for i = 1 to length($cc$-$chain$) do
    let $cs$ be the $i$-th call site in $cc$-$chain$
    $pre = pre \cup$ pre-$cs$-region
    $post = post \cup$ post-$cs$-region
end for
Set $usf = \emptyset$ /* used static fields */
Set $dsf = \emptyset$ /* defined static fields */
/* step 2.1: update the written set $dsf$ */
for all statements $p \in pre$ do
    if $p$ is a call site then
        for all methods $m$ that $p$ could invoke do
            $dsf = dsf \cup def\_map.get(m)$
        end for
    end if
    if $p$ writes $f$ or an object reachable from $f$ then
        $dsf = dsf \cup \{f\}$
    end if
end for
/* step 2.2: update the read set $usf$ */
for all statements $p \in post$ do
    if $p$ is a call site then
        for all methods $m$ that $p$ could invoke do
            $usf = usf \cup use\_map.get(m)$
        end for
    end if
    if $p$ reads $f$ then
        $usf = usf \cup \{f\}$
    end if
end for
return $usf \cap dsf$
```

Figure 3: Algorithm for static fields selection.

the technical details of the analysis algorithm; conceptually, it works in a manner similar to escape analysis [3, 32].

After *use_map* and *def_map* are computed, the algorithm from Figure 3 is used to select the static fields that need to be captured. The algorithm first computes all statements that could potentially execute before the checkpoint (contained in *pre*), and all statements that could potentially execute after the checkpoint (contained in *post*). Each call site *cs* in the CC-chain is inspected, updating *pre* and *post* with the pre-*cs*-region and post-*cs*-region, respectively.

The second step is to inspect each statement in *pre* and *post*, and check if it could potentially write (for *pre*) or read (for *post*) a static field. If a statement has a direct write effect on a static field f or an object reachable from f, f is included in *dsf* ("defined static fields"). Adding f to *usf* ("used static fields") is necessary only if the statement directly reads f. For call sites, the precomputed *use_map* or *def_map* are used to update *usf* or *dsf* with the set of static fields that all possible target methods could read or write. The algorithm returns the intersection of *usf* and *dsf*.

► *Example.* Suppose in Figure 1 we have precomputed that *processCmdLine* has both a write effect on static field *G.instance* (because line 6 in *soot_options_Options* modifies the *G* object reachable from the field) and a read effect on it (because line 4 reads the value of the field). Therefore, both *def_map* and *use_map* contain the pair (*processCmdLine*, {*G.instance*}). After step 1 of the algorithm, *pre* and *post* are $\{42, 43, 22\text{–}27\}$ and $\{29\text{–}39\}$, respectively. These statements do not have direct effects on static fields. However, line 22 calls *processCmdLine*, and therefore *G.instance* is included in *dsf.* Suppose that method *getPack* (omitted in Figure 1) had a read effect on *G.instance.* Due to the calls to *getPack* in *post*, *G.instance* would be included in *usf.* The intersection of *dsf* and *usf* results in {*G.instance*}, which is the set of static fields that should be recorded/replayed. ◄

```
class A {                        class B {
 void main() {                    Set s;
  Set hs = new HashSet();         B(Set arg) {s = arg;}
  B b = new B(hs);                void m() {
  // --- capt/repl(b)              B r0 = this;
  b.m();                           r0.s = new HashSet();
  // --- extra capt/repl           // --- checkpoint
  if(hs == b.s) {                  // --- capt/repl(r0)
      ...                          r0.s.add("");
  }}}                             }}
```

Figure 4: Post-checkpoint capture and replay.

3.2 Post-Checkpoint Capture and Replay

When a primitive-type local variable or static field is captured, the corresponding value is written to disk. For a reference-type value, the corresponding heap object is written by capturing its primitive-type fields, and then recursively writing the heap objects pointed-to by the reference-type fields. The implementation details of the object writer and reader can be found in Section 5.

We use a hash map to record all objects (reachable from captured variables) that have already been written. When writing an object that is directly pointed to by a variable or is transitively reachable from that variable, if the object can be found in the table, we simply write a reference to the existing object (i.e., the address of the object in the table). Hence, the potential aliasing relationships are still maintained when objects are later read from disk.

Replaying only at CDMPs and the checkpoint is not enough to guarantee correct execution after the CP container returns. For example, consider the program fragment in Figure 4. We capture/replay variables before call site *b.m()*, which is a CDMP, and at the checkpoint. After call *b.m()* returns, the replaying execution fails because local *hs* is not restored. To solve this problem, we need to additionally capture/replay variables immediately after each call site in the CC-chain — in this case, after *b.m()* in *main.* The variables that need to be captured/replayed include only the local variables that will be used in the rest of the method.

When writing a local variable, two situations could occur. First, it is possible that all objects in its object graph do not exist in the log file. Hence, we write the entire new object graph to the log. When the object graph is read later during the replaying phase, every object in the graph has exactly the same content as it had in the capturing phase. Second, suppose that an object *o* in the graph has already been written (either at the checkpoint or by the post-callsite capture somewhere deeper in the CC-chain). Therefore, when the recursion in the object writer reaches this object, it simply writes a reference to it, and does not consider its fields. This leads to two subcases. First, if in the capturing phase *o* has not been updated between the time it was written to disk and the current point, the state of *o* seen here during the replaying phase is up-to-date. Second, suppose that in the capturing phase *o* has been updated somewhere between the time it was written to disk and the current point. Hence, during capture, the reference to o we write at the current point refers to an old object. In fact, this does not create consistency issues: during the replaying phase, the reference to o obtained from the log file maps to a reference to an object which has already been loaded at some earlier moment of time, and this object has automatically been updated to its correct state by the execution of the post-checkpoint code in the original program.

3.3 Additional Issues

If a method in the CC-chain has multiple callers, we have to replicate the method so that the capture/replay operations do not affect the invocation of this method from a caller not in the chain. Hence, we can instrument only the replicated method and leave the original one unmodified.

It is possible for a single checkpoint to have multiple run-time instances — for example, if the checkpoint or some call site on the CC-chain is inside a loop. Since the loop condition will be included in the set of captured CDMPs, the replaying phase replicates the iterative behavior, and reaches the checkpoint multiple times. In the capturing phase, when the checkpoint is reached for the i-th time, the necessary state at this particular moment is recorded on disk, and later used during the replaying phase for that same i-th run-time instance of the checkpoint.

One limitation of the proposed technique is that it does not guarantee the correctness of the execution due to dependencies on external resources. For example, suppose that the post-checkpoint execution depends on the state of external resources such as files, databases, etc. Such external state is not part of the application state that is captured by our approach, and the replaying version cannot be guaranteed to replicate the execution of the original program. Existing open-source serialization libraries may be useful for handling I/O streams and external stateful entities such as files. Another limitation is that executions that directly use unique-per-execution values such as the system clock or object hash codes cannot be replicated precisely. In addition, our current implementation considers only single-threaded programs; future work will have to address the handling of execution interleaving for multi-threaded programs. Finally, if code modifications are later introduced in the replaying version by the programmer (e.g., for debugging purposes), they cannot create new cross-checkpoint dependencies — for example, if in the original program a local variable is not read after the checkpoint, this variable cannot be read in the post-checkpoint region in the replaying version.

4. MULTIPLE EXECUTION REGIONS

When debugging or testing a long-running program, programmers may be interested in multiple execution regions. For example, Soot contains several packs of analyses and transformations, including a whole-program pack and a body pack. If programmers are interested in the these two packs, they may want to replay only their executions. This cannot be achieved by taking a single checkpoint in the program. In this section we generalize our approach to allow taking checkpoints for multiple execution regions.

An *execution region* is designated by a *starting point* and an *ending point*, which are specified by two CC-chains. The region includes all statements executed after the staring point and before the ending point. The single checkpoint described earlier is a special case of an execution region, with the starting point being the checkpoint and the ending point being the last statement of the program. We use the same control-dependence-based slicing algorithm to remove statements in front of the starting point of the first region, after the ending point of the last region, and between the ending point of one region and the starting point of the next region. Hence, the execution of each region is directly connected to that of the next region. If a region has an overlap with an exceptional trap, we have to preserve the trap handler so that the exceptions thrown from the trap can be correctly handled. We need to capture/replay variables only at the CDMPs of the starting chain of each region, because the ending chain is solely used to indicate the region boundary.

The complication of using multiple execution regions is that we have to replicate all methods in the two chains of each region, so that capturing/replaying for each region does not influence the execution of another region. For example, suppose there are two chains $main \rightarrow a \rightarrow b$ and $main \rightarrow c \rightarrow a \rightarrow b$. If a can be called from another method d, which is not in the chains, we have to replicate a and b twice, to ensure that (1) replaying in the first chain does not influence the execution of the second chain, and (2) replaying in both chains does not influence the invocation of a from d.

However, naively replicating every method in the chains can easily lead to an explosion in program size, especially if execution regions are specified with long chains. An important observation is that call chains often are quite similar to each other. For example, if two chains are $main \rightarrow a \rightarrow b \rightarrow c$ and $main \rightarrow a \rightarrow b \rightarrow d$, it is enough to replicate a and b once. Furthermore, if a has just a single caller $main$, we do not need to replicate any methods.

Based on this observation, we use an algorithm that merges the call chains in the following manner. Using top-down traversal, the algorithm inspects each level of the call chains. Suppose that two chains contain the same method m at the same level. If this happens at the first level, we just merge the two chains into a tree with root m. Otherwise, if m's parents in the two chains have already been merged, the two m nodes are also merged. The algorithm produces a forest with merged chains. As a result, the number of replicated methods can be significantly reduced.

5. IMPLEMENTATION TECHNIQUES

We have implemented the proposed approach in our JCP framework based on Soot. This section briefly discusses several implementation techniques.

Serializer and loader. Although the Java libraries provide classes *ObjectInputStream* and *ObjectOutputStream* for object reading and writing, several restrictions prevented us from using them directly. We built our own *ObjectReader* and *ObjectWriter* by modifying these two classes. (Others have addressed this issue by employing existing serialization libraries [10].) For example, we removed the check for existence of non-arg constructors. When an object is loaded and its class does not have a non-arg constructor, *sun.misc.Unsafe* is used to allocate heap space without calling any constructors. As another example, we replaced the original recursive implementation with a worklist algorithm, because deep recursion can cause stack overflow.

Writing an object to disk and then reading it back destroys its hash code (if the object's class does not declare a *hashCode* method) because the hash code is based on the object's internal address in the JVM. This affects data structures relying on object hash codes, such as *Hashtable* and *HashMap*. When an object is read from disk, we detect objects of types *Hashtable* and *HashMap*, including their subtypes, and handle this problem similarly to the default implementation in the Java libraries. After all (transitive) fields of the object are read and appropriately set, we retrieve each entry and insert it into a new *Hashtable* or *HashMap* object. Eventually, we replace the internal table in the original object with the table in the new object. Our experiments indicate that this operation takes almost 30% of the total time needed to recover the state from disk.

Checkpoint specification language. We provide a pattern specification language for region specification; the grammar of this language is shown below:

```
region := pt ’&’ pt
pt := Line_ID ’,’ Class_Name ’,’ chain | EMPTY
chain := me | chain ’;’ me | ’*’
me := Class_Name ’.’ method_decl |
      Class_Name ’.’ method_decl ’:’ Line_ID
method_decl := Method_Name |
               Method_Name ’(’ Param_List ’)’
```

An ampersand `&` is used to split the starting point and the ending point of a region. For example, we can specify the checkpoint in Figure 1 using the pattern

```
28, Main, Main.main(String[]):44;Main.run(String[]) &
```

which means that the starting point of the region is at line 28 in class *Main* and the ending point is at the end of the program. The CC-chain is *Main.main(String[])* $\rightarrow$ *Main.run(String[])*, and the call site in *main* that calls *run* is at line 44. The parameter list of the method and the call site line ID can be omitted, if the method name is unique in the class, and the call site is unique in the method. A wild card (*) can be used to specify a chain of arbitrary length, if that chain is unique in the program.

6. EXPERIMENTAL STUDY

To evaluate our proposal for language-level checkpointing/replaying, we performed a variety of experiments, focusing on the effectiveness and efficiency of the static analyses, and on the run-time performance of the checkpointing version and the replaying version. In particular, our experiments consider the following research questions:

- How effective are the static analyses in reducing the number of variables that are captured/replayed?
- How many new statements are introduced to the capturing version, and how many irrelevant statements are removed from the replaying version?
- What is the cost of the static analyses?
- What run-time overhead does the instrumentation introduce in the capturing version, and how much performance speed-up is gained in the replaying version?

We performed two studies: one focused on the static analyses and the other one on the run-time performance. The first study used the 15 Java programs shown in Table 1. For each program, the table shows the number of classes, methods, statements in Soot's intermediate representation, and non-comment non-blank lines of code. For *soot-2.2.3*, the numbers also account for the *polyglot* and *jasmin* libraries.

In the experiments we used multiple execution regions (as described in Section 4). For all programs except *soot*, we defined between one and four regions. The regions were chosen

Program	*#Classes*	*#Methods*	*#Stmts*	*#LOC*
socksproxy	24	261	4439	2592
socksecho	25	295	5305	3044
jtar-1.21	65	319	7123	8997
compress	41	327	7535	4548
jb-6.1	45	548	7538	4418
db	32	317	7567	4641
jlex-1.2.6	26	162	8250	5591
javacup-0.10j	41	408	9753	5621
violet	127	666	9930	6569
raytrace	54	458	10306	5962
jflex-1.4.1	75	568	15614	9635
jess	180	973	17927	10189
sablecc-2.18.2	260	2241	26573	21503
muffin-0.9.3	278	2351	38139	27652
soot-2.2.3	2738	227333	322356	116458

Table 1: Analyzed programs.

based on what we judged to be boundaries of major functionalities of the program, respecting the restrictions outlined in Section 3.3. Because we were not familiar with the internals of these programs, it was quite time consuming to manually inspect the source code to decide what could be an appropriate region; for this reason, the number of chosen regions was relatively small. For *soot*, with which we are fairly familiar, we ran the analyses five times. The set of regions was extended with new regions for each successive run; for the last run, there were a total of ten regions. Each region crossed over a Soot implementation of a static analysis (e.g., the class hierarchy analysis `cg.cha` and the static inliner `wtop.si`).

The second study investigated the run-time performance of the versions of *soot-2.2.3*. This is by far the largest application in our data set; because it is a long-running program (e.g., more than an hour in our experiments), it is representative of the applications for which checkpointing is likely to be most desirable and useful.

6.1 Study 1: Static Analyses

Table 2(a) shows the number of specified execution regions *#R* and the number of capturing/replaying instrumentation points *#IP*. Table 2(b) shows the total number of cloned methods *#CM* and the percentage Red_{CM} of method replication reduction achieved by the call chain merging approach outlined in Section 4, based on the total number of methods in the input chains: $Red_{CM} = (\#total - \#CM)/\#total$. Clearly, call chain merging can significantly reduce the number of methods that need to be replicated. For example, for *soot(5)*, many of the regions crossed entire packs (e.g., the call graph building pack `cg` and the whole-program Jimple transformation pack `wjtp`). As a result, the starting and ending chains of these regions were essentially the same: $main \rightarrow run \rightarrow runPacks$. Our approach merged these chains and completely avoided replication for these regions.

Table 2(b) shows the number *#LO* of local variables (including formal parameters) captured/replayed in all regions; column Red_{LO} is the reduction from the total number of locals in all regions to *#LO*. In general, a relatively small number of locals were captured/replayed for each program. One possible reason is that most methods that we inspected were close to the top of the call graph (due to our lack of in-depth understanding of the internals of the programs), and the checkpoints that were defined were relatively close to the start of the call chain, resulting in the small number

Program	(a)		(b)							(c)		(d)
	#R	*#IP*	*#CM*	Red_{CM}	*#LO*	Red_{LO}	*#SF*	Red_{SF}	*%Lib*	*#RemSt*	*#InstrSt*	*Time* (s)
socksproxy	3	11	1	92.9%	27	91.3%	316	70.0%	100%	3754	1077	136+118
socksecho	3	14	0	100%	57	71.8%	404	60.6%	100%	1276	1129	117+136
jtar-1.21	2	4	0	100%	4	91.7%	119	81.9%	100%	12	267	123+135
compress	1	6	2	75%	18	84.2%	95	51.0%	81.1%	2069	59	90+19
jb-6.1	3	5	0	100%	27	80.7%	249	60.8%	90.8%	1649	533	70+19
db	2	5	1	90%	20	60.8%	97	75.3%	79.4%	1203	237	81+18
jlex-1.2.6	3	8	0	100%	12	97.6%	231	53.6%	100%	1425	698	65+38
javacup-0.10j	4	9	4	80.0%	176	56.9%	460	53.3%	66.1%	7861	1117	70+29
violet	4	9	1	90%	64	86.0%	690	50.8%	100%	235	1587	150+210
raytrace	3	10	2	92.3%	135	65.0%	2	99.7%	0%	3139	190	111 + 109
jflex-1.4.1	2	8	3	75%	24	92.3%	246	70.5%	93.5%	12292	623	160+180
jess	3	8	3	78.6%	36	67.3%	92	77.2%	84.8%	4314	287	100+30
sablecc-2.18.2	4	11	0	100%	76	86.3%	350	53.1%	84.6%	228	811	80+51
muffin-0.9.3	3	20	0	100%	57	85.2%	366	66.7%	92.9%	12242	998	142+127
soot-2.2.3(1)	1	2	0	100%	2	98.4%	125	50.3%	64.8%	52687	268	553+1597
soot-2.2.3(2)	2	4	0	100%	6	98.6%	263	64.0%	61.6%	176548	568	576+1522
soot-2.2.3(3)	4	15	4	82%	68	91.7%	538	54.1%	60.0%	212274	1254	528+1584
soot-2.2.3(4)	6	29	8	92.2%	198	80.7%	814	56.2%	59.7%	275293	1947	542+1414
soot-2.2.3(5)	10	35	19	92.6%	420	75.3%	1366	53.4%	59.3%	303275	3104	539+1604

Table 2: Static analyses: (a) regions and instrumentation points (b) replicated methods; captured local variables and static fields (c) removed and inserted statements (d) analysis running time.

of instrumentation points. For programs where the starting/ending points of a major program functionality were located close to *main* (e.g., in *soot* there were only two call-graph-edge hops between *main* and *runPacks*), the checkpoints were likely to be close to *main* and the number of local variables to be recorded could be expected to be small. For programs where checkpoints were taken in methods far from *main*, the number of local variables could be fairly large. Of course, the variables listed in *#LO* may directly or transitively reference a large number of heap objects that also need to be recorded.

Table 2(b) also shows the number of captured/replayed static fields *#SF* and the reduction Red_{SF} from the total number of static fields that could be accessed directly or transitively by *main* to *#SF*. *%Lib* is the percentage of fields from *#SF* that were declared in the Java libraries. The values of Red_{SF} were greater than 50% for all programs, which shows that the analyses effectively reduced the number of static fields that needed to be captured. As expected, most of the selected static fields were declared in library classes. To ensure the correctness of our approach, we had to record/replay all these fields. In reality, the majority of these fields most likely do not affect the execution of the application code. For example, a call to *System.out.println* reads 259 static fields, and writes many of them. If there are two such statements in the program, one before the checkpoint and the other after it, we have to record and replay the intersection of the reading set and writing set, which is still a large set. Clearly, future work should address this issue. Note that for *raytrace*, only two static fields were selected; after manual inspection, we determined that the chosen regions did not call any library methods.

Table 2(c) shows the number *#RemSt* of statements that were removed in the replaying version, in methods on the call chain and in application methods that were invoked directly or transitively by removed call sites, as well as the number *#InstrSt* of statements that were inserted by JCP at all instrumentation points. For all programs except *jtar* and *sablecc*, the number of removed statements was much larger than the number of statements that were introduced.

Run	*#Objects*	*Space*	*%Heap*	$Time_c$ (s)	$Time_r$ (s)
1	4610958	36.2M	36.3%	4695.3	4643.5
2	65648481	745M	73.2%	4712.2	4410.5
3	65648481	745M	73.2%	4688.4	4387.3
4	77739311	806.4M	70.0%	4770.1	511.5
5	77767256	806.5M	63.5%	4972.8	533.1
6	75668735	795.3M	72.8%	4661.6	411.5

Table 3: Run-time performance for *soot*.

For *jtar*, the regions were close to the beginning of the program, and there were few statements executed before the checkpoint. For *sablecc*, the statements that were removed between regions did not contain many call sites.

Column *Time* shows the running time of the analyses, separated into two components. The first part is the running time of Spark (recall from Section 3 that we used Spark's output), and the second part is the time used by our analyses to compute the checkpointing version and the replaying version. The analysis running time depends on the size of the program and the locations of the checkpoints. The major component (on average 84%) of the running time was the computation of the Mod/Use effects for static fields. In future work we plan to refine the algorithm with a variety of static analysis techniques (e.g., memoization, equivalence analysis [17], precomputed library summaries [25], etc.)

6.2 Study 2: Run-Time Performance

Our second study considered the run-time performance of the original version, the checkpointing version, and the replaying version. In this experiment, we ran *soot* with *soot* itself as the input, enabling phases `cg.spark`, `wjtp`, `wjop.ji`, `wjap.uft`, `jtp`, and `jop.cp` [29]. We ran *soot* six times; for each run, we took a checkpoint at the end of a different phase, closer and closer to the end of the program. The locations of the six checkpoints were (1) before whole-program packs, (2) after `cg`, (3) after `wjtp`, (4) after `wjop`, (5) after `wjap`, and (6) after body packs.

The execution time of the original version was 4665.7 seconds. For each run, Table 3 shows the number *#Objects* of objects that are recorded/restored; the amount *Space* of

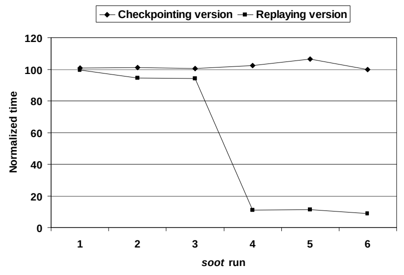

Figure 5: Normalized execution times.

used disk space; the ratio *%Heap* between *Space* and the size of the entire heap; the time $Time_c$ of the checkpointing version, including all bookkeeping and I/O; and the time $Time_r$ of the replaying version, including all recovery code and I/O. Figure 5 shows the normalized execution times for runs of the checkpointing version and the replaying version (the basis at 100% is the original version). Clearly, the execution of the capturing version was close to the original version: the average run-time overhead was 1.8%.

The replaying time was reduced significantly when the distance between the checkpoint and the end of the program became shorter. These preliminary results indicate that the proposed technique is a promising candidate for testing, debugging, and dynamic slicing of long-running applications; in particular, it may be able to assist a programmer to focus on functionality that is executed after some expensive computations. These savings could be important even if the static analyses for generating the capturing/replaying versions have non-trivial cost (which is the case for *soot*, as indicated by the last column in Table 2). The expectation is that the replaying version will be executed multiple times (e.g., with multiple tests from a test suite; for several debugging runs in manual debugging; for repeated execution during delta debugging), and the one-time cost of running the analyses will be amortized over multiple replaying runs.

Note that a large number of objects were saved/loaded for the last five runs, and large files were generated on disk. In these runs, static field *soot.G.instance* was always selected for capturing and replaying. Soot uses this field as a global object manager, through which most heap objects could be reached. Mutating almost any object leads to the selection of *soot.G.instance*, and saving this field requires writing of the majority of heap objects.

In future work, we plan to use additional long-running programs to evaluate the performance of the capturing version and the replaying version. It is particularly interesting to consider the relationships among the number of objects captured and replayed, the locations of checkpoints, and program-specific characteristics.

7. RELATED WORK

Checkpointing/replaying at the system level. Checkpointing/rolling back is an old technique [1] that was originally use for fault tolerance for distributed systems [11]. There is a large body of later work that addresses the efficiency of taking checkpoints [2]. Virtual machine logging and replaying is used to detect intrusions [9] and to debug the guest operating systems [13]. For multi-processed systems or multi-threaded programs, special care needs to be taken to replay non-deterministic executions [26].

Checkpointing at the application level. Application-level checkpointing has recently gained popularity for debugging and testing [24, 22], dynamic slicing [36], and race detection [5, 6]. There is also work on deterministic replay of multi-threaded programs [23, 4]. User-driven language-level checkpointing techniques have been proposed for recording data of interest by instrumenting a program at checkpoints [15, 12]. Our approach also falls into this category of work. However, these existing techniques focus only on recording data, without considering the replaying problem. We employ static analyses to compute a set of program points along the execution path to the checkpoint, and a conservative subset of program state at each point, so that capturing this state can ensure the correct replay of the execution.

Capture and replay. Capture and replay is a lightweight technique that simulates the behaviors of "uninteresting" components by capturing and replaying the interactions between them and the component of interest. It is a special form of the checkpointing/replaying technology, with checkpoints being specified at component boundaries. Capture and replay has been used to debug relevant components by isolating the interactions between them and the rest of the system [20, 19]. Our approach is orthogonal to this work, as we focus on partitioning the execution with respect to temporal properties (before/after programmer-specified execution moments), while [20, 19] considers structural partitioning (inside vs. outsize of an interesting component). Our approach may be more convenient when the structure of the program does not directly reflect its functional behavior. For example, in manual debugging, the programmer may be interested in separating the correct phases of the application from the incorrect ones, and defining checkpoints based on the temporal boundaries between these phases. The test refactoring approach from [27] describes a capture-and-replay technique similar to [20, 19], in which system tests are converted to unit tests that are more focused and efficient. The approaches in [20, 27, 19] can be regarded as action-based, while our technique is state-based [10].

The work closest to ours is the carving-and-replay framework from [10] for generating differential unit tests from system tests. This approach considers program states that need to be saved or loaded before and after calls to a method of interest, and preserves only a subset of the state by applying state projections. To achieve quicker replay, the projections may result in reduced fault detection or unexecutable tests; these tradeoffs define a general framework for carving and replaying. Because checkpointing requires precise replication of the captured execution, in general we cannot exploit such tradeoffs, and instead we employ the interprocedural analyses described in Section 3 to reduce the size of the recorded state. The state recorded in [10] is based on the heap graphs reachable from the formal parameters of the method of interest; in our approach, objects reachable from locals and static fields also need to be taken into account. Since we want to replay the entire program and not just the unit under test, our approach recreates the state of the run-time call stack that leads to and continues from the checkpoint.

8. CONCLUSIONS AND FUTURE WORK

We propose a context-sensitive checkpointing and replaying technique that works at the language level without OS or JVM support. Our approach uses static analyses to identify instrumentation points, determine the partial state that should be captured/replayed at each instrumentation point, and generate the checkpointing version and the replaying version. An initial experimental study indicates that significant savings could be achieved for the replaying version, with small run-time overhead for the capturing version.

Clearly, there is a wide range of open questions for future work. The analysis of static fields can be improved in terms of cost and precision. As Table 2 indicates, static fields from library classes are of particular interest. One possibility is to use precomputed summary information about the libraries, based on [25]. Additional precision refinements are also worth investigating, as well as the use of context-sensitive points-to analyses. The run-time support for object reading and writing can be improved (e.g., by using non-blocking I/O, memory-mapped buffers, or serialization libraries used in previous work [10]). Finally, extensive studies on long-running computation- and memory-intensive applications are needed, in order to investigate the generality of our initial results. Numerous factors — such as call chain length, nesting in loops, characteristics of the state that "flows" through the checkpoint, component/functionality/phase boundaries — are obviously of great importance for the usefulness of the proposed techniques, and they should be investigated carefully in future studies.

Acknowledgments. We would like to thank the ESEC/FSE reviewers for many valuable comments and suggestions.

9. REFERENCES

[1] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. In *Symp. Operating Systems Principles*, pages 90–99, 1983.

[2] Y. Chen, J. S. Plank, and K. Li. CLIP: A checkpointing tool for message-passing parallel programs. In *Conf. Supercomputing*, pages 1–11, 1997.

[3] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 1–19, 1999.

[4] J.-D. Choi, B. Alpern, T. Ngo, and J. Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *Int. Parallel and Distributed Processing Symp.*, page 10023a, 2001.

[5] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Conf. Programming Language Design and Implementation*, pages 258–269, 2002.

[6] M. Christiaens, J.-D. Choi, M. Ronsse, and K. D. Bosschere. Record/play in the presence of benign data races. In *Int. Conf. Parallel and Distributed Processing Techniques and Applications*, pages 1200–1206, 2002.

[7] H. Cleve and A. Zeller. Locating causes of program failures. In *Int. Conf. Software Engineering*, pages 342–351, 2005.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Symp. Principles of Programming Languages*, pages 25–35, 1989.

[9] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Symp. Operating Systems Design and Implementation*, pages 211–224, 2002.

[10] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *Symp. Foundations of Software Engineering*, pages 253–264, 2006.

[11] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[12] M. Kasbekar, C. R. Das, S. Yajnik, R. Klemm, and Y. Huang. Issues in the design of a reflective library for checkpointing C++ objects. In *Symp. Reliable Distributed Systems*, page 224, 1999.

[13] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Technical Conf.*, pages 1–15, 2005.

[14] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, Oct. 1988.

[15] J. L. Lawall and G. Muller. Efficient incremental checkpointing of Java programs. In *Int. Conf. Dependable Systems and Networks*, pages 61–70, 2000.

[16] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *Int. Conf. Compiler Construction*, LNCS 2622, pages 153–169, 2003.

[17] D. Liang and M. J. Harrold. Equivalence analysis: A general technique to improve the efficiency of data-flow analyses in the presence of pointers. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 39–46, 1999.

[18] R. H. B. Netzer and M. H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In *Conf. Programming Language Design and Implementation*, pages 313–325, 1994.

[19] A. Orso, S. Joshi, M. Burger, and A. Zeller. Isolating relevant component interactions with JINSI. In *Int. Workshop on Dynamic Analysis*, pages 3–9, 2006.

[20] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *Int. Workshop on Dynamic Analysis*, pages 29–35, 2005.

[21] D. Z. Pan and M. A. Linton. Supporting reverse execution for parallel programs. In *Workshop on Parallel and Distributed Debugging*, pages 124–129, 1988.

[22] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Symp. Operating Systems Principles*, pages 235–248, 2005.

[23] M. Ronsse, K. D. Bosschere, M. Christiaens, J. C. de Kergommeaux, and D. Kranzlmüller. Record/replay for nondeterministic program executions. *Communications of the ACM*, 46(9):62–67, 2003.

[24] M. Ronsse, K. D. Bosschere, and J. C. de Kergommeaux. Execution replay and debugging. In *Int. Workshop Automated Debugging*, 2000.

[25] A. Rountev, S. Kagan, and T. Marlowe. Interprocedural dataflow analysis in the presence of large libraries. In *Int. Conf. Compiler Construction*, LNCS 3923, pages 2–16, 2006.

[26] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Conf. Programming Language Design and Implementation*, pages 258–266, 1996.

[27] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *Int. Conf. Automated Software Engineering*, pages 114–123, 2005.

[28] Y. Saito. Jockey: A user-space library for record-replay debugging. In *Int. Symp. Automated and Analysis-Driven Debugging*, pages 69–76, 2005.

[29] `http://www.sable.mcgill.ca/soot`.

[30] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conf.*, pages 29–44, 2004.

[31] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Int. Conf. Compiler Construction*, LNCS 1781, pages 18–34, 2000.

[32] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 187–206, 1999.

[33] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Symp. Foundations of Software Engineering*, pages 253–267, 1999.

[34] A. Zeller. Isolating cause-effect chains from computer programs. In *Symp. Foundations of Software Engineering*, pages 1–10, 2002.

[35] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *Int. Symp. Automated and Analysis-Driven Debugging*, pages 33–42, 2005.

[36] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *Symp. Foundations of Software Engineering*, pages 81–91, 2006.