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.
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 runtime 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 runtime 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 (Java Checkpointing) 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`. 
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 directly from 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-

---

**Figure 1: Soot startup example.**

```java
class G { ... ...
    static G instance = new G();
    Options op;
    static G v() { return instance; }
    Options soot_options_Options() {
        if (op == null) op = new Options();
        return op;
    }
    G g = G.v();
    return g.soot_options_Options();
    if (args.length != 0) {
        Main m = new Main();
        static void main(String[] args) {
            ... Options.v().parse(args) ... ...
            if (Options.v().whole_jimple()) { // phase 3
                Set body_packs = getBpacks();
                Set wp_packs = getWpacks();
                loadNecessaryClasses(); // phase 2
                processCmdLine(args); // phase 1
                void run(String[] args) {
                    // phase 1
                    processCmdLine(args);
                    loadNecessaryClasses();
                    Set wp_packs = getWpacks();
                    Set body_packs = getBpacks();
                    if (Options.v().whole_jimple()) { // phase 3
                        getPack("cg").apply();
                        // --- checkpoint ---
                        getPack("wjtp").apply();
                        getPack("wjop").apply();
                        getPack("wipj").apply();
                        ... retrieveAllBodies(); // phase 4
                        for (Iterator i = body_packs.iterator();
                            i.hasNext();){ // phase 5
                            String s = (String)body_packs.next();
                            getPack(s).apply();
                            ...
                            ... Options.v().parse(args) ... ...
                            if (args.length != 0) {
                                Main m = new Main();
                                n.run(args); ...
                                G g = G.v();
                                return g.soot_options_Options();
                            }
                        }
                    }
                }
            }
        }
    }
```
control 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. 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 during replay. For a call site in the CC-chain, the environment contains the run-time type of the receiver object if the instance is an 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 sufficient 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 employs 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 Cm 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 Cm, f should be included in set def_map.get(m). When a caller of m is processed, we only propagate the objects from Cm 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
1: Select_Static_Fields(Map use_map, Map def_map)
2: Set pre = \emptyset /* set of statements */
3: Set post = \emptyset /* set of statements */
4: /* step 1: compute statement set */
5: for i = 1 to length((cc-chain))
6: let cs be the i-th call site in cc-chain
7: pre = pre ∪ pre-cs-region
8: post = post ∪ post-cs-region
9: end for
10: Set usf = \emptyset /* used static fields */
11: Set dsf = \emptyset /* defined static fields */
12: /* step 2.1: update the written set dsf */
13: for all statements p ∈ pre do
14: if p is a call site then
15: for all methods m that p could invoke do
16: dsf = dsf ∪ def_map.get(m)
17: end for
18: end if
19: if p writes f or an object reachable from f then
20: dsf = dsf ∪ \{f\}
21: end if
22: end for
23: /* step 2.2: update the read set usf */
24: for all statements p ∈ post do
25: if p is a call site then
26: for all methods m that p could invoke do
27: usf = usf ∪ use_map.get(m)
28: end for
29: end if
30: if p reads f then
31: usf = usf ∪ \{f\}
32: end if
33: end for
34: return usf ∩ dsf

Figure 3: Algorithm for static fields selection.

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

Figure 4: Post-checkpoint capture and replay.

{G.instance}). After step 1 of the algorithm, pre and post are \{42, 43, 22–27\} and \{29–39\}, 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. ▶

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 ob-
ject 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 oper-
ations 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 depen-
dencies on external resources. For example, suppose that
the post-checkpoint execution depends on the state of ex-
ternal 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 guar-
anteed 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 ob-
ject 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, pro-
grammers 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, specified by two CC-chains. The region in-
cludes all statements executed after the starting point and
before the ending point. The single checkpoint described
earlier is a special case of an execution region, with the start-
ing 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 end-
ing 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 excep-
tional 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 → a → b and main →
c → a → 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 impor-
tant observation is that call chains often are quite similar to
each other. For example, if two chains are main → a → b →
c and main → a → b → 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 sev-
eral implementation techniques.

Serializer and loader. Although the Java libraries pro-
vide 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. (Oth-
er’s have addressed this issue by employing existing serial-
ization 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 call-
ing 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 hash-
Code 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 *HashSet* and *HashMap*. When an object is read from disk, we detect objects of types *HashSet* 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 *HashSet* 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 'k' pt
pt := Line_ID ',' Class_Name ',' chain | EMPTY
chain := se | 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(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[]) → 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 wildcard (*) 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.

<table>
<thead>
<tr>
<th>Program</th>
<th>#Classes</th>
<th>#Methods</th>
<th>#Stmts</th>
<th>#LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>socksproxy</td>
<td>24</td>
<td>261</td>
<td>4439</td>
<td>2592</td>
</tr>
<tr>
<td>socksseo</td>
<td>25</td>
<td>295</td>
<td>5035</td>
<td>3044</td>
</tr>
<tr>
<td>jar-1.21</td>
<td>65</td>
<td>319</td>
<td>7123</td>
<td>8997</td>
</tr>
<tr>
<td>compress</td>
<td>41</td>
<td>327</td>
<td>7535</td>
<td>4548</td>
</tr>
<tr>
<td>ib-0.1</td>
<td>45</td>
<td>548</td>
<td>7535</td>
<td>4548</td>
</tr>
<tr>
<td>db</td>
<td>32</td>
<td>317</td>
<td>7567</td>
<td>4548</td>
</tr>
<tr>
<td>jflex-1.2.6</td>
<td>26</td>
<td>162</td>
<td>8290</td>
<td>5991</td>
</tr>
<tr>
<td>javacp-0.10</td>
<td>41</td>
<td>498</td>
<td>9763</td>
<td>5921</td>
</tr>
<tr>
<td>wtop</td>
<td>127</td>
<td>666</td>
<td>9950</td>
<td>6699</td>
</tr>
<tr>
<td>raytrace</td>
<td>54</td>
<td>458</td>
<td>10306</td>
<td>5962</td>
</tr>
<tr>
<td>jflex-1.4.1</td>
<td>75</td>
<td>568</td>
<td>15014</td>
<td>9635</td>
</tr>
<tr>
<td>jess</td>
<td>180</td>
<td>973</td>
<td>17927</td>
<td>10189</td>
</tr>
<tr>
<td>sablecc-2.18.2</td>
<td>260</td>
<td>2241</td>
<td>26013</td>
<td>21503</td>
</tr>
<tr>
<td>mullin-0.9.3</td>
<td>278</td>
<td>2351</td>
<td>38139</td>
<td>27602</td>
</tr>
<tr>
<td>soot-2.2.3</td>
<td>2748</td>
<td>22733</td>
<td>322586</td>
<td>110458</td>
</tr>
</tbody>
</table>

Table 1: Analyzed programs.

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 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 RedCM 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: RedCM = (#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 → run → 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 RedLO 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 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 directly or transitively by removed call sites, as well as the variables listed in #SF that were declared in the Java libraries. The values of #RemSt 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 250 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 #InstrSt 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

<table>
<thead>
<tr>
<th>Program</th>
<th>#R</th>
<th>#IP</th>
<th>#CM</th>
<th>Red_CM</th>
<th>Red_LO</th>
<th>%LO</th>
<th>#SF</th>
<th>Red_SF</th>
<th>%Lib</th>
<th>#RemSt</th>
<th>#InstrSt</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sootproxy</td>
<td>3</td>
<td>11</td>
<td>1</td>
<td>92.9%</td>
<td>91.3%</td>
<td>316</td>
<td>70.6%</td>
<td>100%</td>
<td>3754</td>
<td>1077</td>
<td>136+118</td>
<td>70+19</td>
</tr>
<tr>
<td>sookeseho</td>
<td>3</td>
<td>14</td>
<td>0</td>
<td>100%</td>
<td>57%</td>
<td>341</td>
<td>60.6%</td>
<td>100%</td>
<td>966</td>
<td>1129</td>
<td>117+136</td>
<td>80+19</td>
</tr>
<tr>
<td>jtar-1.21</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>100%</td>
<td>46%</td>
<td>119</td>
<td>81.9%</td>
<td>100%</td>
<td>12</td>
<td>267</td>
<td>123+135</td>
<td>80+19</td>
</tr>
<tr>
<td>compress</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>97%</td>
<td>84%</td>
<td>95</td>
<td>81.1%</td>
<td>100%</td>
<td>1299</td>
<td>1533</td>
<td>98+19</td>
<td>70+19</td>
</tr>
<tr>
<td>db</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>100%</td>
<td>22%</td>
<td>210</td>
<td>60.8%</td>
<td>100%</td>
<td>1649</td>
<td>533</td>
<td>70+19</td>
<td>81+18</td>
</tr>
<tr>
<td>flex-1.2.6</td>
<td>3</td>
<td>8</td>
<td>0</td>
<td>100%</td>
<td>97%</td>
<td>231</td>
<td>53.6%</td>
<td>100%</td>
<td>1425</td>
<td>698</td>
<td>65+38</td>
<td>70+29</td>
</tr>
<tr>
<td>javacc-0.10</td>
<td>4</td>
<td>9</td>
<td>4</td>
<td>80.9%</td>
<td>76%</td>
<td>166</td>
<td>53.3%</td>
<td>66.1%</td>
<td>256</td>
<td>1117</td>
<td>79+40</td>
<td>79+40</td>
</tr>
<tr>
<td>violet</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>90%</td>
<td>64</td>
<td>690</td>
<td>30.8%</td>
<td>100%</td>
<td>235</td>
<td>1587</td>
<td>150+210</td>
<td>150+210</td>
</tr>
<tr>
<td>raytrace</td>
<td>3</td>
<td>10</td>
<td>2</td>
<td>92.3%</td>
<td>65%</td>
<td>2</td>
<td>99.7%</td>
<td>0%</td>
<td>3139</td>
<td>190</td>
<td>111+109</td>
<td>111+109</td>
</tr>
<tr>
<td>jflex-1.4.1</td>
<td>2</td>
<td>8</td>
<td>3</td>
<td>75%</td>
<td>71%</td>
<td>246</td>
<td>70.5%</td>
<td>93.5%</td>
<td>1292</td>
<td>623</td>
<td>100+180</td>
<td>100+180</td>
</tr>
<tr>
<td>jess</td>
<td>3</td>
<td>8</td>
<td>3</td>
<td>78%</td>
<td>67%</td>
<td>92</td>
<td>77.2%</td>
<td>84.8%</td>
<td>4314</td>
<td>287</td>
<td>100+30</td>
<td>100+30</td>
</tr>
<tr>
<td>sablecc-2.18.2</td>
<td>4</td>
<td>11</td>
<td>9</td>
<td>100%</td>
<td>76</td>
<td>350</td>
<td>86.3%</td>
<td>84.6%</td>
<td>228</td>
<td>914</td>
<td>80+61</td>
<td>100+61</td>
</tr>
<tr>
<td>muffin-0.9.3</td>
<td>3</td>
<td>20</td>
<td>20</td>
<td>100%</td>
<td>76</td>
<td>366</td>
<td>66.6%</td>
<td>92.9%</td>
<td>1224</td>
<td>998</td>
<td>124+127</td>
<td>124+127</td>
</tr>
<tr>
<td>soot-2.2.3(1)</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>100%</td>
<td>2</td>
<td>126</td>
<td>50.3%</td>
<td>64.8%</td>
<td>5268</td>
<td>268</td>
<td>550+1507</td>
<td>550+1507</td>
</tr>
<tr>
<td>soot-2.2.3(2)</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>100%</td>
<td>6</td>
<td>263</td>
<td>54.1%</td>
<td>60.0%</td>
<td>1224</td>
<td>1254</td>
<td>528+1584</td>
<td>528+1584</td>
</tr>
<tr>
<td>soot-2.2.3(3)</td>
<td>4</td>
<td>15</td>
<td>4</td>
<td>82%</td>
<td>68</td>
<td>538</td>
<td>54.1%</td>
<td>60.0%</td>
<td>21274</td>
<td>1254</td>
<td>528+1584</td>
<td>528+1584</td>
</tr>
<tr>
<td>soot-2.2.3(4)</td>
<td>6</td>
<td>29</td>
<td>8</td>
<td>92.2%</td>
<td>80.7%</td>
<td>814</td>
<td>56.2%</td>
<td>59.7%</td>
<td>27529</td>
<td>1947</td>
<td>542+1414</td>
<td>542+1414</td>
</tr>
<tr>
<td>soot-2.2.3(5)</td>
<td>10</td>
<td>35</td>
<td>19</td>
<td>92.6%</td>
<td>75.3%</td>
<td>3466</td>
<td>53.4%</td>
<td>59.3%</td>
<td>30325</td>
<td>3104</td>
<td>559+1604</td>
<td>559+1604</td>
</tr>
</tbody>
</table>

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.

In our second study, we 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, jtp, wxjap, 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 wxjap, and (6) after body packs.

Table 3: Run-time performance for soot.

<table>
<thead>
<tr>
<th>Run</th>
<th>#Objects</th>
<th>Space</th>
<th>%Heap</th>
<th>Time (s)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4610958</td>
<td>36.2M</td>
<td>36.3%</td>
<td>4695.3</td>
<td>4643.5</td>
</tr>
<tr>
<td>2</td>
<td>6604851</td>
<td>74.5M</td>
<td>63.4%</td>
<td>1142.2</td>
<td>1106.5</td>
</tr>
<tr>
<td>3</td>
<td>5864851</td>
<td>74.5M</td>
<td>63.4%</td>
<td>1088.4</td>
<td>1058.3</td>
</tr>
<tr>
<td>4</td>
<td>77739311</td>
<td>806.4M</td>
<td>70.0%</td>
<td>4761.1</td>
<td>4115.5</td>
</tr>
<tr>
<td>5</td>
<td>77767256</td>
<td>806.5M</td>
<td>63.5%</td>
<td>4972.8</td>
<td>533.1</td>
</tr>
<tr>
<td>6</td>
<td>75668735</td>
<td>795.3M</td>
<td>72.8%</td>
<td>4661.6</td>
<td>4115.5</td>
</tr>
</tbody>
</table>
Replaying version and the replaying version. It is particularly interest-
the majority of heap objects.

of reached. Mutating almost any object leads to the selection
object manager, through which most heap objects could be
for capturing and replaying. Soot uses this field as a global
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 ex-
cution 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 dis-
tance 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, de-
bugging, and dynamic slicing of long-running applications;
in particular, it may be able to assist a programmer to fo-
cus on functionality that is executed after some expensive
computations. These savings could be important even if the
static analyses for generating the capturing/replaying ver-
ions 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 de-
bugging 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 ver-
sion and the replaying version. It is particularly interest-
ing to consider the relationships among the number of ob-
jects captured and replayed, the locations of checkpoints,
and program-specific characteristics.

7. RELATED WORK

Checkpointing at the system level. Check-
pointing/rolling back is an old technique [1] that was ori-
ginally use for fault tolerance for distributed systems [11].
There is a large body of later work that addresses the ef-
ciciency 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 sys-
tems 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 de-
bugging 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 conserva-
tive 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 be-
tween them and the component of interest. It is a special
form of the checkpointing/replaying technology, with check-
points 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 exec-
ution moments), while [20, 19] considers structural par-
titioning (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 be-
havior. For example, in manual debugging, the programmer
may be interested in separating the correct phases of the ap-
lication 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 fram-
eork from [10] for generating differential unit tests from sys-
tem 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 re-
playing. Because checkpointing requires precise replication
of the captured execution, in general we cannot exploit such
tradeoffs, and instead we employ the interprocedural analy-
ses 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 in-
terest; 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 un-
der 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 replay-
ing technique that works at the language level without OS
Acknowledgments. We would like to thank the ESEC/FSE reviewers for many valuable comments and suggestions.

9. REFERENCES


