

CROCHET: Checkpoint and Rollback via Lightweight Heap Traversal on Stock JVMs

Jonathan Bell

George Mason University, Fairfax, VA, USA
bellj@gmu.edu
 <https://orcid.org/0000-0002-1187-9298>

Luís Pina

George Mason University, Fairfax, VA, USA
lpina2@gmu.edu
 <https://orcid.org/0000-0003-4585-5259>

Abstract

Checkpoint/rollback (CR) mechanisms create snapshots of the state of a running application, allowing it to later be restored to that checkpointed snapshot. Support for checkpoint/rollback enables many program analyses and software engineering techniques, including test generation, fault tolerance, and speculative execution.

Fully automatic CR support is built into some modern operating systems. However, such systems perform checkpoints at the coarse granularity of whole pages of virtual memory, which imposes relatively high overhead to incrementally capture the changing state of a process, and makes it difficult for applications to checkpoint only some logical portions of their state. CR systems implemented at the application level and with a finer granularity typically require complex developer support to identify: (1) where checkpoints can take place, and (2) which program state needs to be copied. A popular compromise is to implement CR support in managed runtime environments, e.g. the Java Virtual Machine (JVM), but this typically requires specialized, non-standard runtime environments, limiting portability and adoption of this approach.

In this paper, we present a novel approach for *Checkpoint ROLLbaCk via lightweigHt HEap Traversal* (CROCHET), which enables fully automatic fine-grained lightweight checkpoints within unmodified commodity JVMs (specifically Oracle's HotSpot and OpenJDK). Leveraging key insights about the internal design common to modern JVMs, CROCHET works entirely through bytecode rewriting and standard debug APIs, utilizing special proxy objects to perform a lazy heap traversal that starts at the root references and traverses the heap as objects are accessed, copying or restoring state as needed and removing each proxy immediately after it is used. We evaluated CROCHET on the DaCapo benchmark suite, finding it to have very low runtime overhead in steady state (ranging from no overhead to 1.29x slowdown), and that it often outperforms a state-of-the-art system-level checkpoint tool when creating large checkpoints.

2012 ACM Subject Classification Software and its engineering → Frameworks

Keywords and phrases Checkpoint rollback, runtime systems, dynamic analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.17

Supplement Material Code available at <https://github.com/gmu-swe/crochet>

Acknowledgements We would like to thank the anonymous reviewers for their feedback. Riley Spahn and Michael Hicks provided many helpful comments on this document as well.

 © Jonathan Bell and Luís Pina;
licensed under Creative Commons License CC-BY
32nd European Conference on Object-Oriented Programming (ECOOP 2018).

Editor: Todd Millstein; Article No. 17; pp. 17:1–17:31
 Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Checkpoint/rollback (CR) tools capture the state of an application and store it in some serialized form, allowing the application to later resume execution by returning to that same state. CR tools have been employed to support many tasks, including fault tolerance [50, 46], input generation and testing [53, 49], and process migration [44, 48, 38, 16, 26, 9]. For instance, fault-tolerance tools can checkpoint at critical system decision points, allowing for automated recovery in the event of an otherwise unrecoverable failure. As another example, an input fuzzer can run a program unimpeded until the program reaches an interesting function f , and then perturb f 's input, using checkpoint and rollback to re-execute f many times soundly (i.e. while holding all other state constant). Similarly, most tools that perform code synthesis or automated program repair [31, 28, 40, 45] benefit greatly from speculative execution – testing whether the generated code meets the correct post-conditions, and, if not, resetting the program state to generate a more suitable replacement.

Typically, these CR tools rely on support from the operating system (OS), such as POSIX `fork` and various memory management functions. To perform a checkpoint, a CR tool write-protects all pages that the application uses. When the application modifies its memory, the OS notifies the CR tool, which copies the application's data as needed. Alternatively, an application can perform checkpoints by forking and resuming execution on the child process. Due to the copy-on-write nature of the `fork` system call, the parent process holds a checkpoint of the heap. Later, to rollback, the child terminates, effectively discarding its changes to the program state; and the parent forks again, resuming execution on a new child with the program state at the time of the original checkpoint (i.e. `fork`).

Although both of these approaches impose no overhead in the steady state (when no checkpoint is performed), they are inefficient when checkpointing many sparsely populated pages [18]. That is, even though an application may overwrite only 4KB in total, such a CR tool may need to copy up to 16MB if the application overwrites a single byte on 4,000 different pages. Furthermore, mapping the state of an OS-level checkpoint back to the JVM (e.g., for comparing two different executions) is a complex problem in itself [12]. Rather than rely on OS support for lightweight checkpoints, we examine the case of checkpointing in managed language runtime environments, specifically, the Java Virtual Machine (JVM).

Prior work in JVM checkpointing required a specialized, custom JVM [18, 26, 9], or developer support [55]. Our goal is to provide efficient, fine-grained, and incremental checkpoint support within the JVM, using only commercial, stock, off-the-shelf, state-of-the-art JVMs (e.g., Oracle HotSpot and OpenJDK). Guided by key insights into the JVM Just-In-Time (JIT) compiler behavior and the typical object memory layout, we present CROCHET: Checkpoint ROllbaCk with lightweight HEap Traversal for the JVM. CROCHET is a system for in-JVM checkpoint and rollback, providing copy-on-access semantics for *individual variables* (on the heap and stack) that imposes *very low steady-state overhead* and requires no modifications to the JVM. CROCHET allows developers to checkpoint either: (1) the state reachable from all current heap roots (i.e. static fields, stack pointers, even objects held by the garbage collector for finalization), or (2) an object graph encapsulated by a few well identified roots (e.g., a list encapsulated by its head as root). CROCHET also can manipulate active stack frames, allowing it both to checkpoint values on the stack and to resume execution from a rollback (restoring the entire stack, creating and destroying frames as necessary). Moreover, CROCHET is thread-safe, and fully automatic. It allows developers to checkpoint or rollback dynamically at *any time* in execution, without requiring any advance annotations or restrictions on what data to include in that checkpoint.

At its core, CROCHET uses a novel *lazy heap traversal* algorithm that provides a general-purpose page-fault-like mechanism within the JVM, generating traps at the granularity of individual objects which can be enabled/disabled dynamically to checkpoint very large object graphs *in parallel* with the program’s execution and *without* pausing all threads while the checkpoint takes place. We demonstrate that CROCHET shows negligible runtime overhead in a steady state on both Oracle’s HotSpot JVM and OpenJDK, and reasonable performance to checkpoint and rollback an application.

We describe the design and implementation of a prototype of CROCHET that does not require any Java-specific features, and is thus directly applicable to any JVM-based language. Through bytecode rewriting, CROCHET leverages a novel deployment of automated *proxy types*, allowing it to checkpoint and rollback individual objects very efficiently and with very little steady state overhead (ranging from no overhead to 1.29x slowdown on the DaCapo benchmark suite [7]). By paying this marginal steady state overhead, CROCHET can create fine-grained checkpoints very efficiently (average overhead of 1.49x to checkpoint each benchmark state), often outperforming a state-of-the-art process-level checkpoint system (CRIU [16], average overhead of 2.25x to perform the same checkpoint).

In summary, the main contributions of this paper are:

- A general purpose approach to modify the runtime behavior of live objects in a JVM with very low overhead. This approach could be used to enable general ‘run-once’ dynamic analyses that are enabled infrequently and impose very low overhead when disabled.
- A general and efficient approach to checkpoint and restore the heap and stack state of a running application in a JVM.
- A detailed description and an extensive evaluation of our open-source implementation of this technique, CROCHET.
- Several case studies on possible applications that benefit directly from our approach.

2 Design

We set out to design CROCHET with several key goals in mind:

Goal 1 Require no modifications to the JVM itself;

Goal 2 Provide very low runtime overhead when a checkpoint/rollback is not in progress, and only a minimal slowdown when doing so;

Goal 3 Provide efficient checkpoints (i.e. copy only the data needed);

Goal 4 Allow developers to request a checkpoint or rollback at any arbitrary time.

With CROCHET, developers decide (dynamically) to checkpoint all heap and stack structures, only heap roots, or only a specified set of objects using the following high level interface (respectively): `checkpointAllRoots()`, `checkpointHeapRoots()`, and `checkpoint(Objec... objects)`; and matching `rollback` functions. We expect that checkpoints and rollbacks will all occur within a single, continuously running JVM. We could imagine CROCHET being extended to asynchronously flush its checkpoints to disk, allowing rollbacks to occur in a separate process. We do not intend to directly support checkpointing state outside of the JVM (e.g. files and network connections), since if such behavior were desired, we could easily integrate existing systems (e.g., versioning filesystems).

CROCHET’s design is heavily influenced by our primary self-imposed constraint (Goal 1): it must operate entirely within the bounds of the API exposed by the JVM, without requiring any modifications to the JVM itself. Before presenting CROCHET, we first present three strawman approaches for implementing checkpoint and rollback within the JVM that fail to reach all of these goals. The simplest approach – **Strawman 1** – is to pause execution of

all threads immediately upon call to `checkpoint`, collect all variables, copy them, and then resume execution. Upon rollback, pause all threads again, and replace all variables with the previously collected copies. This simple approach trivially satisfies Goals 1, 2 and 4: from the time that `checkpoint` is called, all writes are subject to being replaced by their original values. However, this approach is inefficient, copying every single variable in the JVM, including those that may not ever be changed. Moreover, Strawman 1 pauses all threads in the JVM for the duration of an arbitrarily large checkpoint, which clearly defeats Goal 3.

Strawman 2 provides a lazier approach: Guard all data accesses (i.e. field reads/writes, array reads/writes, local variable reads/writes), checking at the time-of-access if the variable needs to be saved or restored, and then doing so. The lazy Strawman 2 likely requires far less storage, as it copies only the minimum set of variables that change after the checkpoint. Similarly, Strawman 2 can be implemented with per-object locking, allowing other threads to make progress while they are not touching the same objects being checkpointed. However, Strawman 2 introduces prohibitive runtime: before any read or write, Strawman 2 needs to check if a checkpoint or rollback is taking place, and, if so, if the variable being accessed should be copied. Intercepting all field accesses this way can introduce up to 50% overhead on steady-state [42]. Therefore, Strawman 2 defeats Goal 2 by imposing a constant performance overhead, even when no checkpoint or rollback is occurring.

The copy-on-access semantics of Strawman 2 are similar to those that an Operating System (OS) uses when executing the `fork` system call: After `fork` is called, the child process shares its memory pages with the parent process (albeit with copy-on-write semantics). If the child attempts to write to any of those pages, a page fault is trapped and the page is copied and mapped to the child. For an OS, page access checks already occur as part of the memory address translation process, regardless of whether `fork` has been called or not. **Strawman 3** improves on **Strawman 2** by using OS-level `fork` support to provide inexpensive CR. However, `fork` does not duplicate all parent threads (only the forking thread is alive in the child); and all kernel state about a process, which results in some state being shared between parent and child (e.g., `epoll` descriptors).¹ Furthermore, mapping OS-level page fault handlers to variables in the JVM is not trivial. Dealing with these two issues would surely require modifications to the JVM, and likely to the OS, thus defeating Goal 1 (some JVM migration techniques do exactly this [9]). Moreover, if the objects that need to be saved populate many pages sparsely, this approach copies much more data than strictly necessary [18], thus defeating Goal 3.

CROCHET leverages an observation that the JVM already performs various checks before accessing data, and that we can exploit these checks. When performing dynamic dispatch for methods and fields overridden by several different classes, the JVM must decide which concrete implementation of the method/field to choose. For instance, the JVM selects different methods on the same callsite for method `toString`, depending on the type of the receiver object (e.g., `Integer` versus `LinkedList` versus `Object`). Even if the JVM can prove that a call site is monomorphic, it relies on profiling data to predict the likely receiver type. Further, monomorphic call sites can become polymorphic due to class loading. Hence, when the JVM optimizes a (non-static) call site or field access, rather than directly linking a specific method to call or field to access, it maintains instead a small lookup table, to point from class types to the specific code to be invoked.²

¹ <https://lkml.org/lkml/2007/10/27/25>

² <https://wiki.openjdk.java.net/display/HotSpot/PerformanceTechniques> provides a nice summary of JVM method optimization techniques

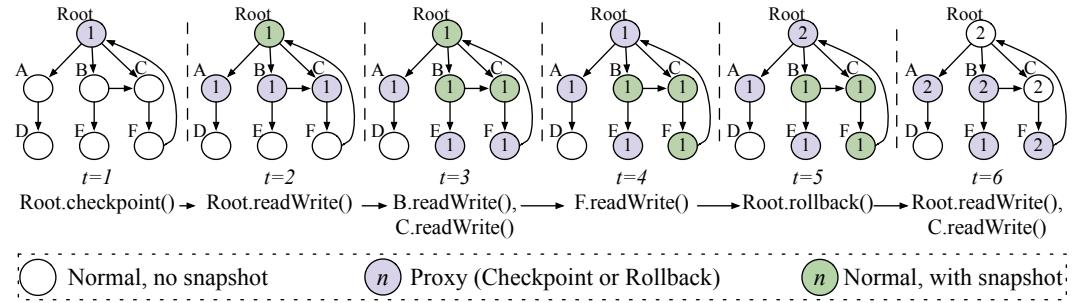


Figure 1 High-level operation of CROCHET, showing the lazy traversal and propagation algorithm in six steps, as objects are manipulated after a checkpoint and a rollback.

Importantly, this means that the JVM already issues checks at every field and method access – and CROCHET exploits these checks³. CROCHET adds an empty method, `onReadWrite`, to each class and instruments all field accesses to call that method first. This empty `onReadWrite` method is then inlined by the JVM, effectively optimizing away any invocation overhead. This results in negligible steady-state overhead (often no overhead on DaCapo, ranging up to at worst, 1.13x steady-state slowdown). Later, when performing checkpoints, CROCHET generates *proxy classes* that extend the original classes and override the empty `onReadWrite` method with specific behavior (copying the object and propagating the traversal). CROCHET can then turn an object into a proxy simply by changing its type from its normal class *C* to the corresponding proxy class *C_p* (we explain the surprisingly simple mechanism to change the type of an object in §4.2).

Rather than change every object into a proxy object when checkpoint or rollback are invoked (which would require pausing all threads in order to do soundly), CROCHET performs a *lazy heap traversal*, as shown in Figure 1. During this traversal, every object is in one of three states: *Normal*, *Checkpoint* or *Rollback*. To start a traversal, CROCHET first transforms all heap roots into their proxy types by calling `Checkpoint` (transforming to *Checkpoint*) or `Rollback` (transforming to *Rollback*) on each root. Proxied objects have a special behavior when they are read or written (as shown in the right half of Figure 1): first, the object checkpoints (or rolls back) itself, and then it transforms all objects directly reachable by it into their corresponding proxy type.

Figure 1 shows how this lazy traversal propagates in a heap with one root object (*root*) and six other objects (*A* – *F*). A checkpoint takes place at instant *t* = 1, followed by manipulating (i.e. reading and/or writing) the *root* at *t* = 2, then objects *B* and *C* at *t* = 3, and *F* at *t* = 4. A rollback takes place at *t* = 5, followed by manipulating the *root* and *C* at *t* = 6. The background denotes the state of each object: white for regular objects without a snapshot present, purple for proxies, and green for objects with a snapshot present. Besides its state, each object *O* keeps a version counter with the number of the most recent traversal that reached *O*. Note that after each checkpoint/rollback there is always at least one proxy between the *root* and each object not yet reached by the traversal. This is in fact one of three invariants that are key for CROCHET’s correctness, as we elaborate in §3.

CROCHET uses a wait-free traversal algorithm to propagate checkpoint proxies, allowing it to safely and efficiently perform its operations in multi-threaded code; rollback operations are blocking, but due to the locality of Java objects, often experience little to no contention

³ Notable exceptions are private or static field/method accesses.

(§6). Further, since these proxy objects remove themselves from the object graph, most of the steady-state overhead that such proxies would otherwise introduce is eliminated, often resulting in zero runtime overhead in the absence of checkpoint or rollback operations (§5.2).

3 Lazy Heap Traversal

We now describe the algorithm that CROCHET uses to lazily checkpoint and rollback the heap. CROCHET provides simple, flat-nested checkpoints: When performing two checkpoints in a row, CROCHET discards the object graph copy of the first checkpoint and keeps the copy of the second checkpoint. Hence, when an application calls checkpoint several times, and then eventually calls rollback, such a rollback restores values captured by the last checkpoint only. We leave supporting arbitrarily nested checkpoints to future work. While CROCHET requires pausing all application threads to collect all root references, the remainder of the algorithm is mostly non-blocking.

3.1 Invariants

Throughout the entirety of program execution, CROCHET maintains the following invariants:

1. **Identity:** Version numbers are not reused between different checkpoints and rollbacks;
2. **Total order:** New checkpoints and rollbacks introduce higher version numbers;
3. **Continuity:** For every object O , either (1) O has been reached by the current traversal, or (2) there is at least one proxy object (i.e. object with status CHECKPOINT or ROLLBACK) with the highest version on every path that reaches O .

Invariant 1 (**Identity**) identifies each checkpoint and rollback uniquely. Invariant 2 (**Total order**) provides a simple total order of all the checkpoints and rollbacks that is easy to check when propagating proxies throughout the object graph. Finally, Invariant 3 (**Continuity**) ensures that proxies will mediate every first interaction with objects during a checkpoint or rollback. CROCHET maintains these invariants even in the presence of multi-threading. The continuity invariant is especially important in multi-threading, as it ensures that two threads which might be racing to checkpoint or rollback the same object will be racing to perform *the exact same operation*. An important consequence of these invariants is that a proxy object can never access objects with a version higher than itself.

Tracking the status of heap traversal. To support its lazy heap traversal, CROCHET needs to track three facts about each object: (1) its **version** (the current version of the object), (2) its **snapshot** (a copy of the object, if it has been checkpointed), and (3) its **status** (representing its status in the traversal: either CHECKPOINT or ROLLBACK, or NONE to indicate the object is not proxied). The checkpoint and rollback algorithms are implemented by automatically generated methods for each class: `onReadWrite`, `onCheckpoint`, `onRollback`, `copyFieldsTo` and `copyFieldsFrom`. CROCHET rewrites all field accesses to first call `onReadWrite`. `onReadWrite` is used to trigger checkpointing or rolling back (by calling `onCheckpoint` or `onRollback` respectively). `copyFieldsTo` and `copyFieldsFrom` are utility methods that allow CROCHET to copy the fields of each object; `propogateCheckpoint` and `propagateRollback` are the key methods used to advance the frontier of a heap traversal that we will now describe.

These methods behave differently depending on the status of the object that they are invoked on. Throughout this paper, we use dynamic dispatch notation in our examples. For

instance: Invoking `onReadWrite` on object `obj` results in method `CHECKPOINT.onReadWrite` being called when `obj.status` is `CHECKPOINT`. We make wide use of the `CompareAndSwap` (CAS) primitive, with the notation $CAS(f, v_o, v_n)$, which atomically updates field f to value v_n if f 's current value is v_o . CAS operations do not succeed if another thread updated field f from the previously observed value v_o . Finally, we omit similar methods for the sake of brevity; a complete reference containing all of the methods appears in Appendix A.

3.2 Algorithm for checkpointing

Figure 1 provides a high-level outline of how CROCHET's lazy traversal works, we will refer again to it in order to describe the checkpoint algorithm in detail. For simplicity, we first present the algorithm assuming a single-threaded execution (in which all compare-and-swap operations succeed); we will later provide a thorough argument about thread-safety in §3.4. To checkpoint the heap, the applications calls method `onCheckpoint` on all root references (§4.5 describes how those roots are found): object `root` in our example, which is in the `NORMAL` state at this point. Listing 1 shows the pseudo-code for method `NORMAL.onCheckpoint`. This is a fast operation that simply turns objects into proxies by changing their state (line 8), thus deferring the snapshot until it is needed.

In terms of the three invariants described above: CROCHET automatically manages version counters, based on how many times checkpoint or rollback operations have been started (on a global count), making Invariant 1 (Identity) and 2 (Total order) easy to enforce. Line 7 updates the version before updating the state, which does not violate Invariant 3, as objects only become proxies with the correct version. The opposite order would create a window during which a proxy would have a version lower than the highest, thus violating Invariant 3.

Once all root references are turned into proxies this way, Invariant 3 is established and the program can resume execution ($t = 1$). Next, the program manipulates the `root` object ($t = 2$), which triggers the invocation of method `CHECKPOINT.onReadWrite`. This method creates a snapshot of the object (lines 19–22); then propagates the checkpoint to all fields (line 26), effectively pushing the frontier of proxies one level forward in the object graph; and, finally, makes the object not a proxy (line 28). The last step does not violate Invariant 3 because all objects referred to by fields are now proxies themselves.

The program keeps executing, propagating proxies as it manipulates more objects. In our example (Figure 1) the program manipulates objects `B` and `C` in that order at $t = 3$; and object `F` at $t = 4$. Note that manipulating object `B` leads to invoking method `CHECKPOINT.onCheckpoint` on object `C`, which already is a proxy. This method is a fast operation that simply updates the version number when needed. In this case, it simply exits on line 33. However, consider if the example instead issued another checkpoint at $t = 2$, and then manipulated the `root` object. In that case, CROCHET would propagate new proxies to outdated proxies (object `A – C`), simply updating their version on line 35. Finally, note that manipulating object `F` turns the `root` back into a proxy at $t = 4$, as the check on line 4 fails. This behavior is correct but inefficient; we present an optimization to avoid this case in §3.5.

3.3 Algorithm for rolling-back

Performing a rollback is the dual of performing a checkpoint. At time $t = 5$, the program uses CROCHET to rollback to the earlier checkpoint. Similarly to the checkpoint, CROCHET starts the rollback by calling method `NORMAL.onRollback` on the `root` reference; which is the same as method `NORMAL.onCheckpoint`, but replaces `CHECKPOINT` with `ROLLBACK`. Again, Invariant 3 is established once all root references are turned into proxies.

17:8 CROCHET: Checkpoint/Rollback via Lightweight Heap Traversal on Stock JVMs

```

1 NORMAL.onCheckpoint(int version){
2     int curV = this.version;
3     if (curV==version &&
4         this.status==CHECKPOINT)
5         return;
6
7     CAS(this.version,curV,version);
8     CAS(this.status,NONE,CHECKPOINT);
9 }
10
11 CHECKPOINT.onReadWrite() {
12     int curV = this.version;
13
14     Object snap = this.snapshot;
15     if (snap==NULL ||
16         snap.version<curV) {
17         // Allocates empty object
18         // Without running constructor
19         Object newSnap = ...
20         this.copyFieldsTo(newSnap);
21         newSnap.version = curV;
22         CAS(this.snapshot,snap,newSnap);
23     }
24
25     for (Field f in this)
26         f.onCheckpoint(curV);
27
28     CAS(this.status, CHECKPOINT,
29          NORMAL);
30
31 CHECKPOINT.onCheckpoint(int vers) {
32     int curV = this.version;
33     if (curV == vers) return;
34
35     CAS(this.version,curV,vers);
36 }
37 ROLLBACK.onReadWrite() {
38     int curV = this.version;
39
40     Object snap = this.snapshot;
41     if (snap != NULL &&
42         snap.version<curV) {
43         synchronized (snap) {
44             snap = this.snapshot;
45             if (snap != NULL &&
46                 snap.version<curV) {
47                 this.copyFieldsFrom(snap);
48                 snap.version = curV;
49             }
50         }
51     }
52
53     for (Field f in this)
54         f.onRollback(curV);
55
56     CAS(this.status, ROLLBACK, NORMAL
57 );
58
59 ROLLBACK.onCheckpoint(int vers) {
60     int curV = this.version;
61     if (curV==vers &&
62         this.status==CHECKPOINT)
63         return;
64
65     this.onReadWrite();
66
67     CAS(this.version,curV,vers);
68     CAS(this.status,ROLLBACK,
69          CHECKPOINT);
70 }

```

Listing 1 Pseudo-code for checkpoint algorithm.

Listing 2 Pseudo-code for rollback algorithm.

After this invocation, the program manipulates the *root* object at $t = 6$, thus invoking method `ROLLBACK.onReadWrite`. At this time, CROCHET reverts the state of the *root* object to the snapshot saved at $t = 1$ (lines 40 – 51). Then, CROCHET propagates proxies one level into the object graph (lines 53–54). Finally, the algorithm makes the *root* object not a proxy (line 56), which, as before, does not violate Invariant 3. Continuing the example, the program then manipulates object *C*, thus causing it to be rolled back to the saved version and propagating the proxies one more level in.

To understand the need for line 65, consider the case of the program issuing another checkpoint, at $t = 6$; and manipulating the *root* object. This results in CROCHET invoking method `ROLLBACK.onCheckpoint` for proxied object *B*, which must be rolled back to the snapshot taken at $t = 3$. Line 65 thus performs the needed rollback; the rest of the method is similar to method `NORMAL.onCheckpoint`.

Some methods are very similar and are thus omitted: method `NORMAL.onRollback` is similar to `NORMAL.onCheckpoint` as explained above, method `ROLLBACK.onRollback` is similar to `CHECKPOINT.onCheckpoint`, and method `CHECKPOINT.onRollback` is the same as method `NORMAL.onRollback`. A complete reference containing all of these methods appears in Appendix A.

3.4 Thread safety

Before scanning for root references, CROCHET pauses all threads to call checkpoint/rollback on each root, and then resumes all threads. Thereafter, multiple threads may race to perform the same checkpoint or rollback of the same (non-root) object, but it is impossible for different checkpoints and rollbacks to race with each other, or for the underlying program to race with the checkpoint/rollback of root references. We outline here a brief argument for CROCHET’s thread safety, but leave a formal proof to future work. For additional support, our evaluation (§5) extensively tested checkpoint/rollback on multi-threaded applications.

CROCHET uses atomic compare-and-swap (CAS) operations to update values that are visible to other threads. For instance, when checkpointing a non proxied object in `NORMAL.onCheckpoint`, the algorithm first checks if there is any work left to do (line 3). If the check fails, the algorithm uses two CAS operations to update the version and the status from their expected values. A failed CAS just means that another thread performed that CAS; no recovery operation needs to take place, and no CAS’ need to be retried.

Note that it is possible that another thread manipulates the same object between the check on line 3 and the first CAS at line 7 more than just performing the two CAS operations. For instance, the other thread may have already created a snapshot of this object by calling `onReadWrite` immediately after `onCheckpoint`, setting the status back to `NONE`. This interleaving is safe as the CAS operation on `onCheckpoint` that updates the version at line 7 will always fail, thus preventing the object from reaching an inconsistent state in violation of an invariant. Unfortunately, the object may cycle between states `CHECKPOINT` and `NORMAL`, which results in wasted work and increased overhead; §3.5 presents an optimization that addresses this problem.

The same argument for thread safety applies to `CHECKPOINT.onCheckpoint`, `CHECKPOINT.onRollback`, `NORMAL.onRollback`, `ROLLBACK.onCheckpoint` and `ROLLBACK.onRollback`. The argument for thread-safety when performing a checkpoint is straightforward. First, the algorithm creates a snapshot, which is private to each thread (line 19). Then, all threads race to update the object snapshot to their private copy with a CAS operation (line 22). One thread wins and keeps its snapshot, all the other threads discard their (equivalent) snapshots.

If an object is arbitrarily changed during the snapshotting process (which results in an invalid snapshot), CROCHET discards that snapshot. Consider two threads racing to update an object’s fields: both threads may race through `CHECKPOINT.onReadWrite`. Neither thread will be able to update the object’s fields until `CHECKPOINT.onReadWrite` returns. For one thread to mutate that object’s fields during the other thread’s snapshot stage, the first thread must have completed `CHECKPOINT.onReadWrite`, and recorded its snapshot. Hence, the second (invalid) snapshot is discarded, since it failed its CAS operation at line 22.

Then, the algorithm propagates the checkpoint to all the objects `obj` refers to (line 26). Invariants 1 and 2 ensure that all operations are idempotent, so it is safe for several threads to propagate the checkpoint, even if the object changes as explained in the previous paragraph. Finally, the checkpoint algorithm can revert the status back to `NORMAL` (line 28), because the previous loop ensures that Invariant 3 is not broken by doing so: the frontier of proxy objects has advanced on level forward in the object graph.

The argument for thread-safety when performing a rollback is more complex. It is similar to performing a checkpoint, but overwriting the object with its snapshot is not idempotent: Consider two threads T_1 and T_2 racing to rollback the same object. T_1 performs the rollback to completion and then executes application code that writes to a field of the object. Then, T_2 is scheduled and erroneously overwrites the object once again.

```

1 NORMAL.onCheckpoint(int v) {
2     int curV = this.version;
3     if (curV == version) return;
4     //realV =(curV < 0) ? curV*-1 : curV;
5     CAS(this.version, curV, v*-1);
6     CAS(this.status, NORMAL, CHECKPOINT);
7     CAS(this.version, v*-1, v);
8 }

```

■ Listing 3 Optimization to avoid redundant proxy propagation.

The algorithm avoids such erroneous executions by acquiring a monitor on the snapshot (line 43). This allows the first thread to overwrite the object with its snapshot *and* set the snapshot to `null` *in one atomic step*. Contending threads, after acquiring the monitor, realize that the object is already rolled back (line 46) and proceed without changing anything.

The rest of the rollback algorithm is similar to the checkpoint algorithm and the same argument for thread-safety applies. Note that the progress condition of the checkpoint algorithm is *wait-freedom* because, regardless of scheduling, there is always a finite bound on the number of steps for a snapshot to be created and, therefore, for the algorithm to make progress. The algorithm for rollback is, of course, *blocking* due to the use of monitors. In the future, we plan to remove monitors from the algorithm and we speculate that the resulting algorithm will be *lock-free*, and not wait-free, due to the non idempotent nature of rollback.

3.5 Optimizations

The algorithm presented in Listings 1 and 2 is correct, but not efficient. In particular, the condition that detects if any work is needed for an object on line 3 fails to detect the case in which an object is re-discovered when propagating a checkpoint. Consider the following example, again from Figure 1: At $t = 4$, the program manipulates a field of Object F , calling `NORMAL.onCheckpoint` on the *root* object. At this point, there is no more work to be done for the *root* object. Yet, the early return on line 4 fails to detect this case, and proxies the *root* object, which will repeat the whole sequence of proxy propagations from $t = 1$ until $t = 4$. Our early experiments showed that this case happens frequently in practice and we added an optimization to improve the efficiency for this common case.

Ideally, checking the version of an object should be enough to decide if there is work to be done. However, removing the second check on line 4 is not safe. Suppose thread T_1 is scheduled out of execution after updating the version of an object o on line 7 but before updating its status on line 8. Thread T_2 now reads the updated version and decides that o does not require more work to be done. The objects that o refers to are now accessible to thread T_2 without a proxy between them and a root reference, i.e. Invariant 3 does not hold.

This problem can be solved trivially by updating both status and version in one atomic step (i.e. double-CAS). Given that support for such an atomic operation is not common, our algorithm uses the optimization shown in Listing 3 instead. The idea is to update the version with an intermediate value first, then update the status, then update the version to the correct value. Any thread that sees the intermediate version can extract the correct version from it. We applied the same optimization to all early return checks on the algorithm (lines 5, 33, and 63). When the actual version is needed (i.e. further uses of variable `curV` outside of a CAS), the pseudo-code on Listing 3 shows how to extract it into variable `realV`.

```

1 class OriginalClass { // Original class
2 OriginalClass f1; int f2;
3 int sum() { return f1.sum() + f2; }
4 }
5 class OriginalClass { // Generated NORMAL class
6 OriginalClass f1; int f2;
7 int sum() { f1.onReadWrite(); return f1.sum() + f2; }
8
9 int \[version; OriginalClass \]snapshot;
10
11 void \[onReadWrite() { /* empty */ }
12 void \]onCheckpoint(int v) { NORMAL.onCheckpoint(v); }
13 void \[onRollback(int v) { NORMAL.onRollback(v); }
14 }
15 class OriginalClass\]PROXY extends OriginalClass {
16 // Generated PROXY class, extends the generated NORMAL class above
17 void \[onReadWrite(){ if (version % 0) ROLLBACK.onReadWrite(this);
18 else CHECKPOINT.onReadWrite(this); }
19 void \]onCheckpoint(int v); // Similar to onReadWrite
20 void \[onRollback(int v); // Similar to onReadWrite
21 }

```

Listing 4 An example class `OriginalClass`, shown before (top) and after (bottom) processing by CROCHET, including generated classes. Receivers `NORMAL`, `CHECKPOINT`, and `ROLLBACK` refer to the algorithms that will follow in Listings 1 and 2.

4 Implementation

CROCHET is implemented entirely within the confines of the JVM. Most of CROCHET is implemented through bytecode instrumentation using ASM [8], but some small components are written in C (mostly for stack introspection and manipulation), using the standard JVMTI [36] interface. While CROCHET can dynamically instrument bytecode on-the-fly (as it is loaded into the JVM), it is necessary to bootstrap the system by (automatically) instrumenting a complete copy of the JVM offline.

4.1 Class modifications and instrumentation

Listing 4 shows how CROCHET modifies the original bytecode of a Java program (shown as source code for the sake of clarity). CROCHET adds two fields to every class: (1) `version`, (2) and `snapshot`. To track the `status` of each object, CROCHET generates a proxy class that extends the original class and overrides the methods `onReadWrite`, `onCheckpoint`, and `onRollback` with the appropriate behavior. To update the value of field `status` of an object, CROCHET changes the class to which that object belongs. As an implementation optimization, CROCHET generates a just single proxy class for both `CHECKPOINT` and `ROLLBACK` and then uses the `version` to decide which state the object is in: Even numbered versions are `ROLLBACK`, odd numbered versions are `CHECKPOINT`. This reduces the overall number of classes that CROCHET needs to generate, which improves performance by minimizing the number of invocation targets that the JVM needs to consider at each call site.

CROCHET uses `sun.misc.Unsafe.defineAnonymousClass`, which loads classes quicker than regular classloaders⁴, performing load-time patching of a class's constant pool to

⁴ Java 8 implements lambda routines through `defineAnonymousClass`: https://blogs.oracle.com/jrose/entry/anonymous_classes_in_the_vm

efficiently define proxy classes at run time. CROCHET uses a single definition for its proxy class and patches it to load proxy classes that extend different host classes without requiring any additional class generation. CROCHET rewrites the bytecode of the reflection layer and `sun.misc.Unsafe` to intercept field accesses and insert calls to `onReadWrite` dynamically, as required for the algorithm to operate correctly. Similarly, CROCHET’s runtime library intercepts reflective calls that inspect various classes to hide its presence (e.g. masking the extra fields and methods that it creates).

4.2 Changing object types

CROCHET is able to change the class of an existing object by leveraging an observation about the JVM object layout, building on our recent work in Dynamic Software Update systems for Java. Rubah [43] found that the type of an object could be changed at runtime by overwriting the header of that object, replacing the value stored in the *klass* sector. To ensure compatibility, the new class type must have an identical number and layout of fields to the old, but there are otherwise effectively no other restrictions [41]. This change can be made at any point, subject to the limitation that the code that modifies the object cannot become inlined with other code that needs to know its type (which is easily avoided by ensuring that the code making the swap is a sufficiently large method). Hence, CROCHET transitions objects from their regular definition into their proxy type using `sun.misc.Unsafe`’s `.putInt` function. The *klass* value is stored 8 bytes into the object header: CROCHET simply replaces the *klass* on each object with the desired *klass*. The format of the object header is well documented [35], and represents the tightest coupling of CROCHET to the JVM, although we expect CROCHET to be adjustable to eventual object layout changes. CROCHET caches instances of both normal objects and their corresponding proxy object to allow it to always be able to map between the desired Java Class and the corresponding *klass* value. CROCHET does not cache *klass* values, as they are subject to change as classes are loaded and unloaded. In addition to the evaluation performed by Rubah [43] of this mechanism, we validated it empirically on the most common JVMs (Oracle and OpenJDK), and found it to hold.

4.3 Static Fields

Static fields are heap roots and pose a unique challenge as they are accessed directly without a receiver object that can be proxied. CROCHET must detect when a static field is first dereferenced (which would then cause it to be copied, either through checkpointing or rolling back). A naïve solution *immediately* checkpoints or rollbacks all static fields, which does not require CROCHET to monitor static fields. However, to do so safely, CROCHET would perform these copies while the entire application execution is paused, likely creating significant performance overhead. Instead, CROCHET wraps accesses to static fields using a special helper class, a `StaticFieldHelper`, which allows it to lazily detect when static fields are accessed. There is one `StaticFieldHelper` generated for each class, which has slots for all of the original class’ static fields to store them as regular instance fields. The `StaticFieldHelper` thus allows CROCHET to treat static fields as any other field: the `StaticFieldHelper` implements all of the methods (e.g., `onCheckpoint`, `onRollback`, etc.) that any other class would, and maintains its own proxy state (*Normal*, *Checkpoint* or *Rollback*). CROCHET generates these helper classes on-the-fly, storing the instance of the class in a static member field of each class for efficient retrieval, and rewrites all bytecode to use the instance fields of static helpers instead of the original static fields.

4.4 Wrapping arrays and non-instrumentable types

CROCHET relies on adding fields and methods to classes in order to modify the behavior of all of the various references that can exist in the JVM to perform its lazy heap traversal. Unfortunately, it is *not* possible to modify the behavior of all possible references directly; in particular, references from an array to its elements, or from an object that CROCHET was unable to modify for other reasons. Arrays in the JVM are lists of contiguous references and have an associated class to represent their type, but that class cannot be modified, and there is no lightweight mechanism to directly apply the proxy concept outlined above. Similarly, there are a small number of classes that cannot be modified due to tight coupling from the JVM internals to the class layouts (e.g., native code accessing hard-coded field offsets in `java.lang.Double`, `java.lang.Object`) [4].

For all non-modifiable types (objects or arrays), CROCHET: (1) creates wrappers that track the state of that reference type, and (2) checkpoints and rollbacks these instances eagerly when they are discovered in a traversal. CROCHET maintains a relatively performant (and thread-safe) lookup table between the non-modifiable instance and its corresponding wrapper using JVTI's object tagging. Typically in our traversal, accessing a proxied object O_1 with field f pointing to object O_2 would cause O_1 to be copied, and O_2 to become a proxy. However, if O_2 is actually an instance of a non-modifiable type, then O_2 is copied immediately, and anything that O_2 points to is transitioned into a proxy. By eagerly propagating proxies into these types, CROCHET accesses the wrapper only during a checkpoint or rollback traversal. These eager checkpoints and rollbacks are implemented using the same algorithm as for other types, and the same thread safety argument applies to them.

4.5 Finding the Root References

To perform a *complete* heap traversal in the JVM, CROCHET needs to first identify all of the various roots that point into the heap. CROCHET considers as roots: (1) static fields of loaded classes, (2) objects in finalizer queues, (3) live instances of `java.lang.Thread`, and (4) objects referenced in, values held by, and monitors held by stack frames. The simplest type of root to collect are static fields: CROCHET simply enumerates all initialized classes, collecting their static field helpers and transitioning them into proxies. CROCHET performs its traversal of objects in finalizer queues by special-casing the finalizer queue itself, causing it to propagate proxies directly to its referees when they are accessed (rather than only when checkpoint is called). This way, CROCHET can correctly traverse objects as they are removed from the finalizer queue, regardless of what had been previously held a reference to these objects. Each live thread in the JVM has a corresponding `java.lang.Thread` object: CROCHET transitions them all into proxies. Stack references are handled as described below.

4.6 Stack references

CROCHET supports three configurations for handling stack state: (1) capture complete stack state of all threads (the variables in each stack frame, plus the complete stack trace), allowing for execution to be rolled back to that same exact instruction; (2) capture complete state for the current stack frame *only*, allowing execution to be rolled back only within this same method execution; or (3) capture only (developer-specified) stack variables of the current stack frame. In the Configuration 1, CROCHET checkpoints stack variables and the stack trace of each thread, and can roll the application back to an identical state (reproducing the same exact stack trace). CROCHET also captures all active monitors held by each thread, and at rollback ensures that those threads hold exactly the same monitors. Configuration 2

```

1 //Original Code
2 void someFunc(int i, int[] ar)
3 {
4     int j = i + 1;
5     ar[i] = ar[i] - j;
6     j--;
7     otherFunc(j, ar); //Checkpoint
    is called by otherFunc
8     ar[i] = 10;
9 }

```

Listing 5 An example function, `someFunc`, that will be in the stack trace when a checkpoint and is called (somewhere deeper in the stack, within the invocation of `otherFunc`.

```

1 //Checkpoint code
2 void someFunc(int i, int[] ar)
3 {
4     boolean captureStack = false;
5     int j = i + 1;
6     ar[i] = ar[i] - j;
7     j--;
8     otherFunc(j, ar);
9     if(captureStack)
10     Checkpointer.captureStack();
11     ar[i] = 10;
12 }

```

Listing 6 The same code from Listing 5, but with the checkpoint code added. When `checkpoint` is called, CROCHET sets the `captureStack` boolean variable in each method active to `true`, causing it to capture stack variables. Not shown: code to capture active values on the operand stack.

```

1 //Rollback code
2 void someFunc(int i, int[] ar)
3 {
4     int j;
5     boolean captureStack = false;
6     if(Rollbacker.doRollback())
7     {
8         i = Rollbacker.localInt();
9         ar = Rollbacker.localIntArray();
10        j = Rollbacker.localInt();
11        Rollbacker.removeRollbackCode();
12    }
13    else
14    {
15        j = i + 1;
16        ar[i] = ar[i] - j;
17        j--;
18        otherFunc(j, ar);
19        if(captureStack)
20            Checkpointer.captureStack();
21    }
22    ar[i] = 10;
23 }

```

Listing 7 The same code as shown in Listing 5, with dynamically generated rollback instrumentation. `doRollback` dynamically decides if the current invocation of this function `someFunc` should jump-forwards; `removeRollbackCode` determines dynamically if this method body has no more rollbacks to perform, and if so, removes the specialized rollback code using bytecode HotSwap. Not shown: code to restore active values on the operand stack.

is a relaxation of Configuration 1: rather than capture *all* stack variables, only the variables of the method that calls `Checkpoint` need to be captured. Configuration 3 is a further relaxation: CROCHET only calls `Checkpoint` on a pre-defined set of variables.

The call stack holds method invocations and their frames; the JVMTI tooling interface [36] allows CROCHET to manipulate (non-native) call stack frames (i.e. read and write local variables, method arguments, and held monitors; and pop frames; similar to how a debugger would do the same). Each stack frame includes an *operand stack*, which is used to pass arguments to JVM instructions and read their results. The operand stack is not accessible by any JVM debugging or reflection interface, so CROCHET accesses and manipulates it through bytecode instrumentation.

Checkpointing: CROCHET pauses all threads and creates a copy of the local variables and operand stack on every call stack frame, calling the normal `onCheckpoint` function for reference types. CROCHET also records each monitor (lock) held in each stack frame. In Configuration 2 (checkpointing only the calling frame), CROCHET makes these transformations only in the calling frame; in the case of the Configuration 3 (checkpointing only selected variables), CROCHET makes none of these transformations. Listings 5 and 6 show an example of how CROCHET modifies the program to support stack checkpoints. CROCHET inserts an extra flag per method as a local variable and checks it after every method invocation. If

the flag is set, the injected code will capture the current operand stack to local variables, checkpoint all local variables (which now includes the operand stack), reset the flag, and restore the operand stack as needed. To checkpoint the stack, CROCHET pauses all threads, toggles this flag for all active frames, and resumes all threads. Each active method will now see the flag set, checkpoint the relevant data, and then continue to execute. If the flag is set in a stack frame that is never re-activated before rollback (e.g. it is deep in the call stack and not returned to), then it would never be possible for those stack variables to have been modified, and hence CROCHET will never checkpoint it. CROCHET thus records the complete stack trace of each thread along with the variables in each stack frame.

Rolling back. In configurations 2 and 3 (which only involve checkpointing in the stack frame that called checkpoint), rolling back is straightforward: CROCHET calls `rollback` on each of the previously checkpointed objects and resets their values. To support Configuration 1 (capturing the complete stack), CROCHET pauses all threads and pops all stack frames so that they can be overwritten by the checkpointed stack. CROCHET then transforms the code for each method on the checkpointed stack trace (using Java’s `ReTransformClass` functionality [37]), adding “roll-forward” code to skip all instructions until reaching the correct method invocation (active at the time of the checkpoint), and then restore all local variables and the operand stack with the values on the checkpointed frame. Listing 7 shows an example of this transformation (without locks or operand stack values). Then, CROCHET resumes each thread in sequence, guiding it to the correct point in execution, through calls of method `onRollback` on objects that are replaced on the stack, and then pauses it again. Once the rollback is complete (and all threads have stack frames equivalent to the checkpoint state), CROCHET removes the generated roll-forward code from the active methods and resumes all threads.

4.7 Limitations

Our implementation of CROCHET does not capture native code behavior through JNI. Native code that reads and writes fields does not trigger those references to be traversed, checkpointed, or rolled back. Similarly, root references held by native code are not considered during traversal. During the evaluation we present in §5, we did not find these limitations to be a concern. Still, these limitations can be removed by replacing JNI functions with wrappers (similarly to how CROCHET handles reflection). CROCHET’s stack checkpointing does not handle JVM-internal threads that are invisible from Java code (e.g., compiler threads), although it does consider JVM-managed threads (e.g., finalizer threads). CROCHET’s approach is fully compatible with the JVM’s garbage collector, and functions correctly even while a garbage collection occurs.

CROCHET does not checkpoint state kept in Java class loaders. That is, if an application: (1) performs a checkpoint, (2) loads class *Foo*, and (3) performs a rollback; then class *Foo* will still be loaded. This limitation is due to tight coupling between class loaders and JVM internals. We believe that this is not a significant limitation of CROCHET, and, if desired, classes could easily be re-initialized between checkpoints using a system like VMVM [5].

Checkpoints also impact the garbage collection and finalization of objects. As expected, CROCHET must ensure that all objects reachable at the start of a checkpoint remain reachable until the matching rollback happens, by keeping a reference to each such object. Allowing these objects to be garbage collected (before a rollback) would defeat the correctness of the checkpoint/rollback semantics of CROCHET, and hence, CROCHET retains references to them.

CROCHET’s implementation relies on the “unsupported” `sun.misc.Unsafe` library. While exposed for public use, its use is discouraged by the developers of the JVM, and none of its functionality is guaranteed to continue to exist in future versions of Java. While there has been much controversy around the JDK’s support for `sun.misc.Unsafe` [32], it is still included in Java 9, and there do not appear to be immediate plans to remove or deprecate the specific functionality used by CROCHET.⁵

Our eager handling of arrays and non-modifiable types requires more copies than strictly necessary – an entire array will be copied even if only one element is overwritten. However, given the constraints imposed by the JVM, we found this to be the most efficient approach to capturing references through arrays.

Finally, as discussed in the context of our design goals (§2), we do not consider the state of a system outside of the JVM (leaked through, for instance, file descriptors or network sockets). We envision that CROCHET could be integrated directly with versioning file systems and other low level approaches to capture this state.

5 Experimental Evaluation

We have conducted a thorough evaluation of CROCHET’s performance, using both micro benchmarks that we have crafted to expose specific performance scenarios, and macro benchmarks that simulate realistic workloads on large-scale apps (such as Apache Tomcat). We set out to answer five primary research questions:

- RQ1:** What is the steady state overhead imposed by the tooling to enable CROCHET’s lightweight heap traversal?
- RQ2:** What is the cost of performing a checkpoint with CROCHET?
- RQ3:** What is the cost of performing a complete checkpoint and rollback of a real application?
- RQ4:** Does CROCHET correctly checkpoint and rollback complicated data structures in the JVM?
- RQ5:** How does CROCHET compare to state-of-the-art software approaches that provide support for checkpoint/rollback?

We conducted all of our evaluations on a machine equipped with two Intel(R) Xeon(R) CPU E5-2650L v3 (each with 12 physical cores, 24 logical) running Ubuntu 16.04 (Linux 4.4.0-121-generic) and 128GB of RAM. We used Oracle’s HotSpot JVM version 1.8.0_66, as it is the latest version that macro benchmarks *tomcat* and *eclipse* run with.⁶ On all JVM configurations we used a max heap size (`-Xmx`) of 10GB and no other JVM options.

5.1 Microbenchmarks

We used several of the collections classes provided by the Java runtime environment to measure the steady-state overhead that CROCHET introduces, the cost of performing checkpoints, CROCHET’s correctness, and how it compares to CRIU and DeepClone (RQ1, RQ2, RQ4, and RQ5, respectively). We used the following data-structures: `HashMap` (HM in Table 1), `TreeMap` (TM), and `LinkedHashMap` (LHM) from the `java.util` package; and `ConcurrentHashMap` (CHM) from the `java.util.concurrent` package.

Each benchmark execution consists of 3 steps. First, the benchmark fills the data-structure with *SIZE* entries that map random integers, *keys*, to newly created objects with no fields,

⁵ <http://openjdk.java.net/jeps/260>

⁶ https://bugzilla.redhat.com/show_bug.cgi?id=1337940

Table 1 Microbenchmark results showing run time comparison between a baseline Java 8 JVM (shown as runtime in msec with a 95% confidence interval) and the relative slowdowns imposed by CROCHET without checkpoints, and checkpoint/rollback using CROCHET (CROCHET_{CP}), DeepClone, and CRIU. The last row shows the average, minimum, and maximum overhead for each configuration.

Structure	Hotspot 8 (ms)	Relative Slowdown			
		CROCHET	CROCHET _{CP}	DeepClone	CRIU
CHM 10	65.84 (64.30, 67.39)	1.06 (1.03, 1.08)	1.35 (1.31, 1.38)	1.96 (1.90, 2.02)	1.74 (1.70, 1.79)
CHM 25	64.72 (64.34, 65.10)	1.10 (1.09, 1.11)	1.41 (1.40, 1.42)	2.03 (1.99, 2.08)	1.82 (1.80, 1.84)
CHM 50	65.18 (64.82, 65.55)	1.09 (1.08, 1.10)	1.44 (1.44, 1.45)	2.05 (2.00, 2.10)	1.83 (1.82, 1.85)
CHM 100	65.36 (64.99, 65.73)	1.11 (1.10, 1.12)	1.50 (1.49, 1.51)	1.98 (1.94, 2.03)	1.85 (1.84, 1.86)
HM 10	62.11 (61.70, 62.53)	1.08 (1.07, 1.10)	1.31 (1.30, 1.32)	1.92 (1.89, 1.94)	1.51 (1.50, 1.53)
HM 25	64.23 (63.86, 64.60)	1.04 (1.02, 1.06)	1.31 (1.30, 1.31)	1.88 (1.86, 1.90)	1.50 (1.48, 1.51)
HM 50	64.94 (64.59, 65.29)	1.04 (1.01, 1.06)	1.30 (1.29, 1.31)	1.88 (1.86, 1.89)	1.50 (1.49, 1.52)
HM 100	67.14 (66.22, 68.06)	1.01 (0.98, 1.05)	1.29 (1.27, 1.31)	1.83 (1.80, 1.86)	1.49 (1.47, 1.52)
LHM 10	51.70 (51.24, 52.16)	1.12 (1.10, 1.13)	1.54 (1.52, 1.55)	2.25 (2.22, 2.28)	2.04 (1.97, 2.11)
LHM 25	54.39 (53.86, 54.93)	1.09 (1.08, 1.11)	1.53 (1.51, 1.55)	2.15 (2.12, 2.18)	1.62 (1.57, 1.66)
LHM 50	56.54 (56.13, 56.94)	1.08 (1.07, 1.09)	1.48 (1.47, 1.50)	2.10 (2.07, 2.12)	1.62 (1.54, 1.71)
LHM 100	58.49 (57.97, 59.01)	1.07 (1.06, 1.08)	1.49 (1.48, 1.51)	2.07 (2.04, 2.10)	1.56 (1.54, 1.58)
TM 10	151.55 (149.93, 153.17)	1.03 (1.01, 1.04)	1.09 (1.07, 1.10)	1.45 (1.43, 1.48)	1.35 (1.33, 1.37)
TM 25	153.20 (151.34, 155.06)	1.02 (1.01, 1.04)	1.12 (1.10, 1.14)	1.45 (1.43, 1.48)	1.36 (1.34, 1.38)
TM 50	157.10 (154.43, 159.77)	1.03 (1.01, 1.05)	1.17 (1.14, 1.19)	1.45 (1.42, 1.48)	1.36 (1.34, 1.39)
TM 100	161.23 (159.11, 163.36)	1.06 (1.05, 1.08)	1.22 (1.21, 1.24)	1.44 (1.41, 1.47)	1.35 (1.33, 1.37)
Average		1.06 (0.98, 1.13)	1.35 (1.07, 1.55)	1.87 (1.41, 2.28)	1.59 (1.33, 2.11)

values. The range of the keys is $SPACE = [0, SIZE \times 2]$. Second, the benchmark creates a checksum by XOR-ring all the hash-codes of the keys (as defined in `Integer.hashCode`) with the values (identity hash-code). Third, the main loop performs $SIZE$ operations on the map, chosen randomly between get, put, delete, and replace. Note that, by construction, the hit-rate of each operation is 50%. We used the microbenchmark framework Caliper⁷ to vary $SIZE$ to measure the execution time of a single execution of work. Caliper monitors the JVM and discards individual runs that involve garbage-collection and all runs with non-optimized JIT code, thus reporting execution times with a nanosecond accuracy. It performs warm-up runs to ensure the code is JITed, and then takes 10 trials for each $SIZE$ it selects. We used CROCHET to perform a checkpoint between steps 2 and 3, and a rollback after step 3. On a separate execution, we computed another checksum of the resulting data-structure and compared it with the original checksum, to ensure that the rollback mechanism is working correctly.

To measure the performance when the workload requires only a subset of the data-structure, we configured step 3 to use a subset of $SPACE$: 10%, 25%, 50%, or 100%.

We compare CROCHET with the popular Java DeepClone library,⁸ used by other Java tools that would benefit from our approach [13, 12]. This library immediately copies every field of every object using automatically-generated code. Note that, in this case, the checksums do not match because the cloning mechanism does not maintain the identity hash-code of the serialized objects. We also compare CROCHET with the state-of-the-art process-level checkpoint and rollback tool, CRIU [16]. Since the task is to simply serialize the data structure under test, we use CROCHET’s `checkpointObjects(...)` routine to checkpoint only the

⁷ <https://docs.google.com/document/d/1M0e2UNf1ZxixotjB09r4FKzJG07VyhrXxbKqwX1LAzo/pub>

⁸ <https://github.com/kostaskougios/cloning>

data structures under test (and not collect all roots). Note that CRIU is, by definition, doing more work than CROCHET and the DeepClone library, since it is checkpointing the entire process. In the macrobenchmark evaluations that follow (§5.2), we compare it with CROCHET also configured to checkpoint entire applications, but here we want to intentionally demonstrate the overhead of a process-level checkpoint technique when only one portion of an application needs to be checkpoind.

We executed this entire process 20 times for each configuration (on top of the 10 runs that Caliper does for each *SIZE* it selects). Table 1 shows the results of our microbenchmark evaluation, showing the average raw time to run the benchmark (for the baseline, HotSpot 8 configuration) as well as the average slowdown imposed by each configuration ($Time_{configuration}/Time_{HotSpot8}$ with 95% confidence intervals). Each row represents a different data structure paired with a different *SPACE* value (e.g. CHM 25 represents the ConcurrentHashMap benchmark at *SPACE*=25%). When not using checkpoint/rollback, CROCHET imposes a very modest overhead, with a maximum slowdown of 1.11x, averaging 1.06x (RQ1). CROCHET performs checkpoint and rollback at a moderate cost (from 1.09x to 1.54x slowdown) (RQ2) for correct checkpoints, i.e. matching checksums (RQ4). Moreover, CROCHET beats the direct competition, often by a wide margin.

5.2 Macrobenchmarks

Our synthetic microbenchmarks shed light on the raw performance of CROCHET on data structures. To measure CROCHET’s performance in realistic workloads, we also evaluated it using the DaCapo benchmark suite, version 9.12-bach [7]. We followed the benchmark authors’ best practices: We ran a warmup phase before collecting results, repeating the benchmark execution until the measured time reaches a coefficient of variation of 2.0 at most over a sliding window of three executions. We repeated this process 20 times per benchmark and present 95% confidence intervals for timings.

We measured the performance of four configurations: (1) baseline JVM without CROCHET, (2) CROCHET without performing checkpoints, (3) CROCHET calling `checkpointHeapRoots` at the start of the benchmark, and (4) using the system-level checkpoint/restart tool *CRIU* [16] to perform a checkpoint at the start of the benchmark. We excluded the benchmark *tradessoap* because it failed to converge on a stable time even for the normal execution. We did not use the DeepClone library since it was incompatible with most of the benchmarks.

Table 2 shows the results of our macrobenchmark evaluation. We report a 95% confidence interval for the average benchmark execution time in the baseline (HotSpot 8) configuration, and 95% confidence intervals for the average slowdown factor. In the steady-state (not performing any checkpoints), the results mostly mirror the microbenchmark results reported in §5.1, with a maximum slowdown of 1.29x. Note that CROCHET imposes a slowdown higher than 1.07 only on two benchmarks (*jython* and *tomcat*), which can be explained by the common use of reflection in these benchmarks (which CROCHET needs to intercept). On average, over all the benchmarks, CROCHET imposes a slowdown of just 1.06x. CROCHET performs checkpoints at a modest cost, with an overhead ranging from 1.01x–3.45x (RQ2). Note that CROCHET imposes a relatively constant overhead when each class is loaded, which results in a higher slowdown for short benchmarks: *fop* and *xalan*. Overall, CROCHET performs checkpoints at an average cost of 1.49x.

In comparison, CRIU’s checkpoints imposed a slowdown ranging from 1.08x–5.36x, *always* higher than CROCHET. CRIU does not impose any steady-state overhead, and only becomes active during the call to checkpoint. However, CRIU dumps the entire resident heap of the JVM, which causes its performance to vary widely with (1) the size of that resident

Table 2 Macrobenchmark results showing slowdown (newTime/originalTime) comparison between a baseline Java 8 JVM, the CROCHET system (without checkpoint ever called), the CROCHET system (with checkpoint called on all heap roots at the start of the benchmark), and CRIU (with checkpoint called just at the start of the benchmark). For CRIU, we report the size of the dump file. We show 95% confidence intervals for all timings. The last row shows the average, minimum, and maximum overhead for each configuration.

Benchmark	HotSpot 8 Run time (ms)	Relative Slowdown			CRIU Dump	
		Crochet		CRIU		
		No checkpoint	Checkpoint			
avrora	4,274 (4,208, 4,341)	1.01 (0.98, 1.03)	1.01 (0.98, 1.04)	1.08 (1.06, 1.11)	192.8 MB	
batik	1,256 (1,242, 1,270)	1.03 (1.01, 1.04)	1.23 (1.21, 1.26)	1.33 (1.31, 1.35)	420.2 MB	
eclipse	18,658 (18,523, 18,794)	1.01 (1.00, 1.02)	1.26 (1.24, 1.27)	1.17 (1.16, 1.18)	3.3 GB	
fop	223 (214, 233)	1.07 (1.02, 1.12)	2.06 (1.94, 2.19)	2.85 (2.73, 2.98)	385.5 MB	
h2	6,451 (6,413, 6,489)	1.04 (1.03, 1.05)	1.15 (1.14, 1.17)	1.21 (1.20, 1.22)	1.8 GB	
jython	3,285 (3,060, 3,510)	1.21 (1.09, 1.34)	1.60 (1.49, 1.72)	1.57 (1.45, 1.71)	1.5 GB	
luindex	761 (749, 773)	1.01 (0.99, 1.04)	1.10 (1.08, 1.12)	1.31 (1.28, 1.33)	176.5 MB	
lusearch	605 (601, 610)	1.01 (1.00, 1.01)	1.11 (1.09, 1.13)	5.36 (5.31, 5.40)	3.7 GB	
pmd	1,417 (1,402, 1,432)	1.04 (1.02, 1.05)	1.12 (1.10, 1.13)	1.63 (1.61, 1.65)	1.0 GB	
sunflow	944 (929, 959)	1.02 (0.99, 1.04)	1.13 (1.10, 1.15)	3.37 (3.27, 3.47)	3.0 GB	
tomcat	988 (979, 996)	1.29 (1.27, 1.30)	1.88 (1.86, 1.91)	2.27 (2.15, 2.40)	1.2 GB	
tradebeans	6,618 (6,535, 6,701)	1.03 (1.01, 1.05)	1.21 (1.19, 1.22)	1.29 (1.26, 1.33)	2.2 GB	
xalan	288 (276, 300)	1.02 (0.97, 1.07)	3.45 (3.28, 3.63)	4.75 (4.55, 4.96)	1.2 GB	
Average		1.06 (0.97, 1.34)	1.49 (0.98, 3.63)	2.25 (1.06, 5.40)		

heap (which may include lots of garbage), and (2) the duration of the benchmark. For instance, in the case of *lusearch* (5.36x slowdown), CRIU had to dump 3.66GB of data, and the underlying benchmark took only 605 msec in the baseline configuration. Compare this to CRIU’s performance on *eclipse* (1.17x slowdown), where CRIU dumped a similar amount of data (3.25GB), but where the dump time was hidden in the significantly longer native benchmark execution time (18,658 msec). Note that CROCHET imposed a slowdown of just 1.11x for *lusearch*.

5.3 Transactional benchmarks

State-of-the-art Software Transactional Memories (STMs) intercept all data accesses (i.e. field/arrays/local variables reads/writes) to provide each thread executing a transaction a consistent view of the program state, and to isolate the changes that each thread performs in its separate transaction. Changes made by a transaction T become globally visible to new transactions when T finishes and commits successfully. Depending on the changes made by other transactions that commit between T ’s start and finish, T may fail to commit; in which case the STM reverts all changes made by T . Transactions may finish by a voluntary abort, with the same outcome of an unsuccessful commit.

Assuming a single-threaded application, STMs can be used as an implementation of Strawman 2, described in §2, as follows: To checkpoint, start a new transaction; to rollback, abort the current transaction. We used the STMBenchmark [24] to evaluate the feasibility of such an approach in comparison to CROCHET.

STMBenchmark creates a realistic object graph that resembles the heap of a Computer Assisted Drawing (CAD) application, and then issues several concurrent operations that manipulate different regions of the object graph. Table 3 shows the results for this experiment, the following text explains each column in detail (higher is better).

Table 3 STM comparison results, showing the baseline number of operations/sec completed without any concurrency control strategy (*HotSpot Ops/Sec*), the fraction (relative to that baseline) of operations completed with a transaction manager enabled but issuing no transactions (*No TX*), the fraction of operations completed with a single transaction issued to implement checkpoint/rollback (*One TX*), and the fraction of operations completed using the STMs to enforce atomicity (*Many TX*). The first row shows the results with CROCHET used to perform checkpoint/rollback (*One TX*), and without performing any checkpoint/rollback (*No TX*). More operations is better.

Configuration	HotSpot Ops/Sec	% of baseline operations/sec		
		No TX	One TX	Many TX
crochet	243 (242, 244)	0.96 (0.96, 0.97)	0.91 (0.91, 0.91)	
deuce-lsa	237 (237, 238)	1.01 (1.00, 1.01)	0.06 (0.06, 0.06)	0.15 (0.15, 0.16)
deuce-tl2	237 (237, 238)	1.00 (1.00, 1.01)	0.01 (0.01, 0.01)	0.13 (0.13, 0.13)
jvstm	237 (237, 238)	0.34 (0.34, 0.35)	0.45 (0.45, 0.45)	0.45 (0.45, 0.45)

The STM Bench7 workload consists of different operations that read and write several different parts of the object graph, and should be atomic. STM Bench7 ships with a backend that uses no concurrency control to ensure such atomicity – *no_lock*. This is our baseline, executed on a native JVM and with a single thread: Column **HotSpot Ops/Sec**.

STM Bench7 also ships with backends to isolate concurrent operations with transactions using existing STMs: Deuce [21] and JVSTM [10]. Deuce is an STM framework that supports several synchronization algorithms, we used the two it ships with: LSA [47] and TL2 [19]. We used each STM backend as the concurrency control mechanism to ensure that each workload operation is atomic: Column **Many TX**. As before, we used a single thread. Note that this is the typical way to compare STM implementations with STM Bench7, but since CROCHET does not (out of the box) enforce atomicity, we cannot use it as a point of comparison.

To perform checkpoint/rollback, we modified STM Bench7 to checkpoint the object graph at the start of the workload, and to roll it back at the end: Column **One TX**. We used CROCHET to perform such checkpoint/rollback with the *no_lock* backend. We also used each STM to checkpoint the object graph by issuing a single transaction at the start of the workload, and keeping that transaction active throughout the whole workload. When the workload finishes, we rollback by aborting that single transaction. Again, we used a single thread. Note that we also added a checksum before and after the workload, and used it to ensure that the checkpoint/rollback mechanism worked as intended. Finally, we ran all the STMs experiments without creating any transactions, and CROCHET without performing any checkpoint/rollback, to measure the cost of the ability to perform checkpoint/rollback when not used: Column **No TX**.

We ran Deuce and JVSTM on Java 7 (Oracle HotSpot 1.7.0_80, the latest version), and CROCHET on the same Java 1.8.0_66 from the previous experiments (CROCHET is not backwards compatible with Java 7). We configured STM Bench7 to run the *read-write* workload during 30 seconds with structural modifications disabled, as they slow down all STMs excessively. We repeated all runs 20 times and present 95% confidence intervals for all results. Table 3 shows the results for this experiment, showing the baseline number of operations performed with no locking under each baseline JVM and the relative fraction of those operations performed under each configuration (higher is better).

We also compared CROCHET to the recent XJ hybrid software transactional memory system [14]. XJ requires a custom-patched version of OpenJDK (specifically, OpenJDK1.7u40) to leverage hardware transactional memory features. We were not able to successfully run XJ outside of the VM that the authors provided (the custom version of OpenJDK

Table 4 Results of the synchroBench benchmark, with checkpoint/rollback provided by CROCHET and XJ; compared against JDK 1.8.0 without checkpoints. More operations/sec is better.

Datastructure	HotSpot (ops/sec)	% of baseline ops/sec performed	
		Crochet	XJ
SequentialHashIntSet	11.60 (11.52, 11.67)	0.91 (0.90, 0.92)	
ClosedHashIntSet	1.31 (1.31, 1.32)	0.91 (0.90, 0.92)	0.52 (0.50, 0.54)

does not build on recent versions of the Linux kernel, ignoring the build error generates a JVM that consumes all available memory for any Java program we tried). Furthermore, XJ requires all programs run with it to be pre-processed, and the pre-processor fails to process STMBench7 our micro-benchmarks and Caliper. Therefore, we were able to conduct an evaluation only inside of the provided VirtualBox VM and only with the provided synchroBench benchmark. XJ requires that data-structures be hand-annotated in order to be check-pointed, and hence was not easily amenable to the STMBench7 workloads that we used. We used two data-structures with synchroBench, both hash-sets that hold integers: `hashtables.SequentialHashIntSet`, a sequential hash-set; and `hashtables.xj.ClosedHashIntSet`, a hash-set that supports *closed transactions* (i.e. the same type of transactions as JVSTM and Deuce, explained earlier). `SequentialHashIntSet` represents a typical data structure that would be used with CROCHET, while `ClosedHashIntSet` represents a data structure provided by the XJ authors with the correct annotations. We provide both data structures to demonstrate both the performance difference between CROCHET and XJ on a shared data structure, and baseline performance within this configuration. The workload initialized the data-structure with 65,536 elements and issued 95% read, 5% write operations during 5 seconds. We modified it to perform a checkpoint and checksum before the workload, rollback and another checksum after the workload, and ensure the two checksums match. To checkpoint/rollback, we used CROCHET and the single transaction technique described above for STMBench7. We allowed the benchmark to warm up during 60 seconds, and repeated the timed workload 100 times. Table 4 shows the average and 95% confidence interval for each configuration.

This experiment shows the prohibitive performance cost of using STMs to support checkpoint/rollback. Besides the performance penalty, using STMs for this purpose has further limitations: it does not support checkpointing state shared between threads (transactions naturally isolate it), and it requires manual modification of the target application (i.e. identifying transactions and transactional data, or even using different/slower data-structures). CROCHET, on the other hand, has a much lower performance cost, does not require manual changes to the application using it, and supports multi-threaded checkpoints.

6 Case Studies

At its core, CROCHET provides the ability to dynamically change the behavior of any object or code in the JVM, with negligible steady-state overhead, and requiring only a minimal pause to perform the initial update. Whereas the JVM provides a hotswap code functionality to redefine methods of all objects of a given class, CROCHET is able to redefine methods on a *per-object* basis. In particular, CROCHET uses this technique to provide lightweight checkpoint and rollback functionality. However, the technique we present in this paper is more general and, in this section, we describe several of the various applications that immediately stand to benefit from CROCHET, as well as several that stand to benefit from its high level instrumentation approach.

Table 5 Execution time (and relative overhead) of four different fuzzing strategies: native (no isolation between runs), crochet-baseline (no isolation between runs, but with CROCHET running), crochet (with CROCHET providing isolation), and restart (restarting the server between each run).

Configuration	Exec Time (ms)	Rel Overhead
native	41.52 (41.37, 41.66)	
crochet-baseline	42.69 (40.32, 45.07)	1.03 (0.97, 1.09)
crochet	43.77 (43.65, 43.89)	1.05 (1.05, 1.06)
restart	78.33 (78.20, 78.46)	1.89 (1.88, 1.89)

6.1 Fuzzing and Test Generation

There are a wide range of approaches toward generating inputs to test program behavior. For instance, symbolic analysis tools, such as KLEE [11], JPF [53] and CUTE/JCUTE [49] generate new inputs systematically to explore different program behavior based on path constraints. Other tools are search-based, turning input generation into an optimization problem that maximizes code coverage [22]. Yet other tools take a simpler approach: fuzzers perturb known inputs to generate new ones [39, 27]. A key challenge for these tools is scalability: there can be an immense input space to search.

All of these tools typically re-execute a program many times from a given point in execution, for instance, to explore different inputs to a function, or to force a program to follow a different branch. Prior approaches either re-execute the program from the beginning each time, or maintain a symbolic heap. EvoSuite generates entire test stubs, and re-executes those test stubs, changing them between executions to expose new behavior. KLEE, JPF, CUTE and JCUTE all maintain a symbolic heap – a map from variables to a collection of values, one per different program state. Both approaches are inefficient: Re-running a program implies the cost of running that execution, and redirecting all heap accesses through a map adds a significant performance penalty (100x is common [54]).

CROCHET allows these approaches to explore different inputs to the same function much more efficiently: Perform a checkpoint when reaching the function to explore for the first time, explore one point in the input space by calling the function once, observe the results, then rollback to the previous checkpoint, and call the function again with another input. Note that CROCHET is also useful if the function is easy to reach (e.g., request handling loop on a server): Typical black-box fuzzers (i.e. based on the format of the requests accepted by the server) assume that each request is independent from all that precede it. If this assumption does not hold, the fuzzer may discover some input that crashes the server on a fuzzing run, but not in isolation; thus limiting the usefulness of such techniques. Using CROCHET to perform a checkpoint just after the server starts, and rolling back to that pristine program state after each fuzzed command, ensures that all errors found will be reproducible in isolation from just the fuzzed command.

As a case study, we modified an existing FTP black-box fuzzer⁹ to perform the same fuzzing run on three scenarios: (1) fuzz all commands on the same server, (2) restart the server after each fuzzed command, and (3) checkpoint the initial server state and rollback after each command. We modified an FTP server written in Java¹⁰ to checkpoint its heap on startup, and to rollback when signaled by the fuzzer. Table 5 shows the results for a short fuzzing run, with 122 total FTP commands sent, as the average of 100 runs and the

⁹ `ftp_pre_post` shipped with Metasploit 4.16.31: <https://github.com/rapid7/metasploit-framework>

¹⁰ CrossFTP version 1.07: <https://sourceforge.net/projects/crossftpserver/>

respective 95% confidence interval. The fuzzing run takes takes 41.52 seconds to execute under Scenario 1, 78.33 under Scenario 2 (an increase of 89%), and just 43.77 seconds under Scenario 3. The results are encouraging, showing that that CROCHET imposes no measurable overhead to ensure proper input isolation.

6.2 Checkpoint/Rollback as an Application Service

Transactional applications may benefit from system-provided checkpoint and rollback abstractions. Database applications naturally fit this format. As a case study, we considered the H2 database, which is an in-memory SQL database written in Java.

The DaCapo benchmark suite contains an H2 benchmark, in which the benchmark driver: (1) creates an in-memory database, (2) populates it with test data from the TPC-C benchmark, (3) performs a number of TPC-C operations using multiple threads, (4) computes a checksum of the database and compare it with the expected value, and, finally, (5) restores the initial state of the database after 2. Each iteration of the benchmark repeats steps 3–5.

To reset the state of the benchmark, DaCapo’s H2 benchmark duplicates a number of columns on each table to hold the original data. The benchmark workload does not use those columns. At step 5, the benchmark resets the original columns by copying the data from their duplicate columns. Step 5 is developer-provided customized code to checkpoint and reset the state of the database. Therefore, it provides a perfect opportunity to test CROCHET. We wrote our own version of the H2 benchmark that uses CROCHET’s generic support to checkpoint and rollback in-memory data (i.e. the database tables) *instead* of the customized code that adds the duplicated columns.

We ran this benchmark in the same environment as specified in the previous section. It completed correctly (passing all checks) averaging $8,256 \pm 91\text{ms}$, adding a slowdown of around 1600ms (1.3x). In doing so, CROCHET proxied a total of 4,185,705 objects and copied 1,009,321 objects totaling just over 88MB, yielding a throughput of 90MB/sec copied. CROCHET was not able to beat the performance of the custom, efficient SQL queries that reset the state in bulk, which is not surprising. However, CROCHET’s approach for supporting efficient lightweight checkpoints is general enough for supporting any other in-memory Java implementation of the same TPC-C benchmark, even if it is not a SQL database (i.e. a key-value store).

6.3 Other Applications

There are also a variety of other potential applications for CROCHET which would require additional development, but could be very promising.

Time Travel Debugging. Time travel debuggers [2, 52, 3, 20, 30] allow developers to “step backward” (in execution) while debugging, which is accomplished through a combination of checkpoint-rollback and deterministic replay techniques: checkpoints are taken at regular intervals, and deterministic replay is used to fast-forward from the nearest checkpoint to the desired point of execution. While there are time travel debuggers for other VM-based languages (e.g. TARDIS for .NET [2]; JARDIS and ReJS for JS [3, 52]), there is currently no time travel debugger for the JVM. CROCHET could be used as the underlying checkpoint mechanism for a new time travel debugger for the JVM, *without* requiring changes to the JVM like these other systems.

Fault Tolerance. Existing high-level approaches for fault tolerance can also benefit from CROCHET’s approach. For instance, systems like ASSURE [50], Rx [46], ARMOR [13], and Mx [25] provide fault tolerance by performing regular checkpoints and detecting when the app fails. When a failure is detected, these systems tolerate it by reverting back to the most recent checkpoint and generating error recovery code.

A main limitation in these approaches is the regularity of checkpoints, which are based at the OS or VM level and, as explained in §2, do not map well to managed language and runtime environments, such as the JVM. Using CROCHET could lead to increased performance, and increased precision in the recovery procedure stemming from the fine-grained object-level information available about the data being restored.

Existing tools that are specialized to the JVM employ Java serialization to make frequent snapshots of specific variables that are considered important. ReCrashJ [1] makes a complete copy of method parameters as functions are called, providing developers with a log of the values of each parameter passed to each function if the application crashes. Capturing the complete object graph of each parameter to each function is quite expensive, imposing a slowdown of over 1,000x. Instead of eagerly capturing each parameter, CROCHET could be used to lazily capture them, only just before they are modified.

Smalltalk `become:`. The Smalltalk language provides a unique method, `become:`, that swaps the identities of its receiver and its argument. `become:` is a powerful global operation that updates all variables that refer to the receiver so that now they refer to the argument, and vice-versa. For instance, the Smalltalk implementation uses the `become:` method to increase the capacity of fixed-size of data-structures (e.g., array-lists and hash-maps): it simply allocates a new, larger, data-structure, copies all objects from the smaller data-structure, and then invokes `large become: small` [23]. All references to the original collection are replaced transparently by references to the new one.

Unfortunately, implementing `become:` is costly. Early implementations of Smalltalk keep a global object table, and represent references as indexes into such a table. `become:` was implemented as a simple pointer swap on the object table. However, modern high-level language VMs do not use such a global object table, relying on a garbage-collected heap instead. Instead, we might need to wrap every single object in a proxy object, and correct all references to that object to reference through the proxy. Or perhaps, we could modify the garbage collector to swap all references on the heap to an object `become:`’ing another through a full GC cycle.

Instead, CROCHET allows for an efficient `become:` through a lazy heap traversal that simply compares each object traversed with the receiver of `become:` and replaces it with the argument, and the same for replacing the argument with the receiver. Unlike the traditional approach, this approach does not require a pause that is proportional to the size of the heap.

7 Related Work

There has been a considerable body of research investigating the implementation and application of checkpoint and recovery tools. The tools can generally be divided into those that operate with operating system (and memory management unit) support, and those that operate primarily with developer support. Notable system-level tools include libpkt [44], Jockey [48], ZAP [38] and CRIU [16]. libpkt [44] and Jockey [48] create a fork of the process being checkpointed and use page faults to detect memory writes as they occur, performing incremental checkpoints. CROCHET is similar in spirit to these systems, as it also takes

incremental checkpoints, but checkpoints at the granularity of individual objects in the JVM, rather than entire pages of memory. ZAP [38] and CRIU [16] focus on checkpointing for process migration, while CROCHET’s goal is to enable recovery within the same process as the checkpoint. We compared CROCHET with CRIU in §5.2.

Specifically targeting the JVM, Cunei and Vitek proposed an approach to checkpointing that is optimized for latency, utilizing a mirror of memory contents to satisfy both checkpoints and write requests concurrently [18]. While this approach required that the source-code of the JVM was modified, CROCHET requires no modifications to the JVM. Cunei and Vitek argued that checkpointing at the granularity of pages could impose higher than expected overheads when (relatively small) objects were sparsely distributed among (relatively larger) pages [18]. We make the same argument, and checkpoint at the granularity of individual objects, rather than memory pages.

Bringing developers into the loop, Xu et al. requires them to specify checkpoint and rollback locations statically for their Java CheckPoint system (JCP) [55]. JCP then performs an offline static analysis to determine which variables need to be included in each checkpoint. JCP also only supports single threaded applications. On the other hand, CROCHET supports multithreaded applications and does not require static specification of checkpoint and rollback sites. Upon rollback, JCP replays the (as computed) minimal slice of code needed to get the execution back to the same point where the checkpoint was called, while CROCHET generates custom code to bring the execution back to the same point, requiring for a stack depth of n only n method calls.

TARDIS [2] supports efficient time-travel debugging (i.e. ‘step backwards’) in the .NET CLR by piggybacking opportunistically on the garbage collector to create regular checkpoints. TARDIS requires modifications to the runtime (equivalent to modifying the JVM) to achieve its efficient checkpoints. Similarly, JARDIS [3] and ReJS [52] support time travel debugging in JS (ChakraCore), and collect snapshots very similarly to TARDIS. We believe that CROCHET could be extended (perhaps in combination with some deterministic record-and-replay tool such as Chronicler [6]) to support efficient time travel debugging in the JVM, without requiring modifications to the JVM.

There are also several systems targeting JVM migration, such as JAVMM [26] and ALMA [9], which bring insights from generic VM migration [51] into the JVM. These systems create a complete checkpoint of a running JVM, transfer that checkpoint to another JVM (perhaps on another machine) and then resume execution from that checkpoint. On the one hand, these systems also consider file and network state, and CROCHET does not. On the other, CROCHET works on stock JVMs, whereas both of these systems require modifications to the JVM.

There are also a variety of related systems that capture partial execution information in order to reproduce crashing executions, either capturing partial checkpoints [34, 15] or partial trace information [17, 56, 29, 15]. CROCHET could be used to support fault reproduction tools in the JVM as well.

CROCHET is also related (in implementation and design) to several non-checkpoint JVM-based systems. For instance, Rubah [43] provides dynamic software update in unmodified JVMs, and employs a lazy-heap traversal that inspired CROCHET’s approach. CROCHET explores the notion of proxies far more widely and generically, focusing on the general performance and application of proxies to implement object-level page faults. Instant pickles [33] is an approach for pickling (serializing) objects in Scala. Instant pickles uses statically generated code to serialize and deserialize objects in the JVM faster than the JVM’s dynamic serialization can. CROCHET uses a similar approach for copying objects.

8 Conclusion

The ability to perform fast, lightweight, fine-grained checkpoints on the JVM is not only useful to provide rich semantics to application developers, but also instrumental to support sophisticated automatic tools for applications such as fuzzing and fault tolerance. In this paper, we presented CROCHET, which significantly improves the state-of-the-art on this topic: CROCHET works on existing stock JVMs through bytecode rewriting and standard debug APIs, and the cost of running CROCHET when not using its checkpoint/rollback capabilities is very low. CROCHET automatically identifies the minimal state to be copied in a checkpoint fully automatically and works correctly with multi-threaded programs. CROCHET enjoys good performance with minimal pauses when performing checkpoints due to its lazy heap traversal algorithm. We believe CROCHET provides an adequate solution to a pressing problem that, in turn, will enable the realistic deployment of other tools that require efficient checkpoint/rollback support on an unmodified JVM.

References

- 1 Shay Artzi, Sunghun Kim, and Michael D. Ernst. Recrash: Making software failures reproducible by preserving object states. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 542–565, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-70592-5_23.
- 2 Earl T. Barr and Mark Marron. Tardis: Affordable time-travel debugging in managed runtimes. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 67–82, New York, NY, USA, 2014. ACM. doi:10.1145/2660193.2660209.
- 3 Earl T. Barr, Mark Marron, Ed Maurer, Dan Moseley, and Gaurav Seth. Time-travel debugging for javascript/node.js. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 1003–1007, New York, NY, USA, 2016. ACM. doi:10.1145/2950290.2983933.
- 4 Jonathan Bell and Gail Kaiser. Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs. In *ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 83–101, New York, NY, USA, October 2014. ACM. doi:10.1145/2660193.2660212.
- 5 Jonathan Bell and Gail Kaiser. Unit Test Virtualization with VMVM. In *36th International Conference on Software Engineering*, ICSE 2014, pages 550–561, New York, NY, USA, June 2014. ACM. ACM SIGSOFT Distinguished Paper Award. doi:10.1145/2568225.2568248.
- 6 Jonathan Bell, Nikhil Sarda, and Gail Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 362–371, Piscataway, NJ, USA, 2013. IEEE Press. URL: <http://dl.acm.org/citation.cfm?id=2486788.2486836>.
- 7 Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06*, pages 169–190, New York, NY, USA, 2006. ACM. doi:10.1145/1167473.1167488.
- 8 Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

- 9 Rodrigo Bruno and Paulo Ferreira. Alma: Gc-assisted jvm live migration for java server applications. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, pages 5:1–5:14, New York, NY, USA, 2016. ACM. doi:10.1145/2988336.2988341.
- 10 João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, 2006. doi:10.1016/j.scico.2006.05.009.
- 11 Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association. URL: <http://dl.acm.org/citation.cfm?id=1855741.1855756>.
- 12 Antonio Carzaniga, Alessandra Gorla, Alberto Goffi, Andrea Mattavelli, and Mauro Pezzè. Cross-checking Oracles from Intrinsic Software Redundancy. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 931–942, 2014.
- 13 Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Mauro Pezzè, and Nicolò Perino. Automatic Recovery from Runtime Failures. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 782–791, 2013.
- 14 Keith Chapman, Antony L. Hosking, and J. Eliot B. Moss. Hybrid stm/htm for nested transactions on openjdk. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 660–676, New York, NY, USA, 2016. ACM. doi:10.1145/2983990.2984029.
- 15 Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Partial replay of long-running applications. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 135–145. ACM, 2011. doi:10.1145/2025113.2025135.
- 16 Jonathan Corbet. Checkpoint/restart (mostly) in user space. *LWN.Net*, 2011.
- 17 Olivier Crameri, Ricardo Bianchini, and Willy Zwaenepoel. Striking a new balance between program instrumentation and debugging time. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 199–214. ACM, 2011. doi:10.1145/1966445.1966464.
- 18 Antonio Cunei and Jan Vitek. A new approach to real-time checkpointing. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, pages 68–77, New York, NY, USA, 2006. ACM. doi:10.1145/1134760.1134771.
- 19 Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11864219_14.
- 20 George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI '02, pages 211–224. ACM, 2002. doi:10.1145/1060289.1060309.
- 21 P. Felber, G. Korland, and N. Shavit. Deuce: Noninvasive concurrency with a java stm. In *Electronic Proceedings of the workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, 2010.
- 22 Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM. doi:10.1145/2025113.2025179.
- 23 Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
- 24 Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: A benchmark for software transactional memory. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European*

- Conference on Computer Systems 2007*, EuroSys '07, pages 315–324, New York, NY, USA, 2007. ACM. doi:10.1145/1272996.1273029.
- 25 Petr Hosek and Cristian Cadar. Safe software updates via multi-version execution. In *International Conference on Software Engineering (ICSE 2013)*, pages 612–621, 5 2013.
- 26 Kai-Yuan Hou, Kang G. Shin, and Jan-Lung Sung. Application-assisted live migration of virtual machines with java applications. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 15:1–15:15, New York, NY, USA, 2015. ACM. doi:10.1145/2741948.2741950.
- 27 Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K. Chang. Ocat: object capture-based automated testing. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 159–170. ACM, 2010. doi:10.1145/1831708.1831729.
- 28 Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 389–400, New York, NY, USA, 2011. ACM. doi:10.1145/1993498.1993544.
- 29 Wei Jin and Alessandro Orso. Bugredux: reproducing field failures for in-house debugging. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 474–484, Piscataway, NJ, USA, 2012. IEEE Press. URL: <http://dl.acm.org/citation.cfm?id=2337223.2337279>.
- 30 Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association. URL: <http://dl.acm.org/citation.cfm?id=1247360.1247361>.
- 31 Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for \$8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 3–13, Piscataway, NJ, USA, 2012. IEEE Press. URL: <http://dl.acm.org/citation.cfm?id=2337223.2337225>.
- 32 Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. Use at your own risk: The java unsafe api in the wild. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 695–710, New York, NY, USA, 2015. ACM. doi:10.1145/2814270.2814313.
- 33 Heather Miller, Philipp Haller, Eugene Burmako, and Martin Odersky. Instant pickles: Generating object-oriented pickler combinators for fast and extensible serialization. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 183–202, New York, NY, USA, 2013. ACM. doi:10.1145/2509136.2509547.
- 34 Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Recording application-level execution for deterministic replay debugging. *IEEE Micro*, 26(1):100–109, 2006. doi: 2006-02-1702:00:03.800.
- 35 OpenJDK Team. CompressedOOPS. <https://wiki.openjdk.java.net/display/HotSpot/CompressedOops>.
- 36 Oracle. Jvm tool interface. <http://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html>, 2013.
- 37 Oracle Corporation. Instrumentation API for the Java Platform SE 7. [https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html#retransformClasses\(java.lang.Class...\)](https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html#retransformClasses(java.lang.Class...)). Accessed on 2018/01/11.

- 38 Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: A system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36(SI):361–376, dec 2002. doi:10.1145/844128.844162.
- 39 Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA ’07, pages 815–816. ACM, 2007. doi:10.1145/1297846.1297902.
- 40 Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. *IEEE Trans. Softw. Eng.*, 40(5):427–449, 2014. doi:10.1109/TSE.2014.2312918.
- 41 Luís Pina. *Practical Dynamic Software Updating*. PhD thesis, Instituto Superior Técnico, University of Lisbon, 2016.
- 42 Luís Pina and João Cachopo. Atomic dynamic upgrades using software transactional memory. In *Proceedings of the 4th International Workshop on Hot Topics in Software Upgrades*, HotSWUp. IEEE, 2012.
- 43 Luís Pina, Luís Veiga, and Michael Hicks. Rubah: DSU for Java on a Stock JVM. In *OOPSLA*, 2014. doi:10.1145/2660193.2660220.
- 44 James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under unix. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON’95, pages 18–18, Berkeley, CA, USA, 1995. USENIX Association. URL: <http://dl.acm.org/citation.cfm?id=1267411.1267429>.
- 45 Yuhua Qi, Xiaoguang Mao, and Yan Lei. Efficient automated program repair through fault-recorded testing prioritization. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM ’13, pages 180–189, Washington, DC, USA, 2013. IEEE Computer Society. doi:10.1109/ICSM.2013.29.
- 46 Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP ’05, pages 235–248, New York, NY, USA, 2005. ACM. doi:10.1145/1095810.1095833.
- 47 Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC’06, pages 284–298, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11864219_20.
- 48 Yasushi Saito. Jockey: A user-space library for record-replay debugging. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, AADEBUG’05, pages 69–76, New York, NY, USA, 2005. ACM. doi:10.1145/1085130.1085139.
- 49 Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM. doi:10.1145/1081706.1081750.
- 50 Stelios Sidiropoulos, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. Assure: Automatic software self-healing using rescue points. *SIGARCH Comput. Archit. News*, 37(1):37–48, 2009. doi:10.1145/2528521.1508250.
- 51 Petter Svärd, Benoit Hudzia, Johan Tordsson, and Erik Elmroth. Evaluation of delta compression techniques for efficient live migration of large virtual machines. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE ’11, pages 111–120, New York, NY, USA, 2011. ACM. doi:10.1145/1952682.1952698.

17:30 CROCHET: Checkpoint/Rollback via Lightweight Heap Traversal on Stock JVMs

- 52 John Vilk, James Mickens, and Mark Marron. A gray box approach for high-fidelity, high-speed time-travel debugging. Technical report, Microsoft Research, June 2016. URL: <https://www.microsoft.com/en-us/research/publication/gray-box-approach-high-fidelity-high-speed-time-travel-debugging/>.
- 53 Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Llerda. Model checking programs. *Automated Software Engg.*, 10(2):203–232, apr 2003. doi:10.1023/A:1022920129859.
- 54 Matej Vitásek, Walter Binder, and Matthias Hauswirth. Shadowdata: Shadowing heap objects in java. In *Proceedings of the 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE ’13, pages 17–24, New York, NY, USA, 2013. ACM. doi:10.1145/2462029.2462032.
- 55 Guoqing Xu, Atanas Rountev, Yan Tang, and Feng Qin. Efficient checkpointing of java software using context-sensitive capture and replay. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE ’07, pages 85–94, New York, NY, USA, 2007. ACM. doi:10.1145/1287624.1287638.
- 56 Cristian Zamfir and George Cadea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems*, EuroSys ’10, pages 321–334. ACM, 2010. doi:10.1145/1755913.1755946.

A Full pseudo-code for the checkpoint/rollback algorithm

```

1 NORMAL.onReadWrite() { /*empty*/ }      53
2                                         54
3 NORMAL.onCheckpoint(int version){       55 NORMAL.onRollback(int version){
4   int curV = this.version;             56   int curV = this.version;
5   if (curV==version &&           57   if (curV==version &&
6   this.status==CHECKPOINT)           58     this.status==ROLLBACK)
7     return;                           59   return;
8                                         60
9   CAS(this.version,curV,version);    61   CAS(this.version,curV,version);
10  CAS(this.status,NONE,CHECKPOINT); 62   CAS(this.status,NONE,ROLLBACK);
11 }                                     63 }
12                                         64
13 CHECKPOINT.onReadWrite() {            65 ROLLBACK.onReadWrite() {
14   int curV = this.version;          66   int curV = this.version;
15                                         67
16   Object snap = this.snapshot;     68   Object snap = this.snapshot;
17   if (snap==NULL ||              69   if (snap != NULL &&
18     snap.version<curV) {          70     snap.version<curV) {
19     // Allocates empty object    71     synchronized(snap) {
20     // Without running constructor 72       snap = this.snapshot;
21     Object newSnap = ...          73       if (snap != NULL &&
22     this.copyFieldsTo(newSnap);   74         snap.version<curV) {
23     newSnap.version = curV;      75         this.copyFieldsFrom(snap);
24     CAS(this.snapshot,snap,newSnap); 76         snap.version = curV;
25   }                                77     }
26                                         78   }
27                                         79 }
28                                         80
29   for (Field f in this)          81   for (Field f in this)
30     f.onCheckpoint(curV);        82     f.onRollback(curV);
31                                         83
32   CAS(this.status, CHECKPOINT,   84   CAS(this.status, ROLLBACK, NORMAL
33     NORMAL);                  85   );
34                                         86
35 CHECKPOINT.onCheckpoint(int vers) { 87 ROLLBACK.onRollback(int vers) {
36   int curV = this.version;        88   int curV = this.version;
37   if (curV == vers) return;      89   if (curV == vers) return;
38                                         90
39   CAS(this.version,curV,vers);  91   CAS(this.version,curV,vers);
40 }                                     92 }
41                                         93
42 ROLLBACK.onCheckpoint(int vers) {  94 CHECKPOINT.onRollback(int vers) {
43   int curV = this.version;        95   int curV = this.version;
44   if (curV==vers &&           96   if (curV==vers &&
45     this.status==CHECKPOINT)    97     this.status==ROLLBACK)
46     return;                      98   return;
47                                         99
48   ROLLBACK.onReadWrite();        100
49                                         101
50   CAS(this.version,curV,vers);  102  CAS(this.version,curV,vers);
51   CAS(this.status,ROLLBACK,     103  CAS(this.status,CHECKPOINT,
52     CHECKPOINT);                104  ROLLBACK);
53 }
```