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

Jonathan Bell & Luís Pina
George Mason University
{bellj,lpina2}@gmu.edu

Fork us on GitHub

github.com/gmu-swe/crochet

Why checkpoint/rollback?

Buffer: SET key val SET key val SET key val

methodToFuzz("SET key val")
methodToFuzz("SET key val")
methodToFuzz("SET key val")
↓
✗
↓
generateTest("SET key val")

- Fuzzer calls `methodToFuzz` repeatedly, potentially trying new inputs
- `methodToFuzz` keeps buffer with all inputs
- Unsound due to shared state: Each fuzzing iteration depends on all previous

Buffer: SET key val

CROCHET.checkpoint()
methodToFuzz("SET key val")
CROCHET.rollback()
methodToFuzz("SET key val")

- *Checkpoint* initial state: Empty buffer
- *Rollback* state after each fuzzed input: Discard all buffer changes
- Fuzzed inputs are independent, sound fuzzing

Design Goals

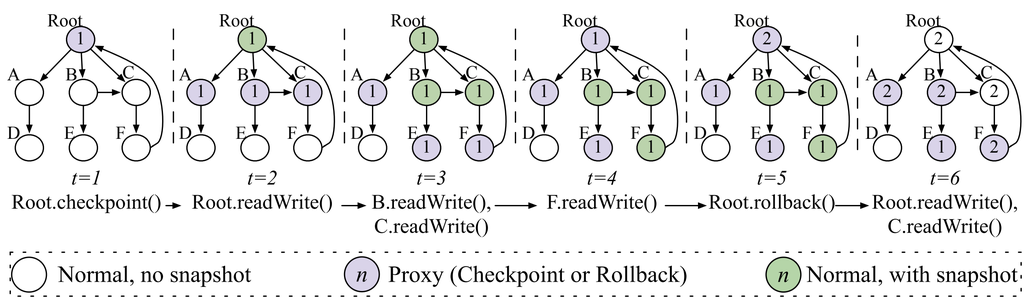
No JVM changes Excludes `fork` due to multi-threaded JVM
- Bytecode rewritting + standard debug API (JVMTI)
- Use `sun.misc.Unsafe` to access the object model

Efficient Excludes `CRIU`, which dumps the whole heap
- Lazy heap traversal with Copy-On-Access

Checkpoint/rollback dynamically Without static limitations
- Supports multithreaded heap traversal after root collection

Low overhead Excludes *STMs* using single transaction as checkpoint due to high instrumentation overhead
- 6% steady-state, 49% when checkpointing

Lazy Heap Traversal



- Each object has: Version, snapshot (if present), and state (normal/proxy)
- Checkpoint/rollback: Collect root references and install proxies
- Proxy deref: Create snapshot or overwrite object, propagate proxies

Bytecode Rewritting

```
01 class List {  // Original code highlighted
02   List next; int i;
03
04   int sum() {
05     next.$$onReadWrite();
06     return i + next.sum();
07   }
08
09   List $$snapshot;
10   int $$version;
11   // But no status!
12
13   // Empty
14   void $$onReadWrite();
15
16   // Turn into proxy
17   void $$onCheckpoint(int v);
18   void $$onRollback(int v);
19
20   void $$copyFieldsTo(C dst);
21 }
```

```
01 class $ListProxy extends List {
02   // No extra fields
03   // Proxy has same size
04
05   // Proxies simply change the
06   // implementation of methods
07   // onReadWrite, onCheckpoint, etc.
08
09   // The JIT inlines non-proxy code
10   // and emits efficient proxy checking
11   // as part of dynamic invocation
12
13   // Snap, propagate, revert proxy
14   void $$onReadWrite();
15
16   // Update version
17   void $$onCheckpoint(int v);
18   void $$onRollback(int v);
19
20
21 }
```

Proxies on/off

```
01 List l = new List();
   [ _mark | _klass | i | next ]
02 List p = new $$ListProxy();
   [ _mark | _klass | i | next ]
03 Field f = List.class.getField("i");
04 int off_i = Unsafe.getOffset(f); // 8
05 int proxy_mark  = Unsafe.getLong(p, 0);
06 int proxy_klass = Unsafe.getLong(p, 4);
07 Unsafe.setLong(l, 4, proxy_klass);
08 assert(l instanceof $$ListProxy);
```

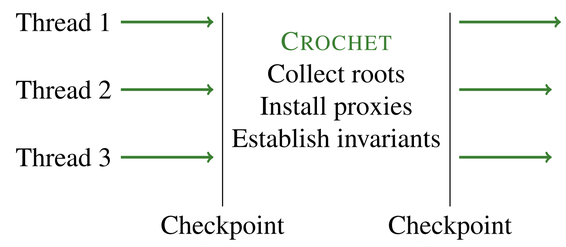
- Objects have JVM-metadata:
  - `_mark` keeps data for: GC, lock, hash-code
  - `_klass` keeps the class of objects
- Meta-data accessible through `sun.misc.Unsafe` pointer-arithmetic-like operations
- Changing `_klass` effectively changes the class of an object: **install proxies dynamically**

- The JIT/GC assumes the program does not change object meta-data
- When inlining, the JIT checks classes at the start of methods
- **This trick breaks that assumption!** This results is JVM crash (`SEGFAULT`) when invoking methods on `l` after line 8 because `l` has an unexpected vtable
- **Preventing line 7 to be inlined** fixes this problem (*i.e.*, put it on large method)

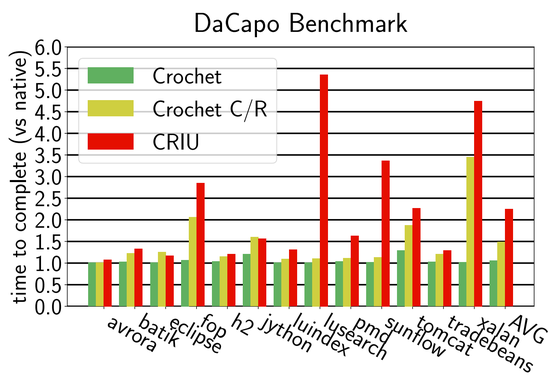
Root Reference Collection

Where are the root references?
- Static fields of loaded classes
  - Instrument class loading
- Live thread objects
  - Instrument thread creation
- Call stack (function args, local vars)
  - Use standard debugger API
- Operand stack (bytecode instruction operands)
  - Tricky, gets compiled away
  - Instrument end of basic blocks to dump current stack when a special flag is set

When checkpoint/rollback happens?
- Need to barrier sync to ensure threads do not race with CROCHET;
- Threads can access the heap concurrently after, CROCHET is thread-safe.



Performance

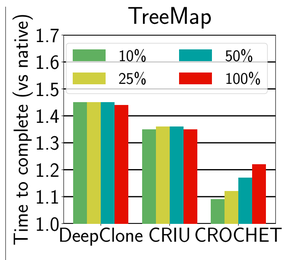


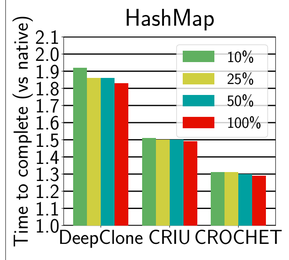
CROCHET No checkpoint

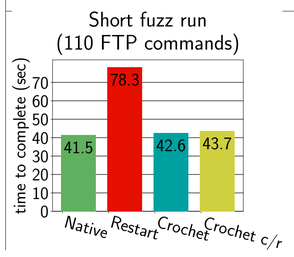
CROCHET C/R Populate benchmark data, checkpoint with CROCHET, run benchmark

CRIU Same, but checkpoint with CRIU[2]

- Very short execution (<300ms): *fop*, *xalan*
  - Constant costs look like high overhead
- Fast (<1s) and big heap (>3G): *lusearch*, *sunflow*
  - Bad for CRIU, great for CROCHET
- Lots of reflection: *jython*, *eclipse*, *tomcat*
  - CROCHET does not optimize reflection







- Fill structure with data, then checkpoint, then access % of structure, compared with **DeepClone**[1] and **CRIU**[2]
- CROCHET is efficient on TreeMap, requiring more effort to checkpoint higher %
- HashMap shows how CROCHET copy the hash-table array eagerly, but still faster than competition

- Fuzz FTP server: **Native**, **CROCHET**;
- Isolate inputs by restarting the server: **Restart**;
- Isolate by checkpoint/rollback: **CROCHET C/R**;

Applications

Checkpoint/Rollback Fuzz testing, code/test generation, time-travel debugging, as application service, etc.

Dynamic Software Update To update, install new code and use proxies to update heap lazily (see Rubah)[3]

Smalltalk `become`: `a.become(b)` means all refs to a become refs to b, and vice-versa; proxies update refs lazily

Dynamic AOP Proxies can add methods, enabling `around` advices and per-object cut-points

[1] DeepClone https://github.com/kostaskougios/cloning
[2] CRIU: Checkpoint/Rollback in User Space: https://www.criu.org/
[3] Pina, Luís and Veiga, Luís and Hicks, Michael. *Rubah: DSU for Java on a Stock JVM*.