

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

Jonathan Bell and Luís Pina

{bellj / lpina2}@gmu.edu
George Mason University

European Conference on Object-Oriented Programming



July 20, 2018

Fuzzing a method

```
01
02     String cmd ;
03
04         ... // Lots of uninteresting code
05
06
07
08             methodToFuzz(cmd);
09
10
11
12
13
14
15
```

Fuzzing a method

```
01
02     String cmd ;
03
04         ... // Lots of uninteresting code
05         while (true) {
06             cmd = mutate(cmd);
07
08                 methodToFuzz(cmd);
09
10
11
12         }
13
14
15
```

Fuzzing a method

```
01
02     String cmd ;
03
04         ... // Lots of uninteresting code
05             while (true) {
06                 cmd = mutate(cmd);
07                 try {
08                     methodToFuzz(cmd);
09                 } catch (Throwable t) {
10                     generateTest(cmd);
11                 }
12             }
13
14
15
```

Fuzzing a method

Example test

```
generateTest("SET key val")
```

Fuzzing a method

Example test

```
generateTest("SET key val")
```



```
methodToFuzz("SET key val")
```

Fuzzing a method

Example test

generateTest("SET key val")



methodToFuzz("SET key val")



Fuzzing a method

Example test

generateTest("SET key val")



methodToFuzz("SET key val")



Fuzzing a method

State is a problem

Buffer:

methodToFuzz("SET key val")

Fuzzing a method

State is a problem

Buffer: SET key val SET key val

methodToFuzz("SET key val")

methodToFuzz("SET key val")

Fuzzing a method

State is a problem

Buffer: SET key val SET key val SET key val

methodToFuzz("SET key val")

methodToFuzz("SET key val")

methodToFuzz("SET key val")



Fuzzing a method

State is a problem

Buffer: SET key val SET key val SET key val

methodToFuzz("SET key val")

methodToFuzz("SET key val")

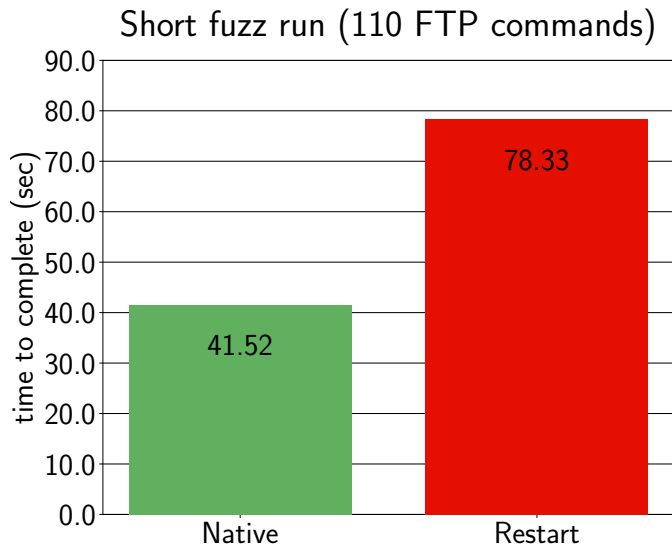
methodToFuzz("SET key val")



generateTest("SET key val")

Fuzzing a method

Restart with fresh state



Checkpoint/Rollback

Buffer:

checkpoint

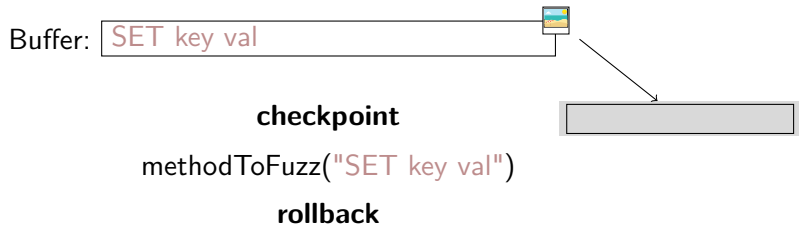
Checkpoint/Rollback



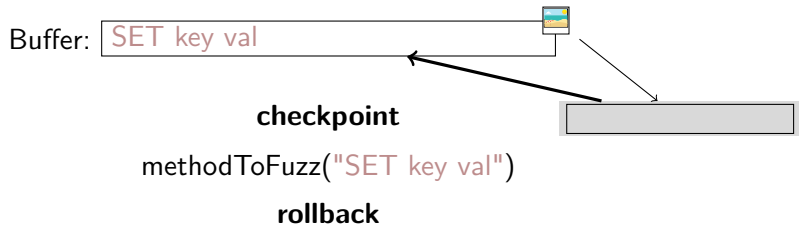
Checkpoint/Rollback



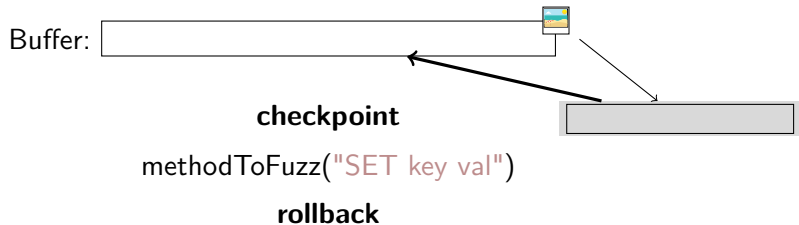
Checkpoint/Rollback



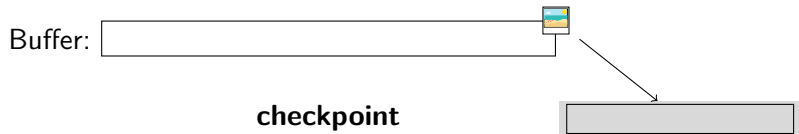
Checkpoint/Rollback



Checkpoint/Rollback



Checkpoint/Rollback



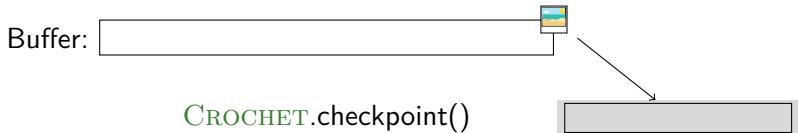
checkpoint

methodToFuzz("SET key val")

rollback

methodToFuzz("SET key val")

Checkpoint/Rollback



`CROCHET.checkpoint()`

`methodToFuzz("SET key val")`

`CROCHET.rollback()`

`methodToFuzz("SET key val")`

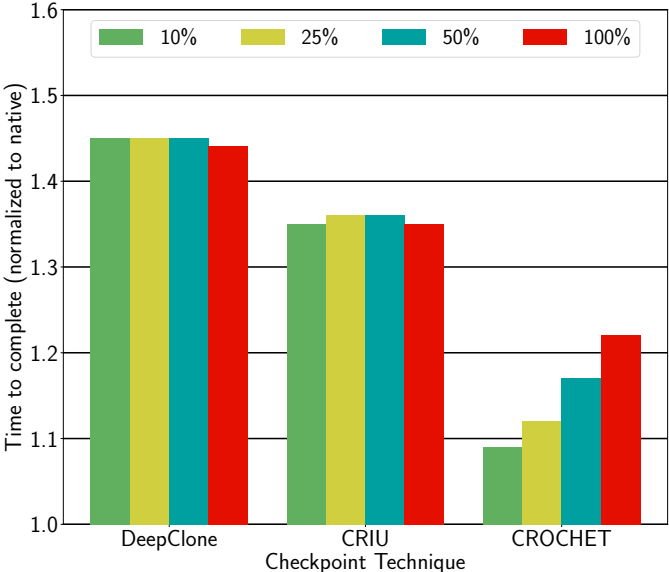
CROCHET

- ▶ No JVM changes (Excludes `fork`)
- ▶ Checkpoint/rollback dynamically
- ▶ Efficient (Excludes `CRIU`)
- ▶ Low overhead (Excludes `STMs`)

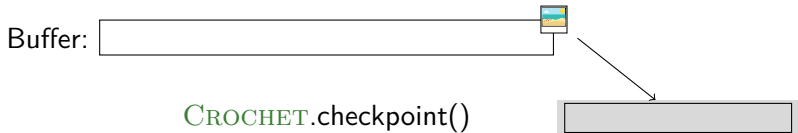
CROCHET

Efficient

Access percent of TreeMap after checkpoint



Checkpoint/Rollback



`CROCHET.checkpoint()`

`methodToFuzz("SET key val")`

`CROCHET.rollback()`

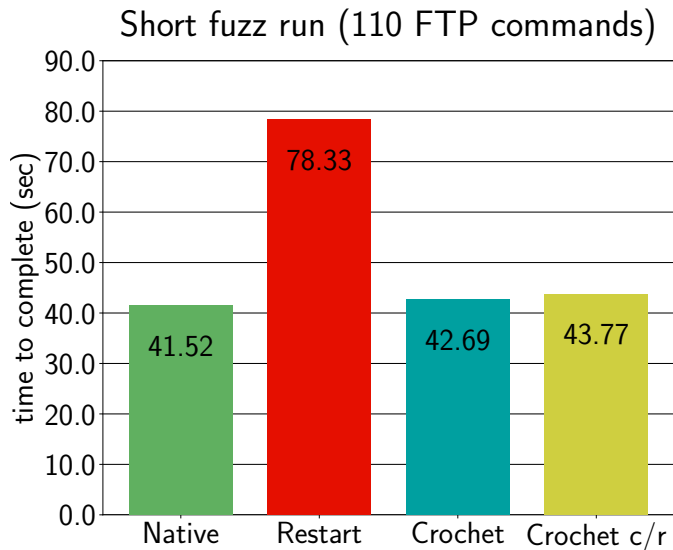
`methodToFuzz("SET key val")`

CROCHET

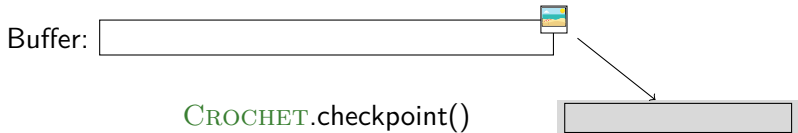
- ▶ No JVM changes (Excludes `fork`)
- ▶ Checkpoint/rollback dynamically
- ▶ Efficient (Excludes `CRIU`)
- ▶ Low overhead (Excludes `STMs`)

CROCHET

Low overhead



Checkpoint/Rollback



`CROCHET.checkpoint()`

`methodToFuzz("SET key val")`

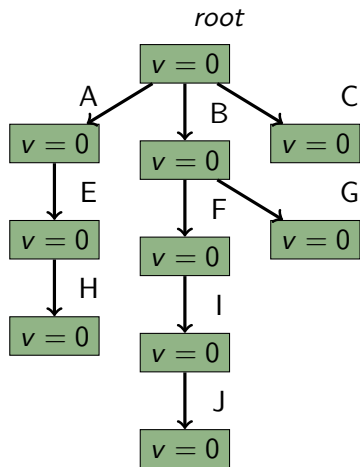
`CROCHET.rollback()`

`methodToFuzz("SET key val")`

CROCHET

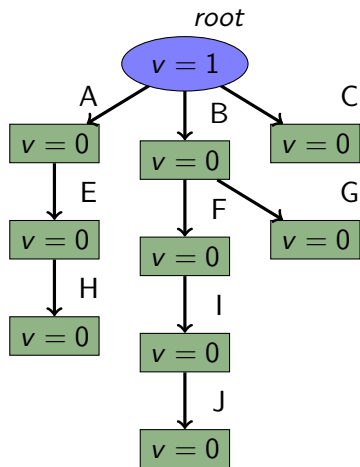
- ▶ No JVM changes (Excludes `fork`)
- ▶ Checkpoint/rollback dynamically
- ▶ Efficient (Excludes `CRIU`)
- ▶ Low overhead (Excludes `STMs`)

Lazy Heap Traversal



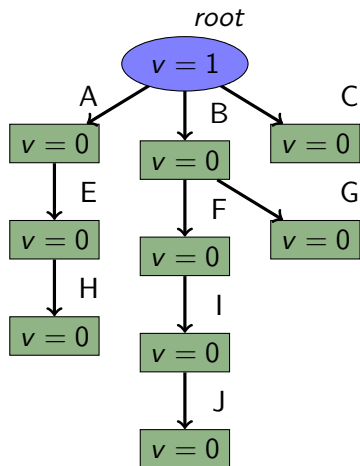
CROCHET.checkpoint()

Lazy Heap Traversal



CROCHET.checkpoint()

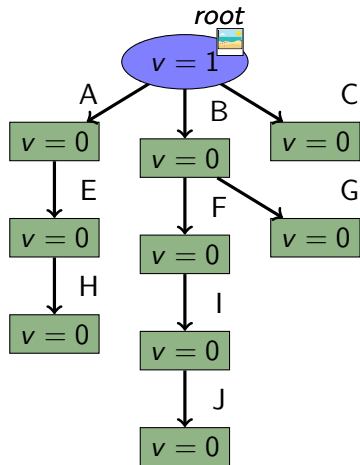
Lazy Heap Traversal



CROCHET.checkpoint()

root.method()

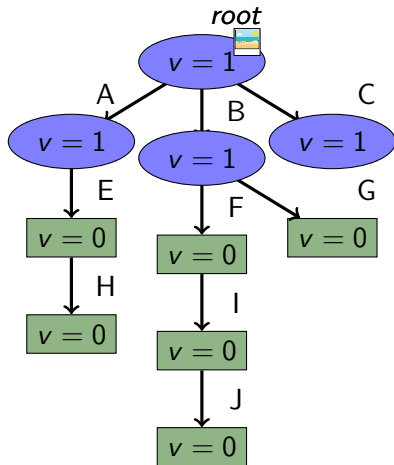
Lazy Heap Traversal



`CROCHET.checkpoint()`

`root.method()`

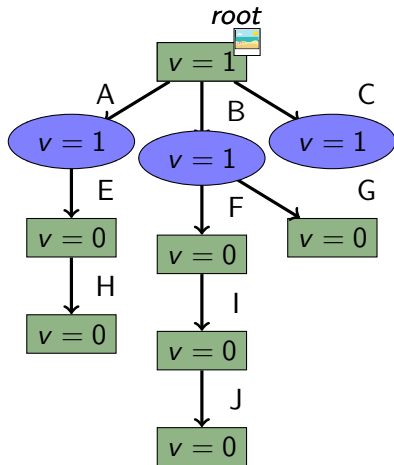
Lazy Heap Traversal



CROCHET.checkpoint()

root.method()

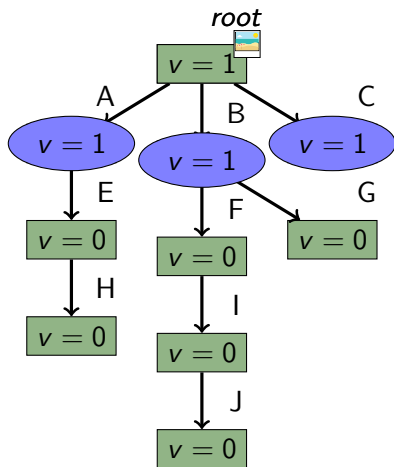
Lazy Heap Traversal



CROCHET.checkpoint()

root.method()

Lazy Heap Traversal

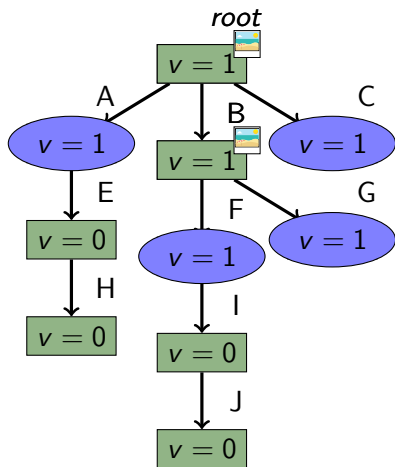


CROCHET.checkpoint()

root.method()

B.method()

Lazy Heap Traversal

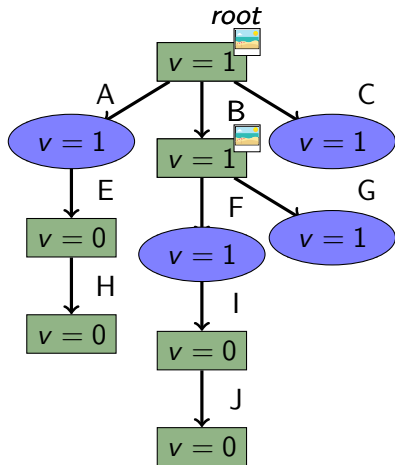


CROCHET.checkpoint()

root.method()

B.method()

Lazy Heap Traversal



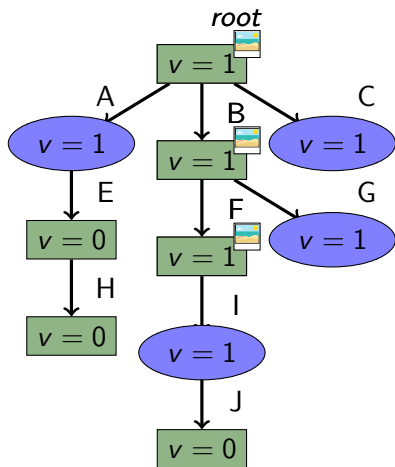
CROCHET.checkpoint()

root.method()

B.method()

F.method()

Lazy Heap Traversal



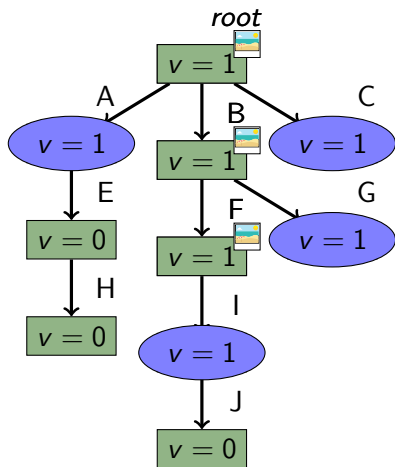
CROCHET.checkpoint()

root.method()

B.method()

F.method()

Lazy Heap Traversal



CROCHET.checkpoint()

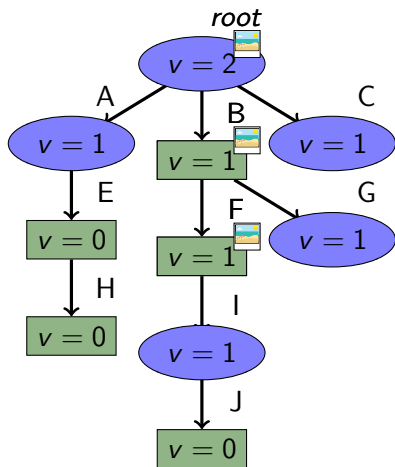
root.method()

B.method()

F.method()

CROCHET.rollback()

Lazy Heap Traversal



CROCHET.checkpoint()

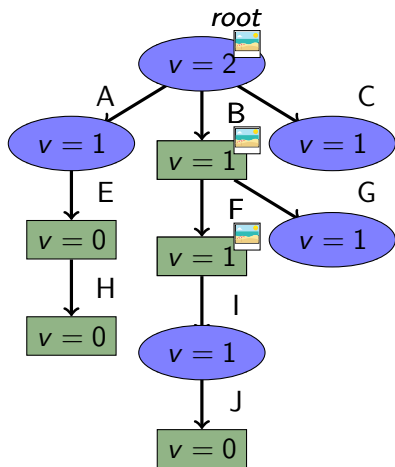
root.method()

B.method()

F.method()

CROCHET.rollback()

Lazy Heap Traversal



CROCHET.checkpoint()

root.method()

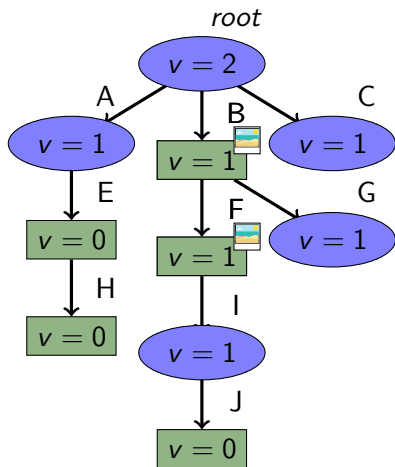
B.method()

F.method()

CROCHET.rollback()

root.method()

Lazy Heap Traversal



CROCHET.checkpoint()

root.method()

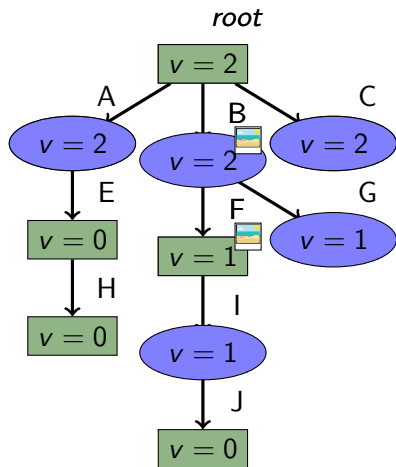
B.method()

F.method()

CROCHET.rollback()

root.method()

Lazy Heap Traversal



CROCHET.checkpoint()

root.method()

B.method()

F.method()

CROCHET.rollback()

root.method()

Lazy Heap Traversal

Tracked object information:

Version 0, 1, ...

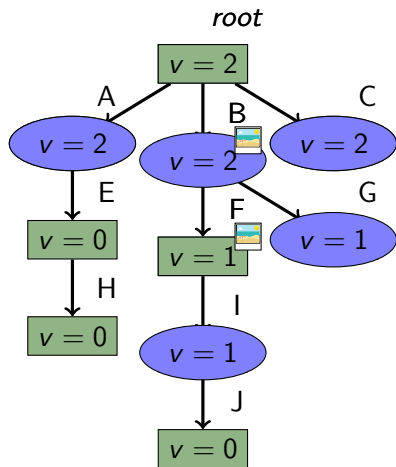
Snapshot Object state to-be-restored, if any

State normal or proxy (Checkpoint/Rollback)

Invariants:

1. Identity: Versions are unique
2. Total Order: Versions increase monotonically
3. Continuity: Proxies mediate access during traversal

Lazy Heap Traversal



CROCHET.checkpoint()

root.method()

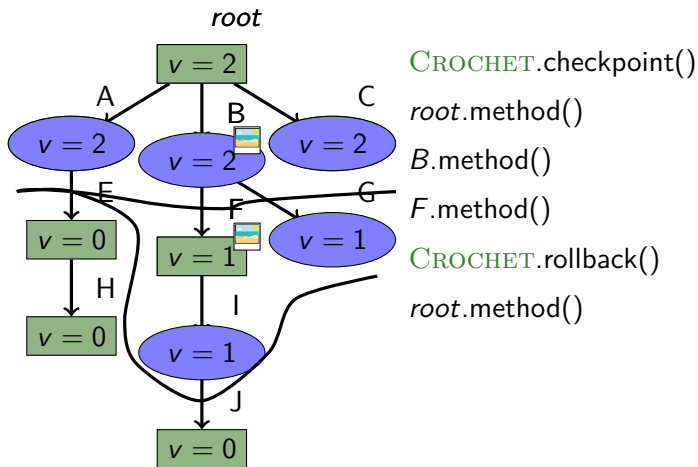
B.method()

F.method()

CROCHET.rollback()

root.method()

Lazy Heap Traversal




Applications

- ▶ Checkpoint/Rollback
 - ▶ Fuzz testing
 - ▶ Code generation
 - ▶ Debugging
 - ▶ As application service
- ▶ Dynamic Software Update
 - ▶ Proxies update objects lazily as reached after update
 - ▶ Rubah
- ▶ Smalltalk become :
 - ▶ `a.become(b)`
 - ▶ All refs to a become refs to b, and vice-versa
 - ▶ Proxies update refs lazily
- ▶ Dynamic AOP
 - ▶ Proxies can add methods, enabling around advices
 - ▶ Per object point-cuts

Goals

CROCHET

- ▶ No JVM changes (Excludes fork)
- ▶ Efficient (Excludes CRIU) 
- ▶ Checkpoint/rollback dynamically
- ▶ Low overhead (Excludes STMs)

Implementation

Bytecode rewriting

```
01 class List {  
02     List next; int i;  
03  
04     int sum()  
05     { return i + next.sum(); }  
06  
07  
08  
09  
10  
11  
12  
13 }
```

Implementation

Bytecode rewriting

```
01 class List {
02     List next; int i;
03
04     int sum()
05     { return i + next.sum(); }
06
07     List $$snapshot; int $$version; // But no status!
08
09
10
11
12
13 }
```

Implementation

Bytecode rewriting

```
01 class List {
02     List next; int i;
03
04     int sum()
05     { return i + next.sum(); }
06
07     List $$snapshot; int $$version; // But no status!
08     void $$onReadWrite(); // Empty
09     void $$onCheckpoint(int version); // Turn into proxy
10     void $$onRollback(int version); // Turn into proxy
11
12     void $$copyFieldsTo(List dest);
13 }
```


Implementation

Bytecode rewriting

```
01 class List {
02     List next; int i;
03
04     int sum()
05     { next.$$onReadWrite(); return i + next.sum(); }
06
07     List $$snapshot; int $$version; // But no status!
08     void $$onReadWrite(); // Empty
09     void $$onCheckpoint(int version); // Turn into proxy
10     void $$onRollback(int version); // Turn into proxy
11
12     void $$copyFieldsTo(List dest);
13 }
```

Implementation

Proxies

```
01 class $$ListProxy extends List {  
02     // No extra fields => Proxy has same size  
03  
04  
05  
06  
07  
08  
09     void $$onReadWrite(); // Snap, propagate, revert proxy  
10     void $$onCheckpoint(int version); // Update version  
11     void $$onRollback(int version); // Update version  
12  
13  
14 }
```

Implementation

Proxy on/off

```
01 List l = new List ();
```

```
02 $$ListProxy p = new $$ListProxy ();
```

Implementation

Proxy on/off

```
01 List l = new List ();
```

_mark	_klass	i	next
-------	--------	---	------

```
02 $$ListProxy p = new $$ListProxy ();
```

_mark	_klass	i	next
-------	--------	---	------

Implementation

Proxy on/off

```
01 List l = new List();
```

_mark	_klass	i	next
-------	--------	---	------

```
02 $$ListProxy p = new $$ListProxy();
```

_mark	_klass	i	next
-------	--------	---	------

```
03 Field f = List.class.getField("i");
```

```
04 int off_i = Unsafe.getOffset(f); // 8
```

Implementation

Proxy on/off

```
01 List l = new List();
```

_mark	_klass	i	next
-------	--------	---	------

```
02 $$ListProxy p = new $$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.getInt(p, 0);
```

```
06 int proxy_klass = Unsafe.getInt(p, 4);
```

Implementation

Proxy on/off

```
01 List l = new List();
```

_mark	_klass	i	next
-------	--------	---	------

```
02 $$ListProxy p = new $$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.getInt(p, 0);
```

```
06 int proxy_klass = Unsafe.getInt(p, 4);
```

```
07 Unsafe.setInt(l, 4, proxy_klass);
```

```
08 assert(l instanceof $$ListProxy);
```

Implementation

Proxy on/off

```
01 List l = new List();
```

_mark	_klass	i	next
-------	--------	---	------

```
02 $$ListProxy p = new $$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.getInt(p, 0);
```

```
06 int proxy_klass = Unsafe.getInt(p, 4);
```

```
07 Unsafe.setInt(l, 4, proxy_klass);
```

```
08 assert(l instanceof $$ListProxy);
```



Implementation

Proxy on/off

```
01 List l = new List();
```

_mark	_klass	i	next
-------	--------	---	------

```
02 $$ListProxy p = new $$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.getInt(p, 0);
```

```
06 int proxy_klass = Unsafe.getInt(p, 4);
```

```
07 Unsafe.setInt(l, 4, proxy_klass);
```

```
08 assert(l instanceof $$ListProxy);
```



Goals

CROCHET

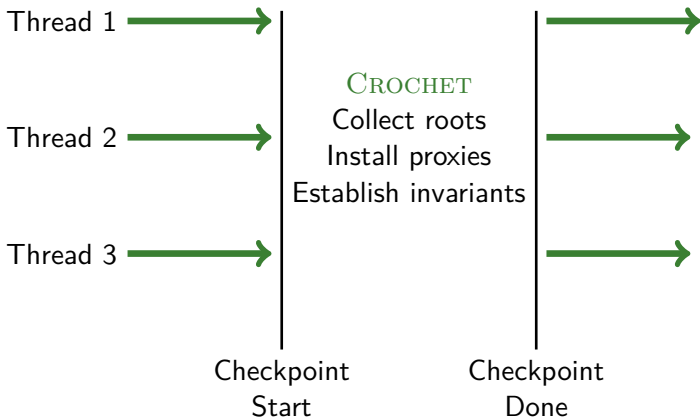
- ▶ No JVM changes (Excludes fork) ✓
- ▶ Efficient (Excludes CRIU) ✓
- ▶ Checkpoint/rollback dynamically
- ▶ Low overhead (Excludes STMs)

Root reference collection

- ▶ 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

Multithreading

- ▶ CROCHET is thread-safe
- ▶ Operations use Compare-And-Swap to synchronize
 - ▶ Rollback not idempotent, needs locks
- ▶ Root collection must be done with threads stopped
 - ▶ Barrier-sync to checkpoint/rollback whole heap



Goals

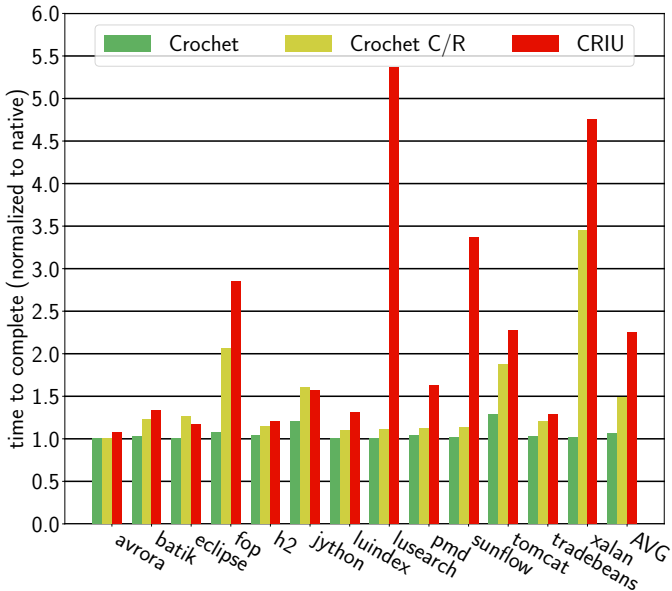
CROCHET

- ▶ No JVM changes (Excludes fork) ✓
- ▶ Efficient (Excludes CRIU) ✓
- ▶ Checkpoint/rollback dynamically ✓
- ▶ Low overhead (Excludes STMs)

CROCHET

Performance

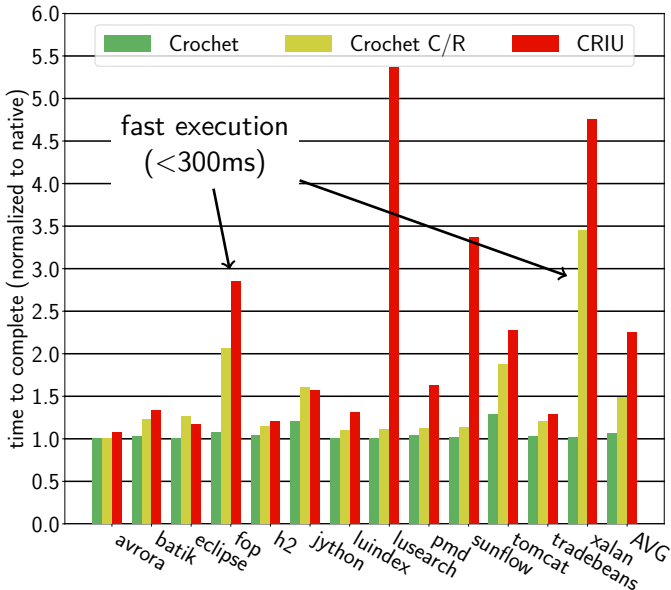
DaCapo Benchmark



CROCHET

Performance

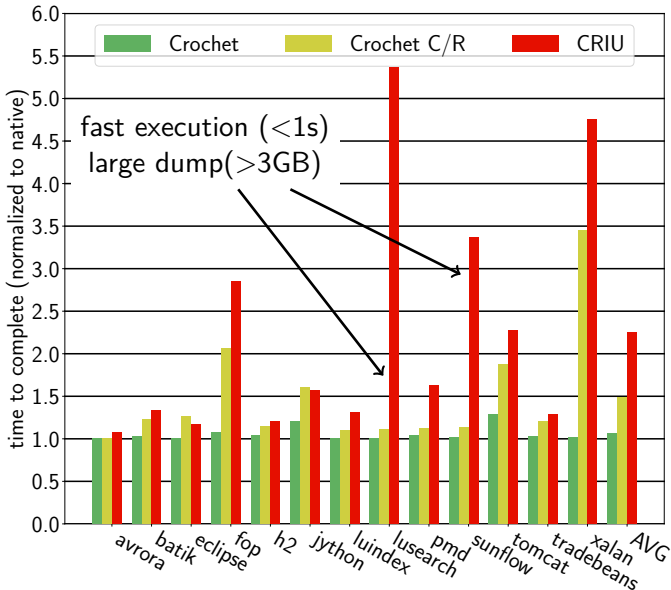
DaCapo Benchmark



CROCHET

Performance

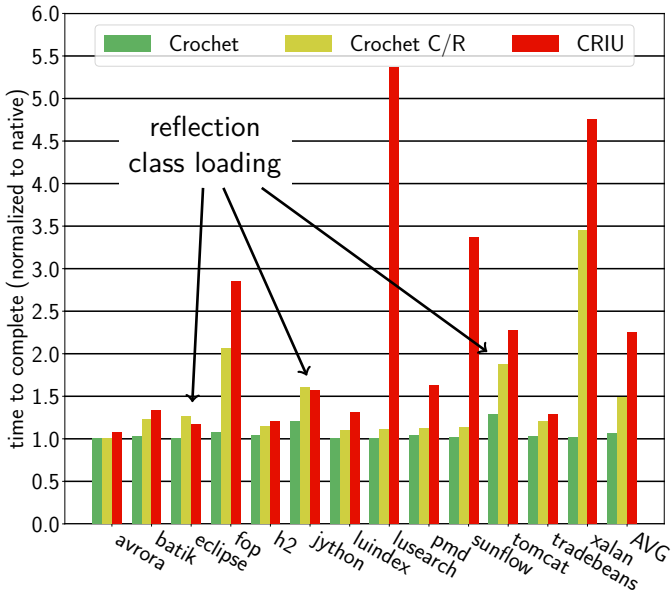
DaCapo Benchmark



CROCHET

Performance

DaCapo Benchmark



Goals

CROCHET

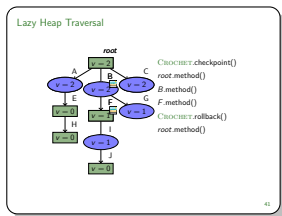
- ▶ No JVM changes (Excludes fork) ✓
- ▶ Efficient (Excludes CRIU) ✓
- ▶ Checkpoint/rollback dynamically ✓
- ▶ Low overhead (Excludes STMs) ✓

CROCHET

Limitations

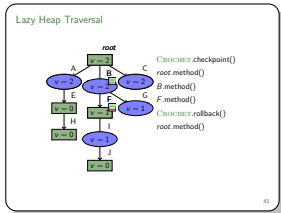
- ▶ Not checkpointed/rolled-back:
 - ▶ Native code and references only held inside native code
 - ▶ Class loaders
 - ▶ State kept outside of the JVM (e.g., files, sockets)
- ▶ Reliance on “unsupported” `sun.misc.Unsafe`
- ▶ Eager checkpoint/rollback of arrays and some classes (e.g., `java.lang.Exception`)

Conclusion



Efficient
lazy heap traversal

Conclusion



Efficient
lazy heap traversal

Implementation

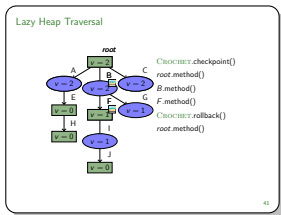
Proxy on/off

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

57

Stock JVM

Conclusion



Efficient
lazy heap traversal

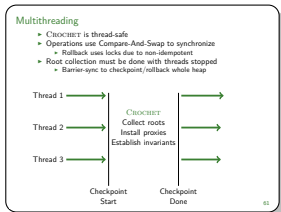
Implementation

Proxy on/off

```
01 List l1 = new List();
   _mark _class 1 next
02 $ListProxy p = new $ListProxy();
   _mark _class 1 next
03 Field f = List.class.getField("");
04 int off_i = Unsafe.getOffset(f); // 8
05 int proxy_mark = Unsafe.getLong(p, 0);
06 int proxy_class = Unsafe.getLong(p, 4);
   Unsafe.setLong(l1, 4, proxy_class);
   assert(1 instanceof $ListProxy);
```

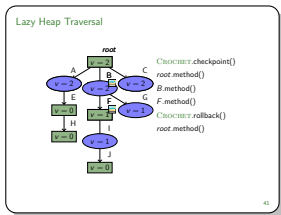
57

Stock JVM



Dynamic checkpoint/rollback

Conclusion




Efficient
lazy heap traversal

Implementation

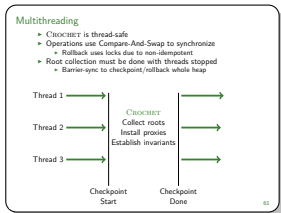
Proxy on/off

```
01 List l = new List();  
   _mark_class 1 next  
02 $$ListProxy p = new $$ListProxy();  
   _mark_class 1 next  
03 Field f = List.class.getField("");  
04 int off_i = Unsafe.getOffset(f); // 8  
05 int proxy_mark = Unsafe.getLong(p, 0);  
06 int proxy_class = Unsafe.getLong(p, 4);  
   Unsafe.setLong(l, 4, proxy_class);  
   assert(! instanceof $$ListProxy);
```

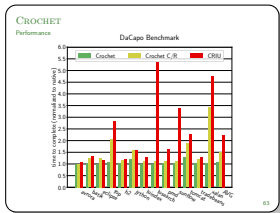


57

Stock JVM



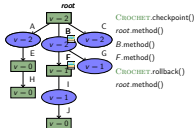
Dynamic checkpoint/rollback



Low overhead

Conclusion

Lazy Heap Traversal



Efficient
lazy heap traversal

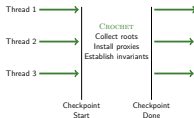
Applications

- ▶ Checkpoint/Rollback
 - Fuzz testing
 - Code generation
 - As application service
 - etc
- ▶ Dynamic Software Update
 - Proxies update objects lazily as reached after update
 - Rollback
- ▶ Smalltalk become:
 - a.become(b)
 - All refs to a become refs to b, and vice-versa
 - Proxies update refs lazily
- ▶ Dynamic AOP
 - Proxies can add methods, enabling around advice

Many applications

Multithreading

- ▶ CROCHET is thread-safe
- ▶ Operations use Compare-And-Swap to synchronize
 - Rollback uses locks due to non-idempotent
- ▶ Root collection must be done with threads stopped
 - Barrier-sync to checkpoint/rollback whole heap



Dynamic checkpoint/rollback

Implementation

Proxy on/off

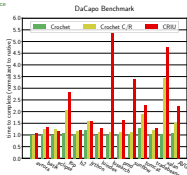
```
01 List l = new List();  
   _mark   _class   i   next  
02 $$ListProxy p = new $$ListProxy();  
   _mark   _class   i   next  
03 Field f = List.class.getField("");  
04 int off_i = Unsafe.getOffset(f); // 8  
05 int proxy_mark = Unsafe.getLong(p, 0);  
06 int proxy_class = Unsafe.getLong(p, 4);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



Stock JVM

CROCHET

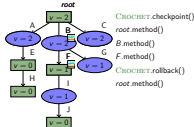
Performance



Low overhead

Conclusion

Lazy Heap Traversal



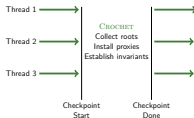
Efficient
lazy heap traversal

Applications

- ▶ Checkpoint/Rollback
 - Fuzz testing
 - Code generation
 - As application service
 - etc
- ▶ Dynamic Software Update
 - Proxies update objects lazily as reached after update
 - Rollback
- ▶ Smalltalk become:
 - a.become(b)
 - All refs to a become refs to b, and vice-versa
 - Proxies update refs lazily
- ▶ Dynamic AOP
 - Proxies can add methods, enabling around advice

Multithreading

- ▶ CHROCHET is thread-safe
- ▶ Operations use Compare-And-Swap to synchronize
 - Rollback uses locks due to non-idempotent
- ▶ Root collection must be done with threads stopped
 - Barrier-sync to checkpoint/rollback whole heap



Dynamic checkpoint/rollback

Implementation

Proxy on/off

```
01 List l = new
   _mark _klass
02 $ListProxy p = new
   _mark _klass
03 Field f = List.class.getField(
04 int off_i = Unsafe.getOffset(
05 int proxy_mark = Unsafe.getLong(
06 int proxy_klass = Unsafe.getLong(
   Unsafe.setLong(1, 4, proxy_klass);
   assert(1 instanceof $ListProxy);
```



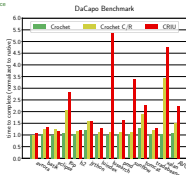
57

Stock JVM

Many applications

CHROCHET

Performance



63

Low overhead

Conclusion

Fork us on GitHub

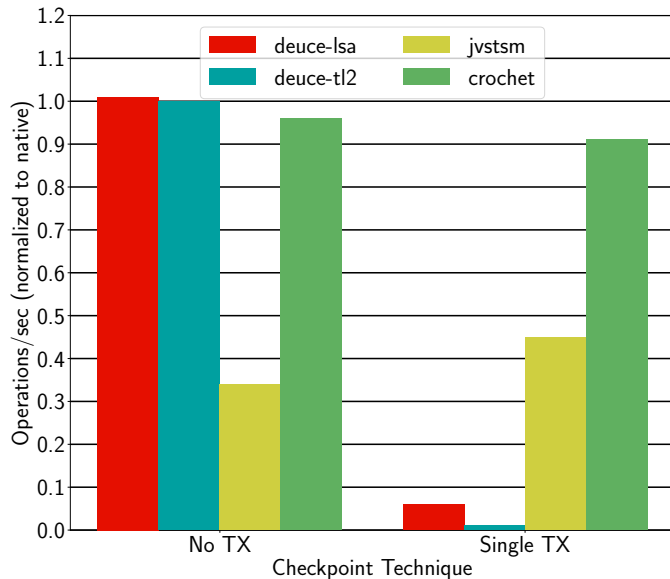
CROCHET

- ▶ No JVM changes
 - ▶ Use bytecode rewriting and standard debug API
- ▶ Efficient
 - ▶ Copy-on-access through thread-safe lazy heap traversal
- ▶ Checkpoint/rollback dynamically
 - ▶ Just barrier-sync threads
- ▶ Low overhead
 - ▶ 6% when not using checkpoint/rollback, 49% otherwise
 - ▶ Beats alternative approaches (CRIU, STM, DeepClone)
- ▶ Many applications
 - ▶ Checkpoint/rollback in itself and to enable other techniques
 - ▶ Specialized proxies for: become:, per-object dynamic AOP, etc.
- ▶ Get CROCHET: <https://github.com/gmu-swe/crochet>

CROCHET

Low overhead

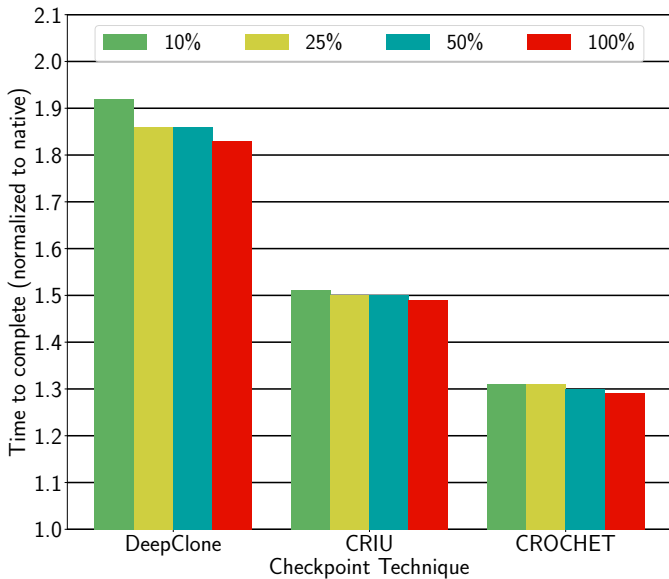
Using STMs for checkpoint/rollback



CROCHET

HashMap

Access percent of HashMap after checkpoint



CROCHET

Operand Stack

```
01 // Original code
02 void someFunc(int i, int[] ar) {
03
04     int j = i + 1;
05     ar[i] = ar[i] - j;
06     j--;
07     otherFunc(j, ar); //Checkpoint called inside otherFunc
08
09
10     ar[i] = 10;
11 }
```

CROCHET

Operand Stack

```
01 //Checkpoint code
02 void someFunc(int i, int[] ar) {
03     boolean captureStack = false;
04     int j = i + 1;
05     ar[i] = ar[i] - j;
06     j--;
07     otherFunc(j, ar);
08     if(captureStack)
09         Checkpointer.captureStack();
10     ar[i] = 10;
11 }
```

CROCHET

Operand Stack

```
01 //Rollback code
02 void someFunc(int i, int[] ar) {
03     int j;
04     boolean captureStack = false;
05     if(Rollbacker.doRollback()) {
06         i = Rollbacker.localInt();
07         ar = Rollbacker.localIntArray();
08         j = Rollbacker.localInt();
09         Rollbacker.removeRollbackCode();
10     } else {
11         // Original code
12         if(captureStack)
13             Checkpointer.captureStack();
14     }
15     ar[i] = 10;
16 }
```

Implementation

Proxy on/off



- ▶ JIT assumes `_klass` does not change
- ▶ That `instanceof` will get JIT'd into `false`
- ▶ Invoking methods on `1` may result in `SEGFALT`
- ▶ Avoid JIT inlining by changing `_klass` on un-inlinable method