Reference Object Processing in On-The-Fly Garbage Collection

Tomoharu Ugawa
Kochi University of Technology
ugawa.tomoharu@kochi-tech.ac.jp

Richard E. Jones Carl G. Ritson
University of Kent
R.E.Jones@kent.ac.uk C.G.Ritson@kent.ac.uk

Abstract

Most proposals for on-the-fly garbage collection ignore the question of Java's weak and other reference types. However, we show that reference types are heavily used in DaCapo benchmarks. Of the few collectors that do address this issue, most block mutators, either globally or individually, while processing reference types. We introduce a new framework for processing reference types on-the-fly in Jikes RVM. Our framework supports both insertion and deletion write barriers. We have model checked our algorithm and incorporated it in our new implementation of the Sapphire on-the-fly collector. Using a deletion barrier, we process references *while mutators are running* in less than three times the time that previous approaches take *while mutators are halted*; our overall execution times are no worse, and often better.

Categories and Subject Descriptors D.3.4 [*Programing Languages*]: Processors—Memory management (garbage collection)

General Terms Algorithms, Languages

Keywords Garbage Collection; Real-time processing; Java; Weak Pointers; Jikes RVM

1. Introduction

The last decade has seen significant changes to the environments in which software is deployed. Developers have turned to managed languages and runtimes for easier deployment and increased security and multi-core processors are ubiquitous. Many applications are sensitive to response time: any pauses may be undesirable or unacceptable. Interactive applications need to respond crisply. Enterprise applications with highly concurrent workloads cannot afford to pause transactions, because delays may lead to either a backlog of re-tried transactions or direct financial loss. Embedded systems may have hard real-time requirements: all operations must complete with a fixed time. Stopping all *mutators* (user threads) in order to reclaim memory (*stop-the-world* or STW) garbage collection (GC) is likely to lead to unacceptable pauses. Even though STW techniques such as generational GC can reduce average pause times, they do not solve the problem of worst-case pause times when the entire heap must be collected.

Incremental GC, which interleaves mutator and collector actions, or *concurrent* GC, which allows mutator and collector threads to run concurrently, address this problem. However, these strategies require careful synchronisation between mutators and collectors for safety (i.e. not reclaiming live objects). Thus, mutators emit *read* or *write barriers* as they access object fields in order to provide the collector with a coherent view of live objects in the heap. Correct termination of a GC cycle is also tricky: the simplest and most widely adopted strategy is to stop all mutator threads briefly to scan their stacks. *On-the-fly* (OTF) collectors, on the other hand, attempt never to stop more than one thread at a time (although even they may have to fall back to STW termination in the face of pathological mutator behaviour).

Problem solved? Unfortunately not. Java provides *reference type* objects — soft, weak and phantom — which can be used for a variety of purposes such as constructing caches, whose contents the collector can reclaim when the system is under memory pressure, or creating canonicalised mappings. Although barriers allow the collector gradually to build up a coherent (if conservative) view of the live objects in the heap, the Java specification requires reference type objects to be dealt with *atomically* by the collector. The simplest solution would be to stop the world while the collector processes reference type objects in the termination phase, but this is unacceptable for a truly on-the-fly collector. Another solution might be to block only those mutators that try to access reference type objects during the GC termination phase; we consider this equally unacceptable.

In this paper, we explore the issues faced by OTF reference type processing, and provide algorithms which we have model-checked and added to our new implementation of the Sapphire OTF copying collector. We find that reference type objects are (possibly surprisingly) heavily used in the DaCapo suite of Java benchmarks [5], although there is substantial variation between programs. Processing reference type objects while mutator threads are halted adds significantly to the time required to terminate a GC cycle. We then show how to process Java's reference type objects in a fully OTF collector. In summary, our contributions are:

- A study of the usage of reference type objects in DaCapo.
- A formalisation of reference type processing in Java.
- A novel algorithm for OTF processing of reference type objects.
- Verification of our algorithm using the SPIN model checker.
- Implementation of the algorithm for the Sapphire OTF copying collector [12] in the Jikes RVM virtual machine.
- Experimental evaluation of both blocking and OTF reference processing methods.

2. Reference Objects

`java.lang.ref` provides *reference object* classes, which support some interaction with the GC. Reference objects may be, in decreasing order of strength, *soft*, *weak* or *phantom*. Soft references are typically used to build caches that the GC can reclaim when

ISMM '14, 12 June, 2014, Edinburgh, UK.

ACM 978-1-4503-2921-7/14/06. . . $15.00.
http://dx.doi.org/10.1145/2602988.2602991

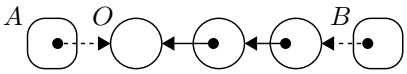

Figure 1: Soft references to be cleared; A and B are soft reference objects. They are assumed to be strongly reachable.

it comes under memory pressure, weak references to implement canonicalised mappings that do not prevent the GC from reclaiming their keys or values, and phantom references for scheduling cleanup actions more flexibly than is possible with finalisation.

The usual Java *strong reference* is created with the `new` operator. Other references are created with the `SoftReference`, `WeakReference` and `PhantomReference` constructors. For instance, the fragment below creates a strong reference `o` to an `Object` (which we call O) and a weak reference `wo` to the same object. Internally, a reference class has a field `referent` that holds a (strong) reference to the reference object's target, i.e. `wo.referent`=`o`:

```
Object o = new Object(); // call this object O
WeakReference wo = new WeakReference(o);
Object maybeNull = wo.get();
...
o = null;
maybeNull = wo.get();
```

O will be preserved by the GC as long as it is reachable by some thread by following a chain of strong references. If O is reachable, `wo.get()` will return a strong reference to O. However, at the second call to `wo.get()`, O may or may not have been reclaimed by the GC, depending on what the elided code marked `...` does. Thus, `wo.get()` may return either a reference to O or `null`.

2.1 Reachability and referent clearing

The different levels of reachability are defined operationally in the `java.lang.ref` package [17].

- An object is *strongly reachable* if it can be reached by some thread without traversing any reference objects. A newly-created object is strongly reachable by its creator thread.
- An object is *softly reachable* if it is not strongly reachable but can be reached by traversing a soft reference.
- An object is *weakly reachable* if it is neither strongly nor softly reachable but can be reached by traversing a weak reference. When the weak references to a weakly-reachable object are cleared, the object becomes eligible for finalization.
- An object is *phantom reachable* if it is neither strongly, softly nor weakly reachable, it has been finalized, and some phantom reference refers to it.
- Otherwise an object is unreachable.

The GC will reclaim any weakly reachable object, and may decide at its discretion to reclaim any softly reachable object. In either case, it must clear the referent field of the reference type.

2.2 The challenge for OTF collection

In a concurrent GC, there may be a race between the collector clearing the reference and a mutator strengthening the reachability of its target by calling `get()`. For this reason, the semantics of reference classes require that, at the time that the GC decides to reclaim a softly reachable object (O in Fig. 1, for example), it must also clear *atomically*

1. *all* soft references to that object (reference from A in Fig. 1), and
2. *all* soft references to other softly-reachable objects from which that object is reachable through a chain of strong references (reference from B in Fig. 1).

At the same time as it clears references or at some later time, the GC may enqueue the reference onto a `ReferenceQueue`; the intention of this mechanism is to notify the program of changes in reachability. Similar semantics apply to weak references / chains of strong and soft references. In contrast, the `get` method of a phantom reference always returns `null`.

Thus, the challenge for OTF processing of soft (resp. weak) references is how to clear not only all soft (weak) references to a softly (weakly) reachable object but also all other soft (weak) soft references to any other softly (weakly)-reachable objects from which that object is reachable through a chain of strong (strong or soft) references in a way that appears atomic to all mutator threads.

3. Related Work

There is a considerable body of work on low pause-time garbage collection, much of it related to real-time systems [1–4, 6–8, 10, 12, 13, 15, 16, 19–24]. Most papers do not address the question of processing reference types.

Azul Systems' Pauseless GC [6] takes the most straightforward approach to avoiding races between mutator and collector threads on reference types: at the end of its marking phase, it stops all mutator threads and does "(in parallel but not concurrent) soft ref processing, weak ref processing, and finalization." Click et al. suggest that they could process references concurrently by having the GC and mutators race to set a value for referent field with a CAS operation if the referent is "not-marked-through". They suggest that if the mutator succeeds the target's reachability is strengthened and the collector knows it, whereas if the GC succeeds the mutator will see the `null`. However, this seems insufficient for correct behaviour as the GC may win one race but lose another, thereby failing to clear atomically all soft references to an object and all soft references to any other softly-reachable objects from which that object is reachable. Reference types are not discussed for other Azul collectors [13, 24]. The Staccato parallel and concurrent real-time compacting collector for multiprocessors [16] also stops the world to process soft, weak and phantom references.

A less intrusive option is to block only those mutators that attempt to call `get()` on reference types. Domani et al. extended the Doliguez-Leroy-Gonthier collector [7] for Java by adding support for finalisation and reference types [8]. They add a read barrier to `get()` that records the referent when called during the marking trace and before weak references have been processed. After the GC has completed its mark phase, it acquires a mutual exclusion lock to prevent any mutator turning a weakly reachable object into a strongly reachable one. Note that having acquired the lock, the GC must check again that marking is complete (since a mutator may have called `get()`) before it clears all reference types with unmarked referents.

The original version of the Metronome real-time collector was incremental, designed for use on a uniprocessor [4]; it did not handle reference types. IBM's first production JVM based on Metronome [1] dealt properly with reference types but was also incremental rather than concurrent.[1] Metronome-TS ("Tax and Spend") [2] is incremental, concurrent and parallel. However, although we believe that it may handle reference types properly in an OTF manner,[2] the paper is reticent on details, stating only: "the

[1] David Grove, personal communication.

[2] David Bacon, personal communication.

full Java language has constructs, such as finalization, weak and soft references, and string interning, which interact in complex ways with memory management. These constructs add phases to the collection cycle: for example, reference clearing can only be considered once marking is over and it is known which objects are reachable."

4. Formal Definition

The semantics of Java's reference types are not described in the Java Language Specification 1.7 [17], which instead refers (section 12.6.2) to the documentation for the package `java.lang.ref`; this package includes the classes `Reference`, `SoftReference`, `WeakReference`, `PhantomReference` and `ReferenceQueue`. Unsurprisingly, the English language descriptions of different levels of reachability given in `java.lang.ref` are unclear and ambiguous.

4.1 Reachability

In this section, we formalise definitions of reachability and, in the next, the actions required when the collector clears reference types. First, let R be a relation on the sets of objects in the heap, `Objects`, and slots in the roots, `Roots`, to objects in the heap,

$$R \subseteq \mathbb{P}((\texttt{Objects} \cup \texttt{Roots}) \times \texttt{Objects})$$

Let $TC(x, R)$ be the transitive closure of a relation R from an object or a slot in the roots, x,

$$TC(x, R) = \{o \in \texttt{Objects} | x R^* o\}$$

and expand TC pointwise:

$$TC(X, R) = \{o \in \texttt{Objects} | \exists x \in X . x R^* o\}$$

We can now define the sets of objects that are strongly, softly, weakly and phantom reachable, given the strong, soft, weak and phantom reference relations $StrR$, $SoftR$, $WeakR$ and $PhantR$. The definition of the set of strongly reachable objects, $StrongReachable$, is straightforward. Objects in this set must be preserved by the GC.

$$StrongReachable = TC(\texttt{Roots}, StrR)$$

However, we believe the `java.lang.ref` definition of 'softly reachable' is misleading: "An object is softly reachable if it is not strongly reachable but can be reached by traversing a soft reference." It does not specify how many soft references we can traverse and when but, worse, it does not require that there be no weak or phantom references in the chain. Without clarifying the kinds of reference that chains may *not* contain, the definitions in the API are inconsistent. We believe the correct specification is,

$$\begin{aligned} SoftReachable &= TC(\texttt{Roots}, StrR \cup SoftR) \\ &\quad - StrongReachable \end{aligned}$$

The API definition of 'weakly reachable' is similarly unsatisfactory. Instead, we define

$$\begin{aligned} WeakReachable &= TC(\texttt{Roots}, StrR \cup SoftR \cup WeakR) \\ &\quad - StrongReachable - SoftReachable \end{aligned}$$

Finally, we define the set of phantom reachable objects. To be phantom reachable, an object must have been finalised, so we assume a set of objects that have been finalised, $Finalised$.

$$\begin{aligned} &PhantomReachable = \\ &(TC(\texttt{Roots}, StrR \cup SoftR \cup WeakR \cup PhantR) \cap Finalised) \\ &\quad - StrongReachable - SoftReachable - WeakReachable \end{aligned}$$

4.2 Clearing references

The GC clears all references to a weakly reachable object, and can elect to clear references to a softly reachable object. If it decides to do so, it must clear atomically *all* soft (resp. weak) references to the object and *all* soft (weak) references to any other softly (weakly) reachable objects from which that object is reachable through a chain of strong (soft and strong) references. At the same time it will declare all of the formerly weakly-reachable objects to be finalizable and, at that time or later, enqueue those newly-cleared references that are registered with reference queues.

Thus, if the GC decides to clear a soft reference to a softly reachable object o, it must also clear all soft references to the set

$$softToClear(o) = \{w \in SoftReachable | w\ StrR^*\ o\}$$

In Fig. 1, $softToClear(O)$ is the set of all the normal objects.

The API encourages the GC to preserve some set P of recently created or used soft references. To accommodate this, we modify the definition of $StrongReachable$ used to calculate $SoftReachable$ in order to retain objects in P:

$$StrongReachable' = TC(\texttt{Roots} \cup P, StrR)$$

Thus, $SoftReachable'$ represents the set of softly-reachable objects that are referents of soft references we have to clear atomically.

In the case of weak references, the GC must clear all weak references to the set

$$\begin{aligned} weakToClear(o) &= \{w \in WeakReachable | \\ &\quad w\ (StrR \cup SoftR)^*\ o\} \end{aligned}$$

`PhantomReference.get()` always returns `null` so there is no need to clear phantom references.

5. On-the-fly Reference Processing

In this section, we describe our design and implementation of OTF processing of reference types. Java provides three types of reference objects, and the GC must take care to process these in the correct order, but we do not discuss that here. The focus of our work is how to have an OTF GC deal with reference type objects — determining reachability, clearing referents atomically — without blocking the mutators. For simplicity of exposition, we concentrate on how we treat weak references, but our implementation deals properly with all types of reference object.

5.1 Concurrent GC

Concurrent GC algorithms require mutator and GC threads to share a consistent view of the heap. This is typically achieved through *read* or, more commonly, *write barriers*, which impose a small overhead on mutator operations [25]. The synchronisation between mutator and GC is best described through the well-known tricolour abstraction [18]. In this abstraction, every object is assigned one of three colours. *White* objects are unknown to the GC; *grey* objects are known the GC but need to be visited again (e.g. to trace their children); and *black* objects are known to the GC but need not be visited again. Tracing live objects is complete as soon as the heap contains no grey objects: at this point, all strongly reachable objects have been marked black and, furthermore, *if we were to ignore reference types*, any white object can be reclaimed by the GC.

Write barriers can be used to ensure consistency between mutators and collectors by colouring objects. *Insertion* (or 'incremental update') barriers[3] shade grey a white target Dijkstra-style (or the source, Steele-style) of a reference write, thus preserving the strong tricolour invariant:

"There are no pointers from black objects to white objects."

[3] We use the terminology of [14].

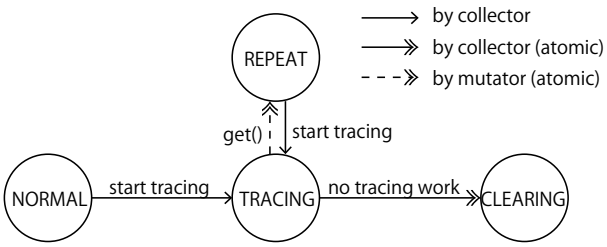

Figure 2: Global reference state transitions.

Deletion (or 'snapshot-at-the-beginning') barriers shade grey the old white target of a deleted reference, preserving the weak tri-colour invariant:

> "All white objects pointed to by a black object are reachable from some grey object, either directly or through a chain of white objects."

Colours can be extended to mutators. The roots of a *grey mutator* may refer to objects of any colour. Thus, the GC must rescan its roots in order to terminate tracing. In an OTF setting, the GC may have to stop a grey mutator repeatedly to scan its roots until is has succeeded in scanning all mutator threads without having discovered a new grey object. In contrast, once the roots of a *black mutator* have been scanned by the collector, they do not need to be scanned again. Insertion barriers preserve the strong invariant for grey mutators but cannot support black mutators [18]. Instead, a black mutator must use a deletion barrier.

5.2 Reference types and termination

The approaches described above are sufficient to ensure safe termination of a GC cycle in the absence of reference types. At the conclusion of the trace, they guarantee that no mutator can reach an unmarked, i.e. white, object through any path of strong references. This is an essential requirement for the safe reclamation of white objects. Note that the GC has traced only strong references at this point; it does not trace the referents of other reference types. However, reference types introduce a wrinkle: a mutator can acquire a strong reference to a weakly reachable object by calling `get()` on a weak reference that has yet to be cleared. There are several known solutions to this problem in a concurrent collector, all of which block mutators to one degree or another.

The simplest [1] is to stop the world while the collector traces again in order to identify the $weakToClear(o)$ set defined in Sec. 4.1; weakly reachable white objects other than this set and its transitive closure under the strong reference relation are marked black. A global flag is then set to indicate that the references have been cleared logically, and the mutators are resumed. Once the flag is set and while the GC is concurrently and physically clearing selected weak references, `WeakReference.get()` returns `null` if the referent is white. Domani et al. [8] relax the requirement to stop the world by having the GC acquire a lock that blocks until the end of the reference processing phase any mutator that calls `get()` while the GC is processing reference types.

However, in a fully OTF collector, we do not want to stop any mutator, other than briefly to scan its roots. We describe our solutions for (a) insertion barriers and (b) deletion barriers next.

5.3 Insertion barrier solution

Our insertion barrier solution is a natural extension to the GC's incremental update termination loop: we repeatedly scan roots and trace until our work queue is empty. We add to the GC a global flag that can take one of four states: `NORMAL`, `TRACING`, `CLEANING` and `REPEAT`. The state transition diagram for this flag is shown in Fig. 2, and pseudocode for the collector and for the `WeakReference.get` method is shown in Figs. 3 and 4.

```
collection() {
  insertionBarrier ← ON;
  transitiveClosureFromRoot();
  while(true) {
    refState ← TRACING;
    handshake();
    transitiveClosureNoRootScan();
    scanRoot();
    if(workQueue.empty() &&
       CAS(refState, TRACING, CLEANING))
      break;
  }
  insertionBarrier ← OFF;
  clearReference();
  refState ← NORMAL;
  handshake();
  reclaim();
}
```

Figure 3: Insertion barrier: collector

```
get() {
  while(true) {
    switch(refState) {
    case NORMAL: case REPEAT:
      return referent;
    case TRACING:
      if (referent=null ∥ COLOR(referent)≠WHITE)
        return referent;
      CAS(refState, TRACING, REPEAT);
      break; /* retry */
    case CLEANING:
      if (referent=null ∥ COLOR(referent)≠WHITE)
        return referent;
      return null;
    }
  }
}
```

Figure 4: Insertion barrier: `WeakReference.get()`

The system is in the `NORMAL` state when GC is not running. To start a new GC cycle, the collector turns on the insertion barrier[4] and traces strong references in the usual way before starting the insertion barrier termination loop. The task of the termination loop is to establish a consistent view, shared between mutators and collectors, of the reachability of objects while the mutators are running and possibly calling `get()`.

The loop starts by setting the tracing state to `TRACING` and handshaking with mutators to ensure that they have all observed this transition. The collector then traces from the work queue of grey objects (`transitiveClosureNoRootScan`) and scans the roots again. If the work queue is empty, the GC attempts to set the tracing state to `CLEANING` using an atomic compare and swap operation (CAS). If this succeeds, tracing has terminated and the

[4] We do not discuss here the subtleties on initiating a GC cycle for an OTF collector; see [14] for details.

```
collection() {
  insertionBarrier ← ON;
  transitiveClosureFromRoot();
  deletionBarrier ← ON;
  handshake();
  scanRoot();
  insertionBarrier ← OFF;
  while(true)
    refState ← TRACING;
    handshake();
    transitiveClosureNoRootScan();
    if(workQueue.empty() &&
       CAS(refState, TRACING, CLEANING))
      break;
  }
  deletionBarrier ← OFF;
  clearReferences();
  refState ← NORMAL;
  handshake();
  reclaim();
}
```

Figure 5: Deletion barrier: collector

```
get() {
  while(true) {
    switch(refState) {
    case NORMAL:
      return referent;
    case REPEAT:
      if (referent=null ∥ COLOR(referent)≠WHITE)
        return referent;
      COLOR(referent) ← GREY;
      return referent;
    case TRACING:
      if (referent=null ∥ COLOR(referent)≠WHITE)
        return referent;
      if (CAS(refState, TRACING, REPEAT)) {
        COLOR(referent) ← GREY;
        return referent;
      }
      break; /* retry */
    case CLEANING:
      if (referent=null ∥ COLOR(referent)≠WHITE)
        return referent;
      return null;
    }
  }
}
```

Figure 6: Deletion barrier: `WeakReference.get()`

collector can proceed to clearing weak objects' referents. At this point, we know precisely which objects to preserve and which weak reference objects' `get` methods should return `null`.

However, while the GC is `TRACING`, a mutator may call `get()` on a weak reference object whose referent is white. In this case, the mutator needs to force the collector to iterate its termination loop again. We do so by adding a read barrier to `get` that atomically sets the GC's state to `REPEAT`; to avoid a race condition, the mutator must also retry `get`'s loop. While the GC is not `TRACING` or has to `REPEAT`, or if the referent is not white, `get()` simply returns the referent. The read barrier does not need to shade the target of the referent as we can defer this to the next root scan. However, once the GC has transitioned to the `CLEANING` state, `get()` must return `null` for any non-white, i.e. black, referent. In this way, it appears to the mutator that all weak reference objects selected for clearing are cleared atomically. Once the weak references have been cleared, the GC resets its state to `NORMAL`, handshakes with the mutators and then reclaims any garbage.

5.4 Deletion barrier solution

Measurements of reference processing with an insertion barrier (Sec. 7) showed that the GC sometimes takes a long time to complete due to the frequent use of `WeakReference.get()`. Perhaps this is not surprising since insertion barrier mutators are grey and so may need to be scanned several times to terminate, even in the absence of reference types. In contrast, a black mutator supported by a deletion barrier needs its stack to be scanned only once.

Our deletion barrier variant for reference object processing is shown in Fig. 5 and 6. We run the Sapphire OTF collector with its insertion barrier on as usual during the initial marking phase. However, before we start to process reference types, we switch on a deletion barrier, handshake with the mutators to ensure that they notice, and scan the roots. Note that an OTF collector requires both insertion and deletion barriers to be on while thread stacks are scanned to blacken the mutators [7]. Once the roots have been scanned, we can turn the insertion barrier off.

As with the insertion barrier, our deletion barrier scheme also requires a termination loop but does not need to rescan mutator roots. Instead, the `get` read barrier must shade the referent in both the `REPEAT` and the `TRACING` states as we cannot rely on a future root scan to discover it.

5.5 Soft and Phantom references

Phantom reference objects are straightforward to process in an OTF manner since there is no interaction between collector and mutator: `PantomReference.get()` always returns null. However, the API encourages the GC to retain recently allocated or used soft reference objects. We treat these by marking them and their strong transitive closures black. We treat any other soft reference objects in the same way as weak reference objects.

6. Model Checking

We verified our framework for OTF reference processing using the SPIN model checker [11]. SPIN exhaustively checks all possible interleavings of processes. Since we deal with soft references as if they are strong or weak references depending on memory pressure, we focused on weak references. We verified the following properties:

P1 (Safety) A mutator will never see a reclaimed object.

P2 (Consistency) Once a `get()` method called on a reference object returns `null`, a mutator will never see the referent of that object.

These properties are from the mutator's view because there can be a variety of implementations of 'clearing'. In our implementation, logically cleared references appeared cleared to mutator. Property **P1** is required regardless of the existence of reference objects. But **P1** also requires that, if a mutator loads a referent of a reference object, the referent has not been reclaimed. Property **P2** implies the atomicity that the API definition requires. Assume that a weak reference to an object o is cleared in spite of retaining another weak reference to an object $w \in weakToClear(o)$. Since the weak reference is cleared, a mutator may see that `get()` returns `null`. But,

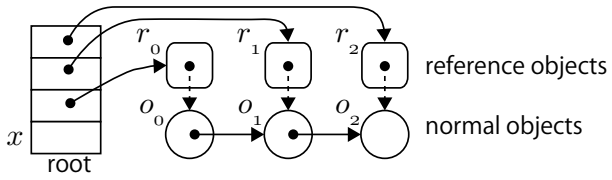

Figure 7: The model

```
while(true) {
  int i = random.nextInt(5);
  switch (i) {
    case 0: x = vr0.get(); break;
    case 1: x = vr1.get(); break;
    case 2: x = vr2.get(); break;
    case 3: if (x ≠ null) x = x.next; break;
    case 4: x = null; break;
  }
}
```

Figure 8: Simple mutator

since o is weakly-reachable through w, the mutator may also see o, contrary to **P2**.□

Since bounded model checking does not deal with infinite state, we checked the properties for the limited model shown in Fig. 7. This model has three pairs of reference and normal objects, namely r_0, r_1, r_2 for references and o_0, o_1, o_2 for the corresponding normal objects. These normal objects are linked in a list, but there are no other strong references to them. We assumed that all reference objects remain directly strongly reachable from the root and that the mutator can always call `get()` methods on them.

Fig. 8 shows the mutator's pseudocode: vr_i is a local variable whose value is a reference object r_i, and x is another local variable. The mutator repeatedly and arbitrarily calls a `get()` method to load the referent to x, loads the 'next' object of x, or clears x. Since we focus on the behaviour of references, the mutator does not write to any object. Thus, our model does not have write barriers.

Fig. 9 shows the model of the `get()` method on the reference object r_i, for a collector using an insertion barrier. This model is faithful to Fig. 4. The return value is passed to the caller through the parameter ret. `mark`$[i]$ and `CLEARED`$[i]$ represent the colour of o_i and whether r_i has been cleared or not, respectively. When `get()` returns o_i, it sets i to ret. In order to check **P2**, the model also puts i and ret in global variables `getRef_arg` and `getRef_ret`.

For the collector side, our model is faithful to the pseudocode in Fig. 3 and 5. At the end of a cycle, the collector reclaims white objects by calling `reclaim()`: we introduce a fourth object state `RECLAIMED`. Our model of `reclaim()` reclaims white objects and reverts the black objects to white. **P1** and **P2** can be interpreted as:

P1 $\Box((x \neq \texttt{NULL}) \implies (\texttt{mark}[x] \neq \texttt{RECLAIMED}))$

P2 $\Box(\texttt{RETNULL}_i \implies \neg\Diamond(x = i))\ \ (i = 1, 2, 3)$

where $\texttt{RETNULL}_i \equiv(\texttt{getRef\_arg}= i)\wedge(\texttt{getRef\_ret}= \texttt{NULL})$.

We have model checked these properties with models both for collectors with an insertion barrier and a deletion barrier. We also tried to model check the termination property.

P3 (Termination) GC eventually terminates.

```
inline getRef(i, ret) {
  do::(refState == NORMAL ||
       refState == REPEAT) ->
      if::CLEARED[i] -> ret = NULL
        ::else       -> ret = i
      fi;
      break
    ::(refState == TRACING) ->
      if::(!CLEARED[i] && (mark[i] == WHITE)) ->
          CAS(refState, TRACING, REPEAT)
          /* continue */
        ::(!CLEARED[i] && (mark[i] != WHITE)) ->
          ret = i;
          break
        ::else ->
          ret = NULL;
          break
      fi
    ::(refState == CLEANING) ->
      if::(!CLEARED[i] && (mark[i] == WHITE)) ->
          ret = NULL
        ::(!CLEARED[i] && (mark[i] != WHITE)) ->
          ret = i
        ::else -> ret = NULL
      fi;
      break
  od;
  d_step { /* d_step is an atomic action */
    getRef_arg = i;
    getRef_ret = ret
  };
}
```

Figure 9: Promela model of a `Reference.get()` method with an insertion barrier

However, we found that, with an insertion barrier, the mutator can continually prevent the collector from breaking out of the termination loop, even if we assume weakly fair scheduling. The reason for this is that, while the collector is tracing or checking if the work queue is empty, a mutator has a chance to load a white referent to a local variable x and then clear x. The mutator changes *refState* to `REPEAT` when it loads a reference with `get()`, thus forcing the collector to trace again. However, if the mutator has cleared x, the collector will not find, and hence shade, a new white referent: the number of white objects is not reduced and so no progress is made. Fortunately, the deletion barrier version *does* make progress, since `get()` shades white objects grey.

7. Evaluation

We built our OTF reference processing framework in Jikes RVM and evaluated it with our new implementation of the Sapphire collector [12], running DaCapo benchmarks that would run (10 from the 2006 and 6 from the 2009 suite). All measurements were performed on a 4-core, 3.4 GHz Intel Core i7-4770 CPU running Ubuntu Linux 12.04.4.

7.1 Reference Type Usage

To understand the behaviour of the benchmarks, we measured how often reference types were used. Fig. 10 shows the number of calls of a `get()` method per second in each 10 ms time window; the x-axis is the normalised elapsed time of the program.

a

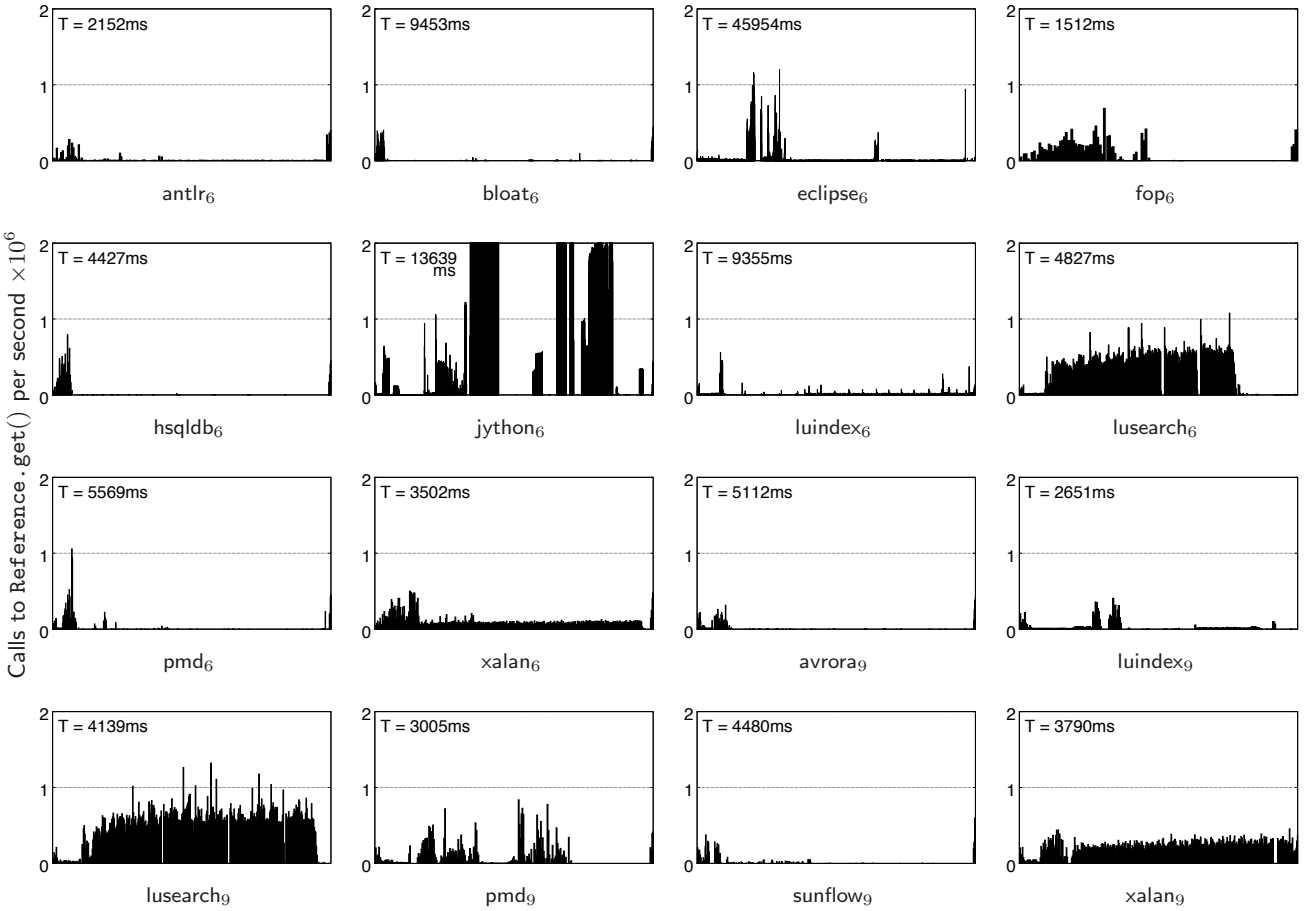

Figure 10: Frequency of calls to `get()` in DaCapo (10 ms quanta); the x-axis is the normalised entire execution time, T.

Contrary to our expectations, some programs used reference types heavily (more than 1 million times per second), but this varies between programs. Most showed a small peak at the beginning of execution; we found that these were due to the Jikes RVM class loader. Often programs made little further use of reference types but jython$_6$ made heavy use of them in particular phases. In contrast, lusearch$_6$, lusearch$_9$, xalan$_6$ and xalan$_9$ made substantial use of reference types for much of their execution. We conclude that

LESSON 1. *An OTF GC must not ignore reference types.*

7.2 Reference Processing Time: blocking schemes

Some implementations block mutator threads during reference type processing. To measure how long a mutator thread would be blocked if reference types were not handled OTF, we implemented two additional variants of reference type processing.

STW The collector stops all mutators during reference processing (tracing to determine reachability and clearing references).

lock The collector acquires a lock to block any mutator that calls `get()` during reference processing. For this variant, we experimented with both insertion and deletion barriers.

To stress the reference type processing mechanisms, we configured Sapphire to perform back-to-back collection cycles, initiating the next as soon as the previous one had completed, using two collector threads. To avoid having any GC cycle fall back from OTF to stop-the-world collection because a mutator is starved of memory, we ran each benchmark in a 1 GB heap.We do not pretend that this is a suitable configuration for normal use. Despite this, we observed starvation in an execution of lusearch$_6$ and lusearch$_9$ due to their high allocation rate. We ignored any pauses not due to reference processing.

Fig. 11 shows frequency distribution histograms of reference processing pauses, using 3 ms bins, in jython$_6$ and DaCapo 2009 benchmarks (luindex$_9$, whose result was similar to avrora$_9$, is omitted to save space), using 'compiler replay' (methods precompiled based on an off-line profile [9], measurements started on benchmark rather than VM startup); each benchmark was executed three times (there was little variation between executions). Note that the **STW** histogram shows the distribution of times taken *to complete the entire reference processing phase*, whereas the **lock** histograms show the distribution of times *for which a mutator was blocked* because it called `get()` in a reference processing phase.

Unsurprisingly, stop-the-world reference processing consistently stopped mutators for only a limited time. Most pauses were less than 6 ms (maximum 9 ms). In contrast, the locking approach sometimes stopped mutators for up to 12 ms in lusearch$_9$ and xalan$_9$, regardless of the barrier used. We also measured reference processing time without compiler replay and taking VM startup into account. Broadly, the distributions were similar, although pauses taking VM startup into account were longer, up to 24 ms in xalan$_9$ (Fig. 12).

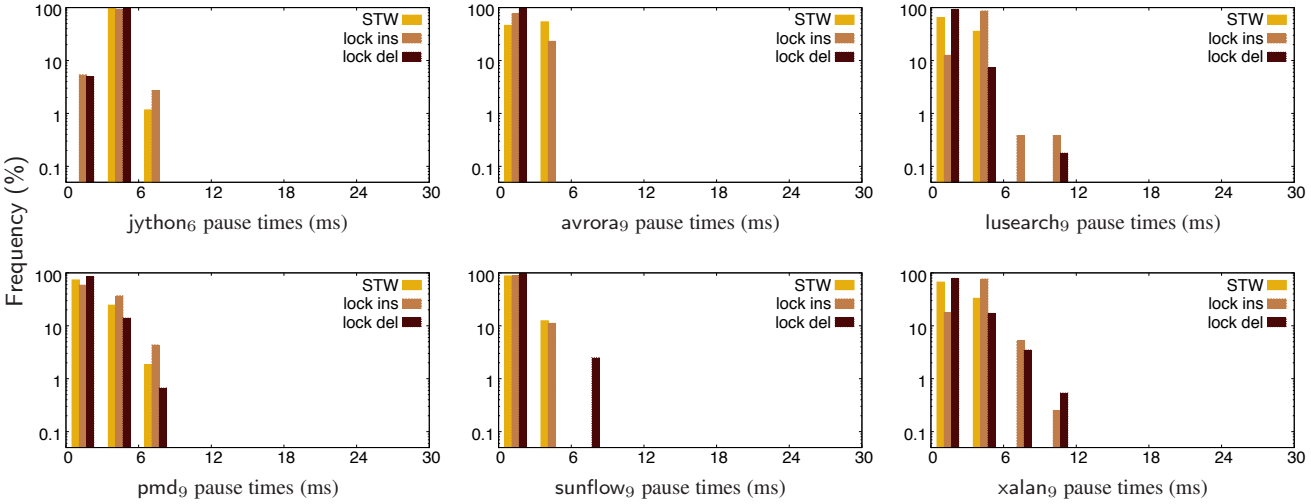

Figure 11: Frequency of pause times (3 ms bins) in jython$_6$ and DaCapo 2009 (compiler replay, measurement starts with benchmark). Note that the y-axis is logarithmic.

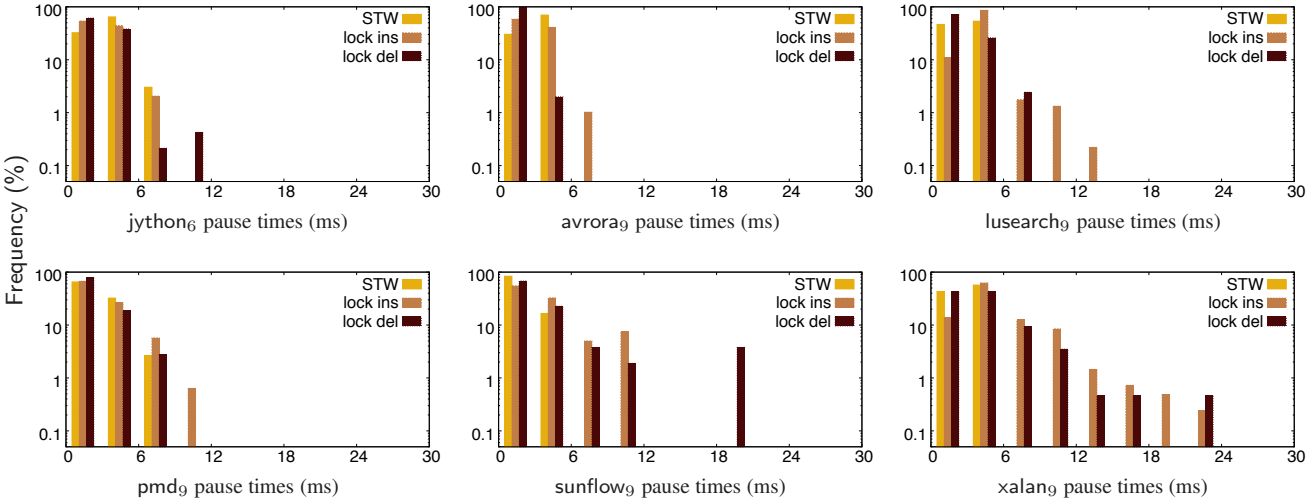

Figure 12: Frequency of pause times (3 ms bins) in jython$_6$ and DaCapo 2009 (without compiler replay, measurement starts with VM). Note that the y-axis is logarithmic.

By processing references in stop-the-world manner, the collector can use all resources available, and so reference processing completes promptly. In contrast, if we allow mutators to run as long as they do not use references, the collector may run more slowly. Handshakes with the mutators will also delay the collectors. Furthermore, write barriers may potentially force collectors to trace again though we did not observe this. All of this extends reference processing time. Thus, the locking approach may block mutators for longer than the stop-the-world approach. In contrast, our OTF reference processing framework does not pause mutators other than to scan an individual mutator's roots.

7.3 OTF Reference Processing Time

Mutator calls to `get()` may force an OTF collector to trace again several times, potentially endlessly (Sec. 6). However, in practice reference processing terminates in a short time. Tab. 1 shows how many times the GC was forced to re-trace, for our different variants of reference processing, using compiler replay. In our measurements, we never observed more than one iteration for the **STW** and **lock** approaches, although this is theoretically possible.

With **OTF**, we observed multiple re-tracings, particularly and unsurprisingly using an insertion barrier. Most benchmarks need to re-trace more than 10 times and lusearch$_9$, which has many mutator threads that allocate at a high rate, needed 62 iterations. One reason for this behaviour is that insertion barrier collectors scan all the mutators even though there may be only few objects to be traced. Scanning mutators takes long time, which increases the chance of mutators forcing the collectors to trace again. In contrast, processing with deletion barriers tended to converge in at most 3 iterations.

Fig. 13 shows the distribution of times for OTF reference processing phases (using compiler replay); the times for **STW** are

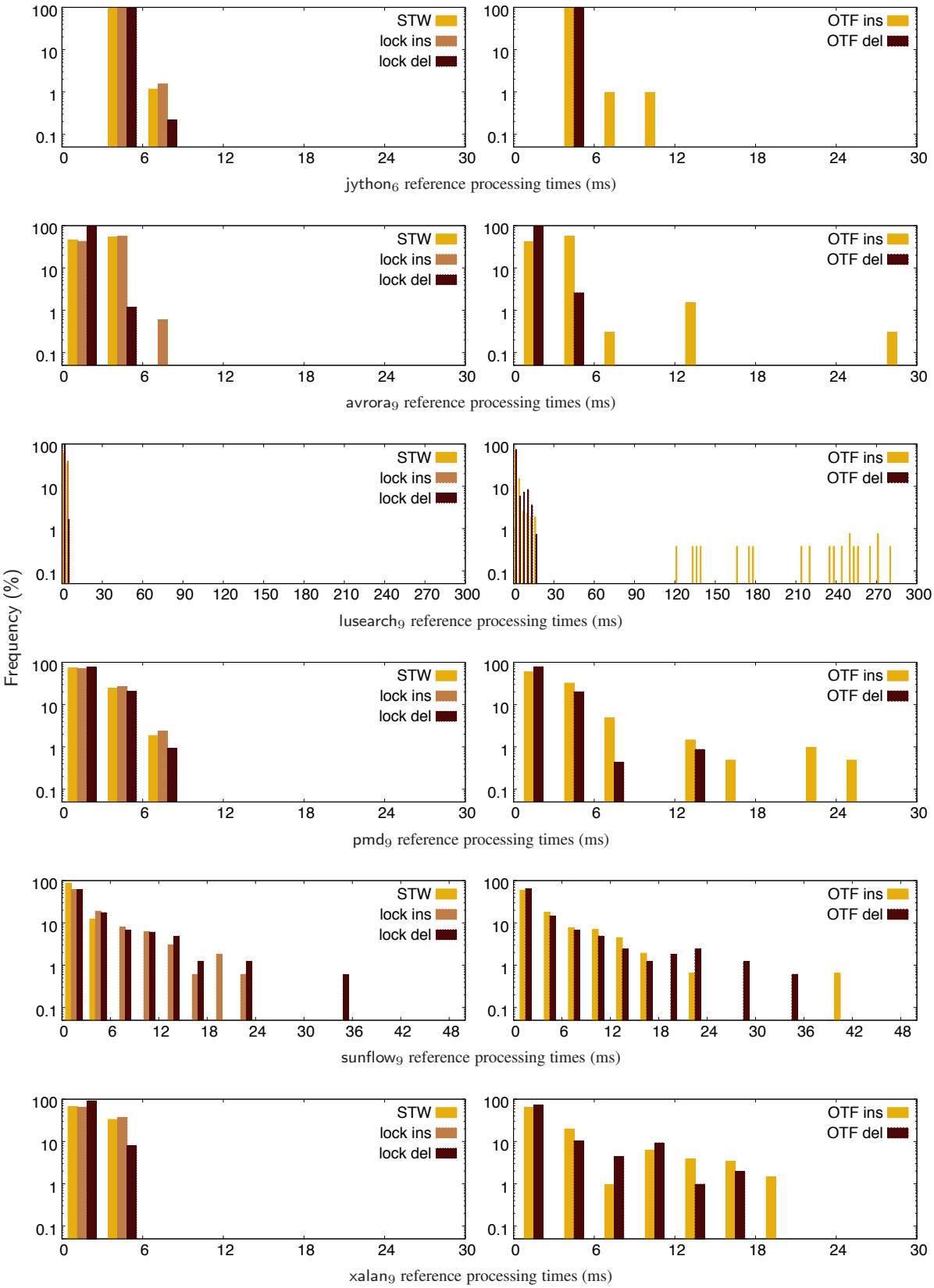

Figure 13: Reference processing time (3 ms bins) in jython$_6$ and DaCapo 2009 (compiler replay, measurement starts with benchmark)

Table 1: Maximum iteration counts (it.) for termination, benchmark execution times (mean t ms. and standard deviation σ over 10 runs, best in bold), number of threads that could be runnable without the reference lock (act) and number of threads blocked by the lock (blk) using compiler replay (measurement starts with benchmark).

	STW			lock ins					lock del					OTF ins			OTF del		
	it	t	σ	it	t	σ	act	blk	it	t	σ	act	blk	it	t	σ	it	t	σ
jython$_6$	1	7775	70	1	7334	50	1.98	0.37	1	9881	63	1.98	0.34	2	7159	71	1	**7047**	66
avrora$_9$	1	5642	199	1	5253	81	3.74	0.23	1	**5238**	56	3.79	0.19	12	5311	54	2	5314	45
luindex$_9$	1	2404	50	1	2343	58	2.00	0.60	1	2372	155	2.00	0.53	13	**2209**	47	1	2270	37
lusearch$_9$	1	4933	46	1	4869	32	3.42	2.06	1	4891	49	3.41	2.00	62	5328	71	2	**4658**	103
pmd$_9$	1	4417	69	1	**4216**	146	3.35	0.91	1	4333	50	3.34	0.79	13	4281	66	3	4581	63
sunflow$_9$	1	5270	246	1	4968	354	5.31	0.26	1	4793	39	5.26	0.22	3	4889	341	1	**4782**	85
xalan$_9$	1	5164	48	1	5039	42	3.29	1.88	1	4850	37	3.19	1.71	2	4877	50	1	**4746**	59

those of Fig. 11. The processing times reflect the number of time the GC has to re-trace; reference processing with insertion barriers shows long tails especially for lusearch$_9$. In contrast, reference processing with deletion barriers finished in under 20 ms except for a small fraction of collections in sunflow$_9$; total reference processing time was always within three times that of the **lock** approach.

LESSON 2. *OTF reference processing phases are longer in the worst case, but with deletion barriers, not by much.*

Even with insertion barriers, they are likely to be acceptable. With OTF processing, mutators continue to run, provided they do not exhaust memory (admittedly this risk is increased with a longer reference processing time), in sharp contrast to the other approaches. Tab. 1 shows the overall execution time in milliseconds (arithmetic mean and standard deviation) of each benchmark. We note that in five out of seven cases OTF reference processing led to the shortest execution times, though often the difference was small. The locking approach blocks only those mutators that call `get()` during the reference processing phase. For this approach, Tab. 1 also shows the average number of active (runnable) threads including those blocked on the reference lock ('act'), and the average number of threads blocked on this lock ('blk'). Of course, all the benchmarks have more threads than 'act' but some of these are blocked for other reasons. Because we have four cores (eight hardware threads) and two GC threads, we note that if $\mathrm{act} - \mathrm{blk} < 2(6)$ then some cores (hardware threads) would be idle.

LESSON 3. *Overall execution time is not increased significantly by processing references OTF, and is often reduced.*

However, Sapphire executes other GC phases OTF, trading pause time for increased overall execution time. The differences due to the way reference types are processed are small in comparison.

8. Conclusion

Reference types are frequently used in a significant number of programs. Our novel reference processing framework for fully OTF collectors works most efficiently with deletion barriers, but also supports with insertion barriers. We process reference types OTF while mutators are running in less than three times the time that previous approaches take while mutators are blocked. Furthermore, OTF reference processing typically reduces overall execution time.

Acknowledgments

We are grateful to Rick Hudson and Intel for a license to implement Sapphire in Jikes RVM, and Laurence Hellyer for his work on Sapphire. We are also grateful for the support of the JSPS KAKENHI Grant Number 25330080, the EPSRC through grant EP/H026975/1 and Google's Summer of Code.

References

[1] J. Auerbach, D. F. Bacon, B. Blainey, P. Cheng, M. Dawson, M. Fulton, D. Grove, D. Hart, and M. Stoodley. Design and implementation of a comprehensive real-time Java virtual machine. In *7th ACM & IEEE International Conference on Embedded Software*, pages 249–258, Salzburg, Austria, Sept. 2007. ACM Press.

[2] J. Auerbach, D. F. Bacon, P. Cheng, D. Grove, B. Biron, C. Gracie, B. McCloskey, A. Micic, and R. Sciampacone. Tax-and-spend: Democratic scheduling for real-time garbage collection. In *8th ACM International Conference on Embedded Software*, pages 245–254, Atlanta, GA, 2008. ACM Press.

[3] H. Azatchi, Y. Levanoni, H. Paz, and E. Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN Notices 38(11), pages 269–281, Anaheim, CA, Nov. 2003. ACM Press.

[4] D. F. Bacon, P. Cheng, and V. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *30th Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices 38(1), pages 285–298, New Orleans, LA, Jan. 2003. ACM Press.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN Notices 41(10), pages 169–190, Portland, OR, Oct. 2006. ACM Press.

[6] C. Click, G. Tene, and M. Wolf. The Pauseless GC algorithm. In M. Hind and J. Vitek, editors, *1st ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 46–56, Chicago, IL, June 2005. ACM Press.

[7] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *21st Annual ACM Symposium on Principles of Programming Languages*, pages 70–83, Portland, OR, Jan. 1994. ACM Press.

[8] T. Domani, E. K. Kolodner, E. Lewis, E. E. Salant, K. Barabash, I. Lahan, E. Petrank, I. Yanover, and Y. Levanoni. Implementing an on-the-fly garbage collector for Java. In C. Chambers and A. L. Hosking, editors, *2nd International Symposium on Memory Management*, ACM SIGPLAN Notices 36(1), pages 155–166, Minneapolis, MN, Oct. 2000. ACM Press.

[9] A. Georges, L. Eeckhout, and D. Buytaert. Java performance evaluation through rigorous replay compilation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN Notices 43(10), pages 367–384, Nashville, TN, Oct. 2008. ACM Press.

[10] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.

[11] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[12] R. L. Hudson and J. E. B. Moss. Sapphire: Copying garbage collection without stopping the world. *Concurrency and Computation: Practice and Experience*, 15(3–5):223–261, 2003.

[13] B. Iyengar, G. Tene, M. Wolf, and E. Gehringer. The Collie: a wait-free compacting collector. In McKinley and Vechev, editors, *11th International Symposium on Memory Management*, pages 85–96, China, June 2012. ACM Press.

[14] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Applied Algorithms and Data Structures. Chapman & Hall, Aug. 2012.

[15] T. Kalibera. Replicating real-time garbage collector for Java. In *7th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, pages 100–109, Madrid, Spain, Sept. 2009. ACM Press.

[16] B. McCloskey, D. F. Bacon, P. Cheng, and D. Grove. Staccato: A parallel and concurrent real-time compacting garbage collector for multiprocessors. IBM Research Report RC24505, IBM Research, 2008.

[17] Oracle Corp. *Java Platform, Standard Edition 7: API Specification*, 2013.

[18] P. P. Pirinen. Barrier techniques for incremental tracing. In S. L. Peyton Jones and R. Jones, editors, *1st International Symposium on Memory Management*, ACM SIGPLAN Notices 34(3), pages 20–25, Vancouver, Canada, Oct. 1998. ACM Press.

[19] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgard. Stopless: A real-time garbage collector for multiprocessors. In G. Morrisett and M. Sagiv, editors, *6th International Symposium on Memory Management*, pages 159–172, Montréal, Canada, Oct. 2007. ACM Press.

[20] F. Pizlo, A. L. Hosking, and J. Vitek. Hierarchical real-time garbage collection. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ACM SIGPLAN Notices 42(7), pages 123–133, San Diego, CA, June 2007. ACM Press.

[21] F. Pizlo, E. Petrank, and B. Steensgaard. A study of concurrent real-time garbage collectors. In R. Gupta and S. P. Amarasinghe, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 43(6), pages 33–44, Tucson, AZ, June 2008. ACM Press.

[22] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: Fragmentation-tolerant real-time garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 45(6), pages 146–159, Toronto, Canada, June 2010. ACM Press.

[23] M. Schoeberl and W. Puffitsch. Non-blocking object copy for real-time garbage collection. In *6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, pages 77–84, Santa Clara, CA, Sept. 2008. ACM Press.

[24] G. Tene, B. Iyengar, and M. Wolf. C4: The continuously concurrent compacting collector. In H. Boehm and D. Bacon, editors, *10th International Symposium on Memory Management*, pages 79–88, San Jose, CA, June 2011. ACM Press.

[25] X. Yang, S. M. Blackburn, D. Frampton, and A. L. Hosking. Barriers reconsidered, friendlier still! In McKinley and Vechev, editors, *11th International Symposium on Memory Management*, pages 37–48, China, June 2012. ACM Press.