Sapphire: Copying GC Without Stopping the World

Richard L. Hudson
Intel Corporation
2200 Mision College Blvd.
Santa Clara, CA 95052-8119
Rick.Hudson@intel.com

J. Eliot B. Moss
Dept. of Computer Science
Univ. of Massachusetts
Amherst, MA 01003-4610
moss@cs.umass.edu

ABSTRACT

Many concurrent garbage collection (GC) algorithms have been devised, but few have been implemented and evaluated, particularly for the Java programming language. Sapphire is an algorithm we have devised for concurrent copying GC. Sapphire stresses minimizing the amount of time any given application thread may need to block to support the collector. In particular, Sapphire is intended to work well in the presence of a large number of application threads, on small- to medium-scale shared memory multiprocessors. A specific problem that Sapphire addresses is not stopping all threads while thread stacks are adjusted to account for copied objects (in GC parlance, the "flip" to the new copies).

Sapphire extends previous algorithms, and is most closely related to replicating copying collection, a GC technique in which application threads observe and update primarily the old copies of objects [13]. The key innovations of Sapphire are: (1) the ability to "flip" one thread at a time (changing the thread's view from the old copies of objects to the new copies), as opposed to needing to stop all threads and flip them at the same time; and (2) avoiding a read barrier.

1. OVERVIEW

Sapphire is a new concurrent copying GC algorithm, designed for type-safe heap-allocating languages. It aims to minimize the time an application thread is blocked during collection. A specific advance Sapphire makes over prior algorithms is incremental "flipping" of threads. Previous algorithms include a step during which all application threads are stopped, their stacks traversed, and pointers in the stacks redirected from old copies of objects to new copies. In systems that might have many threads, we anticipate that this pause will be unacceptable. Sapphire also avoids using a read barrier.

The applications to which Sapphire is targeted are multiprocessor server programs. These might have a large number of threads, each handling a network session, and a considerable amount of shared state in main memory. We are interested in avoiding significant pauses introduced by GC that might be perceptible in variably poor response time. We aimed for an algorithm that will scale to the number of processors for which shared memory is feasible. Sapphire introduces no multiprocessor memory consistency issues, such as the need for additional memory-fence instructions, etc. (see Appendix E).

We organize this paper as follows. This overview section concludes with some useful definitions. Section 2 is the heart of the presentation, discussing the phases of collection and covering the innovations in detail. Section 3 considers how we can combine various of the phases, separated in Section 2 for clarity of exposition. Section 4 relates Sapphire to prior work. Section 5 describes our prototype implementation and measurement results, and concludes. Several appendices consider more specific issues not essential to the primary presentation.

Memory Regions: We define several distinct memory regions. A region may contain *slots* (memory locations that may contain pointers) as well as non-slot data.[1] We assume that all slots in a region can be found without ambiguity.

- U - A region of the heap (i.e., potentially shared among all threads) whose objects are not subject to reclamation in a particular invocation of the collector. U stands for uncollected. For convenience we also include in U all non-thread-specific slots not contained in objects.
- C - A region of the heap (potentially shared among all threads) whose objects are subject to reclamation in a particular invocation of the collector. C stands for collected. C is further divided into:
  - O - Old space: Copies of C objects existing when the collector invocation started.
  - N - New space: New copies of objects surviving the collection.

  C consists only of objects, i.e., it has no "bare" slots, unlike U, which may contain slots not in objects.
- S - Stack: Each thread has a separate stack, private to that thread. The S regions contain slots, but no objects, i.e., there may be no pointers from heap objects into stacks.[2] For convenience we include in S other thread-local slots, such as those corresponding to machine registers holding references.

Java Grande 2001 Palo Alto, CA USA

[1] Our terminology should be familiar to most readers, but we provide a glossary in an appendix.

[2] This still allows purely local objects to be allocated into stacks, so long as references to them cannot "escape" into the heap.

Which objects are collected and not (U and C above) is an arbitrary choice for Sapphire. One might use generational or mature object space (Train) [9] schemes, for example. Sapphire can "piggy-back" some of its needs on a generational collector's write barrier; i.e., Sapphire will need to find pointers from U to C, and a data structure such as a remembered set will save scanning U.

One difference between Sapphire and replicating collection is that we assume that new objects are allocated in U, not C. Note that this helps guarantee termination of marking and copying, since C is not growing. This decision does impose write barrier costs on newly created objects, since Sapphire needs to process their slots that point to C objects.

Sapphire is a purely copying collector, and thus is exact (also called accurate), as opposed to conservative or using ambiguous roots. Perhaps this is obvious, but in Sapphire the collector runs concurrently with all the mutator threads.

2. THE SAPPHIRE ALGORITHM

Sapphire splits into two major groups of phases. The first phases, which we call Mark-and-Copy, (a) determine which O objects are reachable from root slots in the U and S regions and (b) construct copies of the reachable O objects in N. During the Mark-and-Copy phases mutators read and update only the O copies of objects. The O and N copies of any given reachable object are kept loosely synchronized: any changes made by a mutator thread to O copies between two synchronization points will be propagated to the N copies before passing the second synchronization point. This takes advantage of the Java Virtual Machine specification's memory synchronization rules [12].[3] The point is that updates to both copies need not be made atomically and simultaneously. If all mutator threads are at synchronization points, then the O and N copies will be consistent with one another once we are in collection phases in which mutators can observe both O and N copies. We call this property *dynamic consistency* between O and N space.

The second group of phases, called Flip, is concerned with flipping pointers in S and U so that they point to N space and not O space. In Sapphire this group of phases uses a write barrier only (i.e., no read barrier). Sapphire allows unflipped threads to access both O and N copies of objects, even of the same object. Previous concurrent copying collectors either redirect accesses of O objects to their N copies (using a read barrier), or insure that all accesses are to O objects (and flip all at once). Incremental flipping plus having no read barrier admits the possibility of access to both O and N copies at the same time. This possibility requires slightly tighter synchronization of updates to both copies.

It also affects pointer equality comparisons (`==` in Java), since one must be able to respond that pointers to the O and N copies of the same object are equal from the viewpoint of the Java programmer; the equality comparison needed is similar to the implementation of `eq` by Brooks [5]. It is important to note that (a) comparisons with the constant `null` need not do any extra work, and (b) neither do comparisons of two variables that are bit-wise equal or where either variable is `null`. Here is pseudo-code for `==` (which would obviously be inlined, and optimized for the case where one of the arguments is statically 0 (`null`)); it calls `flip-pointer-equal` for the more complicated case:

[3]We assume programs without data races. We describe how to handle Java `volatile` variables in an appendix.

```
// Pointer Comparison
pointer-equal(p, q) {
  if (p == q) return true;
  if (q == 0) return false;
  if (p == 0) return false;
  return flip-pointer-equal(p, q);
  }
```

The `flip-pointer-equal` call does involve what is effectively a read barrier; however, we claim this is a rare operation.

2.1 The Mark-and-Copy Phases: Achieving Dynamic Consistency

The specific phases are: Mark, Allocate, and Copy. Note that in practice a number of these phases can be combined and performed together, as sketched later. However, the algorithmic explanations are clearer if we separate the phases.

A useful way to understand these phases is in terms of the traditional tri-color marking rules (see, e.g., [11]). Under these rules, each slot and object is considered to be *black*, meaning marked and scanned, *gray*, meaning marked but not necessarily scanned, or *white*, meaning not marked. Slots contained within an object have the same color as the object. A single rule restricts colors: a black slot may not point to a white object. Sapphire treats S slots as gray, so they may contain pointers to objects of any color. This implies that storing a reference in a stack slot does not require any work to enforce the color rule. Updates of shared memory (globals and heap objects) do require work, in the form of a write barrier.

Initially we consider all existing objects and slots to be white. As collection proceeds, objects progress in color from white, to gray, to black. In Sapphire, black objects are never turned back to gray and rescanned. The goal of the Mark phases of the collector is to color every reachable C object black. Further, any object unreachable when marking begins will remain white, and the collector will reclaim it eventually. Newly allocated objects are considered to be black.

Except for the dynamic consistency aspects, the Mark-and-Copy phases are essentially a concurrent marking algorithm followed by copying of the marked objects. For this reason, one can easily extend the algorithm to treat such features as weak references and finalization. Since they are orthogonal to the focus of this paper, we do not discuss them further.

Mark Phase: This has three steps: Pre-Mark, to install the Mark Phase write barrier, Root-Mark, to handle non-stack roots, and Heap/Stack-Mark, to complete marking. **Pre-Mark** installs the write barrier, written here in C-like pseudo-code:[4]

```
// Mark Phase Write Barrier
//   this is only for pointer stores
//     the update is *p = q
//     the p slot may be in U or O
//     the q object may be in U or O
mark-phase-write(p, q) {
  *p = q;
  mark-write-barrier(q);
}
mark-write-barrier(q) {
  if (old(q) && !marked(q)) {
```

[4]Note that different phases of the collector employ different write barriers, so we need some way to change the mutator's write barrier as we go, probably either by changing a single global variable read by all the threads, or by changing individual per-thread variables within each thread.

```
    // old && !marked means "white"
    enqueue-object(q);
    // enqueue object for collector
    //   to mark later
  } }
```

Notice that mutators do not perform any marking directly, but rather enqueue objects for the collector to mark. It is useful to consider enqueued objects as being *implicitly* gray; then this write barrier enforces the no-black-points-to-white rule.

Why enqueue rather than having mutators mark directly? Ultimately, we will combine marking with copying, and the mark step will then involve allocating space for a new copy of the object. Having mutators do this allocation leads to a synchronization bottleneck. We avoid this bottleneck by having the collector do the allocation and copying. Further, each mutator has its own queue, so enqueuing has no synchronization overhead. When the collector scans a mutator's stack, it also empties that mutator's queue, by threading it onto a single collector input queue.

The **Root-Mark** step iterates through U slots and "grays" any white C objects referred to by those slots, using `mark-write-barrier`. We consider this to be "blackening" the U slots. Note that as of this step, stores into newly allocated objects, including initializing stores, invoke the `mark-write-barrier`, and thus new objects are treated as black. Since the collector is invoking the write barrier, the relevant objects appear immediately on the collector's input queue.

While one could scan the U region to find the relevant slots, more likely one uses the remembered set data structure built by a generational write barrier, to locate the relevant slots more efficiently [11].

In the **Heap/Stack Mark** step, the collector works from its input queue, a set of *explicitly* gray (marked) objects, and the thread stacks. For each enqueued object, the collector checks if the object is already marked. If it is, the collector discards the queue entry; otherwise, it marks the object and enters it in the explicit gray set for scanning. For each explicitly gray object, its slots are blackened (using `mark-write-barrier` on the slots' referents), and then the object itself is considered black. This is represented by the fact that the object is marked but not in the explicit gray set. The collector will repeatedly proceed until the input queue and the explicit gray set are both empty.

Note that an object may be enqueued for marking more than once, by the same or different mutator threads; however, eventually, the collector will mark the object and it will no longer be enqueued by mutators.

Heap/Stack Mark also involves finding S pointers to O objects. To scan a mutator thread's stack, the collector *briefly* stops the mutator thread at a safe point (about which more later), and scans the thread's stack (and registers) for references to white O objects, invoking the Mark Phase write barrier on each reference. (One might use stack barriers to bound stack scanning pauses [6].) While the thread is stopped, the collector moves the thread's enqueued objects to the collector's queue (using just a few pointer updates to grab the queue all at once). The collector resumes the thread and then processes its input queue and gray set until they are empty again.

While it is easy to scan an individual thread's stack for pointers to white objects, it is harder to see how to reach the situation of having no pointers to white objects in any thread stack. The key problem is that even after a thread's stack has been scanned, the thread can enter more white pointers into its stack, since there is no read barrier preventing that from happening. The key fact needed to understand the solution is that threads cannot write white references into heap objects, because the write barrier will (implicitly) gray the white referent first.

Suppose that between a certain time t_1 and a later time t_2 we have scanned each thread's stack, none of the thread stacks had any white pointers, none of the threads had any enqueued objects, and the collector's input queue and gray set have been empty the whole time. We claim that there are now no white pointers in S or in marked O objects, and thus that marking is complete. We observe that a thread can obtain a white pointer only from a (reachable) gray or white object. There were no objects that were gray between t_1 and t_2, so a thread could obtain a white pointer only from a white object, and the thread must have had a pointer to that object already. But if the thread had any white pointers, it discarded them by the time its stack was scanned, and thus cannot have obtained any white pointers since then. This applies to all threads, so the thread stacks cannot contain any white pointers.

The argument concerning reachable O objects is straightforward. The O objects initially referred to by U slots were all added to the gray set and have been processed, and no additional ones have been added by the write barrier since t_1. A chain of reachability from a black slot to a white object must pass through a gray object (because of the tri-color invariant), and since there are no gray objects, all reachable O objects have been marked.

Here are two potentially useful improvements for stack scanning. First, threads that have been suspended continuously since their last scan in this mark phase need not be rescanned.[5] Second, if we use stack barriers [6], we can avoid rescanning old frames that have not been re-entered by a thread since we last scanned its stack.

Because of the possible and necessary separation of pointer stores from their associated write barriers, stack scanning requires that a thread be in a GC-consistent state, i.e., where every heap store's write barrier has been executed.

2.2 Allocation and Copying

The mark phases establish which O objects are reachable. Once we determine the reachable O objects, we allocate an N copy for each of them (Allocation Phase) and then copy the O copy's contents to the allocated N copy (Copy Phase).

Allocation Phase: Once all reachable O objects have been marked, the collector allocates space for an N copy for each one of them and sets the O copy's forwarding pointer to refer to the space reserved for the N copy. We then say the O copy is *forwarded* to the N copy. Clearly the format of objects must be such as to support a forwarding pointer while still allowing all normal operations on the objects.[6]

If the collector saves a list of the object scanned in the mark phase, then it can use that list to find the O copies. The forwarding pointer slot might serve for the list; if we combine phases

[5] This optimization may be important in the presence of large numbers of threads, most of which are suspended for the short term.

[6] This is different from a stop-the-world collector, which can "clobber" part of the O object as long as the data is preserved in the N copy. In Sapphire we can still clobber a header word, but the mutator will have to follow the forwarding pointer whenever it needs the moved information. Also, installing the forwarding information must be done carefully, so that mutator operations can proceed at any time. This is fairly easy if the collector uses compare-and-swap, retrying as necessary, to install the forwarding address. We "overload" our forwarding on a "thin lock" scheme [2].

(discussed later), then we do not need an explicit list.

Our algorithm also requires us to be able to find the O copy of an object from its N copy. We do this using a hash table, which we discard after collection. If we had instead used back pointers from N copies to O copies, we would have needed to remove them, involving an extra pass over the N copies.

Copy Phase: The Copy Phase needs a new write barrier, to maintain (dynamic) consistency between the O and N copies of objects; the **Pre-Copy step** establishes that write barrier, shown in this pseudo-code:

```
// Copy Phase Write Barrier:
//    handle ptrs, non-ptrs differently
//    p->f = q is the desired update
//      objects p, q may be in U or O
copy-write (p, f, q) {
  p->f = q; copy-write-barrier(p, f, q);}
copy-write-barrier(p, f, q) {
  if (forwarded(p)) {
    pp = forward(p);
    qq = forward(q); // qq==q for non-ptrs
    pp->f = qq; // wrote O first, then N
  } }
forward(p) {
  return (forwarded(p) ?
          forwarding-address(p) : p); }
```

Unlike most copying collector write barriers, this write barrier applies to heap writes of non-pointer values as well as of pointers. It does the same work as replicating copying collection, but with tighter synchronization. It requires work regardless of the generational relationship of the objects when storing a pointer. Finally, note that a pointer in an N copy always points to U or N space, not to O space; we maintain the invariant that N copies never refer to O copies.

The **Copy step** copies the contents of each black O object into its corresponding allocated N copy. If a datum copied is a pointer to an O object, it is first adjusted to point to the N copy of the object.

A tricky thing about this phase is that, as the collector copies object contents, mutators may concurrently be updating the objects. While `copy-write-barrier` will cause the mutators to propagate their updates of O copies to the N copies, the mutators can get into a race with the collector. Since we prefer that the mutator write barrier not be any slower or more complex than it already is, we place the burden of overcoming this race upon the collector, as shown in this pseudo-code:

```
// Collector word copying algorithm
//   Goal: copy *p to *q
//   p points to an O object field
//   q points to the corresponding N field
copy-word(p, q) {
  i = max-cycles; // number of times to
  do {       // try non-atomic copy loop
    wo = *p;
    wn = forward(wo);
      // wn==wo for non-ptrs
    *q = wn;
    if (*p == wo) return;
      // done if no change
  } while (--i > 0);
  wn = *q; // do these reads in order!
  wo = *p;
  wn2 = forward(wo);
    // wn2==wo for non-ptrs
  compare-and-swap (q, wn, wn2);
    // address, old-value, new-value
    // if the compare-and-swap fails, it's
    //   ok, because it means the mutator
    //   copied the new value
}
```

See the related appendix for discussion of the correctness of this algorithm. Note that it assumes that mutators do not attempt updates concurrently with each other (i.e., they use proper locking to avoid data races). We treat Java `volatile` fields in a separate appendix.

2.3 Flip Phases

The later Sapphire phases are: Pre-Flip, Heap-Flip, Thread-Flip, and Post-Flip. The goal of these phases is systematically to eliminate O pointers that may be seen and used by a thread, as follows. First, we install a write barrier that helps keep track of places possibly containing pointers to O objects. We next insure that there are no heap (U region) pointers to O objects. We then start flipping threads at will.

We start with an invariant that N copies do not point to O copies, and then establish and maintain that neither U nor N slots refer to O copies. The Heap-Flip phase does this, by eliminating any U pointers to O copies. Unflipped threads may have pointers to O and N copies, even to the same object, but flipped threads never refer to O copies. The Post-Flip phase simply restores the normal (i.e., not-during-collection) write barrier and reclaims the O region.

As long as there are any unflipped threads, all threads must update both the O and N copies of C objects. However, because of the way in which we take advantage of Java mutual exclusion semantics, the order (O first or N first) does not matter.[7] The main import is that we need a way to "unforward" from an N object to its O copy.

Since unflipped threads may access both O and N copies of the same object, pointer variable equality tests such as `p == q` need to be a little more complex, as previously discussed. Since comparisons with `null` are unaffected, and since most pointer comparisons are probably tests for null pointers, it is unlikely that the more complex pointer equality test will have significant impact.

We contend that the logical aliasing of the two distinct copies of objects is not a problem; see the related appendix.

The Pre-Flip phase installs the Flip Phase Write Barrier; here is the pseudo-code:

```
// Flip Phase Write Barrier
//   p->f = q is the update
//   object p may be in U, O, or N
//   q may be a ptr or non-ptr:
//     omit forwarding for non-ptrs
//   if q is a ptr, it may refer
//     to U, O, or N
flip-write(p, f, q) {
  p->f = q;
  if (forwarded(p)) {
    // true for BOTH O and N copies so
    // that this follows O->N or N->O
    pp = forward(p);
    qq = q;
    if (old(qq)) qq = forward(qq);
      // omit step above for non-ptrs
    pp->f = qq;
```

[7] The situation with volatile fields is more complex; see the appendix.

```
} }
```

The Flip Phase write barrier must be installed before the Heap-Flip phase. Otherwise, unflipped threads might write O pointers in U slots. Likewise, the pointer equality test must be installed now, since the Heap-Flip phase will start to expose N pointers to unflipped threads.[8]

The **Heap-Flip** phase is straightforward: it scans every U slot that might contain an O pointer, fixing O pointers to refer to their N copies. Because of possible races with mutator updates, the collector employs a compare-and-swap operator, ignoring failures (since the mutator thread can only have written an N pointer in this phase). Here is pseudo-code:

```
// Heap-Flip: U space ptr forwarding
//   p points to a U slot,
//     that may point to an O object
flip-heap-pointer(p) {
  q = *p;
  if (old(q))
    compare-and-swap(p, q, forward(q));
    // avoid race with mutator
}
```

The **Thread-Flip** Phase is also straightforward, given the write barrier set by the Pre-Flip phase. To flip a given thread, one replaces all O pointers in the thread's stack and registers with their N versions. This can be done incrementally using stack barriers, as discussed for marking. For flipping S slots, we use `flip-heap-pointer`. Any new threads start flipped.

The **Post-Flip** Phase does "clean up" and freeing. Once all threads have flipped, we can turn off the special write barriers and revert to the normal write barrier used when GC is not running. After guaranteeing that no thread is still executing a flip write barrier (which might try to access on O copy), the collector may then discard the hash table mapping N copies to O copies and reclaim O space.

3. MERGING PHASES

Some Sapphire phases must be strictly ordered and cannot be merged. However, we can merge these Mark-and-Copy phases into a *Replicate* Phase: Root-Mark, Heap/Stack-Mark, Allocate, and Copy. The Pre-Mark phase must precede Replicate; likewise, the Flip phases must follow it and occur in order. The merging is mostly straightforward; here is pseudo-code for the Replicate-Phase write barrier:

```
// Replicate Phase Write Barrier
//   p->f = q is the update
//   object p may be in U or O
//   object q may be in U or O
// This is for ptrs and non-ptrs
replicate-phase-write(p, q) {
  *p = q;
  replicate-write-barrier(p, f, q); }
replicate-write-barrier(p, f, q) {
  copy-write-barrier(p, f, q);
    // above for q ptr or non-ptr
  mark-write-barrier(q);
    // above only if q is a ptr
}
```

[8]The pointer equality test can be installed all the time, if that is more convenient or efficient.

This simply combines the previous Mark and Copy Phase write barriers.

As for the collector, we observe that "marked" is now represented by "forwarded". There are a variety of ways to represent the explicit gray set; we chose to "mark" (forward) recursively, because it avoided multiple scans of the objects.

In the replicate phase, mutators do nothing "special", except use the Replicate Phase write barrier. The collector does the allocation and copying as before, and the write barrier simply enqueues references. The collector thus acts as follows:

1. It scans root slots, heap slots (slots in U that might refer to O objects), and stack slots, and for each one calls `mark-write-barrier`, possibly adding references to its input queue. The order does not affect correctness.
2. For each reference in its input queue, if the referent is not yet forwarded, it allocates a new copy and installs a forwarding pointer. We call these steps `forward-object`.
3. As part of `forward-object`, it recursively processes each slot of the just-forwarded object, forwarding the referent if needed. It also updates the N copy's slot to refer to its referent's N copy as needed. We call this work `scan-slot`.
4. The phase terminates when (a) all roots and U heap slots have been scanned, (b) all N copies have been scanned, and (c) all thread stack slots have been scanned and found to contain no white pointers, while the collector queue has remained empty.

```
// Scan slot
//   called for each field f of each
//     object p needing scanning
//   object p is in N
// We: copy the field from O space,
//   forward the referent, and install
//   the N address
scan-slot(p, f) {
  pp = unforward(p);
  copy-word(&(pp->f), &(p->f));
  forward-object(p->f);
    // above only if p->f is a pointer
  if (forwarded(p->f)) {
    // also only if p->f is a pointer
    v = p->f;
    vv = forward(v);
    compare-and-swap (&(p->f), v, vv);
      // avoid race with mutator
  } }
```

4. RELATED WORK AND DISCUSSION

The most closely related work is that of Nettles and O'Toole [13]. They describe a concurrent copying algorithm called replicating collection that differs from prior work in that (a) it has mutators observe and update only the O copies before the flip, thus avoiding a read barrier, and (b) tracks all writes so that the collector brings the N copies into consistency with the O ones. Sapphire improves replicating collection in at least these significant ways: (a) it uses the mutator write barrier to propagate updates, so the O and N copies are more closely synchronized, (b) it allows mutators to see both O and N copies, and (c) it flips thread stacks and the heap incrementally. As previously mentioned, the latter is especially important when there are many threads. Sapphire also makes better guarantees of termination than does replicating collection, because new objects are not allocated in the region being collected.

There are many previous concurrent and/or incremental copying collectors, most of which use a read barrier to insure that mutators see only the N copies of copied objects. Examples include the algorithms of Baker [3], Brooks [5], and Appel, Ellis, and Li [1], in addition to Nettles and O'Toole. Because of their N-space-only invariant, these algorithms cannot flip incrementally without adding a read barrier. There are some refinements of replicating collection, such as special treatment of immutable data [10] (important for ML, but not as helpful for Java) and of thread-private data [8, 7], but none address incremental flipping while copying the shared heap.

In sum, Sapphire is a new approach to concurrent copying collection, using a different mutator invariant. The first concurrent copying collectors used a read barrier to enforce an N-space invariant, replicating collection uses an O-space invariant but requires atomic flipping, and Sapphire allows mutator references to O and N space and supports incremental flipping. Sapphire does rely on a property particular to Java, namely a memory model and definition of proper synchronization (absence of data races). We exploit that property to avoid making all heap updates atomic during collection. Thus, Sapphire may be appropriate for languages with similar semantics.

Data Races: We say a Java program has a *data race* if two threads can update the same non-volatile field concurrently. Programs with data races may exhibit new behaviors under Sapphire, but the effects are similar to those allowed when optimizing ([12, p. 378]). Further, the values stored into any field are ones stored by some thread, so Sapphire cannot violate Java type-safety. In any case, for properly synchronized programs, Sapphire produces results consistent with the Java Virtual Machine Specification.

5. PROTOTYPE IMPLEMENTATION AND RESULTS

We did a proof-of-algorithm implementation of Sapphire in the Intel Open Run-time Platform (ORP),[9] a complete Java Virtual Machine (JVM) and JIT (just-in-time) compiler for the IA-32. The ORP includes tuned implementations of several GC algorithms, including a stop-the-world copying collector, STW.

Implementing Sapphire in the ORP necessitated inserting write barriers (a) for non-pointers as well as pointers, and (b) at places where a stop-and-collect GC would not need them because it would scan (specifically, the global roots area). We offer measurements of GC pause times (length of time mutator threads are blocked because of GC) to show that Sapphire is indeed effective in minimizing pauses. Since Sapphire has yet to be carefully tuned, these pauses should be taken as upper bounds on what one can achieve with the algorithm. Note that our goal here is to validate the algorithm, not to evaluate performance, since our implementation of Sapphire is not tuned.

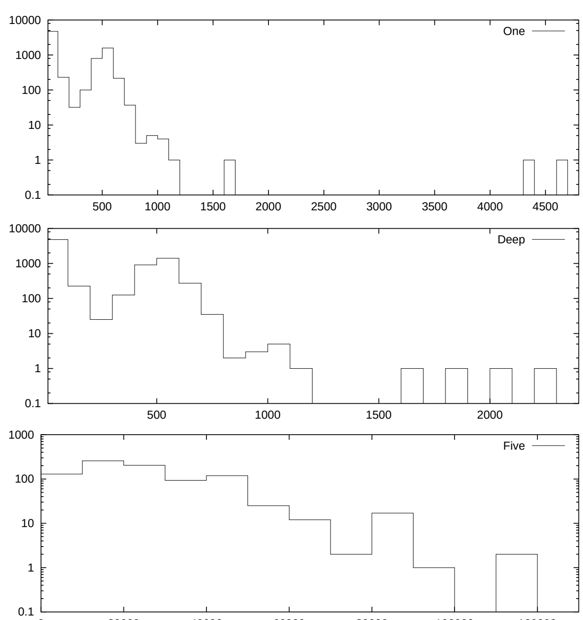
Hardware platform: We used an Intel two-processor Pentium II system running at 300 Mhz with 512 Kb of cache and 256 Mb of main memory (DK440LX motherboard). The operating system was Windows NT Workstation 4.0.

Benchmark programs: To stress test the algorithm, we designed a multi-threaded benchmark program that (a) continuously builds objects and continuously discards them, and (b) can do so using multiple threads. Our benchmark consists of: (1) a GC thread that sleeps until a GC is requested *and* nursery allocation space is low; (2) a GC requesting thread that loops forever requesting a GC and then sleeping 10 milliseconds; and (3) one or more allocating threads. An allocating thread consists of an outer loop iterated 1000 times, and an inner loop that builds 1000 linked lists of 1000 numbered nodes each. The inner loop also traverses each list, and then discards the list; an exception is the first list it builds, which it retains (to force additional collector work). The total space allocated by an allocating thread is 3.2 Gb.

We ran benchmarks with 1 allocating thread and with 5 allocating threads (16 Gb total allocation). We also ran a variant with one allocating thread, but a deep stack (an extra 1000 frames) between the outer and inner loop, to see if the time needed for stack scanning or flipping would change significantly. We call these benchmarks One, Five, and Deep. The heap size was approximately 128 Mb for all benchmarks, so they require a number of collections in order to complete. We ran each benchmark five times and aggregate the results.

Pause times: Since the primary goal of Sapphire is minimizing mutator thread pause times, that is the primary result we include here. First we give tables showing for each benchmark the minimum, maximum, mean, 50%ile (median), 90%ile, 95%ile, and 99%ile pause times, in *microseconds*, across all 5 runs of the program. Second, we show pause time histograms as graphs for each program, again aggregating the data of the 5 runs. *Note that the vertical scale is logarithmic, emphasizing outlying points.* We find that Sapphire meets our goal of very short pauses for One and Deep. It does less well on Five, where it is strongly affected by OS scheduling concerns since we have only 2 CPUs. We believe that similar effects caused the outliers in One and Deep as well. We believe this effect can be controlled; it does not appear to be a fundamental problem.

Pause times (microseconds)							
Prog	Min	Max	Mean	50%	90%	95%	99%
One	66	4710	165	89	525	641	705
Deep	71	2321	169	89	529	643	713
Five	53	122 129	1865	118	567	3464	46 982



[9] The ORP, including source code, is available at http://www.intel.com/research/mrl/orp.

Additional statistics: The table below shows the total time (in seconds) to run each benchmark program 5 times, with the stop-the-world collector (STW) and with Sapphire (Sapph). It also indicates the total number of GCs over 5 runs for each collector, and the total number of pauses in Sapphire runs, to scan a thread's stack. We note that our VM does not allow scanning at any point in mutator code; we simply restart a thread and try again later if it is at an "unsafe" point. The table also indicates the number of pauses that found threads at unsafe points. Our Sapphire write barrier is currently untuned, so our 1-2% slowdown compared with STW can likely be improved. We observe that our STW pauses were generally 17–25 *milliseconds*, much larger than typical Sapphire pauses.

Additional Statistics						
Prog Name	STW secs	GCs	Sapph secs	GCs	Pauses	#Unsafe
One	504	2255	507	2060	23 064	17 593
Deep	504	2240	508	2071	21 749	16 214
Five	505	673	515	540	18 849	11 627

In conclusion, Sapphire offers a new approach to concurrent copying collection, that minimizes thread blocking (pause) times while avoiding a read barrier. We introduced and explained Sapphire, and in addition we have implemented and offer some measurements of the algorithm, strengthening our argument that it is correct and demonstrating that its pause times are indeed small.

APPENDIX

A. TERMINOLOGY

There are some terms, common to the GC literature and/or the Java language or virtual machine, whose definition we repeat here for clarity.

collector, collector thread: One or more loci of control (not necessarily Java threads, but possibly OS threads) that perform garbage collection work (as opposed to executing application code)

flip: Changing slots referring to O objects to refer to the corresponding N copies. This is most usually applied to the flipping of slots in S (thread stacks), but also applies to slots in U.

JNI, Java Native Interface: An interface defined by Sun Microsystems with the intent of allowing native code (e.g., written in C) to access Java objects, but in a controlled fashion allowing copying collectors and such to work in the presence of uncooperative and unknown code. JNI operates by imposing a level of indirection on object accesses. The JNI's table of pointers it has handed out to native code is a group of slots that are roots for collection and thus part of the U region, as previously mentioned.

mutator, mutator thread: One or more loci of control, generally Java threads (or OS threads accessing the heap through a controlled interface such as JNI), that perform application work. A mutator obviously interacts with the collector when the mutator allocates (though such interaction does not generally require close synchronization), updates heap slots, and "flips" its stack from old-space to new-space.

null pointer: A specific pointer value that refers to no object. By convention, the null pointer value is numerically 0 in most systems, but Sapphire does not depend on that.

object: A collection of contiguous memory locations, lying in a single region that can be addressed and accessed via references. Objects do not overlap and may be relocated independently of one another by the collector. In many cases an object as we use the term here corresponds to a Java object, but sometimes we may use multiple low-level objects to represent a single Java object. A typical case of this is a Java object with complex monitor locking happening. An object may contain slots, non-slot data, or both.

pointer: The address of an object, i.e., a reference (q.v.).

reachable: An object is reachable if a root slot points to it, or a reachable object has a slot pointing to it. Put another way, reachability is the transitive closure of reference following, starting from roots.

read barrier: Operations performed when loading a pointer (or possibly when accessing its referent object). It is called a barrier because the operations must be performed before the pointer use proceeds, since the barrier may replace the pointer with another one, etc. Sapphire does not use read barriers.

reference: The address of an object, also called a pointer. Here we generally mean the value; if we mean a memory location containing a reference, we use the word slot.

root: A slot whose referent object (if any), is considered reachable, along with all objects transitively reachable from it. The S and U regions contain roots, which are where collection "starts" in its determination of reachable O objects.

slot: A memory location that may contain a reference (pointer) to an object. It may also refer to no object, i.e., contain the null pointer. As previously mentioned, we assume that memory locations can be categorized into slots and non-slot data correctly and unambiguously.

synchronization point: A point in code, that when reached, entails a synchronization between threads. We also use the term to refer to the reaching of a synchronization point, i.e., an event in time. The Java programming language and the Java virtual machine have precise definitions of required synchronization points and their effects. The principal points are acquisition and release of monitor locks, and reads and writes of volatile variables. Sapphire assumes that user code obeys the protocols (i.e., non-volatile fields are not accessed concurrently, i.e., without monitor locks being held).

write barrier: Operations performed when a datum (typically a pointer) is stored into a heap object. The operations need to be loosely synchronized with the actual update, but the synchronization requirements are generally not as stringent as for a read barrier. Generational collectors use write barriers to detect and record pointers from older to younger generations, so that upon collection they can locate pointers from U to C efficiently. Sapphire uses more complex write barriers in some phases, to bring O and N copies of objects into consistency and to assist in flipping. Some of these write barriers must occur for all updates rather than only those that store pointers.

B. CORRECTNESS OF `COPY-WORD`

We refer to the code of `copy-word`, shown earlier in the paper. We first argue that if the collector executes the first return statement then the copying operation is correct. Call the mutator action of writing `*p` mp, and likewise mq for writing `*q`. Call the collector actions rp, rq, and wq, for reading `*p` and `*q` and writing `*q`, respectively. The collector actions consist of one or more <rp, wq> pairs followed by a final rp. Mutator actions for a single update consist of an <mp, mq> pair, but multiple mutator updates can come, one after another (but *not* interleaved!). Our goal is that once a collector sequence is complete, and any mutator sequence begun before the collector sequence ends, `*q == forward(*p)`.

Consider the possible interleavings of mutator and collector actions on a given slot. The <rp, wq> and <mp, mq> pairs may execute without interleaving, or we may have one of the following orders:

- <rp, mp, wq, mq>: This gives the right outcome with no further work; the mutation occurs logically after the copying. However, when the collector does the second rp, it will see that p changed and will harmlessly recopy, which can occur either before or after mq.

- <rp, mp, mq, wq>: In this case the logically earlier copy operation clobbers the logically later update of q. But the second rp will detect a difference and redo the copying, with the new value.

- <mp, rp, wq, mq> or <mp, rq, mq, wq>: The copy occurs logically after the update; wq and mq write the same value, and the second rp will see no change.

If another update comes along after the mq, the collector may go through another cycle of copying and checking. Since this can happen indefinitely, the code sequence switches to an atomic update strategy. This involves actions rp, rq (reading q), and csq (compare and swap on q), which gives these possible interleavings:

- <rq, mp, mq, rp, csq>, <rq, mp, rp, mq, csq>, <rq, rp, mp, mq, csq>, <mp, rq, mq, rp, csq>, or <mp, rq, rp, mq, csq>: The compare-and-swap fails, leaving the logically later value from the mq (unless the update did not actually change the value, in which case the compare-and-swap will succeed, but not change the value either).

- <rq, mp, rp, csq, mq> or <mp, rq, rp, csq, mq>: The compare-and-swap succeeds, installing the logically newer value. The later mq writes that same value again.

- <rq, rp, mp, csq, mq>: The compare-and-swap succeeds, installing the logically older value. The later mq install the correct final value.

Note that when the compare-and-swap succeeds, another mutator update cannot have begun (the mq happens after the csq). What could possibly allow a csq to succeed when it should have failed? If, after the collector's rq and rp there are further updates that (a) set q to the value read by the rq (i.e., update both p and q) and then (b) update p to some new value (but the write to q has not yet happened), then the csq will succeed. But the final mq cannot have happened before the csq (or the csq would have failed), hence the mq will happen later and q will end up with the correct value.

The compare-and-swap based copying sequence updates q in the same order as p, whereas the other sequence may not update q in the same order (but will give a correct final value when it terminates). This is not of great import here, since mutators do not read q until after this phase is complete.

Note that we rely on Java locking semantics to resolve possible race conditions between mutators. In particular, we assume there is no interleaving of <mp, mq> update pairs from different threads. For consideration of Java volatile variables, see the related appendix.

C. JAVA MONITOR LOCKS

The Java programming language and virtual machine offer means to obtain exclusive access to individual objects, via monitor locks associated with (some) objects. Virtual machine implementations typically acquire a monitor lock by performing an atomic memory operation on some word associated with the synchronized object, which we term the lock word. If mutators run during collection, and the lock word of a copied object moves, we must insure that mutators always direct their locking related operations to the appropriate memory word.

Our design builds on the *thin lock* protocol [2]. In that protocol, the most common lock states are represented by lock values that fit in the lock word of the synchronized object. Less common states require a *fat lock*, a group of words allocated in memory, which we call a *lock object*. To this model we add the notion of a *forwarded lock word:* the lock word of an O copy forwards lock operations to the lock word of the N copy, which may itself be thin or fat. Thus our lock words have four states: thin, fat, forwarded, and meta-locked, while in the original protocol they have only three. In both protocols one uses the meta-locked state to accomplish atomic transitions between more complex lock states. When a lock word is meta-locked, other readers and writers of the word spin until the holder of the meta-lock releases it, by writing a lock word value that is not meta-locked.

To "move" a lock from an O copy to its N copy, we acquire the meta-lock on the O copy lock word, write the old lock value into the N copy, and write a forwarding lock value into the O copy lock word. If the moved lock is fat, we also update the fat lock's back pointer to the synchronized object to refer to the N copy, while holding the meta-lock. This design was relatively simple to add to the existing thin lock implementation of our JVM. It also keeps locks thin almost as often as before.

We allocate lock objects in a non-moving space, but we could copy them, by acquiring the meta-lock so as to gain exclusive access to the lock data structure.

D. VOLATILE FIELDS

Java has a feature whereby one can annotate a field as being volatile. Similar to the semantics of C and C++, this means that each logical read (write) of the volatile field in the source code must turn into exactly one physical read (write) of the field when executed at run time. Volatile fields thus have different memory synchronization properties from ordinary fields: ordinary fields need only be synchronized with memory at each synchronization point. Sapphire takes advantage of the "loose" synchronization of ordinary fields.

Assume that volatile reads and writes must appear to be totally ordered (we discuss weak memory access ordering later). A simple approach to implementing volatile reads and writes is to mimic ordinary reads and writes. Unfortunately, this fails during some Sapphire phases. For example, suppose we have a volatile field X in one object, and a (non-volatile) counter C in some other object, and we always update C via synchronized methods. Say that the object containing X has been copied, so we have two copies of X, namely Xo and Xn. Suppose thread T1 is in the middle of updating X, and has written Xo but not Xn. At this point, T2 comes along and reads the new value in Xo and then increments C. Then T3 increments C and reads Xn, obtaining the old value of X, but at a time (indicated by the counter C) clearly after T2's read of the new value. Threads T2 and T3 have perceived memory update events in inconsistent orders. How do we avoid such inconsistencies?

First, we note that the problem only comes up during those phases in which there are (a) two copies of an object, and (b) at least one mutator can access the old copy and at least one the new copy. Similar to monitor locks, we insure that accesses to any given volatile field are ultimately ordered by accesses to a specific memory word. Here are the details of the strategy:

- Until an O copy of an object is forwarded, accesses to its volatile fields happen on the O copy.
- Once an O copy has an N copy (i.e., it is forwarded), accesses happen to the N copy.
- To make the switch from O copy to N copy atomic and consistent, the collector acquires the O copy meta-lock (which it must do anyway) and copies the volatile fields while holding the meta-lock.
- To insure ordering between volatile writes and collector copying, during the Replicate Phase Sapphire requires mutators to perform volatile writes to unforwarded O copies by first acquiring the O copy meta-lock. Note also that writes to volatile pointer fields must also perform a write barrier, not shown in the code below so as not to distract.
- During the Replicate Phase, volatile *reads* must check the lock field to see if the object is forwarded. If it *is* forwarded, they read the N copy, and (for pointer fields) *unforward* the value read (since during this phase they should see only O references, and N copy volatile pointer fields may contain N references).

  If the object is *not* forwarded, we can perform a volatile read on the O copy. This may appear to be a race condition, but if the object is forwarded and later written, the read is legally serialized before the forwarding, so the value read is consistent with a legal ordering of accesses.

Here is pseudo-code for Replicate Phase volatile accesses:

```
// Replicate Phase Volatile Read
//   p refers to an O copy
//   f is a field
repl-vol-read(p, f) {
top:
  l = p->lock;
  if (metalocked(l)) goto top;
  if (not forwarded(l))
    return vol-read(p->f);
  p = forward(p);
  v = vol-read(p->f);
    // v MAY refer to N space
  return unforward(v);
}

// Replicate Phase Volatile Write
//   p refers to an object
//   f is a field
//   v is the value
repl-vol-write(p, f, v) {
  if (forwarded(p)) {
    p = forward(p);
    v = forward(v);
    vol-write(p->f, v);
    return;
  }
  p = acquire-metalock(p->lock);
    // note: forwards to N copy, if any
  if (p is in N) v = forward(v);
  vol-write(p->f, v);
  release-metalock(p->lock);
}
```

E. WEAK ACCESS ORDERING

Pugh [14] indicates that the Java memory model as described in the original JVM specification ([12]) is problematic. In a recent proposal [15] he suggests a model in which volatile accesses must be sequentially consistent, with each other and with certain synchronization events. On a system that enforces total ordering of all stores and consistency of loads with that order at each processor, we need not do much. However, we were concerned with supporting the IA-64's acquire-release weak ordering model. It turns out that for order-critical accesses (volatiles and monitor locks), we should replace loads with load-acquires, stores with store-releases, and in between a store-release and the next load-acquire, we should insert a memory-fence. It is straightforward to apply these transformations to `copy-word`, etc., and obtain correct behavior of Sapphire in the acquire-release model. Likewise, it is straightforward to determine correct places to insert memory fences for other weak ordering models.

The point is that Sapphire's approach to monitor locks and volatiles still orders accesses through a single word—the object's lock field—and one simply applies whatever techniques one otherwise needs to insure adequate ordering to support the language memory model. In short, Sapphire introduces no new issues.

F. ALIASING

One might be concerned that, during the Flip phases of Sapphire, having distinct pointers refer to what is logically the same object presents new issues of aliases and alias analysis to compilers and hardware. If the update of the "other" copy is deferred, we might have an issue at the hardware level. For example, if we wrote a field via pointer p in O space and read it via pointer q in N space, the read might not reflect the write. Thus, we require that a thread complete updates to both O and N space before proceeding to the next field read or write that might possibly touch the same field. Note that interference from other threads is not an issue, because Java synchronization rules require locking in such cases (see the related appendix for discussion of volatile fields). If we follow the rule of updating both spaces before accessing possibly conflicting fields in the same thread, then hardware alias detection mechanisms will work correctly. The possibility of two physical copies of the same logical object does not affect compiler alias analysis: we could have distinct p and q referring to copies of the same logical object only when p and q could refer to the same physical copy. However, if the compiler inserts run-time tests of pointer equality to conditionalize code based on aliasing, then those equality tests must allow for the possibility of physically distinct copies of the same logical object, i.e., the compiler must emit code for the more complex equality test.

G. GENERATIONAL WRITE BARRIERS

In a generational collector, to avoid scanning the (usually large) older generations when collecting younger generations, one tracks mutator writes using a write barrier. Specifically, when object p is modified to refer to object q, if p is in an older generation than q, we remember that fact. Some write barrier schemes record something about every pointer write. For example, card marking records the region that was modified (the region containing p). Later the information is filtered to determine if one actually created an older-to-younger pointer, and such pointers may be remembered across collections, etc. The important thing to note about the Sapphire scheme is that, unlike most generational schemes, in Sapphire we must apply the write barrier to stores that initialize pointer fields of newly allocated objects. This does not arise from the age relationships of generational collection, but rather with the fact that newly allocated objects are not placed in the C region and we need to know about references to C objects from outside the C region. However, we can arrange the ages of regions so that a generational write barrier will remember the pointers that need to be remembered, as follows. Make the (logical) age of the nursery older than that of the O region, so that we will record references to O objects from nursery objects. In order to end up with the desired remembered pointers at the end of collection, arrange for the age of the N region to be older than the nursery.

While one may do more generational write barrier work in Sapphire than in a collector that includes the nurseries in every collection, it is hard to guarantee termination if one includes the nurseries in C. Also, one should expect that a concurrent collector will do more total work (across all CPUs) than a stop-he-world collector. What one is gaining with Sapphire is minimal disruption and better system utilization.

H. REFINEMENTS TO MARKING FROM STACKS

Marking requires finding S pointers to O objects, i.e., scanning thread stacks. The collector briefly suspends a thread to scan the stack (and registers) for references to white (unmarked) objects and to invoke the mark phase write barrier on them. Here are some useful refinements to this process.

We need not process an entire stack at once. We can process registers, the top frame, and zero or more additional frames, processing the rest after resuming the mutator, in the style of generational stack collection [6]. This may further shorten pauses.

This refinement requires the mutator and the collector to synchronize. In particular, the collector cannot process a frame in which the mutator is running, or a frame from which the mutator has returned. Hence, for the collector to work on frames below a certain point in a thread stack, the collector should install a stack barrier. One can implement barriers by “hijacking” the return address into the frame, making the return address point to a routine that will synchronize with the collector appropriately. (This way the mutator does not need code to check explicitly for needed synchronization.) The collector will remove the stack barrier when it is done scanning, or can move the barrier down the stack incrementally, one or more frames at a time, as it finishes scanning frames for pointers to white (unmarked) objects.

The collector processes suspended threads. It may be possible to remember O-to-N object mappings, and to update long-suspended threads less often, or just at they are awakened. The idea here is to avoid repeated scanning of the stacks of threads that are suspended for a long time. We would need to remember or update the O-to-N maps for objects referred to by suspended threads. It may turn out to be not only simpler but as fast or faster just to record the locations of a suspended thread’s non-null stack references and to update them as part of each collection.

REFERENCES

[1] A. W. Appel, J. R. Ellis, and K. Li. Realtime concurrent collection on stock multiprocessors. In *Proceedings of the ACM SIGPLAN ’88 Conference on Programming Language Design and Implementation*, pages 11–20, Atlanta, Georgia, June 1988. *ACM SIGPLAN Notices 23*, 7 (July 1988).

[2] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for Java. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268, Montreal, Quebec, June 1998. ACM Press.

[3] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, Apr. 1978.

[4] Y. Bekkers and J. Cohen, editors. *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, St. Malo, France, Sept. 1992. Springer-Verlag.

[5] R. A. Brooks. Trading data space for reduced time and code space in real time garbage collection on stock hardware. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 256–262. ACM, 1984.

[6] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 162–173, Montreal, Quebec, June 1998. ACM Press.

[7] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-First ACM Symposium on Principles of Programming Languages*, Portland, Oregon, Jan. 1994.

[8] D. Doligez and X. Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 113–123. ACM Press, Jan. 1993.

[9] R. L. Hudson and J. E. B. Moss. Incremental collection of mature objects. In Bekkers and Cohen [4], pages 388–403.

[10] L. Huelsbergen and J. R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Fourth Annual ACM Symposium on Principles and Practice of Parallel Programming*, volume 28(7) of *ACM SIGPLAN Notices*, pages 73–82, San Diego, CA, May 1993. ACM Press.

[11] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.

[12] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[13] S. M. Nettles, J. W. O’Toole, D. Pierce, and N. Haines. Replication-based incremental copying collection. In Bekkers and Cohen [4].

[14] W. Pugh. Fixing the Java memory model. In *ACM Java Grande Conference*, June 1999.

[15] W. Pugh. Semantics of multithreaded java. Available as www.cs.umd.edu/˜pugh/java/memoryModel/semantics.pdf Oct. 24 2000.