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

March 29, 2002

Abstract

The growing use in concurrent systems built in languages that require garbage collection (GC), such as Java, is raising practical interest in concurrent GC. Sapphire is a new algorithm for concurrent copying GC for Java. It 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.

Sapphire extends previous concurrent copying 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 [Nettles *et al.*, 1992]. 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; (2) exploiting Java semantics and assuming any data races occur on volatile fields, to avoid a barrier on reads of non-volatile fields; and (3) working in concert with virtually any underlying (non-concurrent) copying collection algorithm.

1 Introduction

Sapphire is a new concurrent copying GC algorithm 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 copying 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 avoids using a read barrier on most memory accesses. Further, Sapphire can be used to make almost any non-concurrent copying GC algorithm concurrent.

The applications to which Sapphire is targeted are multiprocessor server programs. These might have a large number of application 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 result 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 addresses multiprocessor memory consistency issues, including weak access ordering, for volatile and non-volatile fields accesses, assuming that all data races occur on volatile variables.

We organize the paper as follows. This overview section defines the memory regions used in Sapphire, and offers some additional useful definitions of terms. 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. In Section 4 we consider in more detail how the collector and mutator threads synchronize their data accesses. The more complex algorithms for handling Java volatile fields appear in Section 5, and Section 6 considers issues of weakly ordered memory accesses as presented in some modern multiprocessors. Section 8 relates Sapphire to prior work. Section 9 describes our prototype implementation and measurement results, and concludes.

1.1 Memory Regions

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, i.e., that collection is *exact* (accurate), not conservative (ambiguous roots, etc.). If an object (slot) lies in the *X* region, we will it an *X* object (slot).

- Uncoll - 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. Uncoll stands for *uncollected*. For convenience we also include in Uncoll all non-thread-specific slots not contained in objects, such as global variables, including Java static fields.
- Old - A region of the heap (potentially shared among all threads) whose objects are subject to reclamation in a particular invocation of the collector. Note that Old, unlike Uncoll, contains only object, no "bare" slots.
- New - A region of the heap (potentially shared among all threads) whose objects are copies of Old objects surviving the current collection. Like Old, New contains no "bare" slots. Note that objects created by the mutator are allocated in Uncoll, not in New; that is, "New" means "new copies of Old objects", not "(brand) new objects".
- Stack - Each thread has a separate stack, private to that thread. The Stack regions contain slots, but no objects, i.e., there can be no pointers from heap objects into stacks.[2] For convenience we include in Stack other thread-local slots, such as those corresponding to machine registers holding references.

Which objects are collected in a given collection and which are not, Old and Uncoll above, is an arbitrary choice for Sapphire. For example, one might use generational or mature object space (Train) [Hudson & Moss, 1992] underlying copying GC schemes. Sapphire can

[1] Our terminology should be familiar to most readers, but we provide definitions in the next section.

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

"piggy-back" some of its needs on a generational collector's write barrier; i.e., Sapphire will need to find pointers from Uncoll to Old, and a data structure such as a remembered set will save scanning Uncoll for such pointers.

One difference between Sapphire and replicating collection is that we assume that objects created by the mutator are allocated in Uncoll, not New. Note that this helps guarantee termination of marking and copying, since Old (and hence New) is bounded in size. This decision does require write barriers on writes to newly created objects, since Sapphire needs to process their slots that point to Old objects.

Sapphire is a purely copying collector, and thus is exact, as previously mentioned. In Sapphire the collector runs concurrently with the mutator threads. Our design as presented assumes a single collector thread; later we discuss how we might parallelize Sapphire collection.

1.2 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)

compare-and-swap: an atomic memory access operation, also known as compare-exchange. We abbreviate it CAS. `CAS(p, old, new)` performs the following steps atomically: It compares the contents of location `p` with the value `old`. If the values are equal, it updates location `p` with the value `new`. The operation returns 1 if it installs the new value and 0 if it does not.

fetch-and-add: An atomic memory access operation. We abbreviate it FA. `FA(p, v)` atomically fetches the current value of location `p` and stores back into `p` that value plus `v`. It is possible to build hardware to allow large numbers of processors to perform FA on the same location simultaneously with very low delay, because the FA operations of multiple processors can be combined from the point of view of the memory.

flip: Changing slots referring to Old objects to refer to the corresponding New copies. This is most usually applied to the flipping of slots in Stack (thread stacks), but also applies to slots in Uncoll.

load-linked: an atomic memory access operation, also called load-with-reservation. We abbreviate it LL. `LL(p)` reads and returns the contents of location `p`. LL is used in conjunction with store-conditionally (q.v.).

mutator, mutator thread: One or more loci of control, generally Java threads, 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 references from Old to New.

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 most 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 (integers, etc.), or both.

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

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

read barrier: Operations performed when dereferencing a pointer (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 when accessing non-volatile slots.

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 term *slot*.

root: A slot whose referent object, if any, is considered reachable. The Stack and Uncoll regions contain roots, which are where collection "starts" in its determination of reachable Old objects.

safe point: A point in the application code at which (a) accurate pointer/non-pointer type information is available for the stack and register contents, and (b) the appropriate write barrier has been performed for each write (i.e., we are not between a write and its corresponding write barrier). Put another way, when a mutator thread is at a safe point, its GC-related information is consistent.

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.

store-conditionally: An atomic memory access operation, used in conjunction with load-linked (described above). We abbreviate it SC. `SC(p, v)` attempts to store value `v` to location `p`. The same processor should have previously read `p` using load-linked (LL). The SC succeeds if no processor has written location `p` since the LL, and fails (without updating the location) otherwise. SC returns 1 if it succeeds and 0 if it fails.

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 does not access *non-volatile* fields 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 Uncoll to Old efficiently. Sapphire uses more complex write barriers in some phases, to bring Old and New 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.

2 The Sapphire Algorithm

We first describe how Sapphire breaks down into phases, next offer an overview of the phases, and then present each phase in detail, including correctness arguments.

Sapphire splits into two major groups of phases. The first phases, which we call Mark-and-Copy, (a) determine which Old objects are reachable from roots in the Uncoll and Stack regions, and (b) construct New copies of the reachable Old objects. During the Mark-and-Copy phases, mutators read and update only the Old copies of objects.[3] The Old and New copies of any given reachable object are kept loosely synchronized: any changes made by a mutator thread to Old copies between two synchronization points will be propagated to the New copies before passing the second synchronization point. This takes advantage of the Java Virtual Machine specification's memory synchronization rules [Lindholm & Yellin, 1997].[4] 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 Old and New copies will be consistent with one another (once we are in collection phases in which mutators can observe both Old and New copies). We call this property *dynamic consistency* between Old and New copies.

The second group of phases, called Flip, is concerned with flipping pointers in Stack and Uncoll so that they point to New copies and not Old copies. In Sapphire this group of phases uses a write barrier only (i.e., no read barrier on reads of non-volatile fields). Sapphire allows unflipped threads to access both Old and New copies of objects, even of the same object. Previous concurrent copying collectors either redirect accesses of Old objects to their New copies (using a read barrier), or insure that all accesses are to Old objects (and flip all at once). Incremental flipping plus having no read barrier admits the possibility of access to both Old and New 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 Old and New 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 [Brooks, 1984]. 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`. Figure 1 gives pseudo-code for `==` (which would obviously be inlined, and probably optimized for the case where one of the arguments is statically `null`); it calls `flip-pointer-equal` for the more complicated case. The `flip-pointer-equal` call does involve what is effectively a read barrier; however, we claim this is a rare operation.[5]

[3] This is not quite true for volatile fields, which we discuss in detail in a separate section.

[4] We assume program have no data races on *non-volatile* variables.

[5] If an application makes heavy use of hash tables, this might not be true, in which case it may be important to engineer pointer comparison more carefully. One can compare object header fields, such as hash codes used to support `Object.hash` in Java, to detect *different* objects quickly. If hash codes are *unique*, then they can also detect *equal* objects quickly.

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

Figure 1: Sapphire extended pointer comparison algorithm

2.1 Overview of Sapphire Phases

Mark-and-Copy: Mark: This phase marks every Old object reachable from roots (Stack and Uncoll). In our design, the collector does all the marking, and works from a queue (set) of objects needing marking. The mutator write barrier insures that when a mutator stores a reference to an unmarked Old object p, p is added to the collector's mark queue. The collector scans any Uncoll slots that might refer to Old objects, and enqueues any Old objects it finds. When the collector removes from the queue a reference to object p, if p is not yet marked then it marks p and enqueues any unmarked Old objects referred to by slots in p.

The trickiest part of marking is dealing with the Stack regions; this is because, since there is no read barrier, mutator stacks may refer to unmarked Old objects. When the collector finds the mark queue empty, it processes each mutator's stack, one at a time. To process a stack, it stops the mutator briefly, scans the stack for references to unmarked Old objects (which it enqueues), and restarts the mutator thread. If the collector makes a pass through all the stacks without enqueuing any objects for marking, then marking is complete (see below for a more detailed termination argument).

Mark-and-Copy: Allocate: Once the Mark phase has determined the set of reachable Old objects, the collector allocates a New copy for each marked Old object. It sets a forwarding pointer in the Old object to refer to the New copy. It also builds a table mapping New copies to Old copies (needed in later phases).

Mark-and-Copy: Copy: Once every marked Old object has a New copy and a forwarding pointer installed, the collector begins copying Old object contents to the New copies. In so doing, it is careful to maintain an invariant that New objects do not refer to Old ones (but rather to corresponding New copies). Beginning in this phase we maintain dynamic consistency: when a mutator updates an Old field (pointer or non-pointer), it updates the corresponding New field as well. Field writes require some synchronization with the collector, since it may be copying at the same time the mutator is updating. For non-volatile fields, the write barrier is fairly simple, both because we assume no data races (i.e., only one mutator writing at a time), and because we put most of the synchronization burden on the collector. For volatile fields, we need more careful synchronization (detailed later), so that we can totally order all writes to a single volatile field, etc. Likewise we need to take care in moving Java object synchronization information from the Old to the New copy.

Flip: Heap-Flip: Beginning in this phase, mutators may "see" both Old and New

copies of objects. The collector processes each Uncoll slot that may refer to an Old object, and "flips" the slot to refer to its New copy. The mutator write barrier also guarantees not to write Old pointers into Uncoll slots. Thus, at the end of the phase, Uncoll has been flipped and will remain so.

Flip: Thread-Flip: In this phase, the collector flips each mutator thread, one at a time. To flip a thread, it stops the thread briefly, scans the stack for references to Old copies, flips those references to refer to the New copies, and restarts the thread. Since the thread can "see" only New and Uncoll slots at that point, and those slots cannot (and will not) refer to Old objects, we need to flip each thread exactly once. During this phase, mutators still need to update both New and Old copies: unflipped threads still read non-volatile fields from Old copies. Hence, we need to be able to find Old copies from New ones, as previously mentioned.

Flip: Reclaim: At this point, the collector reclaims the Old region and the table mapping New to Old copies.

2.2 About Write Barriers

In the phase descriptions below, we speak of "installing" various write barriers, by which we mean causing the mutators to use a particular write barrier. The way we actually implemented the mutator write barrier(s) was as an out-of-line subroutine that checks a global variable indicating the current phase of collection. Thus, in our case, all we need to do to install a particular write barrier is to change the phase variable—*and* to wait until we have observed each thread stop at or pass through a safe point, to insure that no thread is in the middle of a previous phase write barrier. Note that the handshakes are individual (this is *not* a global barrier synchronization), and it is the collector that waits: mutator threads never wait for these handshakes.

One alternative implementation strategy is to have a global variable referring to the currently applicable mutator write barrier subroutine. This has the advantage of avoiding some testing and branching within the routine, and the disadvantage of incurring an indirect subroutine call at each write barrier. The relative costs clearly depend on the expense of different mechanisms on a given hardware platform.

2.3 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 described in Section 3. However, the exposition is clearer if we initially separate the phases.

A useful way to understand these phases is in terms of the traditional tri-color marking rules (see, e.g., [Jones & Lins, 1996]). Under these rules, each slot and object is considered to be *black*, *gray*, or *white*. The colors have these meanings: black: marked and scanned; gray: marked but not necessarily scanned; white: 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 Stack 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.[6] The goal of the Mark phases of the collector is to color every *reachable* Old 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 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, which installs the Mark Phase write barrier, Root-Mark, which handles non-Stack roots, and Heap/Stack-Mark, which completes marking. The **Pre-Mark** step installs the write barrier, shown in Figure 2 in C-like pseudo-code. Notice that we enqueue the *slot* containing the reference to the Old object. When updating a "bare" slot in Uncoll one should insure that `&p[f]` produces the address of the slot (or more likely write an equivalent write barrier tailored to that case). Note that we assume there are primitives available to tell whether an object is Old, New, marked, etc.

```
// Mark Phase Write Barrier
//   this is only for pointer stores
//   objects p and q may be in Uncoll or Old
//   updates the field at offset f
mark-phase-field-write(p, f, q) {
  p[f] = q;
  mark-write-barrier(q);
}

mark-write-barrier(q) {
  if (old(q) && !marked(q)) {
    // old && !marked means "white"
    enqueue-object(q);
    // the collector will mark the object later
  }
}
```

Figure 2: Sapphire mark phase write barrier

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

[6] When we merge phases as described in Section 3, individual *slots* in black New objects may be grayed (enqueued) if a mutator stores an Old pointer into such a slot.

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 normally involves no synchronization, and is not a synchronization bottleneck. When the collector scans a mutator's stack, it also empties that mutator's queue, by threading it onto a single collector input queue. Since a mutator thread is stopped while the collector scans the mutator's stack, we require no further synchronization.

Correctness information for the Pre-Mark Step: The gray set is initially empty before the write barrier is changed in this phase.

Conditions true at the start of the phase: All objects are white. The gray set is empty. All threads have the "standard" write barrier.

Conditions true at the end of the phase: All threads have the mark phase write barrier.

Invariants of the phase: Stack and Uncoll slots are gray. There are no black slots referring to white objects (in particular there are no black slots or black objects, but there may be gray objects); we treat new objects' slots as being gray in this phase. Any gray Old object was reachable at the start of the phase. No objects are allocated into the Old region. Because there are no black slots, the no-black-points-to-white rule holds trivially.

Termination: We assume that any thread created during or after this phase starts with the appropriate write barrier. Hence we need only handshake with each previously existing thread.

The **Root-Mark** step iterates through Uncoll slots and "grays" any white Old objects referred to by those slots, using `mark-write-barrier`. We consider this to be "blackening" the Uncoll 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.

While one could scan the Uncoll 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 [Jones & Lins, 1996].

Correctness information for Root-Mark Step:

Conditions true at the start of the phase: All Uncoll slots are gray.

Conditions true at the end of the phase: All Uncoll slots are black.

Invariants of the phase: Stack slots are gray. All black slots are in Uncoll. Any Old object grayed was reachable from a root. No objects are allocated into the Old region. All threads employ the mark-phase write barrier. Black slots never refer to white objects.

Termination: The set of previously existing Uncoll slots is fixed at the start of the phase, so checking a slot makes progress. New slots are blackened by their initializing stores and are not the collector's responsibility. Thus allocation cannot "get ahead" of the collector in this phase.

In the **Heap/Stack Mark** step, the collector works from its input queue, a separate set of *explicitly* gray (marked) objects, and the thread stacks. For each enqueued object, the collector checks if it 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 Stack pointers to Old objects. To scan a mutator thread's stack, the collector *briefly* stops the mutator thread at a safe point, and scans the thread's stack (and registers) for references to white Old objects, invoking the Mark Phase write barrier on each reference. (One might use stack barriers to bound stack scanning pauses [Cheng *et al.*, 1998].) 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. We claim that there are now no white pointers in Stack or in marked Old 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 Old objects is straightforward. The Old objects initially referred to by Uncoll 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 Old 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.[7] Second, if we use stack barriers [Cheng *et al.*, 1998], 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.

Correctness information for the Heap/Stack-Mark Step:

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

Once the Mark Phase completes, the Mark Phase write barrier may be removed, though it does no harm to correctness if it remains until a different write barrier is required by a later phase.

Conditions true at the start of the phase: All Uncoll slots are black. All Stack slots are gray.

Conditions true at the end of the phase: All Uncoll slots are black. No Stack slot refers to a white object. The gray set is empty. All reachable Old objects are marked and black (some black Old objects may have become unreachable since the collector cycle began). All white Old objects are unreachable.

Invariants of the phase: No objects are allocated into the Old region. All threads employ the Mark-Phase write barrier. Black slots never refer to white objects.

Termination: The set of Old objects does not change during collection. Any given Old object is entered in the explicit gray set at most once, and is therefore scanned and blackened at most once. Likewise, if we scan some thread stacks and find white objects, we add them to the (implicit) gray set and thus make progress. Once we have explicitly grayed every reachable Old object, the write barrier will enter no more slots into the collector's input queue, and stack scans will find no more Old objects to gray.

There are only two possible attacks on progress in marking. One is flooding the queue of implicitly gray objects. Provided we can bound the relative rate of progress of mutators versus the collector, then we can bound the number of times slots referencing a particular Old object are enqueued before the collector marks the Old object. It is a hassle to insure that we have enough queue space, but we can do it. If in practice it seems to be a problem, then we could have mutators set an "enqueued" bit in the objects to prevent or bound the number of duplicates.

The other attack is on stack scanning. If we continually create new threads there might always be stacks not yet scanned by the collector. However, this is not really a problem. Consider the original argument and its time span from t_1 to t_2. Let OT be the set of threads existing at time t_1 and NT be threads created between time t_1 and t_2. We claim that if no thread in OT referred to a white object since t_1, and no objects have been enqueued, then no thread in NT can refer to a white object. For an NT thread to have a pointer to a white object, it would have to load the pointer from the heap,[8] since there is no direct communication between threads.[9] All Old objects reachable from Uncoll slots are black at t_1, and since the gray set remained empty, that property was true from t_1 to t_2. That is, all reachable Old slots (and all Uncoll slots) were black for the whole time. Thus an NT thread cannot have obtained any pointers to white objects.

2.4 The Allocation and Copying Phases: Creating New Copies

The mark phases establish which Old objects are reachable. Once we determine the reachable Old objects, we allocate a New copy for each of them (Allocation Step) and then copy

[8]Note that "the heap" includes "bare slots" in Uncoll (in Java, static fields), implying the need for a write barrier on static fields. We already mentioned that Uncoll slots need a write in Sapphire. Many non-concurrent collectors avoid a write barrier on such global areas, by scanning them during (stop-the-world) collection. Concurrent collectors typically employ a write barrier on global variables.

[9]We must make sure that when threads are spawned, any pointers passed to them are through memory structures, to which write barriers apply.

the Old copy's contents to the allocated New copy (Copy Step).

Allocation Step: Once each reachable Old object has been marked, the collector allocates space for a New copy for it and sets the Old copy's forwarding pointer to refer to the space reserved for the New copy. We then say the Old copy is *forwarded* to the New copy. Clearly the format of objects must be such as to support a forwarding pointer while still allowing all normal operations on the objects.[10]

If the collector saves a list of the objects scanned in the Mark Phase, then it can use that list to find the Old copies. The forwarding pointer slot might serve for the list; if we combine phases (discussed later), then we do not need an explicit list.

Our algorithm also requires us to be able to find the Old copy of an object from its New copy. We do this using a hash table, which we discard after collection. If we instead use back pointers from New copies to Old copies, we would need to remove them later, involving either an extra pass over the New copies or increasing the object header size to provide space for the back pointer all the time.

Correctness information for the Allocation Step:

Conditions true at the start of the phase: No Old objects have New copies allocated.

Conditions true at the end of the phase: Each black Old object has space allocated for a New copy. The mapping between black Old objects and their New copies is one-to-one and onto, Old-to-New via forwarding pointers, and New-to-Old via a separate table.

Invariants of the phase: No new objects are allocated into the Old region. All reachable Old objects are black. The mapping between black Old objects and their New copies is one-to-one, and onto the New copies. If an Old object has a New copy, the New copy has room for the Old object's data.

Termination: The set of black Old objects does not change during this step. Each allocation reduces the set of black Old objects without New copies.

Copy Step: The Copy Step needs a new write barrier, to maintain (dynamic) consistency between the Old and New copies of objects; the **Pre-Copy Step** establishes that write barrier, shown in Figure 3. There are two notations we use in the figure that require further explanation. First, the lines marked with `($)` are memory accessing operations that must be done in the indicated order, as discussed further in Section 4. Second, when an operation or routine returns (or accepts) multiple values packed together, we write the packed set of values as `[v1:v2:...:VN]`. In this case, we pack the information about forwarding from Old to New copies together, returning the forwarding address and forwarded flag as a unit. This is to make clear that both values are read or written using a single memory access.

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 a New copy always points to an Uncoll or New copy, not to an Old one; we maintain the invariant

[10] This is different from a stop-the-world collector, which can "clobber" part of the Old object as long as the data is preserved in the New 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 CAS, retrying as necessary, to install the forwarding address. We "overload" our forwarding on a "thin lock" scheme [Bacon *et al.*, 1998].

```
// Copy Phase Write Barrier:
//    handles pointers, non-pointers differently
//    p[f] = q is the desired update
//    objects p, q may be in Uncoll or Old
copy-write (p, f, q) {
  p[f] = q;  // ($) do the write itself, then ...
  copy-write-barrier(p, f, q); // the write barrier
}

copy-write-barrier(p, f, q) {
  [pp:forwarded] = forwarding-info(p);  // ($)
  if (forwarded) {
    qq = q;
    qq = forward(qq); // omit this for non-pointers
    pp[f] = qq;       // ($) write New after Old
  }
}

forward(p) {
  // Note: forwarding-info exists only if p is Old
  [faddr:forw] = forwarding-info(p);
  return (forw ? faddr : p);
}
```

Figure 3: Sapphire copy phase write barrier

that New copies never refer to Old copies. (Note: when we merge phases in Section 3 this is no longer true.)

We consider *volatile* fields separately, in Section 5, so as not to obscure the basic operation of the algorithm.

Correctness information for the Pre-Copy Step:

Conditions true at the start of the step: Each black Old object has a unique corresponding New copy allocated. No thread has the copy phase write barrier installed. Contents of New objects are undefined.

Conditions true at the end of the step: Every thread uses the copy phase write barrier.

Invariants of the phase: No new objects are allocated into the Old region. All reachable Old objects are black. The mapping between black Old objects and their New copies is one-to-one and onto. If an Old object has a New copy, the New copy has room for the Old object's data. No pointer stored into a New object refers to an Old object. No mutator thread reads from a New copy.

Termination: The set of threads existing at the start of the phase is fixed and finite, and each new thread has its write barrier set appropriately as the thread is created. Thus as each thread is switched to the new write barrier we reduce a fixed set.

The **Copy Step** copies the contents of each black Old object into its corresponding

allocated New copy. If a datum copied is a pointer to an Old object, it is first adjusted to point to the New 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 Old copies to the New 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 Figure 4.

```
// Collector word copying algorithm
//   Goal: copy *p to *q
//   p points to an Old object field
//   q points to the corresponding New field
copy-word(p, q) {
  i = max-cycles; // times to try non-atomic loop
  while (i-- > 0) {
    vn = *p;       // ($) vn: value for the New copy
    vn = forward(vn);  // omit for non-pointers
    *q = vn;       // ($) write New
    vo = *p;       // ($) read Old value into vo
    if (vo == vn)  // if equal, we are done
      return;
  }
  LL(q);  // ($) ignore value: LL only for effect
  vn = *p;         // ($) must read after the LL
  vn = forward(vn);  // omit for non-pointers
  SC(q, vn);       // ($) SC succeeds only if *q has
}                  // not been updated since the LL
```

Figure 4: Collector's word copying algorithm

The non-atomic copying loop is based on Lamport's algorithm [Lamport, 1977]. If used as the sole strategy, it could loop indefinitely, though it was what we used in our implementation and we had no problems with it. The LL/SC technique is guaranteed to complete, and could certainly be used by itself (i.e., the code above is intended to illustrate options rather than to offer the best performance). CAS is less suitable, since it would require retrying like the copy loop (see the discussion in Section 4).

Note that `copy-word` assumes that mutators do not attempt updates of *non-volatile fields* concurrently with each other (i.e., they use proper locking to avoid data races). We treat Java `volatile` fields in Section 5, to avoid disrupting the flow of the presentation.

Correctness information for Copy Step:

Conditions true at the start of the phase: Each black Old object has a unique corresponding New copy allocated. Contents of New copies are undefined.

Conditions true at the end of the phase: Contents of New copies are "dynamically consistent" with their (unique) Old copies. More precisely, when no mutator is in the

middle of write barrier code for a given slot, the New and Old copies of that slot have consistent values. For references to copied objects, "consistent" means that the New value is the forwarded version of the Old value; for other pointers and for non-pointer data, "consistent" means that the values are bit-wise equal.

Invariants of the phase: All threads use the Copy Phase write barrier. No new objects are allocated into the Old region. All reachable Old objects are black. The mapping between black Old objects and their New copies is one-to-one and onto. If an Old object has a New copy, the New copy has room for the Old object's data. No pointer stored into a New object refers to an Old object. No mutator thread reads from a New copy.

Termination: There is a fixed set of slots to be copied. The copying routine terminates after a fixed maximum time for each slot.

2.5 The Flip Phases: Abandoning Old Objects

The Flip phase steps are: Pre-Flip, Heap-Flip, Thread-Flip, and Reclaim. The goal of these steps is systematically to eliminate Old pointers that may be seen and used by a thread, as follows. First, we install a write barrier that insures that we do not write Old pointers into New or Uncoll slots. We next insure that there are no heap (Uncoll region) pointers to Old objects. We then flip each thread.

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

As long as there are any unflipped threads, all threads must update both the Old and New copies of objects having two copies. However, because of the way in which we take advantage of Java mutual exclusion semantics, the order (Old first or New first) does not matter.[11] The main import is that we need a way to "unforward" from a New object to its Old copy.

Since unflipped threads may access both Old and New copies of the same object, pointer variable equality tests such as `p == q` need to be a little more complex, as previously discussed.

A final point to consider is whether the aliasing introduced by having two distinct copies of the same object could be a problem. We require that both the Old and New copies of a field be updated before the same thread undertakes an access that might turn out to be to the same field. With this rule, the underlying hardware will provide the correct value to the thread, because it writes both copies and then reads one or the other. There is no problem with other threads because we do not allow them to access non-volatile fields without doing Java synchronization, and we handle volatile fields such that one copy or the other is authoritative.

The Pre-Flip Step installs the Flip Phase Write Barrier, shown in Figure 5. The `get-old-copy` routine uses a hash table to map a New copy to its corresponding Old copy.

[11] The situation with volatile fields is more complex; see the separate section on them.

```
// Flip Phase Write Barrier
//   p[f] = q is the update
//   object p may be in Uncoll, Old, or New
//   q may be a pointer or non-pointer; if a
//     pointer, it may refer to Uncoll, Old, or New
flip-write(p, f, q) {
  qq = q;  // qq is value to write in both copies
  qq = forward(qq);   // omit for non-pointers
  p[f] = qq;
  if (old(p)) {
    pp = forward(p);  // follow Old->New
    pp[f] = qq;
  }
  else if (new(p)) {
    pp = get-old-copy(p);  // follow New->Old
    pp[f] = qq;
  }
}
```

Figure 5: Flip phase write barrier

The Flip Phase write barrier must be installed before the Heap-Flip phase. Otherwise, unflipped threads might write Old pointers in Uncoll or New slots. Likewise, the extended pointer equality test must be installed now, since the Heap-Flip phase will start to expose New pointers to unflipped threads. (The extended pointer equality test can be used all the time, if that is more convenient or efficient.)

Figure 6 gives pseudo-code for one way of implementing pointer equality tests:[12]

Correctness information for Pre-Flip Step:

Conditions true at the start of the phase: New object contents are dynamically consistent with their Old copies. All mutator threads use the Copy Phase write barrier.

Conditions true at the end of the phase: All mutators use the Flip Phase write barrier. No further Old pointers will be written (anywhere).

Invariants of the phase: No new objects are allocated into the Old region. All reachable Old objects are black, and have a unique corresponding New copy, with which they are dynamically consistent. No New object refers to an Old object.

Termination: There is a fixed set of threads to be processed, and processing each one takes no more than a fixed number of operations. (New threads will be spawned with the new write barrier, so termination is not threatened by thread creation.)

The **Heap-Flip** Step is straightforward: it scans every Uncoll slot that might contain

[12] This pointer equality test assumes that the thread is not suspended in the middle of the test while the collector completes collection and a new collection starts. If that could happen, we could end up with an Old version of `p` but a forwarded version of `q`, and the test could then give the wrong answer. The easiest fix is to insure that threads in this code advance to the end of it before collection completes. Such thread advancing requirements apply to other Sapphire pseudo-code fragments as well, i.e., any write barriers. Our handshake mechanisms accomplish this automatically.

```
// Flip Phase Pointer Equality Test
//   p == q is the desired test
//   objects p and q may be in Uncoll, Old, or New
//   we assume p != null, q != null, p != q
//     (in the sense of bit patterns)
flip-pointer-equal(p, q) {
  if (old(p)) p = forward(p);
  if (old(q)) q = forward(q);
  return (p == q);
}
```

Figure 6: A possible Flip phase pointer equality test

an Old pointer, fixing Old pointers to refer to their New copies. Because of possible races with mutator updates, the collector employs CAS, ignoring failures since a conflicting mutator thread can only write a New pointer in this phase. Figure 7 gives pseudo-code.

```
// Heap-Flip: Uncoll pointer forwarding
//   p points to a Uncoll slot, which
//     may point to an Old object
flip-heap-pointer(p) {
  q = *p;  // ($)
  if (old(q))  // CAS to avoid race with mutator
    CAS(p, q, forward(q));
}
```

Figure 7: Routine to flip slots in Uncoll

Correctness information for Heap-Flip Step:

Conditions true at the start of the phase: No store to Uncoll or New slots stores an Old pointer (but Uncoll objects may contain Old pointers).

Conditions true at the end of the phase: Uncoll objects (and New objects) contain no Old pointers.

Invariants of the phase: No new objects are allocated into the Old region. All reachable Old objects are black, and have a unique corresponding New copy, with which they are dynamically consistent. No New object refers to an Old object. No stores to Uncoll or New slots store an Old pointer (because all mutators use the Flip Phase write barrier).

Termination: There is a fixed set of slots to be processed, and processing each one takes a bounded number of operations.

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

Correctness information for Thread-Flip Step:

Conditions true at the start of the phase: Stack slots may refer to Old objects.

Conditions true at the end of the phase: Stack slots do not refer to Old objects.

Invariants of the phase: No new objects are allocated into the Old region. All reachable Old objects are black, and have a unique corresponding New copy, with which they are dynamically consistent. No New object refers to an Old object. No stores to Uncoll or New store an Old pointer (because all mutators use the flip-phase write barrier).

Termination: There is a fixed set of threads to process (those existing at the start of the phase), each thread's stack has a fixed number of slots, and processing each slot takes a bounded number of operations.

The **Reclaim** Step 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 Old copy), at last the collector may discard the hash table mapping New copies to Old copies and reclaim the space holding Old copies.

Correctness information for the Reclaim Step:

Conditions true at the start of the phase: New objects have a reverse mapping to their Old counterparts. All threads use the Flip Phase write barrier.

Conditions true at the end of the phase: No New object has a reverse mapping to an Old object. All threads use the normal write barrier.

Invariants of the phase: No Stack, Uncoll, or New slot refers to an Old object. All allocation occurs in Uncoll space.

Termination: There is a fixed and finite set of threads to process (those existing at the start of the phase), and a finite New-to-Old mapping data structure to reclaim, each requiring a finite amount of work.

3 Merging Phases

Some Sapphire phases must be strictly ordered and cannot be merged. However, we can merge some 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. Figure 8 gives pseudo-code for the Replicate-Phase write barrier, which essentially combines the previous Mark and Copy Phase write barriers. However, if we write an Old pointer into a New slot, in addition to enqueuing the object for copying, we enqueue the *slot*, on a separate queue so that it can be fixed later. For clarity, in this case we present separate versions for pointers and non-pointers.

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; in our implementation 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 roots, heap slots (slots in Uncoll that might refer to Old objects), and stack

```
// Replicate Phase Write Barrier
//   p[f] = q is the update
//   object p may be in Uncoll or Old
//   object q may be in Uncoll or Old
// This is for pointers and non-pointers
replicate-phase-write(p, f, q) {
  p[f] = q;
  // use appropriate write barrier below
  replicate-write-barrier-X(p, f, q);
}

// Use this routine if q is a pointer
replicate-write-barrier-pointer(p, f, q) {
  [qq:forward-q] = forwarding-info(q);
  if (!forward-q) qq = q;
  [pp:forward-p] = forwarding-info(p);  // ($)
  if (forward-p) {
    p = pp;
    p[f] = qq;      // ($) write New after Old
  }
  if (old(q) && !forward-q) {
    enqueue-object(q);      // insure q copied
    if (!old(p))
      enqueue-slot(&p[f]);  // insure slot fixed
  }
}

// Use this routine if q is a non-pointer
replicate-write-barrier-non-pointer(p, f, q) {
  [pp:forwarded] = forwarding-info(p);  // ($)
  if (forwarded) {
    pp[f] = q;       // ($) write New after Old
  }
}
```

Figure 8: Replicate phase write barrier

slots, and for each one calls `mark-write-barrier`, possibly adding objects to its object queue. The order does not affect correctness.

2. For each object in its input queue, if the object is not yet copied, it allocates a New copy and installs a forwarding pointer. We call these steps `forward-object`.
3. As part of `forward-object`, it processes each slot of the just-forwarded object, recursively, forwarding the referent if needed. It also updates the New copy's slot to refer to its referent's New copy as needed. We call this work `scan-slot`.

4. The phase terminates when (a) all roots and Uncoll heap slots have been scanned, (b) all thread stack slots have been scanned and found to contain no white pointers, while the collector object queue has remained empty, (c) all New copies have been scanned, and (d) all enqueued New slots have been scanned. As previously noted, condition (d), may require slots to be scanned more than once. However, assuming the collector processes the object queue first, it will eventually copy and forward each reachable Old object, and then the mutator write barrier will no longer enqueue New slots. Hence there is not a termination problem.

```
// Scan slot
//   called on each field p[f] needing scanning
//   object p is in New
// We: - copy the field from the Old copy,
//     - forward the referent, and
//     - install the new address
scan-slot(p, f) {
  pp = get-old-copy(p);
  copy-word(&pp[f], &p[f]);
  // do the rest only for a pointer slot
  v = p[f];
  forward-object(v);
    // forward-object: see description in text
  [vv:forwarded] = forwarding-info(v);
  if (forwarded)  // CAS to avoid race with mutator
    CAS (&p[f], v, vv);
}
```

Figure 9: Replicate phase procedure for scanning a New slot

Correctness information for the Replicate Phase:

Conditions true at the start of the phase: All objects and slots are white or implicitly gray (enqueued). The explicit gray set is empty.

Conditions true at the end of the phase: All reachable Old objects are black, having a unique corresponding New copy, and the copies are dynamically consistent.

Invariants of the phase: Black slots do not point to white objects. No objects are allocated into the Old region.

Termination: For the moment, ignore the re-processing of slots from the slot queue. We process each Uncoll slot at most once and each Old object slot at most once. We may process thread stacks multiple times, but if so, it is because we are finding more references to uncopied Old objects and thus making progress. Now consider the slots in the slot queue. Once we have copied all reachable Old objects, the mutators will no longer enqueue New slots. The collector can then empty the slot queue.

4 Discussion of Synchronization Issues

The most subtle aspects of Sapphire concern synchronization around collector copying that is concurrent with mutator field access. Let us consider first non-volatile fields, where the mutator uses `copy-write-barrier` and the collector copies with `copy-word`.

Notation and concepts: An *event* happens (conceptually) at an instant of time, and may be ordered with respect to other events: $x \rightarrow y$ means event x occurs before y. We assume that $\rightarrow$ is a partial order over events. Memory reads and writes are events, some of which are ordered by $\rightarrow$.[13]

Correctness of `copy-word` with `copy-write-barrier`: The mutator's action in `copy-write-barrier` is simple: it writes a field in the Old copy and then writes the corresponding field in the New copy.[14] The collector's action can be more complex. Consider first just the while loop in `copy-word` (Figure 4). On each iteration it consists of: read-Old, write-New, read-Old, where the second read-Old checks to see if the Old field has changed since the first read. Suppose that the second read-Old returns the same value as the first read-Old. Then either the field has not changed (and `copy-word` copied it correctly), or it has been updated, possibly multiple times, but most recently with the same value as the first read-Old. Let MWO and MWN be the mutator writes to the Old and New copies, for the invocation of `copy-write-barrier` that wrote the value seen by read-Old. Let CRO1, CWN, and CRO2 be the collector's first read-Old, write-New, and second read-Old accesses, respectively. We have MWO $\rightarrow$ MWN and CRO1 $\rightarrow$ CWN $\rightarrow$ CRO2 from the logic of the code, and MWO $\rightarrow$ CRO2 by definition. However, MWN and CWN write the *same value*, so their relative ordering does not matter.

It would be *correct* to use just a do-forever loop in place of the while loop, but continual mutator updates of the field could prevent collector copying progress.[15] To obtain a strong guarantee progress, the second part of `copy-word` uses an atomic update primitive, LL/SC, to accomplish the copying in a fixed number of steps. This might be more expensive than the read-write-compare actions of the while loop. Let us call the collector accesses CLN (LL New), CRO (read Old), and CSN (SC New); they are ordered CLN $\rightarrow$ CRO $\rightarrow$ CSN. A complication is that there may be more than one mutator write barrier overlapping the invocation of `copy-word`, even though the mutator write barriers cannot overlap with each other.

Consider the MWN of the latest completed overlapping invocation of the write barrier, and its corresponding MWO access. There are several cases that can occur:

- CSN fails: Any time the collector SC fails, it means the mutator did a (correct) update, so it is fine that the collector takes no further action.
- MWO $\rightarrow$ CRO, and CSN succeeds: The collector "sees" the new value and installs it. MWN must come later (because the SC worked), and redundantly installs the same value.
- CRO $\rightarrow$ MWO, and CSN succeeds: The mutator has not yet applied its update

[13] It is not our task at the moment to define the Java memory model in its entirety.

[14] In later phases, these writes may occur in the opposite order, but during the Copy (or Replicate) Phase, they will always happen in the order Old then New.

[15] We used this technique and had no problems in practice.

with MWN, so even though the collector writes a stale value, the correct value will appear in due time.

We note that CAS is *not* a suitable replacement for the SC operator, since a series of multiple mutator writes could "conspire" to leave the value read by CLN and the collector would then incorrectly store a stale value. One could use CAS in a loop similar to the while loop, to reduce the probability of looping.

Correctness of Java Synchronization and Copying/Forwarding: Mutators might invoke Java monitor lock operations concurrently with collector copying of an object. We assume a "thin locks" strategy for implementing Java monitor locks [Bacon *et al.*, 1998]. In this strategy, a monitor lock operation is accomplished either by a single atomic state transition, using CAS (or an alternative such as LL/SC or FA), or by using CAS (or alternative) to set a *meta-lock* on the lock word and release it afterwards. By design one guarantees that meta-locks are held only for short periods of time, to accomplish atomic state transitions in the Java monitor lock structures.

We have the collector set the meta-lock before setting the forwarding pointer in a copied object. While it holds the meta-lock, the collector copies the monitor lock information from the Old to the New copy, an then installs the forwarding pointer. We store the forwarding pointer in the lock/status word of the object, and give it a distinct state, "forwarded". If a thread encounters a forwarded object, it simply uses the New copy's lock word. Our design strategy had minimal impact on the locking code in the system.

We found it convenient to place monitor lock queueing data structures in a non-moving, explicitly managed, space. This eliminated any need to copy them carefully, etc. Since they are fixed size and usually few in number, this design decision is quite practical.

5 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 memory read (write) of the field when the code is executed at run time. Non-volatile fields, in contrast, may be accessed from copies in thread-local memory (stack, registers), modulo certain synchronization constraints.

Volatile fields also have different memory synchronization properties from non-volatile fields: a thread's physical memory accesses to volatile fields must occur essentially in the order given by the code. Further, volatile and non-volatile accesses must be performed in the proper order with respect to Java monitor lock actions (acquisition and release of monitor locks). This is needed to insure that threads that acquire a lock see the values written by previous lock holders, etc. (Non-volatile accesses must also be ordered with respect to lock actions.)

We have purposely avoided getting into all the details, since it is clear that the original Java memory model (JMM) specification has substantial problems, as described by Pugh [Pugh, 1999]. The proposed new model, described in a document by Manson and Pugh [Manson & Pugh, 2001], deals with several issues. First, the new model allows more compiler optimizations pertaining to accesses to non-volatile fields. The old model *appeared* to allow substantial "caching" of values in the stack and registers, supporting typical well-known optimizations, but in fact the ordering requirements in the presence of multiple

threads severely undermined this. Second, the new model allows optimizing away more synchronization operations, such as those on provably thread-local synchronized objects, which previously required memory fences to enforce correct ordering of accesses to other variables. Third, the new model clarifies how to insure safe initialization of objects (i.e., that non-type-safe values can never be read, even in the presence of concurrency). The JMM of Manson and Pugh, or a later draft, is quite likely to be accepted, so we base our discussion on it.

A key point for us around the JMM is that the analyses of possible behaviors in the original JMM reveal that data race accesses to non-volatile fields are not very useful, because they do not have strong enough synchronization and ordering properties. In the past, various people had proposed synchronization strategies using field accesses *without* Java monitor locks. We now know that these can fail using current compilers and machines that reorder memory accesses, and that enforcing the orderings of the original model can be prohibitively expensive. The new model essentially requires such fields to be declared volatile to be useful. This is why we feel justified in not handling data races on non-volatile fields in Sapphire. Further, there are now tools emerging to assist in detecting and removing data races from Java programs, so there are methodologies available for building race-free programs that will avoid the anomalies that Sapphire's handling of non-volatile fields might introduce into programs with data races. Note that the worst that Sapphire can do is leave an older value of a non-volatile field installed in the new copy of an object, so Sapphire's handling of non-volatile fields does not break Java type safety.

A further point is relevant here: standard benchmark suites, such as SPECjvm, do not make direct use of volatiles at all (some JVMs may use volatiles within library implementations), and neither do the benchmarks we present later. Unlike the rest of Sapphire, the algorithms for volatiles that we offer below have not been implemented or tested.

5.1 A Volatile Field Protocol

We now consider a protocol for accessing and copying volatile fields. We aim to minimize the impact on mutator reads and writes, possibly at the expense of more complex collector copying. However, since mutator volatile accesses to the same field demand total ordering of the writes, and ordering of reads with respect to writes, we need to insure adequate ordering of accesses while somehow moving the focus of physical memory activity from the old copy of an object to the new copy.

We present three versions of our design, a simpler one based on CAS synchronization that is not scalable to high contention situations, a more complex but more scalable design based on fetch-and-add (FA) synchronization, and a version of the FA design that synchronizes at the level of individual fields rather than whole objects (similar to the technique of Blelloch and Cheng [Blelloch & Cheng, 1999]). All three designs are based on three phases of activity:

(1) before the collector begins to copy an object (or a particular field), volatile accesses occur on the *Old* copy of the field;

(2) while the collector copies the field, volatile accesses are locked out;

(3) when the collector is done copying the (volatile field or fields of the) object, volatile accesses occur on the *New* copy of the field.

Further, we can characterize the operations abstractly as shown in the code fragments

```
// Volatile read
//   reads volatile field f of object p
//   p may be Old, New, or Uncoll
volatile-read(p, f) {
  a = get-lock-address(p);
  v = p[f]; // ($) must read memory
  limit = wait-while-copying(a, p, f);
  [pp:forw] = forwarding-info(p); // ($)
  if (forw && f >= limit)
    v = pp[f];                    // ($)
  return v;
}

// Volatile write
//   writes nv to volatile field f of object p
//   p may be Old, New, or Uncoll
void volatile-write(p, f, nv) {
  a = get-lock-address(p);
  nv = forward(nv);  // omit for non-pointers
  [limit:flag] = write-enter(a, p, f);
  [pp:forw] = forwarding-info(p); // ($)
  if (forw && f >= limit) p = pp;
  p[f] = nv;         // ($) must write memory
  write-exit(a, p, flag);
}
```

Figure 10: Mutator procedures for accessing volatile fields

given in Figures 10 and 11. The designs implement some subroutines differently. All designs associate with each object a *lock word* used for synchronizing volatile accesses. One may use the same lock word for multiple objects, e.g., to save space, though such combining may block mutators more often in some of the designs.

Note that the mutator routines for reading and writing volatile fields also work when passed references to New copies, as happens in the Flip phases. In the Flip phases, copying (or replication in the case of merged phases) is complete, so all volatile accesses go to the New copies. If the routines are passed a New reference, they access the New copy directly; if passed an Old reference, they forward to the New copy.

5.1.1 CAS Design

We now offer the CAS design. The lock word has two fields, written `[count:busy]`, where `count` is an integer count of the number of volatile writes in progress, and `busy` is a one-bit flag indicating that copying is in progress. In this design all volatile accesses to an object are held back while the object's volatile fields are copied. These operations could easily be recast using LL/SC. Figures 12 and 13 give the code.

```
// Copy volatile fields
//   copies volatile fields of p to New copy pp
//   installs forwarding pointer
void copy-volatile-fields(p, pp) {
  a = get-lock-address(p);
  copy-volatiles-start(a, p);
  install-forwarding-pointer(p, pp);
  for each volatile field f of p do {
    // processes fields from largest f to smallest
    copy-one-volatile-start(a, f);
    v = p[f];          // ($) must read memory
    v = forward(v);    // omit for non-pointers
    pp[f] = v;         // ($) must write memory
    copy-one-volatile-end(a);
  }
  copy-volatiles-end(a);
}
```

Figure 11: Collector routine to copy volatile fields

Correctness of `volatile-read`*:* There are three cases of interest. (1) Read old copy, lock word not busy, object not forwarded: the field read occurred on the old copy before copying could have started (Phase 1), and memory ordering occurred on the old copy field. (2) Read old copy, lock word initially busy, wait, object forwarded, read new copy: the value returned is from the new copy field, read after copying is complete (Phase 2 then Phase 3). (3) Read old copy, lock word not busy, object forwarded, read new copy: this situation is after copying, and it returns the new field value (Phase 3).

Correctness of `volatile-write`*:* There are two cases: `write-enter` completes in Phase 1, or it completes in Phase 3. In the Phase 1 case, the object is not forwarded, and the routine writes the old copy field. Because the collector's copying waits until `count` is 0, the write precedes the copier's read. In the Phase 3 case, which may occur after waiting in `write-enter` for Phase 2 to complete, the object is forwarded and we write the new copy. This occurs after the copier copies the field from the old to the new copy.

Correctness of `copy-volatile-fields`*:* First, `copy-volatiles-start` marks the lock word as busy. This prevents any new writers from entering (they will wait in `write-enter`). Next, it waits for the lock word's count to become 0, signifying that no writers are in the middle of `volatile-write`. At this point, the copier is serialized after preceding writers. The copier then copies the volatile fields from the old copy to the new copy, and installs the forwarding pointer. Finally, using `copy-volatiles-end`, it turns busy off and allows waiting readers and writers to proceed. Now they will read and update the new copy, properly serialized after the copying.

```
// Wait while copying
//   called by volatile reader:
//     prevents reading during copying
//   a is the address of the lock word
//   p[f] is the field being accessed
//   returns "limit": larger offsets have been copied
int wait-while-copying(a, p, f) {
  // CAS design ignores p, f; always returns limit -1
  do {
    // below, a - indicates an ignored field
    [-:busy] = *a;  // ($) must read memory
  } while (busy);
  return -1;
}

// Write enter
//   called by volatile writer, before writing
//   a is the address of the lock word
//   p[f] is the field being updated
//   Effect: wait while busy, then increment count
//   returns limit (like wait-while-copying) and flag
int write-enter(a, p, f) {
  // CAS design ignores p, f; always returns [-1:0]
  do {
    do {
      [count:busy] = *a;
    while (busy);
    // here, saw not busy
    } while (!CAS(a, [count:0], [count+1:0]));
  return [-1:0];
}

// Write exit
//   called by volatile writer, after writing
//   a is the address of the lock word for object p
//   flag is the value from write-enter
//   Effect: decrement count atomically
void write-exit(a, p, flag) {
  // CAS design ignores p and flag
  do {
    [count:busy] = *l;  // note: count > 0 here
  } while (!CAS(a, [count:busy], [count-1:busy]));
}
```

Figure 12: Mutator support routines for CAS volatile design

```
// Copy volatiles start
//   called by collector:
//     before copying any volatile fields of p
//   a is the address of the lock word for object p
//   Effect: set busy and wait for zero count
void copy-volatiles-start(a, p) {
  // CAS design ignores p
  do {
    // Invariant: busy == 0
    [count:busy] = *a;
  } while (!CAS(a, [count:busy], [count:1])); // ($)
  // here, succeeded in setting busy
  while (count != 0) {
    [count:-] = *a;  // ($) must do a read
  }
  // Postcondition: count == 0, busy == 1
}

// Copy volatiles end
//   called by collector:
//     after copying volatile fields of p
//   a is the address of the lock word
//   Effect: clear busy
void copy-volatiles-end(a) {
  // Precondition: count == 0, busy == 1
  *a = [0:0];  // ($) must do a write
}

// Copy one volatile start
//   called by collector:
//     before copying volatile field f of p
//   a is the address of the lock word
//   Effect: not used in CAS design
void copy-one-volatile-start(a, f) { }

// Copy one volatile end
//   called by collector:
//     after copying volatile field f of p
//   a is the address of the lock word
//   Effect: not used in CAS design
void copy-one-volatile-end(a) { }
```

Figure 13: Collector support routines for CAS volatile design

5.1.2 Fetch-and-Add (FA) Design

In the FA design, the lock word consists of three fields, written `[xcnt:ncnt:busy]`. The `busy` field indicates that the copier is active, as in the CAS design. The other two fields are counts, each large enough to count somewhat more than the maximum number of threads allowed in the system. The `ncnt` field counts the number of threads that have *entered* to write a field, and the `xcnt` field counts the number that have *exited*.

We maintain the counts modulo N. Given that T is the maximum number of threads allowed in the system, and that the count fields are k bits wide, we require that $T < N < 2^k - T$. The first constraint is so that up to T threads can be waiting. The second requirement is to prevent the count field from overflowing into the next field.

When a count is incremented to N, the incrementer will subtract N from the count, to maintain it essentially modulo N. Any incrementer that sees the count greater than N will wait for the count to be lowered, thus avoiding overflow.

Figures 14 and 15 give the subroutines. As in the previous design, we do not use either of `copy-one-volatile-start` and `copy-one-volatile-end`. Note that where we have done `FA(a, 0)`, depending on the hardware it might work simply to read the lock word `a`.

Correctness: The FA design's correctness follows easily from the correctness of the CAS design, since the CAS `count` field is just the difference between the entry and exit counts of the FA design (`count` $=$ (`xcnt` $-$ `ncnt` mod N).

5.1.3 Second Fetch-and-Add (FA) Design

In this design, the lock word has three count fields, `xcnt` and `ncnt` somewhat like before, and `bcnt`, for those threads that encounter the lock word with `busy` set to 1. (The `xcnt` field is the exit count for the ones that find `busy` equal to 0). The lock word also has fields called `obj`, to indicate the object being copied, `off`, for the offset being worked on, and `busy`, set until a field is done, and then reset until starting the next field. The overall format is `[bcnt:xcnt:ncnt:obj:off:busy]`.

Note that this design may require that FA be available on a double-word quantity, given that `obj` will require most of a machine word, etc. We maintain the counts modulo N as in the previous design. Figures 16 and 17 give the subroutines.

The advantage of this FA design is that waiting is always bounded by a small number of steps in another thread. A design that is intermediate between the FA designs would cause mutators to wait only when the particular object they are accessing is being copied, using the `bcnt` and `xcnt` technique of the second design, but omitted the `off` field, etc.

Correctness: This design is admittedly more subtle than the first FA design. Understanding it hinges on realizing that the `busy` bit controls alternation between two distinct modes for volatile field writers: writers that see `busy` equal to 1 increment `bcnt` on exit (called "busy" writers), and writers that see 0 increment `xcnt` (called "non-busy" writers).

When the copier changes the value of `busy`, it always waits for all previous-mode threads to exit before proceeding. Thus, when it sets `busy`, it waits for all previous non-busy threads to exit. It detects this condition when `xcnt` is sufficiently incremented. Note that any writers entering after the change to `busy` will increment `bcnt` on exit. Thus once `xcnt` reaches the target value, it is stable (until `busy` is set to 0 again). When `busy`

```
int wait-while-copying(a, p, f) {
  // FA design ignores p, f; always returns limit -1
  do {
    [-:-:busy] = FA(a, [0:0:0]);  // ($) busy?
  } while (busy);
  return -1;
}

int write-enter(a, p, f) {
  // Effect: increment entry count; wait while busy
  // ignores p, f; always returns [-1:0]
  [-:ncnt:busy] = FA(a, [0:1:0]);    // ($)
  // handle entry count wraparound
  if (ncnt == N)
    [-:-:busy] = FA(a, -[0:N:0]); // ($) wrap mod N
  else if (ncnt > N) {
    do {  // wait here prevents overflow
      [-:e:busy] = FA(a, [0:0:0]);   // ($)
    } while (e > N);
  }

  // wait while busy
  while (busy) {
    [-:-:busy] = FA(a, [0:0:0]);     // ($)
  }
  return [-1:0];
}

void write-exit(a, p, flag) {
  // Effect: increment xcnt
  // this design ignores p and flag
  [xcnt:-:-] = FA(a, [1:0:0]); // ($)
  // handle xcnt wrap around here:
  // as for ncnt in write-enter (details omitted)
}
```

Figure 14: Mutator support routines for FA volatile design

```
void copy-volatiles-start(a, p) {
  // sets busy, waits for previous writers to finish
  // this design ignores p
  // Precondition: busy == 0

  // set busy; sample xcnt, ncnt
  [xcnt:ncnt:-] = FA(l, [0:0:1]);  // ($)

  // wait for current writers to finish
  while (xcnt mod N != ncnt mod N) {
    // do NOT change local variable ncnt!
    [xcnt:-:-] = FA(l, [0:0:0]);   // ($)
  }
}

copy-volatiles-end(l) {
  // Precondition: busy == 1
  [-:-:-] = FA(l, -[0:0:1]);  // ($) clears busy
}
```

Figure 15: Collector support routines for FA volatile design

changes from 1 to 0 in `copy-one-volatile-end`, the same things happens, with the roles of `xcnt` and `bcnt` reversed. (At first it might seem that we need not wait in the 1-to-0 case, but we need to insure that all the "busy" writers have exited before the next 0-to-1 transition.) In each case it is important that (a) the counts are sampled atomically, and (b) the sampling is atomic with respect to the change to `busy`.

In the 0-to-1 transition, the copier waits until all writers that entered before the transition have exited. All writers that enter *after* the transition will wait, if they desire to access the field to be copied; and likewise for readers that check after the transition. Here is a table of the various cases, to make clear when Old or New copy fields are read/written; we write the cases in the temporal order in which they occur, before, during, and after processing object `p`:

Condition	Copy	Phase
`p != obj && !forwarded(p)`	Old	1
`p == obj && f < off`	Old	1
`p == obj && f == off && busy`	(wait)	2
`p == obj && f == off && !busy`	New	3
`p == obj && f > off`	New	3
`p != obj && forwarded(p)`	New	3

```
int wait-while-copying(a, p, f) {
  do {
    [-:-:-:obj:off:busy] =
                  FA(a, [0:0:0:0:0:0]); // ($)
  } while (obj == p && off == f && busy);
  // wait only for same object & field to be copied
  return (obj == p) ? off : -1;
}

int write-enter(a, p, f) {
  // increment entry count; wait while busy
  [-:-:ncnt:obj:off:busy] =
                  FA(a, [0:0:1:0:0:0]); // ($)
  // handle ncnt count wraparound (details omitted)

  // wait only if this field is being updated
  flag = busy
  while (obj == p && off == f && busy) {
    [-:-:-:obj:off:busy] =
                  FA(a, [0:0:0:0:0:0]); // ($)
  }
  limit = (obj == p) ? off : -1;
  return [limit:flag];
}

void write-exit(a, p, flag) {
  // increment bcnt or xcnt, according to flag
  if (flag) {
    [bcnt:-:-:-:-:-] = FA(l, [1:0:0:0:0:0]); // ($)
    // handle bcnt wraparound (details omitted)
  } else {
    [-:xcnt:-:-:-:-] = FA(l, [0:1:0:0:0:0]); // ($)
    // handle xcnt wraparound (details omitted)
  }
}
```

Figure 16: Mutator support routines for second FA volatile design

```
static int currobj = 0;
static int lastoff = -1;

void copy-volatiles-start(a, p) {
  // switch to object p, with a big offset
  // At entry: obj==0, off==0, busy==0
  currobj = p;
  lastoff = value larger than any field offset for p
  [-:-:-:-:-:-:-] = FA(a, [0:0:0:p:lastoff:0]); // ($)
}

void copy-one-volatile-start(a, f) {
  // set offset and busy
  // wait for previous (non-"busy") threads to exit
  // At entry: lastoff>f, currobj==obj, busy==0
  delta = lastoff - f;      // delta > 0
  // change to offset f, setting busy
  [bcnt:xcnt:ncnt:-:-:-] =
    FA(a, [0:0:0:0:0:1]-[0:0:0:0:delta:0]);  // ($)
  lastoff = f;
  // wait for non-busy entered threads to exit (xcnt)
  while ((bcnt + xcnt) mod N != ncnt mod N)
    [-:xcnt:-:-:-:-] = FA(a, [0:0:0:0:0:0]); // ($)
    // do NOT update local variables bcnt or ncnt!
}

void copy-one-volatile-end(a) {
  // clear busy
  // wait for previous "busy" threads to exit
  // Precondition: busy == 1
  [bcnt:xcnt:ncnt:-:-:-] =
         FA(a, -[0:0:0:0:0:1]);  // ($) clears busy
  // wait for busy entered threads to exit (bcnt)
  while ((bcnt + xcnt) mod N != ncnt mod N)
    [bcnt:-:-:-:-:-] = FA(a, [0:0:0:0:0:0]);
    // do NOT update local variables xcnt or ncnt!
}

copy-volatiles-end(a) {
  // At entry: obj==currobj, off==lastoff, busy==0
  [-:-:-:-:-:-] =          // ($) clear obj and off
             FA(a, -[0:0:0:currobj:lastoff:0]);
}
```

Figure 17: Collector support routines for second FA volatile design

6 Weak Access Ordering

Many modern CPUs may perform memory accesses in an order different from the issuing order of the accessing instructions. This allows hardware to proceed with fewer interlocks and synchronizations, but sometimes requires the programmer (or compiler) to insert explicit ordering instructions (such as "memory fences"). We have marked in the pseudo-code, with `($)`, those memory accesses that need to be ordered for Sapphire to work correctly. Obtaining the desired order is an architecture-specific concern.

7 Approaches to Parallelizing Sapphire

In the unmerged version of Sapphire, it would be fairly easy to parallelize marking, allocating, copying, and flipping. For example, in marking chunks of queue entries coming from the mutators could be work items in a shared work pool. Allocating and copying work could be divided up according the subregions of the Old region, and one would eliminate allocation bottlenecks in the usual way, by having collector threads request blocks of space from a shared pool, and then allocating into a thread-local block without need to synchronize allocation. One can parallelize flipping according to spatial subregions, such as individual thread stacks. In the version of Sapphire with merged phases, one would proceed similarly.

The one place where one would need to adjust our design a bit is in the copying of volatile fields. The algorithms we presented will continue to work so long as two collector threads do not try to copy fields of objects that share the same lock word. By suitable design of the mapping of objects to lock words and of copying work to collector threads, one can readily meet this constraint. However, if it appears desirable to remove the constraint, then the copying algorithms need to be adjusted so that a copier will wait if the lock word is in use by another copier. These adjustments are straightforward.

8 Related Work and Discussion

There is considerable literature on concurrent GC, so we limit discussion mostly to more recent and more closely related works, focusing on concurrent GC algorithms for shared memory multiprocessors applicable to well-known programming languages. For a broad overview, see the book by Jones and Lins [Jones & Lins, 1996]. Three basic approaches to GC are reference counting, mark-sweep (possibly with compaction added), and copying; all these approaches have concurrent versions, which we now consider.

8.1 Reference Counting

This approach has the virtue of making it easy to spread the work out and reduce mutator interruptions. Two recent works [Bacon *et al.*, 2001, Levanoni & Petrank, 2001] describe reference counting GC algorithms for Java, with implementation and experimental results. We note that a major challenge in implementing concurrent reference counting is reducing the synchronization required for updating reference counts and for triggering reclamation. Like Mark-Sweep, reference counting does not move or compact objects, so it can lead to

fragmentation and does not admit heap reorganization to improve locality in the memory hierarchy. The collector of Bacon et al. includes a concurrent cyclic garbage detector so that it also reclaims cycles, a potential weakness of reference counting. The maximum mutator pause reported by Bacon et al. was 2.6 ms. While we did not achieve a maximum pause that low, we believe that Sapphire has potential for maximum pauses an order of magnitude smaller (100–200 μs), as discussed with our experimental results.

8.2 Mark-Sweep

Concurrent mark-sweep collection has a venerable history, including what we believe to be the first concurrent GC algorithm published [Steele, 1975] (which also included compaction), and the classic paper on on-the-fly GC [Dijkstra *et al.*, 1978]. More recently, Doligez and Gonthier [Doligez & Gonthier, 1994] generalized Dijkstra's design, which supports only a single mutator and a single collector, to support multiple mutator processes, and to address a number of issues of practical performance import. Quite recently, there have been reports on several implementations of the Doligez-Gonthier algorithm within Java virtual machines. The first, DKP [Domani *et al.*, 2000a], does not report maximum pause times, being more concerned with whether concurrent collection will offer an overall reduction in elapsed time for (mostly) single-threaded benchmarks when run on a multiprocessor. The second, DKLS [Domani *et al.*, 2000b], does not support generations, pays some particular attention to weakly ordered memory accesses, and achieves maximum pause times in the range of tens to hundreds of milliseconds. The third, PD [Printezis & Detlefs, 2000], is described as "mostly concurrent", but achieves maximum pauses on the order of tens of milliseconds.

8.3 Copying

Concurrent copying collectors are most directly comparable to Sapphire in terms of algorithmic approach. They have roots in the Steele collector [Steele, 1975] and Baker's incremental copying algorithm [Baker, 1978]. Brooks [Brooks, 1984] did an early implementation that achieved atomic redirection to new copies by *always* following a pointer in the object header to find the current copy of an object. This level of indirection occurs on every access to an object field, and would likely have significant performance impact if run on modern processors. It amounts to a kind of read barrier.

A few years later, Appel, Ellis, and Li [Appel *et al.*, 1988] used the clever trick of adjusting virtual memory page protections to avoid executing extra mutator instructions for a read barrier—but the read barrier still exists and might significantly block mutator progress. This collector, like Baker's incremental algorithm, enforces an invariant that the mutator "sees" only *new* copies of copied objects (called the ToSpace invariant). It is the job of the read barrier to insure that any attempt to read an old copy results in redirection to the new copy.

O'Toole and Nettles [O'Toole & Nettles, 1993] observed that changing the invariant could produce a concurrent copying algorithm *with no read barrier*, called replicating collection. Their invariant is that the mutator "sees" only *old* copies of objects (the FromSpace invariant). Once the collector has copied as many reachable objects as it can, it stops the mutator briefly, to flip mutator root references to point to new copies. Clearly the

FromSpace invariant requires an atomic flip at some point.

There is an additional subtlety in replicating collection: the mutator can change previously copied objects. O'Toole and Nettles leveraged the existing store-list (a list of updates to existing objects, used for generational collection in Standard ML of New Jersey) to have the collector apply the updates to the new copies, and also copy any additional objects that may be found in the process. This latter step interferes with guaranteeing termination of collection or the absence of large pauses, since the mutator allocates new objects in FromSpace, and thus can keep generating object copying work for the collector. (The store list generates collector work as well.)

Blelloch and Cheng developed a variant of concurrent replicating collection with provably real-time behavior [Blelloch & Cheng, 1999]—a significant achievement in the realm of GC. While this variant is of theoretical interest, it has a number of practical limitations: (1) no provision of global variables or ordinary procedure call stacks (assumes a "stack in the heap model" such as in Standard ML of New Jersey); (2) no generations; and (3) no support for Java features such as synchronized objects (locking). In more recent work [Cheng & Blelloch, 2001], they reported on an implementation that dealt with these issues, either by sacrificing the real-time bounds or by doubling the space requirements for pointers. The issue is achieving an atomic flip for global, stack, and inter-generation variables; their approach is for each pointer to have two copies, with a single global bit determining which copy the mutator should use. This difficulty seems inherent in replicating collection's FromSpace invariant.

8.4 Sapphire

Our algorithm differs from concurrent copying and concurrent replication in that we impose neither the FromSpace nor the ToSpace invariant. We maintain both copies, synchronized loosely (adequate for non-volatile fields in Java), but more tightly than provided by the original replicating collector's use of a store-list as a queue of updates from the mutator to the collector. Maintaining both copies allows us to do flipping incrementally rather than as a single atomic act.

Our volatile field reads do involve a read barrier. Thus it is clear that we are exploiting Java's semantics to avoid most read barriers, since in other languages (such as C) one would have to treat all fields as volatile in the Java sense when working in a concurrent setting.[16] We do require that any data races in a program occur on *volatile*, not ordinary, fields. However, if this requirement is violated, the worst that happens is that a mutator may observe another mutator's field writes out of order. We break neither type safety nor correctness of garbage collection. We further note that there are emerging tools and coding disciplines that either prevent, or detect and allow one to reproduce and eliminate, harmful data races in programs.

[16] C and most languages simply do not define well what *should* happen. The Java Language Specification tried to define it, but at least initially got it wrong.

9 Prototype Implementation and Results

We did a demonstration implementation of Sapphire in the Intel Open Run-time Platform (ORP),[17] 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. *We emphasize that the primary purpose of our implementation is to show that Sapphire works, not to evaluate thoroughly its actual or potential performance.*

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.

9.1 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.

9.2 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 arrangement consists of: (1) a GC thread (built in to the system and not part of the benchmark code) that sleeps until a GC is requested *and* nursery allocation space is low; (2) a GC requesting thread (part of the benchmark) 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. Note that the allocating threads do little work and create huge volumes of garbage.

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.

We developed our own benchmark programs primarily because our purpose was to stress test Sapphire. Also, many existing benchmarks do not have very interesting allo-

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

cation and garbage collection behavior. For example, some standard benchmarks do very little allocation and can be run with *no* garbage collection on machines with commonly available quantities of memory. Some others allocate, but create little garbage. Finally, testing Sapphire with threads that allocate and create garbage at maximal rates gives a sense of the algorithm's behavior under those extremal conditions.

9.3 Pause Times

Since the primary goal of Sapphire is minimizing mutator thread pause times, that is the primary result we include here. Table 1 shows 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. Figure 18 presents 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.

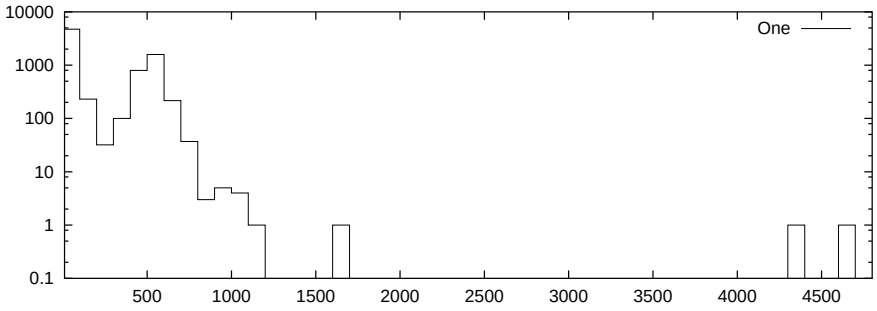
Program	Minimum	Maximum	Mean	50%ile	90%ile	95%ile	99%ile
One	66	4710	165	89	525	641	705
Deep	71	2321	169	89	529	643	713
Five	53	122 129	1865	118	567	3464	46 982

Table 1: Pause time statistics for the benchmarks (in μs)

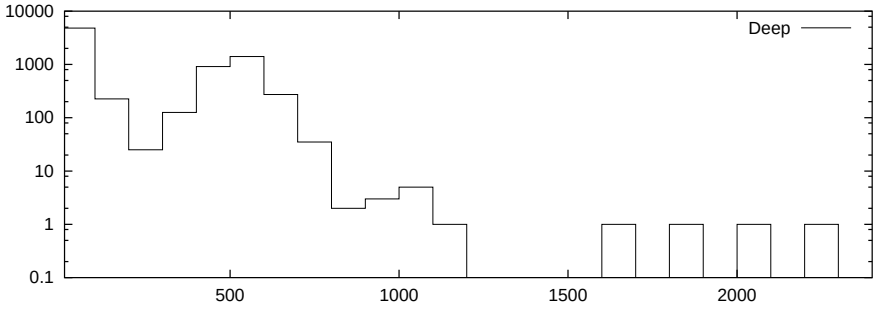
9.4 Additional Statistics

Table 2 shows the total time (in seconds) to run each benchmark program 5 times, with the stop-the-world collector (STW) and with Sapphire. 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 [Stichnoth *et al.*, 1999]. The table also indicates the number of pauses that found threads at unsafe points (these pauses are very short). 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*, orders of magnitude larger than typical Sapphire pauses.

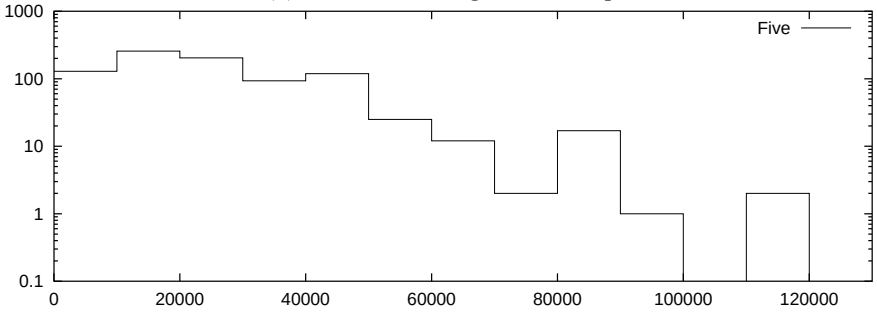
One interesting thing in these results is that the Sapphire collector is actually invoked *fewer times* than the STW collector. Both collectors are triggered when the heap is half full. The STW collector runs to completion before the mutators advance. In Sapphire, however, the mutators continue to run and as they do, they cause more objects to become garbage—some of which Sapphire can actually reclaim in the current invocation. Because Sapphire reclaims more space per invocation, it therefore is invoked less often.



(a) Pause time histogram for One



(b) Pause time histogram for Deep



(c) Pause time histogram for Five

Figure 18: Pause time histograms for the benchmarks

Program Name	Stop-the-World		Sapphire			
	Time (s)	GCs	Time (s)	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

Table 2: Additional statistics comparing Sapphire and Stop-the-World

9.5 Conclusion

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.

References

[Appel *et al.*, 1988] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices 23*, 7 (1988), 11–20.

[Bacon *et al.*, 2001] David Bacon, Dick Attanasio, Han Lee, and Stephen Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In [PLDI, 2001], pp. 92–103.

[Bacon *et al.*, 1998] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: Featherweight synchronization for Java. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Montreal, Quebec, June 1998), ACM Press, pp. 258–268.

[Baker, 1978] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM 21*, 4 (1978), 280–94. Also AI Laboratory Working Paper 139, 1977.

[Bekkers & Cohen, 1992] Yves Bekkers and Jacques Cohen, Eds. *International Workshop on Memory Management* (St. Malo, France, Sept. 1992), no. 637 in Lecture Notes in Computer Science, Springer-Verlag.

[Blelloch & Cheng, 1999] Guy E. Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation* (Atlanta, May 1999), ACM SIGPLAN Notices, ACM Press, pp. 104–117.

[Brooks, 1984] Rodney 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* (1984), ACM, pp. 256–262.

[Cheng & Blelloch, 2001] Perry Cheng and Guy Blelloch. A parallel, real-time garbage collector. In [PLDI, 2001], pp. 125–136.

[Cheng *et al.*, 1998] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Montreal, Quebec, June 1998), ACM Press, pp. 162–173.

[Dijkstra *et al.*, 1978] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM 21*, 11 (Nov. 1978), 965–975.

[Doligez & Gonthier, 1994] Damien Doligez and Georges 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).

[Domani *et al.*, 2000a] Tamar Domani, Elliot Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation* (Vancouver, June 2000), ACM SIGPLAN Notices, ACM Press, pp. 274–284.

[Domani *et al.*, 2000b] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Elliot E. Salant, Katherine Barabash, Itai Lahan, Erez Petrank, Igor Yanover, and Yossi Levanoni. Implementing an on-the-fly garbage collector for Java. In [Hosking, 2000].

[Hosking, 2000] Tony Hosking, Ed. *ISMM 2000 Proceedings of the Second International Symposium on Memory Management* (Minneapolis, MN, Oct. 2000), vol. 36(1) of *ACM SIGPLAN Notices*, ACM Press.

[Hudson & Moss, 1992] Richard L. Hudson and J. Eliot B. Moss. Incremental collection of mature objects. In [Bekkers & Cohen, 1992], pp. 388–403.

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

[Lamport, 1977] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM 20*, 11 (Nov. 1977), 806–811.

[Levanoni & Petrank, 2001] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. In *OOPSLA'01 ACM Conference on Object-Oriented Systems, Languages and Applications* (Tampa, FL, Oct. 2001), vol. 36(10) of *ACM SIGPLAN Notices*, ACM Press, pp. 367–380.

[Lindholm & Yellin, 1997] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[Manson & Pugh, 2001] Jeremy Manson and William Pugh. Semantics of multithreaded Java. Available as www.cs.umd.edu/~pugh/java/memoryModel/semantics.pdf Dec. 13 2001.

[Nettles *et al.*, 1992] Scott M. Nettles, James W. O'Toole, David Pierce, and Nicholas Haines. Replication-based incremental copying collection. In [Bekkers & Cohen, 1992].

[O'Toole & Nettles, 1993] James W. O'Toole and Scott M. Nettles. Concurrent replicating garbage collection. Technical Report MIT–LCS–TR–570 and CMU–CS–93–138, MIT and CMU., 1993. Also LFP94 and OOPSLA93 Workshop on Memory Management and Garbage Collection.

[PLDI, 2001] *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation* (Snowbird, Utah, June 2001), ACM SIGPLAN Notices, ACM Press.

[Printezis & Detlefs, 2000] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In [Hosking, 2000].

[Pugh, 1999] William Pugh. Fixing the Java memory model. In *ACM Java Grande Conference* (June 1999).

[Steele, 1975] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM 18*, 9 (Sept. 1975), 495–508.

[Stichnoth *et al.*, 1999] James M. Stichnoth, Guei-Yuan Lueh, and Michal Cierniak. Support for garbage collection at every instruction in a Java compiler. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation* (1999), pp. 118–127.