Relaxing Safely: Verified On-the-Fly Garbage Collection for x86-TSO

Peter Gammie
NICTA, Australia
peteg42@gmail.com

Antony L. Hosking
Purdue U, USA and NICTA, Australia
hosking@cs.purdue.edu

Kai Engelhardt
UNSW and NICTA, Australia
kaie@cse.unsw.edu.au

Abstract

We report on a machine-checked verification of safety for a state-of-the-art, on-the-fly, concurrent, mark-sweep garbage collector that is designed for multi-core architectures with weak memory consistency. The proof explicitly incorporates the relaxed memory semantics of x86 multiprocessors. To our knowledge, this is the first fully machine-checked proof of safety for such a garbage collector. We couch the proof in a framework that system implementers will find appealing, with the fundamental components of the system specified in a simple and intuitive programming language. The abstract model is detailed enough for its correspondence with an assembly language implementation to be straightforward.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection), Run-time environments; D.4.2 [*Operating Systems*]: Storage Management—Garbage collection; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification

General Terms Algorithms, Design Languages, Reliability, Verification

Keywords formal verification, machine-checked proof, relaxed memory, TSO

1. Introduction

Garbage collectors are now ubiquitous all the way from servers down to mobile and embedded devices, and the safety guarantees they provide for programmers are well understood. However, the trustworthiness of their implementations is less obvious, especially when they exploit concurrency for performance. Particularly worrisome is the fact that modern multi-cores have relaxed/weak memory models. This complicates reasoning which is already intricate even for sequentially-consistent memory.

State-of-the-art garbage collectors for multi-core systems have several design goals. They avoid blocking the progress of *mutator* (application) threads as much as possible. Assuming there is sufficient memory available to satisfy allocation requests, interactions between the application and the collector can generally be made *non-blocking*, given atomic hardware synchronization primitives that suffer at most only finite spurious failures, such as the atomic compare-and-swap (CAS) primitive of x86 or the load-linked/store-conditionally (LL/SC) primitives of ARM/PowerPC. Collection can operate *on-the-fly* (i.e., concurrently) with mutator threads, provided that a protocol for accessing references on the heap is respected. In other words, while the collector must periodically *handshake* with each mutator thread individually to poll for its *mutator roots*, it can do so without *stopping the world*: it has no need to suspend all mutators at the same time. With care the requisite communication can be made non-blocking and even wait-free, and the amount of work per collection cycle bounded.

Here, we present the machine-checked verification of safety for a realistic, on-the-fly, mark-sweep garbage collector that accounts for the relaxed memory model of modern x86 multi-cores. The mark-sweep collector kernel that we verify is realistic: it lies at the heart of the Schism real-time garbage collector [30], which offers superior real-time predictability with good throughput. We model the salient features of the collector algorithm, the x86-TSO memory model [35], and concurrent execution of mutator threads, including precise details of the write barriers that constitute the heap access protocol, and the handshakes that initiate and terminate the collection cycle. Importantly, safety is preserved even for applications that are not themselves data race free: the wait-free write barriers (which are themselves racy) ensure collector safety even in the face of application-level data races. The collector we model runs concurrently with mutator threads, but is not in itself parallel. Our model (and implementation) could, with some effort, be extended to a multi-threaded collector.

1.1 Contributions

Our contributions include:

1. The first (to our knowledge) machine-checked verification of safety for a realistic on-the-fly garbage collector against a relaxed memory model.
2. A comprehensive and unambiguous model that is accessible to system designers and implementers.

Our headline result, formally proved in the Isabelle/HOL proof assistant, asserts that the parallel composition of the garbage collector (GC) with any number of mutator threads (M_i) operating against an x86-TSO memory system (Sys) preserves safety:

$$\mathsf{GC} \parallel \mathsf{M}_1 \parallel \mathsf{M}_2 \ldots \parallel \mathsf{Sys} \models \Box(\forall r. \mathsf{reachable}\ r \rightarrow \mathsf{valid_ref}\ r)$$

There is always an object at every reference reachable from a mutator root.

The verification itself is made easier due to the careful design of the collector algorithm to meet real-time goals of predictability and performance. These impose tighter constraints than some other

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

PLDI'15, June 13–17, 2015, Portland, OR, USA
© 2015 ACM. 978-1-4503-3468-6/15/06...$15.00
http://dx.doi.org/10.1145/2737924.2738006

collector designs that are more relaxed about the latencies of both collector cycle and mutator barriers. For example, at any given time there is only ever one authoritative witness (either the collector itself or one of the mutators) to the liveness of a given object. Abstraction of the collector and its verification in Isabelle/HOL was performed *post hoc*, starting from the pre-existing Schism implementation, in a collaboration between run-time system and formal methods experts. Qualitatively, the run-time system experts gained greater confidence in the correctness of algorithm designs (and not just the particular design modeled here), while the formal methods experts relied on the semi-formal intuitions of the system experts when formulating the invariants that justify this result.

2. The Verified Collector

We verify an on-the-fly mark-sweep tracing collector, similar to that at the heart of the Schism real-time collector [30], which itself is based on the Doligez-Leroy-Gonthier (DLG) on-the-fly collector [7, 8]. Doligez and Gonthier [7] formalize a generic collector design that admits multiple implementation instances. However their work does not directly address real-time responsiveness, nor the details of the fine-grained synchronization between the collector and the mutators, nor the impact of relaxed memory on safety. Real-time goals mean adopting a collector design where both timeliness and predictability are paramount. While the Schism collector adopts these well-known design elements, it uses novel implementation techniques relying on hardware atomic instruction primitives to promote mutator wait-freedom for common mutator operations. We now briefly summarize the basic design elements of our verified collector before considering implementation details. Jones, Hosking, and Moss [15] provide a comprehensive survey of modern garbage collectors.

Mark-Sweep. Our verified collector is an instance of McCarthy's venerable *mark-sweep* scheme [22]: the collector traces reachable objects from some set of root references, generally including references held in mutator roots (i.e., thread stacks and registers) as well as global variables, *marking* all objects transitively reachable from the roots. Tracing proceeds by marking the roots, then scanning through the references held in known marked objects and marking their targets, and repeating these scan and mark steps until no more marked objects are discovered. Marking an object that was not previously marked can be thought of as advancing the *wavefront* of known reachable objects. Once this upper bound on the set of reachable objects has been determined, the collector performs a *sweep* of the heap to free allocated but unmarked objects. We eliminate the need to reset the flag on retained objects by making the interpretation of such marks contingent on a shared flag, which is inverted from one collector cycle to the next [18].

Concurrent. The verified collector is *concurrent* in that mutators can modify the heap while the collector is active. To preserve safety, concurrent collectors use mutator *write barriers*, compiled into the operations used to manipulate reference fields in the heap. One such write barrier is an *incremental update* (or *insertion* barrier [6]: whenever a mutator stores a reference into the heap it also makes sure that the target of that inserted reference is marked. This barrier addresses the following scenario: consider an unmarked object whose reference is stored by a mutator behind the collector wavefront and then all of its references are deleted ahead of the wavefront. Then the collector will never see the reference to mark its target, and mistakenly believe that the (reachable but unmarked) target can safely be freed. With the insertion barrier installed the target is marked at the time the reference was stored. Our implementation accumulates marked objects into thread-private *work-lists* which are later transferred to the collector for tracing.

Timeliness. It is also possible that a mutator loads a reference to an unmarked object that is ahead of the wavefront, and that this root becomes the sole witness to that object's reachability. If marking was to end without further ado then that unmarked but reachable object would be mistakenly freed. One solution to this is for the collector to rescan the mutators' roots before marking terminates. However, such references might hide long chains of unmarked objects, potentially prolonging the marking phase and hence its impact on mutator utilization. Our collector ensures the timely completion of the collection cycle by employing a *snapshot* (or *deletion*) barrier [1, 41], whereby mutators ensure that any references they overwrite target marked objects. Note that establishing exactly which reference is overwritten by a given mutator in the face of racy concurrent updates by other mutators is tricky for weak memory platforms. Thus, once the mutator roots have been sampled, the collector preserves all objects reachable from those roots at that *snapshot*, regardless of subsequent changes in reachability due to concurrent mutator activity. Objects that become unreachable after the snapshot will survive through the current collector cycle, but will be reclaimed at the next. These retained but unreachable objects are referred to as *floating garbage*.

Other sources of unmarked but reachable objects are allocations performed by the mutators after their roots have been scanned. Once again, to avoid rescanning the mutators before terminating marking, the collector adopts a simple solution: newly allocated objects are marked at creation [17].

On-the-Fly. If a snapshotting concurrent collector is to avoid rescanning the mutators before terminating the mark phase, it needs to obtain a coherent view of the mutator roots. The most straightforward way to achieve this is to stop *all* mutator threads before sampling their roots, and afterwards restarting the mutators and initiating concurrent marking. But this imposes relatively long and unpredictable pauses on mutators that degrades their real-time properties. *On-the-fly* collectors instead sample the roots of each mutator separately, and concurrently with other mutator threads. Indeed, on-the-fly collectors need not stop threads at all: the collector can asynchronously prompt each mutator to mark its own roots, and then wait for them all to respond. Similarly to DLG, our collector uses several rounds of *soft handshakes* with the mutators throughout the collection cycle, for several purposes: (a) ensuring that mutators have an up-to-date view of the collector control state (viz., the collector phase, the sense of the marks on the objects in the heap, and the sense of the mark to be used when allocating), (b) marking and querying the mutator roots, and (c) querying for objects marked by mutator write barriers. Mutators respond to soft handshakes only at well-defined *GC-safe points*, compiled into the application code at backward branches and call returns. Moreover, elemental mutator operations such as accessing references in the heap (where writes contain barriers), object allocation, and the handshake handlers themselves, are free of GC-safe points and therefore cannot be interrupted by collector requests. Only while the collector is performing a round of handshakes is the mutator-collector interaction *synchronous*, in which no interleaving of collector operations with mutator operations can occur. At all other times the collector and mutator operate *asynchronously*. However, mutators that have yet to respond to a given handshake still run concurrently with mutators that have already responded. In this respect, handshakes are said to be *ragged*.

For safety, on-the-fly snapshotting collectors *must* use an insertion barrier while the snapshot is being constructed. Otherwise, it is possible for a mutator to store a reference to an unmarked object into a marked (reachable) object behind the collector wavefront. While it is possible to turn this barrier off after sampling the mutator roots, our collector leaves it in place, to avoid complicating the write barrier code sequence with unnecessary conditional code. Moreover,

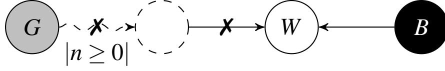


Figure 1. Grey protection. The dashed edge and circle imply that the white object W is reachable from grey object G via a chain of white references: $G \rightarrow_w^* W$. The weak tri-color invariant says that it is safe under this circumstance for a black object B to refer to W, since the collector will eventually trace and mark the chain of white objects from G. The deletion barrier preserves the weak tricolour invariant by greying the target of any edge (marked ✗) deleted in the chain.

an incremental update collector does not need the deletion barrier; but, as noted, omitting it implies repeated rescanning of the mutators until there are no unmarked roots, which unpredictably delays termination of marking. Thus, our heap mutation protocol includes both forms of write barrier.

2.1 Abstractions and Invariants

The global heap invariants preserved by the collector and the mutators (via handshakes and write barriers) are commonly framed in terms of Dijkstra's *tricolor* abstraction [6]. Briefly, tracing collection partitions the heap into *black* (presumed live) and *white* (possibly dead) objects. By analogy with object color, heap references can be said to take the color of their target object (e.g., a white reference is one that refers to a white object). When the collector cycle begins the entire heap is white. When a white object is encountered during tracing it is marked *grey* (reached, not yet processed). At some point the collector must process the *children* of every source grey object (i.e., the target objects to which the fields of the source object refer), marking each white child grey, whereupon the source object is shaded black (reached, processed). Thus, an object is black if the collector has finished processing it, and grey if the collector knows about it but has not yet processed it. The grey objects represent the collector's wavefront as it traces through the heap to find the reachable objects. When there are no more grey objects the trace is complete, and the remaining white objects can safely be freed. Grey objects represent outstanding tracing work that the collector is yet to perform.

Safety requires that the collector find and mark all white objects that are reachable from the roots. So long as each reachable white object is protected from deletion it will not be missed by the collector. It is thus sufficient to preserve the

weak tricolor invariant: any white object pointed to by a black object is always *grey-protected*: it is also reachable from some grey object via a chain of zero or more white objects.

Figure 1 illustrates this invariant, which the deletion barrier preserves by ensuring that no pointer (e.g., to a white object) can be deleted (e.g., in such a chain) without first making its target grey, and so ensuring that the remaining white objects in the chain remain grey-protected. Other than black (B, already seen) and grey (G, still to be processed) objects, only those white objects (W, as yet unseen) that are reachable via a white chain from grey objects will ever be seen by the collector. A mutator that deletes an edge in a chain of white objects can hide live objects from the collector. Here, the solid white object W is live, reachable both from a black object B and from a grey object G via a chain of white objects (dashed, and length $n \geq 0$ to indicate that the chain is possibly empty). A mutator that deletes any of the edges (marked ✗) in the white chain will prevent the collector from seeing the white object W even though it is live. The deletion barrier prevents this from happening.

Initially, (implicitly grey) roots protect *all* reachable objects. The deletion barrier preserves the weak tricolor invariant and so ensures safety. When tracing finishes there are no grey objects, so the invariant implies that there can be no reachable white objects, and it is safe to reclaim all white objects. In contrast, the insertion barrier preserves the

strong tricolor invariant: there are no pointers from black objects to white objects.

While the collector is processing a grey object to blacken it, and after it has been blackened, the insertion barrier prevents mutators from storing white references into it. Thus, the collector need never revisit black objects. When tracing finishes, the strong invariant says there are no references from black to white, so it is safe to reclaim all white objects. It trivially implies the weak invariant.

Our formal proof of safety relies on both the weak and strong invariants. In effect we treat the mutators as black once their roots have been scanned (their roots are never rescanned), and they are permitted to acquire and hold white references in their roots so long as these are grey-protected (weak tricolor). The alternative of using an insertion (read) barrier on mutators whenever they load a reference into their roots from the heap is typically too expensive since heap reads are common. Meanwhile the strong tricolor invariant applies to the heap.

2.2 The Collector

Figure 2 shows the collector in an informal pseudo-code that served as the basis for our formal model in Isabelle/HOL. The collector can be thought of as a non-terminating control loop, where each iteration of the loop performs a mark-sweep cycle. We omit scheduling decisions (i.e., when to trigger a collection). Three control variables shared between the collector and the mutators are subject to relaxed memory effects. Important invariants are indicated in comments. Each round of handshakes asks the mutators to perform some work asynchronously on behalf of the collector (indicated by the **`at`** *m* notation); some simply signal a change in control state, and hence the mutators need merely signal acknowledgement (**`nop`**).

The collector is idle to begin with, and from one collector cycle to the next, so the round of handshakes at lines 3–4 ensures that all mutators are aware that the collector is idle (`phase=Idle`). Moreover, at this point the entire heap is black, including newly allocated objects. While the collector is idle the mutator write barriers are disabled (see §2.3).

There are three rounds of handshakes from that point to enable marking. Each follows a change by the collector to one or more of its control variables. Lines 5–7: The sense of the marks on the objects is flipped by inverting the f_M flag, which changes the heap from black to white. Newly created objects continue to be allocated white. Lines 8–10: The phase variable is set to not idle (`Init`). When the collector phase is not idle the mutator write barriers are enabled. Lines 11–14: The sense of the marks to use for allocation of new objects is changed to black ($f_A \leftarrow f_M$). The only black objects are newly allocated objects, which cannot refer to white objects (`Black`$\not\rightarrow$`White`) as the write barriers are enabled. The only grey objects are those generated by the write barriers. At this point, with the write barriers enabled, and the sense of the marks established, the collector is ready to begin marking. It first uses a round of handshakes to have the mutators concurrently mark and accumulate their (grey) roots to their private work-lists W_m, before transferring them to the collector (lines 15–20). The use of the **`atomic`** keyword indicates that the transfer of the marked roots from each mutator's work-list W_m to the collector's shared work-list W occurs atomically. Importantly, these grey work-lists (each W_m and W) are all disjoint. Thus, in Schism each object header simply contains a field that points to the next element in the work-list. Moreover the atomic transfer of work-list W_m to the collector's W uses wait-free hardware primitives. Upon completion, the critical

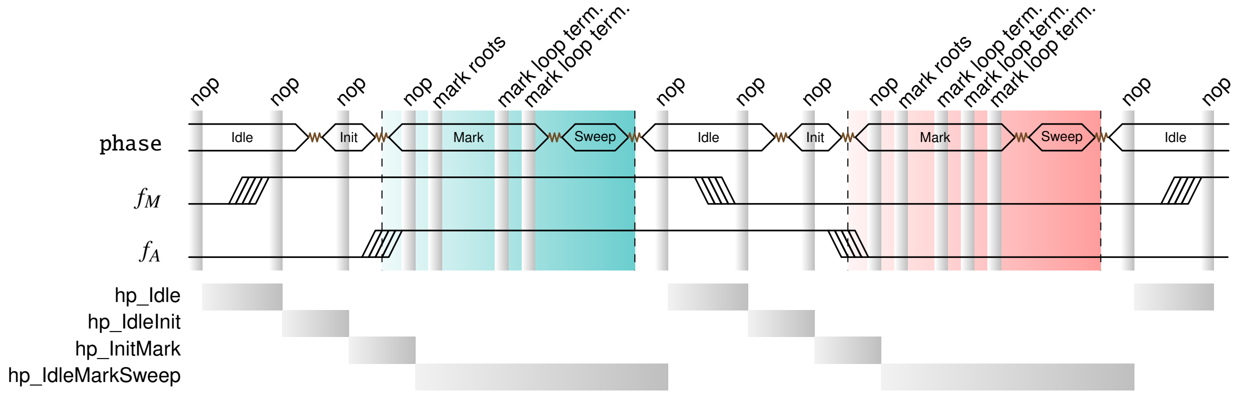


Figure 3. The mutators' view of control state transitions and handshakes. The handshakes are annotated with their types at the top of the diagram. At the bottom is *handshake phase* that the mutator is in, which we discuss in §3.2.

```
1  shared phase ← Idle, f_M ← f_A ← false, W ← ∅
2  loop
3      ∀m ∈ Mutators // Grey = ∅, White = ∅, heap = Black
4          handshake: at m nop
5      f_M ← not f_M
6      ∀m ∈ Mutators // Grey = ∅, Black = ∅, heap = White
7          handshake: at m nop
8      phase ← Init
9      ∀m ∈ Mutators // Black = ∅
10         handshake: at m nop
11     phase ← Mark
12     f_A ← f_M
13     ∀m ∈ Mutators // barriers installed, allocate Black
14         handshake: at m nop
15     ∀m ∈ Mutators
16         handshake: at m
17             // Mutator m marks and returns roots
18             for each ref ∈ roots_m
19                 mark(ref, W_m)
20             atomic W ← W ∪ W_m, W_m ← ∅
21             // Mutator m is now black; with strong_tricolor_inv
22             // establishes reachable_snapshot_inv for m's roots:
23             // reachable ⊆ Black ∪ Grey →*_w White
24     // Mark: loop invariant is reachable_snapshot_inv
25     while W ≠ ∅ // W ⊆ Grey
26         while W ≠ ∅
27             src ← r. r ∈ W
28             for each fld ∈ fields(src)
29                 mark(src.fld, W)
30             W ← W ∖ {src} // blacken: src ∈ Black
31         ∀m ∈ Mutators
32             handshake: at m
33                 // Mutator m reports its grey references
34                 atomic W ← W ∪ W_m, W_m ← ∅
35     // Sweep: Grey = ∅ ∧ reachable_snapshot_inv
36     //        ⇒ reachable ⊆ Black
37     phase ← Sweep
38     refs ← heap
39     while refs ≠ ∅
40         ref ← r. r ∈ refs
41         if flag(ref) ≠ f_M
42             // Free: ref ∈ White ∧ reachable_snapshot_inv
43             //       ⇒ ref ∉ reachable
44             atomic heap ← heap ∖ {ref}
45         refs ← refs ∖ {ref}
46     phase ← Idle
```

Figure 2. Pseudo-code for the collector. Mutators perform their side of soft handshakes as specified by the **at** statements.

invariant is that all objects reachable from the mutator roots are either black (newly allocated) or grey-protected (in the set of grey objects, or white and reachable via a chain of white references from some grey object: $\texttt{Grey}\rightarrow^*_w\texttt{White}$).

Iterative marking (scanning from grey to black) by the collector now proceeds until its private work-list is empty, after which it polls the mutators for their work-lists. This continues so long as there are outstanding grey references (as reported by the mutators). Subtly, entry to the marking loop only requires that if *any* mutator has a non-empty work-list W_m then at least one of them (not necessarily the same one) will report work to the collector. It is possible for a mutator to report no grey roots, before moving past the handshake and shading some objects (grey). It can only have done so if some other mutator also reports a grey root. Similarly, safe termination of the marking loop requires that if no single mutator reports a grey object from the round of handshakes at lines 31–34 then all mutators are free of grey references. Note that it is also possible for the collector to hold all the grey references during the mark loop (lines 23–30), and so this invariant only holds over the handshakes (lines 15–20 and 31–34).

Figure 3 illustrates the control state transitions of the collector and the handshakes that communicate these transitions to the mutators over two example collection cycles. The collector has three distinct active phases while it is not Idle: Init, Mark, and Mark-Sweep. It ensures communication of control state transitions via soft handshakes. The control variable f_M toggles during the Idle phase *before* the transition to Init, while f_A toggles *at* the transition from Init to Mark. The handshake to acquire mutator roots then takes the collector into the marking loop. At least one handshake occurs during the marking loop before it can terminate, and move on to the Sweep phase. Mutators may observe new control states even before the corresponding handshake due to TSO store buffer effects (as illustrated with ⟩w⟨, and ⫽⫽ or ⧹⧹), but all agree on the new state after the handshake round. In other words, at the completion of a round of handshakes the collector knows that each mutator knows the new control state, but—crucially—the mutators themselves remain uncertain about the epistemic state of the other mutators. This lack of common knowledge complicates the formal treatment of non-interference at these state transitions.

Figure 4 shows the anatomy of a handshake in the form of a sequence diagram. The component Sys referred to in the diagram represents both the hardware memory system to which both the collector and mutators perform memory loads and stores and the thread subsystem that implements the handshakes (implemented in Schism using pthread primitives). The collector begins a handshake

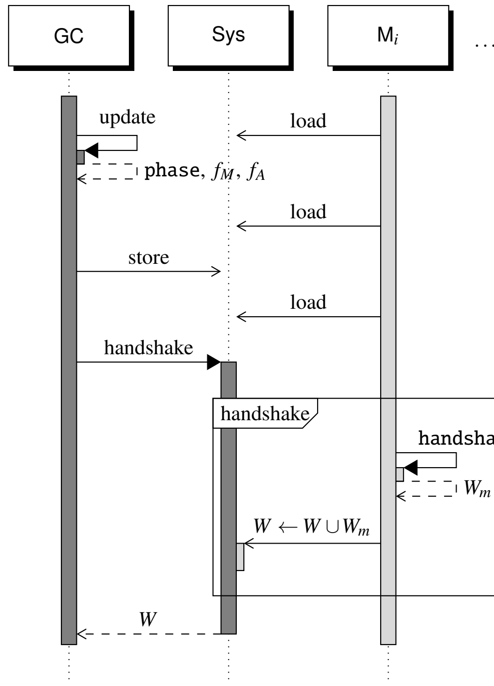


Figure 4. Anatomy of a handshake

by (optionally, depending on the handshake type) updating some subset of the shared control variables (f_A, f_M, or `phase`), before initiating a handshake at the system. Mutators may load stale values for these variables before they handshake. The system ensures that each mutator engages the handshake, in parallel, each (optionally, depending on the handshake type) computing private work sets W_m and then (atomically) merging those private work sets into the global work set W. The system (optionally, depending on the handshake type) returns the global work set W to the collector before the collector can continue. When not performing the handshake a mutator can continue to perform any other mutator operation.

2.3 Marking

Both the collector (in the marking loop) and the mutators (in their write barriers) race to mark and place grey objects atomically and exclusively on their private work-list (collector W or mutator W_m). The mark operation appears in Figure 5. The parameters to `mark` include both the reference to be marked, `ref`, and the thread-private work-list w. If an object does not have the expected mark according to the sense of f_M then the thread attempts to mark it using a *strong* atomic compare-and-swap (CAS). Implemented on x86 as a single locked `CMPXCHG` instruction, we spell out the explicit details of the atomic CAS operation in line with its formal model in x86-TSO: either the comparison at line 6 succeeds and the store at line 8 occurs, or the comparison fails only because some other thread has already succeeded in marking the object (and claiming success). All mutators involved in a race to mark a particular object witness the change from not marked to marked, whether or not they win the race. Only the winning thread places the now-marked object into its work-list (making it grey). Note that the CAS operation is attempted only if the flag yielded by the ordinary memory read is not as expected and if the collector is not idle. Thus, a hardware compare-and-swap instruction, which is typically expensive, occurs *only* when the collector is not idle and when the object of interest does not have the expected flag value. This serves to minimize the performance impact of write barriers on mutator throughput,

```
1  mark(ref, w):
2      expected ← not f_M
3      if flag(ref) = expected
4          if phase ≠ Idle
5              atomic // CAS
6                  if flag(ref) = expected // we win
7                      winner ← true
8                      flag(ref) ← f_M
9                      // ghost_honorary_grey ← ref
10                 else // some other thread won and marked
11                     winner ← false
12             if winner
13                 w ← w ∪ {ref}
14                 // ghost_honorary_grey ← null
```

Figure 5. Pseudo-code for marking

```
1  private
2      roots_m // some set of valid initial roots
3      W_m ← ∅
4
5  Load(src ∈ roots_m, fld ∈ fields(src)) :
6      roots_m ← roots_m ∪ {src.fld}
7
8  Store(dst ∈ roots_m, src ∈ roots_m, fld ∈ fields(src)):
9      mark(src.fld, W_m) // deletion barrier
10     mark(dst, W_m) // insertion barrier
11     src.fld ← dst
12
13 Alloc:
14     atomic
15         ref ← r. r ∉ heap
16         heap ← heap ∪ {ref}
17         flag(ref) ← f_A
18         roots_m ← roots_m ∪ {ref}
19
20 Discard(ref ∈ roots_m):
21     roots_m ← roots_m \ {ref}
```

Figure 6. Pseudo-code for mutator operations

particularly if hardware branch prediction assumes the condition at line 3 evaluates to false.

Some collectors avoid synchronization instructions for marking by representing the marks in a private bit-map on the side, or in a shared byte-map where byte stores by different marking threads can be performed atomically (on most hardware). In that case, the idempotency of marking can be exploited to avoid the need for synchronization, though at the cost of multiple markers possibly reporting the object as marked. However, if concurrently manipulated mark bits are stored in object headers alongside other concurrently manipulated meta-data state (such as for biased or thin locking), or in a shared bit-map, then synchronization is necessary. In our collector we do exploit idempotency to pay the cost of the CAS *only* when there is a race to mark. Once a marker has won the race, all other markers will observe the mark at line 3 and not even attempt the CAS.

Mutators. Each mutator has an arbitrary set of roots, $\texttt{roots}_m$ (i.e., local variables in its stack and registers). These can have non-empty intersection or be empty. In addition to the mutator side of the handshake operations shown in Figure 2, other mutator operations of interest to the collector appear in Figure 6. A mutator can: (a) load a reference from some field of a known root object; (b) store a known root reference to some field of a known root object, along with the appropriate write barriers; (c) allocate a new object in the heap, setting the mark on the object to f_A and add the new reference to its

roots; or (d) discard a reference from its roots. Note that the deletion barrier in `Store` does not load the deleted reference `src.fld` into the mutator's roots.

We do not consider potential interference with the collector by mutators other than those mentioned here; we assume type safety but not any kind of data-race freedom. Note also that we ignore other heap accesses that involve non-reference fields of objects.

2.4 Relaxed Memory: x86-TSO

Schism [30] has been implemented for both the relaxed memory x86 and weak memory ARM/Power architectures. Its correctness is presumed by informal reasoning about the memory models of those platforms. The only requirements are that there be a way to implement strong compare-and-swap (with implied store fence), and that handshakes include strong memory fences: a store fence when the collector initiates a round of handshakes, a load fence at each mutator when it accepts a handshake, a store fence when it finishes the handshake, and a load fence at the collector after all the handshakes complete. These are implicitly performed by the pthread primitives that implement the handshake mechanism. All other loads and stores execute without specific ordering constraints, including mutator loads of collector control variables. Of course, informal arguments do not a proof make, and while we are confident in the correctness of the collector, our purpose here is to establish a machine-checked formal proof of correctness. We do so for x86-TSO [35].

The x86-TSO model postulates the existence of a *store buffer* private to each hardware thread (though hardware need not implement TSO this way). Memory stores are placed in the buffer, which the hardware can asynchronously commit to the shared memory. The size of the buffer is unspecified. Memory loads first consult that thread's store buffer for the memory location to load. The most recent value stored to that location by the thread results, if present in the store buffer. Otherwise, the load consults the shared memory. In this respect, a given thread will see its own stores in the order it has emitted them. But, the stores of other hardware threads may appear arbitrarily interleaved among those local stores, though still in store order. *Locked* instructions like atomic compare-and-swap (x86 locked `CMPXCHG`) flush the store buffers, and also prevent loads from memory by all other hardware threads, ensuring that their update is visible to all the other hardware threads before completion. Similarly, memory fences (x86 `MFENCE`) flush the store buffer of the issuing thread, though each flushed store can still interleave with other non-local memory operations.

Even x86-TSO makes framing correctness and a proof of our collector non-trivial. Past formalizations of concurrent collectors assume sequential consistency (SC). While enforcing SC on x86-TSO is as simple as flushing the store buffers with an `MFENCE` instruction immediately after every store, doing so is expensive; for performance our collector flushes buffers only at handshakes and CAS. All other memory accesses are ordered only by the program text and TSO effects. Some accesses inherently have data races: heap updates by mutators are unordered excepting when the write barrier is triggered to enforce ordering (with CAS), depending on both the collector phase and the sense of the marks on the objects. Our intuitions of correctness rely on knowing that the marks placed on objects during tracing (by both collector and mutators) propagate to memory in a timely way, as enforced by both write barriers and handshakes. We discuss our refinements of the tricolor abstraction (§2) in the following sections.

It is important to note that, while the x86-TSO model has been extensively shown to be *observationally* adequate with respect to real Intel hardware as far as correctness goes [35], it is overly pessimistic about the efficiency of the microarchitectural realization. Thus it is worthless for predicting realtime behavior.

$$\frac{s' \in R\ s}{(\{\!|l|\!\}\ \textsc{LocalOp}\ R \cdot cs, s) \rightarrow_\tau (cs, s')}$$

$$\frac{(c_1 \cdot c_2 \cdot cs, s) \rightarrow_\alpha (cs', s')}{((c_1;;c_2) \cdot cs, s) \rightarrow_\alpha (cs', s')}$$

$$\frac{\alpha = act\ s \qquad s' \in val\ \beta\ s}{(\{\!|l|\!\}\ \textsc{Request}\ act\ val \cdot cs, s) \rightarrow_{«\alpha,\beta»} (cs, s')}$$

$$\frac{(s', \beta) \in act\ \alpha\ s}{(\{\!|l|\!\}\ \textsc{Response}\ act\ val \cdot cs, s) \rightarrow_{»\alpha,\beta«} (cs, s')}$$

Figure 7. An excerpt of CIMP process semantics $_ \rightarrow_\gamma _$, which is a relation between a pair of local states (a list of commands paired with the local data state for the process) and a communication action γ. The latter is one of τ (local computation), $«\alpha,\beta»$ (arising from a REQUEST), or $»\alpha,\beta«$ (from a RESPONSE). The second rule shows how we treat sequential composition using a frame stack.

$$\frac{s_p \rightarrow_\tau s'_p}{s \Rightarrow s(p := s'_p)}$$

$$\frac{s_p \rightarrow_{«\alpha,\beta»} s'_p \qquad s_q \rightarrow_{»\alpha,\beta«} s'_q \qquad p \neq q}{s \Rightarrow s(p := s'_p, q := s'_q)}$$

Figure 8. The CIMP system semantics $_ \Rightarrow _$ is a relation between system states (a function s that maps process names such as p to their local states s_p). The second rule captures rendezvous. The $_(_ := _)$ is Isabelle/HOL's syntax for function update.

3. Formal Verification in Isabelle/HOL

> None of it is new; but sensible old ideas need to be repeated or silly new ones will get all the attention.
>
> Leslie Lamport [20]

The model is the contract that joins the intuitions of the run-time system designers to the details required for formal verification. It must be expressed in a language that is plausible to both communities. To that end we developed the small imperative language CIMP that extends IMP [40] with process-algebra-style rendezvous (synchronous message passing) [25], control and data non-determinism, and (flat) parallel composition of processes. Several of its small-step semantic rules are shown in Figure 7, using frame stacks [29]. We derive an equivalent evaluation-context semantics [9] that supports the generation of verification conditions in terms of atomic actions.

Each process has local control and data states: there is no shared global state. All commands are prefixed with a label which is enclosed in $\{\!|_|\!\}$. Processes update their local data states using LOCALOP R, where R is a set-valued function of the process's local data state. Communication commands have a similar effect, and additionally require a pair of processes to synchronize. We employ the customary mix of deeply embedded commands and shallowly embedded expressions (over local data states).

Process transitions are interleaved only at the top-level, with no action hiding, as shown in the system semantics of Figure 8. This relation on global states (a map from process names to their local states) encodes the communication rendezvous. Intuitively, the sender's REQUEST command determines α as a function (act) of its

definition
mem-TSO :: (*'field*, *'mut*, *'ref*) *gc-com*
where
mem-TSO ≡
  {$\|$''*sys-read*''$\|$} RESPONSE (λ*req s*. { (*s*, *sys-read p mr s*)
      |*p mr*. *req* = (*p*, *ro-Read mr*) ∧ *not-blocked s p* })
  ⊔ {$\|$''*sys-write*''$\|$} RESPONSE (λ*req s*. { (*s*(| *mem-write-buffers* := (*mem-write-buffers s*)(*p* := *mem-write-buffers s p* @ [*w*]) |), *mv-Void*)
      |*p w*. *req* = (*p*, *ro-Write w*) })
  ⊔ {$\|$''*sys-mfence*''$\|$} RESPONSE (λ*req s*. { (*s*, *mv-Void*)
      |*p*. *req* = (*p*, *ro-MFENCE*) ∧ *mem-write-buffers s p* = [] })
  ⊔ {$\|$''*sys-lock*''$\|$} RESPONSE (λ*req s*. { (*s*(| *mem-lock* := *Some p* |), *mv-Void*)
      |*p*. *req* = (*p*, *ro-Lock*) ∧ *mem-lock s* = *None* })
  ⊔ {$\|$''*sys-unlock*''$\|$} RESPONSE (λ*req s*. { (*s*(| *mem-lock* := *None* |), *mv-Void*)
      |*p*. *req* = (*p*, *ro-Unlock*) ∧ *mem-lock s* = *Some p* ∧ *mem-write-buffers s p* = [] })
  ⊔ {$\|$''*sys-dequeue-write-buffer*''$\|$} LOCALOP (λ*s*. { (*do-write-action w s*)(| *mem-write-buffers* := (*mem-write-buffers s*)(*p* := *ws*) |)
      | *p w ws*. *mem-write-buffers s p* = *w* # *ws* ∧ *not-blocked s p* ∧ *p* ≠ *sys* })

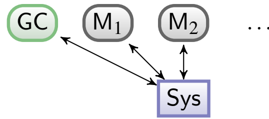
Figure 9. x86-TSO in CIMP, using the auxiliary functions defined by Sewell et al. [35]. The ⊔ operator denotes non-deterministic choice.

local data state, and the receiver's RESPONSE command determines β as a non-deterministic function of α and the latter's local data state. Both processes, upon rendezvous, simultaneously update their local states non-deterministically. This mechanism encodes our intuitions about causality and provides access to Isabelle/HOL's rich datatypes while avoiding the need for π-calculus-style binding or large non-deterministic sums [25]. Channels are not primitive, but can be modelled by a judicious choice of α.

A key strength of this approach is that the atomicity of distinct operations is independent, which is supremely useful during the initial phases of such projects, while leaving open the possibility of atomicity refinement within the same framework.

3.1 Model

We model the software threads as CIMP processes which talk to a single reactive system thread:



The system abstracts and encapsulates the TSO model, allocation, and the synchronization structure of the handshakes, as we discuss in the next section. The variables that the run-time system designers consider to be global reside here. The local states of the software components abstractly represent the program counters, the registers, and the stacks that are thread-local. These include each thread's local copies of the garbage collector's control state that are updated as the pseudo-code suggests. The number of mutators is arbitrary.

Our complete model is available at the Archive of Formal Proof.[1] We focus on the novelties of each component here.

System. We begin with an adaptation of the x86-TSO memory model due to Sewell et al. [35] to our setting. Let us fix an arbitrary non-empty set $\mathscr{R}$ of *references*, and treat the heap in the time-honored manner as a a partial map from $\mathscr{R}$ to objects $\mathscr{O}$ or `NULL`. An *object* $o \in \mathscr{O}$ consists of a garbage collection mark and a partial mapping of fields to $\mathscr{R} \cup \{$`NULL`$\}$. We abstract from any non-reference payloads that an object may have. We use the domain of the heap to track free references. Allocation is treated as an atomic action that creates and initializes an object which is inserted into the heap at an arbitrary free reference. This is the coarsest and least defensible abstraction in the present model.

We encode the transition rules given by Sewell et al. as shown in Figure 9, which diverges only by eliding their fine-grained treatment of x86 registers. The TSO store buffers for each thread are represented as a sequence of pending write actions in the system component, and the TSO lock records the name of the process holding it, if there is one. The system component has only one internal transition, which is to commit the oldest pending write for any thread. We make all of the garbage collector's control variables (f_A, f_M, `phase`) subject to TSO, as well as all operations on objects (including per-object mark bit flags). We exploit CIMP's non-standard handshaking here so that memory requests involve only a single communication step despite requiring bi-directional communication.

The system includes a very straightforward treatment of handshakes: the collector sets the handshake type, issues an `MFENCE`, and, for each mutator m in an arbitrary order, issues a request to the system that sets the bit for m. Each mutator polls its bit at its leisure, and resets it after issuing an `MFENCE` and completing the requested work. The process of completing a handshake includes transferring its work-list to the system for root marking and mark loop termination. Concurrently the collector polls the mutator bits, blocking until all have been reset, and finally loads the system's work-list into its local state.

We ignore the effects of TSO on the handshake state as it has no interesting impact on collector correctness at our level of abstraction, and should be straightforward to resolve during a later atomicity refinement step. Similarly, work-lists are not subject to TSO; we prove that these are disjoint and hence thread-local, which justifies the representation used in the C code discussed in §2.2.

Collector. One would be surprised if the collector were difficult to represent in CIMP given that the language was designed with it in mind, and indeed our formalization of the pseudo-code presented in Figure 2 is barely worth remarking on other than to observe its similarity; we present the mark loop in Figure 10 just for color. Taking inspiration from the public APIs of message-passing μkernels like seL4 [16], we encapsulate the system REQUEST operations so that, after some hefty syntactic sugaring, the resulting model looks very much like an ordinary shared-variable program, albeit one that is parameterized by the semantics of memory.

Mutators. These are less exciting than the collector, being merely a maximally non-deterministic choice amongst the operations mentioned in Figure 6, an `MFENCE`, a local step (notionally refined by any terminating thread-local computation), and the mutator's side of the handshakes tied to collector phase transitions shown in Figure 2. We expect every client of the collector to be a formal refinement of this process, which implies nothing beyond their respect for the heap access protocol for references. The mutator process is parameterized by name, and our safety proof is for an unbounded number.

[1] `http://afp.sf.net/entries/ConcurrentGC.shtml`

```
...
{|...|} WHILE not empty W DO
 {|...|} WHILE not empty W DO
  {|...|} ´tmp-ref :∈ ´W ;;
  {|...|} ´field-set := UNIV ;;
  {|...|} WHILE not empty field-set DO
   {|...|} ´field :∈ ´field-set ;;
   {|...|} ´ref := ´tmp-ref→´field ;;
   {|...|} mark-object ;;
   {|...|} ´field-set := (´field-set − {´field})
  OD ;;
  {|...|} ´W := (´W − {´tmp-ref})
 OD ;;
 {|...|} ragged-safepoint-get-work
OD ;;
...
```

Figure 10. An excerpt of the collector model: the marking loop. The labels on commands have been elided for reasons of space.

3.2 Formal Abstractions and Invariants

> *You can't always write a chord ugly enough to say what you want to say, so sometimes you have to rely on a giraffe filled with whipped cream.*
>
> Frank Zappa

Our verification technique is classical: we develop a single global invariant that holds at all reachable states, following Lamport [19]; see de Roever et al. [5] for an extended account, and Ridge [31] for a comparable development. In practice this statement is a conjunction of *local* assertions, which use the `at` p ℓ predicate to assert that a property holds when control for process p resides at program location ℓ, and *universal* assertions that do not. Both are predicates over the instantaneous global state (i.e., a map from each process's name to its current local state). This approach separates code, assertions, and proofs, which is advantageous for large-scale verification efforts as exemplified by the l4.verified project [16].

We show that the model satisfies this invariant by induction over the set of reachable states induced by the $_ \Rightarrow _$ relation. A custom tactic automatically discharges most verification conditions, exploiting Isabelle's efficient and scalable parallelism to greatly reduce latency while discovering invariants and proofs [39].

As the high-level safety argument has already been presented informally (§2), and our full formal development is publicly available, we proceed here by linking the two: we refine the definitions that underpin the former account, and recount some of the corner cases uncovered by the latter.

Handshakes. We begin by formalizing the phase behavior of the system that is illustrated in Figures 3 and 4. Intuitively, we construct a system-wide program counter that limits the combinatorial possibilities for potentially interfering operations. This takes the form of a tight relation between the most-recent handshake type (one of: noop, mark roots, mark loop termination), the pending-handshake bits of the mutators and ghost state that tracks how many handshakes the collector has initiated and how many each mutator has completed. This ghost state makes it easy to state the invariants about the complex write barriers that run asynchronously to the collector. The handshake type allows us to distinguish between the mark loop termination handshake, which can happen zero or more times, and the handshake that signals the beginning of the next collector cycle.

Each mutator knows that its fellow mutators are in one of the preceding, the same, or the next phase, and that the collector is in the same phase or in one of the two adjacent handshakes.

Coarse TSO Invariants. Our model contains genuine data races, if only because we do not assume that the mutators are data-race free; we discuss others in the following sections. The effect of TSO on the two control variables f_A and f_M is benign as these are only written by the collector, which immediately issues an `MFENCE` as part of the following handshake. In contrast the `phase` variable is not data-race free: there can be several writes pending to it in the collector's write buffer. Therefore we use the handshake state and the state of the collector's TSO store buffer to express our expectations of these variables as shown in Figure 3. We also track the state of the TSO lock with respect to program locations in the various threads.

Collector Predicates and Invariants. Our next step in refining the informal intuitions of correctness is to find an inductive invariant that implies our correctness assertion. To do so we need to account for the effects of TSO on reachability: can a path go via TSO buffers? Note that a pending write may be the only witness to the reachability of some object from some other object. As observed earlier, the write barriers take care of inserting greys as necessary to keep white-reachable objects grey-reachable, and these greys are immediately published by the CAS, which also has the effect of flushing any pending mutations. Therefore we treat references in TSO store buffers as extra roots, and otherwise ignore their effect on paths; a path always goes via the heap.

For similar reasons we also consider the reference marked by the deletion barrier for the duration of the marking operation to be a root (line 9 in Figure 6).

Formally, a reference reaches another if there is a path from the former to the latter through objects on the heap. A reachable reference y is one for which there exists a root that reaches y. Our valid_refs_inv states that there is an object on the heap for all reachable references.

Another subtle point is the interpretation of the colors that underpin the tricolor abstractions of §2.1. Fine-grained modelling of TSO and CAS means that the marking operation cannot atomically take an object from white to grey: (1) the mark may reside in a TSO store buffer until the TSO lock is released by the CAS, and (2) only after the CAS has been won is the reference placed on a work-list. In terms of actual program state, our scheme blackens a white object, and then reverts it to grey. We use ghost state (ghost_honorary_grey) to track this last transition (lines 8–13).

We therefore use the following interpretation of colors: an object is *white* if it is not marked on the heap, *grey* if it is on a work-list or ghost_honorary_grey, and *black* if it is marked on the heap and not on a work-list. These colours overlap: black is certainly disjoint from white and grey, but a reference is both white and grey (ghost_honorary_grey) during the CAS (at line 8, after the write has been issued but not yet committed), and would be considered black at line 12, before it is added to a work-list, if not for the ghost state. With these subtleties addressed, these definitions directly support the classical definitions of the strong and weak tricolor invariants given earlier.

Our valid_W_inv says that if a reference is on a work-list or ghost_honorary_grey for thread p, and the TSO lock is not held by p, then its target object is marked on the heap. It also states that any pending marks use f_M, and the work-lists are disjoint. This captures the abstract effect of the four uses of the marking function shown in Figure 5, and effectively allows us to use colors as in the informal argument.

Marking. The uses of mark (Figure 5) by the collector and in the root marking operation by the mutators are straightforward to verify as the collector's control state is common knowledge at those times. In contrast the two uses in the mutators' store operation run completely asynchronously, and the use of `phase` is not data-race free since the writes that change it from mark to sweep (line 36 in Figure 2), and sweep to idle (line 46) are unsynchronized. Further complicating our reasoning, `phase` can change after we check it

(line 4 in Figure 5), and f_M may flip after the load (line 2). Moreover the reference that is marked by the deletion barrier (line 9 in Figure 6) may not be the reference that presently resides in `src.fld` due to interference by another mutator.

These scenarios are relatively straightforward during the collector's steady-state mark loop. We therefore focus on the initialization phases where the reasoning is more complicated.

Initialization. The goal of initialization is to establish the snapshot before entering the mark loop. Our reachable_snapshot_inv asserts that all reachable references are in the snapshot. Formally the snapshot contains all black and grey-protected white (i.e., grey or white-reachable from a grey) references. The invariant is established by each mutator as it completes the root marking handshake, and depends on all mutators having both barriers installed. Once a mutator has established the reachable_snapshot_inv we consider it black; that is, its roots won't be scanned again on this collection cycle.

This is where we cash in our very precise handshake relation. With reference to the handshake phases shown along the bottom of Figure 3, our sys_phase_inv says that when the collector is in the given phase (i.e., it has initiated the handshake named here but not yet the next one) then the assertion holds:

hp_Idle: If f_A equals f_M then the heap is black otherwise it is white, and there are no grey references.

hp_IdleInit: There are no black references.

hp_InitMark: Until the write to f_A is committed, there are no black references; that is, the mutators allocate white until then. Intuitively, to preserve the strong_tricolor_inv, we must know that all mutators have installed their insertion barriers before setting the allocation flag f_A to f_M, and we also want to set f_A as late as possible to ameliorate floating garbage.

Before recounting the invariants for the mutators, we define the effect of the write barriers. A reference is an *insertion* if it is a reference being written into an object by a write pending in a TSO store buffer. The predicate marked_insertions asserts that all such insertions are marked. Similarly a *deletion* is the reference in an object that will be overwritten by a write pending in a TSO store buffer. The predicate marked_deletions asserts that all such deletions are marked.

Our mutator_phase_inv asserts that, for mutators in the given handshake phase, the assertion holds:

hp_IdleInit: There are no black references.

hp_InitMark: Only marked_insertions holds. Note that a mutator m that has yet to pass this handshake can defeat the deletion barrier of a mutator m' which has passed the handshake by inserting white references into objects. The scenario is as follows: the insertion barrier in m reads `phase=Idle` and does not mark the reference, then the `Store` operation in m' runs to completion, and finally the white insertion by m is committed. Conversely, all mutators that pass this handshake have flushed any white insertions from their TSO store buffers.

hp_IdleMarkSweep: By the commencement of the mark-roots handshake, both marked_insertions and marked_deletions hold for all mutators, and reachable_snapshot_inv holds for all mutators that have completed this handshake. The strong_tricolor_inv guarantees to the black mutators which have marked their roots that white-reachable objects are in fact grey-protected; the only black objects at this point arise from allocation (only the collector takes grey to black, and then only in the mark loop), and the insertion barrier ensures that these cannot be the sole witness to the reachability of white objects.

The local invariants for the mutators' asynchronous mark operations in the `Store` operation that justify these assertions are the most intricate in the entire development.

Termination of Marking. We argue that if the collector's work-list is empty at line 25 in Figure 2 then there are no grey references. The gc_W_empty_mut_inv predicate asserts that, if mutator m has completed the root marking or mark loop termination handshake, the collector's work-list W is empty and m has a non-empty W, then there is a mutator with a non-empty work-list that has yet to complete the handshake. In concert with our other invariants, this implies that the collector's W must become non-empty before it completes the handshake. Note that it is possible for the collector to own all the grey references in the system during the mark loop, and so this predicate is only invariant over those handshakes, when the collector's W is known to start empty.

With no grey references in the system, reachable_snapshot_inv reduces to the assertion that all reachable references are black, and it is therefore safe to free any objects that remain white.

Satisfiability of the Invariants. Finally, the formal development exhibits a small but non-trivial concrete heap that discharges the remaining hypotheses of our model, and further that our invariants are satisfiable. This provides assurance of non-triviality.

4. Concluding Remarks

The goal of this work was to verify a plausible model of the extant garbage collector. Its value is in exhaustively establishing the safety of the synchronization and marking schemes; we conclude that the run-time system experts had the right intuitions (sound and formalizable) and ultimately enough of them. But of course our development is neither the first nor the final word.

Previous Verification Efforts. Our development traces its roots to the classic work of Dijkstra et al. [6], which originated the fundamental tricolor invariants. Formalization of on-the-fly collection became much clearer with the work of Doligez and Gonthier [7]. Both of these assume that memory is sequentially consistent, and do not address the implementation of realistic write barriers and handshakes. Moreover both formulations proved to be unsafe in subtle ways that were only revealed by more formal analyses [11, 12], illustrating the difficulties of devising and reasoning about these algorithms. We note in passing that Doligez and Gonthier [7] used TLA [19], which yields a looser model than ours. For instance, we unnecessarily fix the order of the insertion and deletion barriers in the mutators where they do not.

Hawblitzel and Petrank [13] offer perhaps the most comprehensive verification of a real garbage collector down to assembly code, though their collector is not concurrent. Vechev et al. [38] explore correctness-preserving transformations that allow synthesis of a diverse range of more realistic collectors (less expensive, more concurrent), from an apex abstract collector that is simpler, and easier to prove correct. While interesting as a means to enumerating possible designs, this approach does not yield a collector with the fine-grained concurrency and mutator responsiveness of our collector. A similar approach has been attempted by Pavlovic et al. [28], again without descending to our level of fine-grained detail.

Park and Dill [27] developed a model checker for the SPARC v9 relaxed memory order (RMO) model and applied it to a simple spin lock. Das et al. [4] considered verification of a cache coherence protocol and concurrent garbage collection using predicate abstraction. Perplexingly they did not combine these two works.

Much recent work [10, 21, 23, 24, 36] treats garbage collection as an exercise in separation logic. All of these are sequential; concurrent separation logic remains a work in progress [37].

Representations. We assume that each thread runs on a separate core; i.e., each has its own TSO buffer. This is the most adversarial setup because any reasonable scheduler will flush TSO buffers on context switches; as our assertions are already (necessarily) closed under TSO flushes, schedulers should not introduce any new interference.

More importantly we did not make all of the system state subject to TSO: in particular, the work-lists and (less crucially) the precise details of the implementation of handshakes. We have kept objects and references abstract. We leave these issues to an atomicity-refinement technique.

Least defensibly we have essentially axiomatized allocation as a global and atomic operation, which is far from realistic. Similarly to Doligez and Gonthier [7], we have devised but not yet verified an extension to the model that would allow mutators to gather pools of unallocated references from which to perform fine-grained allocation without synchronizing. For TSO, we can also perform the marking and initialization of the fields at each allocation without the need for an `MFENCE`, because publishing the new reference to other mutators can occur only after the prior initializing stores have been flushed. We have yet to consider process spawning and reaping.

Connection With Reality. Clearly our safety property does not imply every property one may need in a larger setting. For instance, while we have established that the collector does not free live objects, we have not shown that it does not mutilate the heap. (This is obvious for our model as the GC never writes to references.) Moreover a real-time collector like Schism deserves to have analytic timing bounds, but as we have observed, the x86-TSO model is worthless for this purpose. We know that garbage is collected within two cycles of the collector's outer loop, up to liveness of the mutators and hardware, but again we owe this a proof. We concur with Gonthier [11] that a formal treatment of liveness is likely to be complex and adds little value, given the need for dubious fairness hypotheses.

The connection of models such as ours to more executable things has been a major theme for the group we work in [16], and for others. Therefore we are interested in refinement techniques that will take us closer to the C or x86-TSO assembler implementation. The recent work on CompCert-TSO by Sevcík et al. [34] appears to provide a solid foundation for such a development, in concert with the recent work by Jagannathan et al. [14] that would allow us to modularly axiomatize our special operations (handshakes, barriers, alloc, free) at the source-code level, and discharge these assumptions over small pieces of x86 assembler. The details will doubtless be devilish.

Heading in the opposite direction, one might wish for a more compositional story that can be used to show the safety of a larger system. We did not use such program logics (featuring, for instance, ghost state, separation, or ownership) in our present development as they did not appear to help discover invariants; instead we developed *ad hoc* encodings of the concepts we found useful, as did l4.verified [16]. For instance our process-local states are the obvious static separation between thread-local states, the ownership of grey references is explicit in the form of work-lists, and ghost state takes the form of program variables from which a read never occurs. We did not attempt to address compositionality, but note that no such techniques known to us would have reduced our proof effort.

Interestingly, the recent literature does not appear to address the temporal phase structure common in concurrent systems, and indeed our development suffers somewhat from a combinatorial explosion in interference possibilities due to the raggedness of the handshakes. The *communication-closed layers* approach recounted by de Roever et al. [5, Chapter 12] looked promising but did not deliver for us.

Finally we could not avail ourselves of the battery of extant theorems that reduce reasoning about data-race free (DRF) programs on TSO to reasoning about sequentially-consistent memory [3, 26, 32], if only because the collector does not have the luxury of assuming that mutators are DRF. An alternative approach is to transform the model to expose data races to a standard technique for sequential consistency [2]. This may be appealing to those using algorithmic state-traversal techniques but it is less clear how it plays in a deductive setting.

Observations. From our close analysis of this algorithm we know that two of the initialization handshakes can be removed on x86-TSO, but have yet to prove this. We also expect that the insertion barrier can be removed after roots have been marked (i.e., across the mark loop) in exchange for an extra branch in the store barrier. We thank a reviewer for suggesting that this may be more performant.

Further we expect that a verification of safety for ARM/POWER can be carried out along similar lines, despite the complexity of that memory model [33]. What has stopped us so far is the concern that extant models will not have the longevity of x86-TSO.

We have demonstrated here that reasoning about programs with data races on TSO is somewhat tractable, but that may be because the collector is carefully constructed. To scale one certainly wants to exploit the general absence of data races. We encourage other researchers to bring their techniques to bear on collectors of this sophistication, and hope they find our invariants useful.

Acknowledgments

This work was carried out at the Neville Roach Laboratory in Sydney, Australia, while the first two authors were at NICTA. We thank our NICTA colleagues, the anonymous PLDI paper and artifact reviewers, and Shaz Qadeer for their valuable feedback. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program. The second author also receives support from Qualcomm and the National Science Foundation under grants nos. CNS-1161237 and CCF-1408896.

References

[1] S. Abraham and J. Patel. Parallel garbage collection on a virtual memory system. In *International Conference on Parallel Processing*, pages 243–246, University Park, Pennsylvania, Aug. 1987.

[2] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 512–532, Rome, Italy, Mar. 2013. Springer. doi: 10.1007/978-3-642-37036-6_28.

[3] E. Cohen and B. Schirmer. From total store order to sequential consistency: A practical reduction theorem. In *International Conference on Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 403–418, Edinburgh, Scotland, July 2010. Springer. doi: 10.1007/978-3-642-14052-5_28.

[4] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *International Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 160–171, Trento, Italy, July 1999. Springer. doi: 10.1007/3-540-48683-6_16.

[5] W. P. de Roever, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001.

[6] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM*, 21(11):966–975, Nov. 1978. doi: 10.1145/359642.359655.

[7] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 70–83, Portland, Oregon, Jan. 1994. doi: 10.1145/174675.174673.

[8] D. Doligez and X. Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 113–123, Charleston, South Carolina, Jan. 1993. doi: 10.1145/158511.158611.

[9] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2): 235–271, Sept. 1992. doi: 10.1016/0304-3975(92)90014-7.

[10] X. Feng. Local rely-guarantee reasoning. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 315–327, Savannah, Georgia, Jan. 2009. doi: 10.1145/1480881.1480922.

[11] G. Gonthier. Verifying the safety of a practical concurrent garbage collector. In *International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 462–465, New Brunswick, New Jerseu, July–Aug. 1996. Springer. doi: 10.1007/3-540-61474-5_103.

[12] D. Gries. An exercise in proving parallel programs correct. *Commun. ACM*, 20(12):921–930, Dec. 1977. doi: 10.1145/359897.359903.

[13] C. Hawblitzel and E. Petrank. Automated verification of practical garbage collectors. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 441–453, Savannah, GA, Jan. 2009. doi: 10.1145/1480881.1480935.

[14] S. Jagannathan, V. Laporte, G. Petri, D. Pichardie, and J. Vitek. Atomicity refinement for verified compilation. *ACM Trans. Prog. Lang. Syst.*, 36(2):6:1–30, Apr. 2014. doi: 10.1145/2601339.

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

[16] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *ACM SIGOPS Symposium on Operating Systems Principles*, pages 207–220, Big Sky Resort, Montana, Oct. 2009. doi: 10.1145/1629575.1629596.

[17] H. T. Kung and S. W. Song. An efficient parallel garbage collection system and its correctness proof. In *Symposium on Foundations of Computer Science*, pages 120–131, Providence, Rhode Island, Oct. 1977. IEEE. doi: 10.1109/SFCS.1977.5.

[18] L. Lamport. Garbage collection with multiple processes: an exercise in parallelism. In *International Conference on Parallel Processing*, pages 50–54, 1976.

[19] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[20] L. Lamport. Who builds a house without drawing blueprints? *Commun. ACM*, 58(4):38–41, Apr. 2015. doi: 10.1145/2736348.

[21] C. Lin, Y. Chen, and B. Hua. Verification of an incremental garbage collector in Hoare-style logic. *International Journal of Software and Informatics*, 3(1):67–88, Mar. 2009.

[22] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM*, 3(4):184–195, Apr. 1960. doi: 10.1145/367177.367199.

[23] A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 468–479, San Diego, California, June 2007. doi: 10.1145/1250734.1250788.

[24] A. McCreight, T. Chevalier, and A. Tolmach. A certified framework for compiling and executing garbage-collected languages. In *ACM SIGPLAN International Conference on Functional Programming*, pages 273–284, Baltimore, Maryland, Sept. 2010. doi: 10.1145/1863543.1863584.

[25] R. Milner. *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[26] S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *European Conference on Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 478–503, Maribor, Slovenia, June 2010. Springer. doi: 10.1007/978-3-642-14107-2_23.

[27] S. Park and D. L. Dill. An executable specification and verifier for relaxed memory order. *IEEE Transactions on Computers*, 48(2):227–235, 1999. doi: 10.1109/12.752664.

[28] D. Pavlovic, P. Pepper, and D. R. Smith. Formal derivation of concurrent garbage collectors. In *International Conference on Mathematics of Program Construction*, volume 6120, pages 353–376, Québec City, Canada, June 2010. Springer. doi: 10.1007/978-3-642-13321-3_20.

[29] A. M. Pitts. Operational semantics and program equivalence. In *International Summer School on Applied Semantics, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*, pages 378–412. Springer, Caminha, Portugal, Sept. 2000. doi: 10.1007/3-540-45699-6_8.

[30] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: Fragmentation-tolerant real-time garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–159, Toronto, Canada, June 2010. doi: 10.1145/1806596.1806615.

[31] T. Ridge. Verifying distributed systems: the operational approach. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 429–440, Savannah, Georgia, Jan. 2009. doi: 10.1145/1480881.1480934.

[32] T. Ridge. A rely-guarantee proof system for x86-TSO. In *International Conference on Verified Software: Theories, Tools, Experiments*, volume 6217 of *Lecture Notes in Computer Science*, pages 55–70, Edinburgh, Scotland, Aug. 2010. Springer. doi: 10.1007/978-3-642-15057-9_4.

[33] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 175–186, San Jose, California, June 2011. doi: 10.1145/1993498.1993520.

[34] J. Sevcík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22, 2013. doi: 10.1145/2487241.2487248.

[35] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010. doi: 10.1145/1785414.1785443.

[36] N. Torp-Smith, L. Birkedal, and J. C. Reynolds. Local reasoning about a copying garbage collector. *ACM Trans. Prog. Lang. Syst.*, 30(4), July 2008. doi: 10.1145/1377492.1377499.

[37] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 691–707, Portland, Oregon, Oct. 2014. doi: 10.1145/2660193.2660243.

[38] M. T. Vechev, D. F. Bacon, P. Cheng, and D. Grove. Derivation and evaluation of concurrent collectors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 341–353, Ottawa, Canada, June 2007. doi: 10.1145/1133981.1134022.

[39] M. Wenzel. Shared-memory multiprocessing for interactive theorem proving. In *International Conference on Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 418–434, Rennes, France, July 2013. Springer. doi: 10.1007/978-3-642-39634-2_30.

[40] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1993.

[41] T. Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, Mar. 1990. doi: 10.1016/0164-1212(90)90084-Y.