Model Checking Transactional Sapphire

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

Richard Jones
University of Kent
r.e.jones@kent.ac.uk

March 15, 2018

Introduction

This report describes how we verified major aspects of the *Transactional Sapphire* garbage collector for Java. Sapphire uses *replication-based copying*. The heap is divided into two semispaces, and the collector constructs a replica in *tospace* of the live object graph in *fromspace*. It is an *on-the-fly collector*. Mutator and collector threads operate concurrently, without any stop-the-world pauses. The original Sapphire algorithm was due to Hudson and Moss [4, 5]. *Transactional Sapphire* adds faster copying with transactional memory [8], parallel GC, sound processing of reference types by the collector [9] and a simpler yet sound treatment of volatile fields, within a general framework for on-the-fly, concurrent and parallel garbage collection for the widely used Java virtual machine, Jikes RVM/MMTk (`http://jikesrvm.org`).

In this report, we describe how we use the SPIN bounded model checker [3] to verify our implementation, on Intel's x86 relaxed memory architecture, of key aspects of our Sapphire implementation, including its more complicated GC phase changes.

Section 1: Concurrent copying, with either atomic CAS operations or software transactional memory;

Section 2: The change of phase from no collection to marking;

Section 3: The change of phase from copying to flipping semispaces;

Section 4: Reference object processing.

Section 5: Object hashing.

Model checking

Model checking is a verification technique for finite state systems. The model checker will visit all possible states reachable from the initial state of the system, checking whether a given property holds in every state. Since the model checker

visits all possible states, models must be small for verification to complete within reasonable time and space. The SPIN model checker accepts a model written in the Promela language, describing a state transition system as a form of sequential processes communicating via channels. We express properties as assertions injected into the model.

Typically, each of our models comprises a collector thread process and one or more mutator thread processes. In addition, the model for concurrent copying has a memory process that models the x86 relaxed memory architecture. Note that these are processes in the modelling language, and not to be confused with the operating systems concept of a process.

1 Concurrent Copying

The Sapphire heap is divided into two semispaces, and collector constructs a replica in *tospace* of the live object graph held in *fromspace*. Sapphire operates in a number of phases. In the Copying phase, the mutator operates on the *fromspace* version of live objects, before switching to the *tospace* version in the Flip phase. It is essential that these two replicas are eventually consistent. This means that, in Sapphire, the mutator is required to propagate any update to a *fromspace* object to its *tospace* replica by writing to both. In Sapphire, the cost of dealing with races between the mutator and collector is borne by the collector. Our implementation supports two concurrent copying algorithms: using CAS operations to copy, or using software transactional memory (STM).

The Promela models for SPIN in this section check both algorithms. Here, we want to verify that, in the Copying phase, the mutator observes a consistent view of the heap regardless of any action by the collector. This model checks the property that, when a mutator repeatedly writes to and reads from a field of an object while the collector is copying the object (and no other mutator accesses the field), the value that the mutator reads is the value that it has most recently written.

The model is kept as small as possible. There is a single mutator and a single collector in the model. A single object holds a single field that is accessed by the mutator while it is copied by the collector. We consider all possible low-level types of field: a single-word scalar, a double-word scalar, and a reference. Because the STM version algorithm depends on the memory model of the x86 CPU, we model not only the collector and the mutator but also the store buffers of the CPU, which cause reordering.

1.1 Configuration

This model contains six configurations; combinations of two versions of copying algorithm and three types of the field. When `STM` is defined, the model uses the STM version. Otherwise, it uses the CAS version. The type of the field is selected by defining macros; its type is:

- a reference type if `REFERENCE` is defined,

- a double-word scalar type if `DOUBLE_WORD` is defined, or
- a single-word scalar type if neither is defined.

1.2 Abstraction

The mutator and collector threads perform reads and writes that access a single word. Each word is assigned a distinct *address*. We assume conservatively that single-word scalar and pointer values are dealt with by single read/write instructions while a double-word value needs two read/write instructions.

Reads and writes to shared memory are abstracted by the following macros, which we define later when we show how to handle the x86 relaxed memory model.

- *thread*`_READ`$(a,\ x)$, and
- *thread*`_WRITE`$(a,\ v)$,

where *thread* is either `MUTATOR` or `COLLECTOR`, a is an address, x is a local variable, and v is a value. The `READ` macro reads from the address a and stores the value to the local variable x. The `WRITE` macro writes a value v to the address a. In addition, the model uses the *thread*`_MFENCE` macro as a memory barrier.

The mutator and collector threads obtain addresses of words in the field of the object by the following macros.

- `FROM_SPACE_ADDR`(i) and
- `TO_SPACE_ADDR`(i),

where i is an offset of a word from the beginning of the field of the object. For a single-word configurations, i is zero, and for a double-word configuration, i is either zero or one. These macros yield the address of the word in the field of the *fromspace* and *tospace* copies, respectively.

In the model with a reference type field, the values are either `NULL` or an address of either copy of the object. The address of a copy of the object can be obtained by the macros

- `FROM_SPACE_OBJECT` and
- `TO_SPACE_OBJECT`.

In addition, the model uses the `FORWARD`(r) macro that updates r with the other copy of the object referred to by r.

1.3 Mutator for Single-Word, Scalar-Field Configuration

Model 1 shows the model of the mutator for the single-word, scalar-field configuration. The mutator uses Promela's `do` construct to repeatedly choose nondeterministically to perform one of the following actions (both guards of the `do` statement are `true` so both cases are always enabled).

Model 1: Concurrent copying: Mutator model for the single-word, scalar-field configuration.

```
1  proctype mutator() {
2    byte x, r, a, v;
3    do
4      ::true ->
5        if
6        ::true -> x = 0
7        ::true -> x = 1
8        fi;
9        r = x;
10       MUTATOR_WRITE(FROM_SPACE_ADDR(0), r);
11       MUTATOR_WRITE(TO_SPACE_ADDR(0), r);
12       r = 0;
13     ::true ->
14       if
15       ::!flipped ->
16         MUTATOR_READ(FROM_SPACE_ADDR(0),r)
17       ::else ->
18         MUTATOR_READ(TO_SPACE_ADDR(0),r)
19       fi;
20       assert(r == x);
21       r = 0;
22   od
23 }
```

- The mutator chooses zero or one arbitrarily (lines 5–8), and writes this value to the field (line 10). The mutator's Copy phase write barrier requires the mutator to write to the *fromspace* copy and then to the *tospace* copy (line 11).
- The mutator reads from the field and checks if the value read is the one that the mutator has most recently written. During the Copy phase, the mutator reads from the *fromspace* copy (line 16) but it reads from the *tospace* copy after the stack is flipped in the Flip phase (line 18).

1.4 Mutator for Double-Word, Scalar-Field Configuration

Model 2 shows the model of the mutator for the double-word, scalar-field configuration. The model is similar to the one for the single-word, scalar-field configuration, but it writes either [0,1] or [1,0] to the two words. It is important that there is a chance for the collector to work between two writes.

Model 2: Concurrent copying: Mutator model for the double-word, scalar-field configuration.

```
1  proctype mutator() {
2    byte x, r0, r1, a, v;
3    do
4      ::true ->
5        if
6        ::true -> x = 0
7        ::true -> x = 1
8        fi;
9        r0 = x; r1 = 1 - x;                    /* [r0,r1] = [0,1] or [1,0] */
10       MUTATOR_WRITE(FROM_SPACE_ADDR(0), r0);
11       MUTATOR_WRITE(FROM_SPACE_ADDR(1), r1);
12       MUTATOR_WRITE(TO_SPACE_ADDR(0), r0);
13       MUTATOR_WRITE(TO_SPACE_ADDR(1), r1);
14       r0 = 0; r1 = 0;
15     ::true ->
16       if
17       ::!flipped ->
18         MUTATOR_READ(FROM_SPACE_ADDR(0),r0);
19         MUTATOR_READ(FROM_SPACE_ADDR(1),r1);
20       ::else ->
21         MUTATOR_READ(TO_SPACE_ADDR(0),r0);
22         MUTATOR_READ(TO_SPACE_ADDR(1),r1);
23       fi;
24       assert(r0 == x && r1 == 1 - x);
25       r0 = 0; r1 = 0;
26   od
27 }
```

Algorithm 1: The collector's word copying algorithm using CAS

```
1  copyWord(p, q):
2      loop
3          currentValue := *q;
4          toValue := *p                                          $
5          if isPointerField(toValue)
6              toValue := forwardObject(toValue)
7          if toValue == currentValue
8              return
9          if ! CAS(q, currentValue, toValue)                     $
10             return
```

1.5 Mutator for Reference-Field Configuration

Model 3 shows the model of the mutator for the reference-field configuration. The model is similar to the one for the single-word, scalar-field configuration, but there are the following differences.

- The values the mutator may write to a field of the object are `NULL` or a reference to the object itself.
- If the value was a reference, the mutator writes the address of the *fromspace* copy to the *fromspace* copy and that of the *tospace* copy to the *tospace* copy.
- After reading a reference, the mutator checks if the reference read is semantically the same as the reference that it last wrote. Because `x` holds the address of the *fromspace* copy and the *tospace* copy should have a *tospace* address, we convert the address read from *tospace* to its *fromspace* equivalent before the comparison.

1.6 The Collector

Models 4 and 5 show the models of the CAS version and the STM version of the collector. Each copies the field of the object according to their copying protocol. The CAS version collector copies each word according to the protocol shown in Algorithm 1.

For the double-word configuration, where `N_WORDS`=2, the collector copies word by word, modelled with the `do`-loop in Model 4. The offset `i` (from the start of the object) indicates the word to copy. If the field is a pointer field, that is, `REFERENCE` is defined, the model needs to forward the `toValue` read from *fromspace* using the `FORWARD` macro.

If the *fromspace* and *tospace* replicas hold the same value (line 10), the collector moves to the next word (the mutator must have updated both). Otherwise, the collector attempts to update atomically (using a CAS) the value held in the

Model 3: Concurrent copying: Mutator model for the reference-field configuration.

```
1  proctype mutator() {
2    byte x, r, a, v;
3    do
4      ::true ->
5        if
6        ::true -> x = NULL
7        ::true -> x = FROM_SPACE_OBJECT
8        fi;
9        r = x;
10       MUTATOR_WRITE(FROM_SPACE_ADDR(0), r);
11       FORWARD(r);
12       MUTATOR_WRITE(TO_SPACE_ADDR(0), r);
13       r = 0;
14     ::true ->
15       if
16       ::!flipped ->
17         MUTATOR_READ(FROM_SPACE_ADDR(0),r);
18         assert(x == r);
19       ::else ->
20         MUTATOR_READ(TO_SPACE_ADDR(0),r);
21         FORWARD(r);
22         assert(x == r);
23       fi;
24       r = 0;
25   od
26 }
```

Model 4: Concurrent copying: Model of the CAS part of the collector.

```
1   i = 0;
2   do
3     ::(i < N_WORDS) ->
4       COLLECTOR_READ(TO_SPACE_ADDR(i), currentValue);
5       COLLECTOR_READ(FROM_SPACE_ADDR(i), toValue);
6  #ifdef REFERENCE
7       FORWARD(toValue);
8  #endif
9       if
10        ::(toValue == currentValue) -> i++
11        ::else ->
12          atomic { /* CAS */
13            COLLECTOR_MFENCE;
14            COLLECTOR_READ(TO_SPACE_ADDR(i), tmp);
15            if
16              ::(currentValue == tmp) ->
17                COLLECTOR_WRITE(TO_SPACE_ADDR(i), toValue)
18              ::else -> i++
19            fi;
20            COLLECTOR_MFENCE;
21          }
22      fi
23    ::else -> i = 0; break
24  od;
25 SUCCESS:                              /* This label is used in Model 5. */
26  COLLECTOR_MFENCE;
27  flipped = true;
```

tospace replica. The CAS instruction is modelled with an `atomic` block (lines 12–17). If the word is copied successfully, the collector moves to the next word by incrementing `i`. Otherwise, it tries again.

The STM version of the collector first copies the entire field and then checks that the two replicas are semantically equivalent. The protocol is shown in Algorithm 2. The Promela model is shown in Model 5. The first `do`-loop copies the field word by word, and the second verifies. If any word of the *tospace* copy does not match its *fromspace* copy, control passes to `FAIL` to fallback to the CAS version.

Algorithm 2: Collector's code for copying an object using software transactional memory

```
1  copyObjectTransactional(p, q):
2      for i := 0 to words(q)                                  /* copying step */
3          toValue := p[i]
4          if isPointerField(p, i)
5              buf[i] := toValue
6              toValue := forward(toValue)
7          q[i] := toValue
8
9      memoryBarrier
10
11     for i := 0 to  words(q)                                 /* verification step */
12         if isPointerField(p, i)
13             if p[i] != buf[i]
14                 goto FAIL
15         else if p[i] != q[i]
16             goto FAIL
17
18     return
19
20 FAIL:
21     copyObject(p, q)       /* fall back to copying word at a time with CAS */
```

Model 5: Concurrent copying: Model of the STM part of the collector.

```
  i = 0;
  /* Copy */
  do
    ::(i < N_WORDS) ->
      COLLECTOR_READ(FROM_SPACE_ADDR(i), toValue);
#ifdef REFERENCE
      buf[i] = toValue;
      FORWARD(toValue);
#endif
      COLLECTOR_WRITE(TO_SPACE_ADDR(i), toValue);
      i++
    ::else -> i = 0; break
  od;

#ifndef NO_FENCE
  COLLECTOR_MFENCE
#endif

  /* Verify */
  do
    ::(i < N_WORDS) ->
#ifdef REFERENCE
      COLLECTOR_READ(FROM_SPACE_ADDR(i), currentValue);
      if
        ::(currentValue != buf[i]) -> goto FAIL
        ::else -> skip
      fi;
#else
      COLLECTOR_READ(FROM_SPACE_ADDR(i), currentValue);
      COLLECTOR_READ(TO_SPACE_ADDR(i), toValue);
      if
        ::(currentValue != toValue) -> goto FAIL
        ::else -> skip
      fi;
#endif
      i++
    ::else -> i = 0; break
  od;
  goto SUCCESS:
FAIL:
  /* insert Model 4 here.
  * Model 4 has the SUCCESS label as well as the fallback routine.
  *   SUCCESS:
  *     COLLECTOR_MFENCE;
  *     flipped = true;
  */
```

1.7 Memory Model

The x86 architecture [6] implements the total store order (TSO) memory model [1]. We modelled TSO in a standard manner [7, 10].

Figure 6 models the semantics of TSO, and Figure 7 shows the mutator's API through which the mutator accesses shared memory. The collector's API is defined in the same manner.

The contents of the shared variables are stored in the shared memory, `shared`. Every process (mutator or collector) has its own FIFO (`mutator_queue` or `collector_queue`) that models the CPU's store buffer. When a process writes a value v to an address a, the pair of (a, v) is written to the store buffer rather than updating the contents of the shared memory. A dedicated `memory` process retrieves the pair from the store buffers non-deterministically, and updates the shared memory by calling `COMMIT_WRITE`. When a process reads from an address a, it uses the value v of the pair (a, v) if such a pair is in the store buffer, or otherwise it reads from the shared memory. However a process cannot observe the contents of the store buffer because it is modelled with the channel type of the Promela modelling language. In Promela, a construct `chan?x` reads a value from a channel `chan` into a variable `x`, and `chan!y` sends the value of `y` down the channel. Thus, the process writes v to its own local memory, modelled by the arrays `mutator_local_memory` and `collector_local_memory`, indexed by an address, at the same time as it writes to the store buffer, thus allowing it to retrieve the value later from the store buffer. The `mutator_queue_count` and `collector_queue_count` counters keep track of the number of pairs written to each address in the forwarding buffer.

1.8 Results

We checked correctness of our concurrent copying algorithms, both with CAS and STM, and found that there was no error, unless we were to omit the `MFENCE` in the STM version (Model 5, line 16). Thus, this fence is essential, and we cannot omit it. The property we checked was that, for single mutator thread programs, the value the mutator reads from a field of an object was always the most recently written value.

As Sapphire assumes that mutators are data-race free, we needed only a model of a single mutator and a single collector process. As mutator or collection actions on one object/field do not affect any other (as far as garbage collection is concerned), we consider only a single object with a single field. It suffices to have the single mutator write to this field many times, and the collector copy it just once. Since our algorithms work slightly differently for different kinds of field, we checked with every low-level type: a single-word scalar field, a multi-word scalar field and a reference field.

As we use bounded model checking, we cannot formally prove that the algorithms are correct. However, we greatly increased confidence in the algorithms, and we found that omission of the `MFENCE` in the STM algorithm was a bug.

Model 6: Model of TSO memory.

```
#define N_ADDRS   (N_WORDS*2)
#define N_THREADS 2

byte shared[N_ADDRS];

byte mutator_local_memory[N_ADDRS], collector_local_memory[N_ADDRS];
byte mutator_queue_count[N_ADDRS], collector_queue_count[N_ADDRS];

chan mutator_queue = [N_THREADS] of {byte, byte};
chan collector_queue = [N_THREADS] of {byte, byte};

#define COMMIT_WRITE(q, count) \
  (len(q) > 0) -> q?a,v -> shared[(a)-1] = v; count[(a)-1]--

active proctype memory() {
  byte a, v;
endmem:
  do
    ::atomic{COMMIT_WRITE(mutator_queue, mutator_queue_count)}
    ::atomic{COMMIT_WRITE(collector_queue, collector_queue_count)}
  od
}
```

Model 7: Mutator's memory access macros.

```
#define MUTATOR_READ(a, v) \
atomic { \
  if \
    ::mutator_queue_count[(a)-1] == 0 -> v = shared[(a)-1] \
    ::else -> v = mutator_local_memory[(a)-1] \
  fi; \
}

#define MUTATOR_WRITE(a, v) \
atomic { \
  mutator_queue!a,v; \
  mutator_local_memory[(a)-1] = v; \
  mutator_queue_count[(a)-1]++; \
}

#define MUTATOR_MFENCE \
atomic { \
  do \
    ::COMMIT_WRITE(mutator_queue, mutator_queue_count) \
    ::else -> break \
  od \
}
```

2 Phase Change: from NoGC to Mark Phase

The first GC phase of Sapphire is the Mark phase. When Sapphire starts GC, it changes the GC phase from NoGC (`NOGC`) to the Mark (`MARK`) phase through the two intermediate phases: `PREMARK1` and `PREMARK2`. Each mutator holds its own indication of the phase: the mutator's phase dictates the mutator's behaviour when it writes to an object or allocates one. In each intermediate GC phase, the mutator changes its own phase to catch up with the GC phase, and in doing so changes its barrier.

The following table shows the mutator phases and barriers. During the `PREMARK1` phase, the mutator enables the insertion barrier, and during `PREMARK2` phase, it starts allocating objects black.

mutator phase	write barrier	allocation colour
`NOGC`	no barrier	white
`PREMARK1`	ins. barrier	white
`PREMARK2` and `MARK`	ins. barrier	black

Our model represents the behaviour of the collector and the mutators during the phase transition from `NOGC` to `MARK`. This model has multiple mutators so that we can check interactions between mutators in different mutator phases. We used two mutators (`N_MUTATORS = 2`).

This model demonstrates that we need at least two intermediate phases. Strictly speaking, we cannot prove that two intermediate phases suffice because our approach relies on bounded model checking. However, we have reasonable confidence in the correctness and sufficiency of this phase change with two intermediate phases.

2.1 Abstraction and Bounding

This model has a heap in which a limited number of objects can be allocated. Our experimentation reveals that model checking completes in a reasonable time and with reasonable memory consumption if the number of objects (`N_OBJECTS`) is three or fewer. Each object is located at a distinct address. Addresses are represented by integers from 1 to `N_OBJECTS`. Each object has a colour and a single payload field. These can be accessed through the macros

- `COLOUR`(x) and
- `SLOT`(x),

where x is the address of an object.

Colour is one of `WHITE`, `GREY`, `BLACK`, and `NOT_ALLOCATED`. The colours except for `NOT_ALLOCATED` have the same meaning as the standard tricolour abstraction. `NOT_ALLOCATED` means that the object is not allocated. The payload field can hold an address of an object or `NULL`.

```
1  proctype mutator(int id) {
2    int root0 = 1;
3    int root1 = NULL;
4
5  end_mutator:
6    do
7      ::atomic{(g_phase != m_phase[id]) -> m_phase[id] = g_phase};
8      ::atomic{(root0 != NULL) -> read(root0, root0)}
9      ::atomic{(root1 != NULL) -> read(root1, root1)}
10     ::atomic{(root0 != NULL) -> read(root0, root1)}
11     ::atomic{(root1 != NULL) -> read(root1, root0)}
12     ::atomic{(root0 != NULL) -> write(root0, root0)}
13     ::atomic{(root1 != NULL) -> write(root1, root1)}
14     ::atomic{(root0 != NULL) -> write(root0, root1)}
15     ::atomic{(root1 != NULL) -> write(root1, root0)}
16     ::atomic{!IS_HEAP_FULL() -> alloc(root0)}
17     ::atomic{!IS_HEAP_FULL() -> alloc(root1)}
18   od
19 }
```

Model 8: No GC to Marking phase: Mutator model.

2.2 The Mutator

Figure 8 shows the mutator model. The mutator model has two local variables: `root0` and `root1`. All mutators share an object; `root0` of each mutator points to the object at address 1. The mutator model emulates an arbitrary program that reads from an object field, writes to an object field, or allocates an object. We modelled read, write and allocation as atomic operations in order to reduce the number of states to be explored. This does not lose generality because we assume that mutators do not race one another, and the collector does not write to the variables that the mutator accesses in these phases; all the collector does in these phases is to advance the GC phase and wait for mutators to catch up.

In addition to the operations above, the mutator model changes its phase to catch up with the GC phase. The mutator changes its phase at a GC safepoint outside of any read, write or allocation action in this model.

Read, write and allocation operations are defined as: Model 9.

- `read`(p,r) reads a reference from the object that local variable p points to. The read reference is assigned into local variable r.
- `write`(p,q) The mutator writes the value in local variable q to the object to which local variable p points. After writing, the write barrier code `checkAndEnqueue` may make the target object grey depending on the phase.
- `alloc`(r) allocates an object and stores the reference into local variable r. The colour of the object is initialised according to the phase. It is

```
1  inline checkAndEnqueue(q) {
2    if
3      ::(m_phase[id] != NOGC_PHASE && q != NULL && COLOUR(q) == WHITE)->
4        COLOUR(q) = GREY
5      ::else -> skip
6    fi
7  }
8
9  inline write(p, q) {
10   SLOT(p) = q;
11   checkAndEnqueue(q);
12 }
13
14 inline read(p, retval) {
15   retval = SLOT(p);
16 }
17
18 inline alloc(retval) {
19   retval = free_ptr;
20   free_ptr = free_ptr + 1;
21   if
22     ::(m_phase[id] == PREMARK2_PHASE || m_phase[id] == MARK_PHASE) ->
23       COLOUR(retval) = BLACK;
24     ::else ->
25       COLOUR(retval) = WHITE;
26   fi;
27   SLOT(retval) = NULL;
28 }
```

Model 9: Read, write and allocation operations.

worth noting that `alloc` is called only when there is room for allocating a new object: the choice of `alloc` in the `do`-loop in Model 8 is guarded by "`not IS_HEAP_FULL()`", which is defined as "`free_ptr` $<$ `N_OBJECTS`".

2.3 Collector and Phase Advancing

The collector advances the GC phase, and waits for the mutators to catch up. Model 10 shows the model of the phase advance mechanism. `g_phase` and `m_phase` represent the GC phase and the mutator phases of mutators, respectively. `collector` is the collector model. The collector advances the GC phase, and waits for all mutators to change mutator phase. Each mutator changes its mutator phase with

```
7  ::atomic{(g_phase != m_phase[id]) -> m_phase[id] = g_phase};
```

```
1  mtype g_phase = NOGC_PHASE;
2  mtype m_phase[N_MUTATORS];
3
4  inline waitForMutators() {
5    int i = 0;
6    do
7      ::(i < N_MUTATORS) ->
8        (m_phase[i] == g_phase) -> i++
9      ::else -> break
10   od
11 }
12
13 proctype collector() {
14   atomic {
15 #ifdef TYPE_II
16   g_phase = PREMARK1_PHASE;
17   waitForMutators();
18 #endif
19   g_phase = PREMARK2_PHASE;
20   waitForMutators();
21   g_phase = MARK_PHASE;
22   waitForMutators();
23   }
24 }
```

Model 10: No GC to Marking phase: Phase advance

in Model 8.

We can explore both Type I and Type II phase changes, which use one or two intermediate phases, respectively.1 If the `TYPE_II` macro is not defined in Model 10, the collector skips the `PREMARK1` phase, letting each mutator enable the insertion barrier and start allocating black at the same time.

2.4 Verification

As Sapphire uses an insertion barrier with a grey mutator in these phases, we checked that the strong tricolour invariant always holds. This invariant requires that there are no black to white references. Our model includes an observer process (Model 11) that checks, for every object, that if the object is `BLACK` and its slot does not hold `NULL`, then the object pointed from the slot is not `WHITE`. Because the observer can work at any time, our model checks the invariant is held at any time.

```
1  proctype observer() {
2    int i = 1;
3    atomic {
4      do
5      ::(i <= N_OBJECTS) ->
6        assert(!(COLOUR(i) == BLACK && SLOT(i) != NULL) ||
7               COLOUR(SLOT(i)) != WHITE);
8        i = i + 1
9      ::else -> break
10     od
11   }
12 }
```

Model 11: No GC to Marking phase: Observer

2.5 Results

When we used two intermediate phases, that is, we defined the `TYPE_II` macro, model checking showed that the tricolour invariant is always held. In contrast, if we did not define `TYPE_II`, allowing the mutator to enable the insertion barrier and to start allocating black at the same time, the model checker found a counterexample where the strong tricolour invariant did not held. The counterexample discovered the following execution:

```
   mutator 0      mutator 1
1  m_phase[0]= PREMARK2_PHASE
2  root1 = alloc();                    // new object: 2 (black)
3  root0.slot = root1;                 // root0: 1 (white), root1: 2 (black)
4  root1 = root1.slot;
5               root1 = root0.slot;    // root0: 1 (white), root0.slot: 2 (black)
6               root1.slot = root0;    // root1: 2 (black), root0: 1 (white)
```

Initially, `root0` of both mutator threads points to the object 1.

1. At line 1, mutator 0 proceeds to the `PREMARK2` phase while mutator 1 is still in the `NOGC` phase.
2. At line 2, mutator 0 allocates a black object 2. Then, it stores the object into the slot of white object 1. Because the object 1 is shared between two mutators, mutator 2 becomes able to access object 2 through object 1.
3. At line 5, mutator 1 reads the reference to object 2.
4. At line 6, mutator 1 stores a reference to white object 1 in the slot of black object 2. Thus, the strong invariant is violated.

3 Phase Change: from Copy to Flip Phase

In the Copy (`COPY`) and Flip (`FLIP`) phases, the mutator writes to both *fromspace* and *tospace* replicas. In the Copy phase, the mutator assumes that there are no *tospace* references held anywhere except in *tospace*, and it never writes *tospace* reference to anywhere other than *tospace* (Algorithm 3). In contrast, in the Flip phase, the mutator never writes *fromspace* references (Algorithm 4). These two barriers conflict: a mutator in the Flip phase may violate the assumption of a mutator in the Copy phase. As a result, the invariant that *a tospace object never refers to a fromspace object* is violated.

Algorithm 3: Copy phase barrier

```
1  Write_Copy(p, f, q):                                   /* p.f = q */
2      p[f] := q
3      if inFromspace(p)
4          pp = p.forwardingPointer
5          if inFromspace(q)
6              pp[f] := q.forwardingPointer
7          else
8              pp[f] := q
```

Algorithm 4: Flip phase barrier

```
1  Write_Flip(p, f, q):                                   /* p.f = q */
2      if inFromspace(q)
3          q = q.forwardingPointer
4      p[f] := q
5      if inFromspace(p) || inTospace(p)
6          pp := p.forwardingPointer
7          pp[f] := q
8
```

To prevent this, Sapphire uses two intermediate phases: `PREFLIP1` and `PREFLIP2`. The mutator runs with PreFlip phase barrier (Algorithm 5) in the PreFlip1 phase, and it switches to the Flip phase barrier when it enters to PreFlip2 phase. The PreFlip phase barrier carefully checks the destination of references so that it works as if it is the Flip phase barrier if the mutator is writing a *tospace* reference while it works as if it is the Copy phase barrier in other cases. The following table summarises the mutator phases and barriers.

mutator phase	write barrier
`COPY`	Copy phase barrier (Algorithm 3)
`PREFLIP1`	PreFlip phase barrier (Algorithm 5)
`PREFLIP2` and `FLIP`	Flip phase barrier (Algorithm 4)

Algorithm 5: Copy to Flip phase: PreFlip phase barrier

```
1  Write_preFlip(p, f, q):                                   /* p.f = q */
2      if inFromspace(q) && inTospace(p)
3          q := q.forwardingPointer
4      p[f] := q
5      if inFromspace(p) || inTospace(p)
6          pp := p.forwardingPointer
7          if inFromspace(q) || (inTospace(q) && inFromspace(pp))
8              pp[f] := q.forwardingPointer
9          else
10             pp[f] := q
11
```

Our model represents the behaviour of the collector and mutators during the phase transition from `COPY` to `FLIP`. The model has multiple mutators as the model in Section 2 does so that we can check interactions between mutators in different mutator phases. We used two mutators(`N_MUTATORS = 2`).

This model demonstrates that we need at least two intermediate phases. More specifically, if we have a single intermediate phase, that is, if mutators with the Copy phase barrier and mutators with the Flip phase barrier run simultaneously, then the invariant is violated. But if we have two intermediate phases, the model checker does not find an error.

3.1 Abstraction and Bounding

We consider three spaces: *fromspace*, *tospace* and a non-replicated space. We assume that every object in *fromspace* has its copy in *tospace* because these were created in a prior phase. We consider two objects in the replicated spaces (`N_OBJECTS = 2`), and a single object in the non-replicated space; hence, there are five copies of objects. We assign addresses from 0 to 4 to copies of objects. The macros `IN_FROM_SPACE` and `IN_TO_SPACE` are predicates to test whether a given address is in *fromspace* (respectively, *tospace*). In addition, the model uses the `FORWARD`(r) macro to update r with the other copy of object referred to by r if r is in *fromspace* or *tospace*. Otherwise, `FORWARD` does nothing.

Each object has a single field as usual. The field holds an address; initially, each copy holds the address of itself in its field, meaning that each object refers to itself. We do not check the case where a field holds `NULL`. The mutator and collector processes obtain the value in the field with the following macros.

- `FROM_SPACE_OBJECT`(i),
- `TO_SPACE_OBJECT`(i), and
- `NON_REPL_OBJECT`(j),

```
1  proctype mutator(int id) {
2    int root0 = FROM_SPACE_OBJECT(0);
3    int root1 = FROM_SPACE_OBJECT(0);
4    int tmp, p, q;
5
6  end_mutator:
7    do                                                    /* handshake */
8      ::atomic{(g_phase != m_phase[id]) -> m_phase[id] = g_phase;}
9      ::atomic{read(root0, root0);normalise()}
10     ::atomic{read(root0, root1);normalise()}
11     ::atomic{read(root1, root0);normalise()}
12     ::atomic{read(root1, root1);normalise()}
13     ::atomic{write(root0, root0)}
14     ::atomic{write(root0, root1)}
15     ::atomic{write(root1, root0)}
16     ::atomic{write(root1, root1)}
17   od
18 }
19
```

Model 12: Copy to Flip phase: Mutator model.

where i is a number identifying the object in *fromspace* and *tospace* (0 or 1), and j is a number identifying the object in non-replicated space. Because we have a single object in non-replicated space, j should be 0.

3.2 The Mutator

Model 12 shows the main loop of the mutator model. This is similar to the mutator model for the phase change from NoGC to Mark phase in Section 2.

All mutators have two local variables: `root0` and `root1`. Initially, both variables of all mutators refer to the same object in replicated space. Because the GC is in the Copy phase, they hold the address of the *fromspace* replica. This mutator model emulates an arbitrary program that reads from an object or writes to an object, but we do not consider allocation for this model.

As with the mutator model in Section 2, read and write operations are atomic. Because we needed to reduce the state space to be explored by model checking further in orde to complete within a reasonable computational resource, we normalised the states of mutators after `read` operations. Normalisation swaps the values of variables `root0` and `root1` if necessary so that `root0` $\leq$ `root1`. Because this model is symmetry with respect to these local variables, this normalisation does not loose generality.

Reads and writes are modelled in similar way to the model in Section 2, but the write barrier differs.

- `read`(p, r) reads a reference from the object to which local variable p points

into local variable r.

- `write`(p, q) writes the value in local variable q to the object to which local variable p points by calling the write barrier corresponding to the current phase.

Write barriers are defined in Model 13. `write_copy`, `write_preflip`, and `write_flip` are models of the copy phase barrier (Algorithm 3), the PreFlip phase barrier (Algorithm 5) and the Flip phase barrier (Algorithm 4) respectively. They semantically write address of the object `q` to the field of the object `p`. In these models, `raw_write`(p,q) writes the value q to the address p. The modelling of these barriers is straightforward.

3.3 Collector and Phase Advancing

The collector model and the phase advancing mechanism are the same as those for the model for the phase change from NoGC to PreMark in Section 2 up to the phase names.

It is worth noting that this model is also capable of checking Type I and Type II phase changes by undefining and defining the `TYPE_II` macro, as with the collector model in that section.

3.4 Verification

An important invariant of Sapphire is that

> *a tospace object never refers to a fromspace object.*

Our model includes an observer process that checks this invariant. More specifically, the observer process checks the assertion

```
1  assert(!IN_FROM_SPACE(TO_SPACE_OBJECT(i)));
```

for all objects `i` in the replicated space.

3.5 Results

We used two intermediate phases, and checked the invariant that an object in *tospace* never has a reference to *fromspace*. Model checking showed that the invariant always holds. However, with a single intermediate phase, omitting the phase where mutator used the PreFlip phase barrier (Algorithm 5), the model checker found a counterexample. This showed that the intermediate phase with the PreFlip phase barrier is necessary.

The counterexample our model checking with a single intermediate phase found is as follows.

```
1  inline write_copy(p, q) {
2    raw_write(p, q);
3    if
4      ::(IN_FROM_SPACE(p)) ->
5        FORWARD(p);
6        if
7          ::(IN_FROM_SPACE(q)) -> FORWARD(q); raw_write(p, q)
8          ::else -> raw_write(p, q)
9        fi
10     ::else -> skip
11   fi
12 }
13
14 inline write_preflip(p, q) {
15   if
16     ::(IN_FROM_SPACE(q) && IN_TO_SPACE(p)) -> FORWARD(q)
17     ::else -> skip
18   fi;
19   raw_write(p, q);
20   if
21     ::(IN_FROM_SPACE(p) || IN_TO_SPACE(p)) ->
22       FORWARD(p);
23       if
24         ::(IN_FROM_SPACE(q) || (IN_TO_SPACE(q) && IN_FROM_SPACE(p))) ->
25           FORWARD(q);
26           raw_write(p, q)
27         ::else -> raw_write(p, q);
28       fi
29     ::else -> skip
30   fi
31 }
32
33 inline write_flip(p, q) {
34   if
35     ::(IN_FROM_SPACE(q)) -> FORWARD(q)
36     ::else -> skip
37   fi;
38   raw_write(p, q);
39   if
40     ::(IN_FROM_SPACE(p) || IN_TO_SPACE(p)) ->
41       FORWARD(p);
42       raw_write(p, q);
43     ::else -> skip
44   fi
45 }
46
```

Model 13: Copy to Flip phase: Write barriers. `raw_write(`p`,` q`)` writes value q to address p.

```
   mutator 0      mutator 1
1  m_phase[0]= PREFLIP2_PHASE
2  root0.slot = root0;                 // mem[0] := 2, mem[2] := 2
3                 root0 = root0.slot;  // root0 := mem[0]
4                 swap root0 and root1 // root0 := 0, root1 := 2
5                 root1.slot = root0;  // mem[2] := 0
```

Although we performed model checking with two objects in the replicated space and one object in the non-replicated space, the counterexample showed a single replicated object suffices to cause an error. In the counterexample, address 0 is of the *fromspace* copy of the object and 2 is of the *tospace* copy. Initially, both `root0` and `root1` of both mutator threads points to the *fromspace* copy.

1. At line 1, mutator 0 proceeds to the `PREFLIP2` phase while mutator 1 is still in the `COPY` phase.

2. At line 2, mutator 0 writes by calling the Flip phase barrier. The Flip phase barrier writes the address of *tospace* copy to both copies.

3. At line 3, mutator 1 reads from the *fromspace* copy, which has the address of the *tospace* copy.

4. At line 4, the model checking swaps the value of two local variables `root0` and `root1` of mutator 1. This does not lose generality. Now, `root0` points to the *fromspace* copy, and `root1` points to the *tospace* copy.

5. Finally, at line 5, mutator 1 writes by calling the Copy phase barrier. The Copy phase barrier writes the address of the *fromspace* to the *tospace* copy.

As a result, the *tospace* copy points to the *fromspace* copy, violating the invariant.

4 Reference Objects

Java provides references of four (decreasing) levels of strength: strong (i.e. normal), soft, weak, and phantom. Weaker references are implemented by *reference object* classes. Correct handling of reference objects by a concurrent, let alone an on-the-fly, collector is complex. Mutators can acquire a reference through `java.lang.ref.Reference.get`, which returns a strong reference to the referent or `null` if the reference has been cleared (`PhantomReference.get` always returns null). For simplicity, we consider only weak references here.

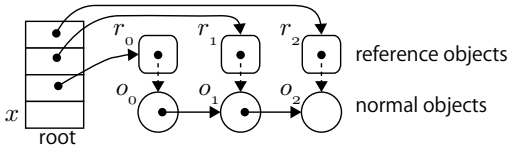

Figure 1: Weak references held in strongly reachable objects r_0, r_1 and r_2 must be cleared atomically.

The challenge for concurrent GC is that there may be a race between the collector clearing a reference and a mutator strengthening the reachability of its referent by calling `get`. For this reason, the semantics of reference classes require that, at the time that the GC decides to reclaim a weakly reachable object (such as o_2 in Figure 1), it must also clear *atomically*

1. *all* weak references to o_2 (e.g. the reference from r_2 in Figure 1), and
2. *all* weak references to other weakly-reachable objects from which o_2 is reachable through a chain of stronger references (e.g. the references in r_0 and r_1).

This prevents a mutator from making o_2 strongly-reachable by retrieving one of the weakly-reachable objects from which o_2 is reachable.

This is relatively straightforward in a stop-the-world context, as mutators are not active while the collector runs. However, in an on-the-fly context, a mutator may call `get` on a weak reference (e.g. A) whose referent O is only weakly-reachable, causing the referent to become strongly-reachable if the reference has not yet been cleared. Once the referent becomes strongly-reachable, the collector must not clear the weak reference, and must retain any objects that just become strongly reachable. The consequence is that single invocation of `get` may affect whether the collector should clear many other weak references spread across the heap. This problem cannot be resolved with just a barrier.

Our Sapphire collector identifies all strongly-reachable objects and all weak references whose referents are only weakly-reachable. This is an iterative process since a mutator may cause a previously weakly-reachable object to become strongly-reachable by calling `get`. Mutators calling `get` communicate with the collector through a global reference-state variable.

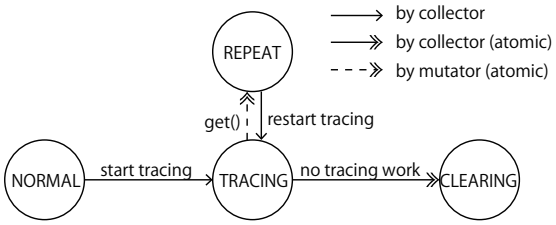

Figure 2: Global reference state transitions.

4.1 Reference processing state transitions

Figure 2 shows the state transition diagram for this global reference state. The collector is in the `NORMAL` state when it is not running. When a collection is triggered, the collector starts `TRACING`, traversing strong references from the roots as usual, trying to mark all strongly-reachable objects. If no mutator calls `get` during the traversal, all strongly-reachable objects will be marked and the collector can proceed to `CLEARING` weak reference objects whose referents are not marked.

However, if a mutator invokes `get` on a reference object with an unmarked referent while the collector is `TRACING`, the collector must process the referent's transitive closure before it moves to `CLEARING`. To resolve the race between calling `get` and the collector proceeding to `CLEARING`, we introduce another state, `REPEAT`. When the collector believes its tracing work is complete, it attempts to change the state to `CLEARING` atomically. Meanwhile, any `get` will attempt to set the global state atomically to `REPEAT`, which will prevent the collector proceeding to `CLEARING`. If the collector fails to proceed to `CLEARING`, it continues `TRACING` from the newly greyed referents. We can expect these to be few and that the number of white objects that can be reached from those grey objects not to be large. Once the collector starts `CLEARING`, mutators are prevented from retrieving any unmarked referent: `get` returns `null`. Thus the collector has *logically* cleared all weak references simultaneously.

It is important to ensure that no mutator obtains a reference to an unmarked referent in the `CLEARING` state. If a `get` method were to be invoked in the `NORMAL` or `REPEAT` state, and if the collector were to change the state to `TRACING` and then to `CLEARING` before the mutator executes the instruction to obtain the referent in `get`, then the mutator would obtain the reference in the `CLEARING` state. This does not happen in our collector because the collector handshakes with mutators in the `TRACING` state to flush the `tobeCopiedQueue` for the mutators' write barrier. The mutator cannot answer the handshake while it is executing `get`. Thus, the transition from `TRACING` to `CLEARING` occurs only when tracing is actually complete.

4.2 Implementation

When a mutator in the `TRACING` state obtains a white (unmarked) referent of a weak reference, the documentation for Java's `java.lang.ref` package specifies that a strong reference is loaded. How this is handled depends on whether the collector uses an insertion or a deletion barrier. If the collector uses insertion barriers, the mutator is grey so it may hold a white reference. This reference will be blackened when the collector loops to terminate, scanning its work queue and mutator roots repeatedly until it finds no grey objects before attempting to set its state to `CLEARING` (Algorithm 6a). If this attempt succeeds, tracing has terminated, and any attempts to `get` an unmarked referent will return `null`.

If the collector uses deletion barriers, mutators are black and cannot load a white reference, as the collector does not rescan roots (Algorithm 6b). Hence, `get` must shade the referent grey to preserve the invariant (Algorithm 7b); `get` with an insertion barrier does not need to do this (Algorithm 7a). However, the collector must still loop to terminate, in this case to process grey objects, if the global state was `REPEAT` when the collector attempted to switch to `CLEARING`.

The deletion barrier solution tends to terminate quickly, as the collector traces only from objects known to be grey. In contrast, a collector using an insertion barrier must scan the roots to discover grey objects before processing them. This also increases the opportunity for mutators to `get` further white referents while the collector is attempting to terminate. Thus, theoretically there is a risk of failure to make progress, for instance if a mutator repeatedly `get`s then drops a white referent. However, termination is guaranteed with the deletion barrier. We used model checking to confirm the correctness of the deletion barrier version. It also identified the risk of non-progress with the insertion barrier.

4.3 Model Checking

To check our algorithms for processing reference types, we verified the following properties:

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

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

These properties are from the mutator's view because there can be a variety of implementations of 'clearing'. In our implementation, logically cleared references appeared cleared to mutator. Property **P1** is required regardless of the existence of reference objects. But **P1** also requires that, if a mutator loads a referent of a reference object, the referent has not been reclaimed. Property **P2** implies the atomicity that the API definition requires.

Since bounded model checking does not deal with infinite state, we checked the properties for the limited model shown in Fig. 1. This model has three pairs of reference and normal objects, namely r_0, r_1, r_2 for references and o_0, o_1, o_2

```
1  collection() {
2    insertionBarrier ← ON;
3    transitiveClosureFromRoot();
4
5
6
7
8    while(true) {
9      refState ← TRACING;
10     handshake();
11     transitiveClosureNoRootScan();
12     scanRoot();
13     if(workQueue.empty() &&
14        CAS(refState,
15            TRACING, CLEARING))
16       break;
17   }
18   insertionBarrier ← OFF;
19   clearReference();
20   refState ← NORMAL;
21   handshake();
22   reclaim();
23 }
```

(a) Insertion barrier

```
1  collection() {
2    insertionBarrier ← ON;
3    transitiveClosureFromRoot();
4    deletionBarrier ← ON;
5    handshake();
6    scanRoot();
7    insertionBarrier ← OFF;
8    while(true)
9      refState ← TRACING;
10     handshake();
11     transitiveClosureNoRootScan();
12
13     if(workQueue.empty() &&
14        CAS(refState,
15            TRACING, CLEARING))
16       break;
17   }
18   deletionBarrier ← OFF;
19   clearReferences();
20   refState ← NORMAL;
21   handshake();
22   reclaim();
23 }
```

(b) Deletion barrier

Algorithm 6: The collector

```
1  get() {
2    while(true) {
3      switch(refState) {
4      case NORMAL:
5        return referent;
6      case REPEAT:
7        return referent;
8
9
10
11
12     case TRACING:
13       if (referent=null
14           ||COLOR(referent)≠WHITE)
15         return referent;
16       CAS(refState, TRACING, REPEAT);
17       break; /* retry */
18
19
20
21     case CLEANING:
22       if (referent=null
23           ||COLOR(referent)≠WHITE)
24         return referent;
25       return null;
26     }
27   }
28 }
```

(a) Insertion barrier

```
1  get() {
2    while(true) {
3      switch(refState) {
4      case NORMAL:
5        return referent;
6      case REPEAT:
7        if (referent=null
8            ||COLOR(referent)≠WHITE)
9          return referent;
10       COLOR(referent) ← GREY;
11       return referent;
12     case TRACING:
13       if (referent=null
14           ||COLOR(referent)≠WHITE)
15         return referent;
16       if (CAS(refState, TRACING, REPEAT)) {
17         COLOR(referent) ← GREY;
18         return referent;
19       }
20       break; /* retry */
21     case CLEANING:
22       if (referent=null
23           ||COLOR(referent)≠WHITE)
24         return referent;
25       return null;
26     }
27   }
28 }
```

(b) Deletion barrier

Algorithm 7: `WeakReference.get()`

```
1  while(true) {
2    i = random.nextInt(5);
3    switch (i) {
4      case 0: x = vr0.get(); break;
5      case 1: x = vr1.get(); break;
6      case 2: x = vr2.get(); break;
7      case 3: if (x ≠ null) x = x.next; break;
8      case 4: x = null; break;
9    }
10 }
```

Algorithm 8: Simple mutator

for the corresponding normal objects. These normal objects are linked in a list, but there are no other strong references to them. We assumed that all reference objects remain directly strongly reachable from the root and that the mutator can always call `get`() methods on them.

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

Model 14 shows the model of the `get`() method on the reference object r_i, for a collector using an insertion barrier. This model is faithful to Algorithm 7a. The return value is passed to the caller through the parameter `ret`. `mark`[`i`] and `CLEARED`[`i`] represent the colour of o_i and whether r_i has been cleared or not, respectively. When `get`() returns o_i, it sets `i` to `ret`. In order to check **P2**, the model also puts `i` and `ret` in global variables `getRef_arg` and `getRef_ret`.

For the collector side, our model is faithful to the pseudocode in Algorithms 6a and 6b. At the end of a cycle, the collector reclaims white objects by calling `reclaim`(): we introduce a fourth object state `RECLAIMED`. Our model of `reclaim`() reclaims white objects and reverts the black objects to white. **P1** and **P2** can be interpreted as:

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

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

where $\texttt{RETNULL}_i \equiv$(`getRef_arg`$= i) \wedge$ (`getRef_ret`$=$ `NULL`).

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

P3 (Termination) GC eventually terminates.

```
inline getReferent(i, ref) {
  do :: (refState == NORMAL) ->
        if
          :: (reference[i] == true) -> ref = REFERENT(i)
          :: else -> ref = -1
        fi;
        break
      :: (refState == REPEAT) ->
#ifdef DELETION_BARRIER
        if :: (reference[i] == true) ->
             if
               :: (mark[REFERENT(i)] == WHITE) -> mark[REFERENT(i)] = GRAY;
               :: else -> skip
             fi;
             ref = REFERENT(i);
           :: else -> ref = -1;
        fi;
        break
#else
        if :: (reference[i] == true) -> ref = REFERENT(i)
           :: else -> ref = -1
        fi;
        break
#endif
      :: (refState == TRACING) ->
        if :: (reference[i] == true) ->
             if :: (mark[REFERENT(i)] == WHITE) ->
                  CAS(refState, TRACING, REPEAT)  /* continue do-loop */
                :: else -> ref = REFERENT(i); break
             fi
           :: else -> ref = -1; break
        fi
      :: (refState == CLEANING) ->
        if :: (reference[i] == true) ->
             assert(mark[REFERENT(i)] != RECLAIMED);
             assert(mark[REFERENT(i)] != GRAY);
             if :: (mark[REFERENT(i)] == WHITE) -> ref = -1
                :: else -> ref = REFERENT(i)
             fi;
           :: else -> ref = -1
        fi;
        break
  od;
  d_step{
    getReferent_arg = i;
    getReferent_ret = ref
  };
}
```

Model 14: Reference processing: `Reference.get()` method

4.4 Results

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

Fortunately, the deletion barrier version *does* make progress, since `get`() shades white objects grey. SPIN confirmed that all three properties, **P1**, **P2** and **P3**, hold for this model, i.e. our implementation is safe, consistent and terminates.

5 Header and Hashcode

Handling of the Java `Object.hashCode` method is tricky in Sapphire. The requirement of Java language is:

> Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified[1].

Sapphire uses address-based hashing, in which addresses of objects are used as their hash codes. But Sapphire moves objects. In a stop-the-world setting, we could record the hash code of an object when the garbage collector moves the object if any mutator had previously obtained its address as its hash code. However, in our on-the-fly setting, the mutator may call the `hashCode` method while the collector is moving the object and, hence, they may race.

We made models of multiple mutators that continuously call the `hashCode` method on the same object while the collector is running in order to check that `hashCode` in our implementation meets its specification. The collector model performs the whole cycle of Sapphire garbage collection continuously because a mutator may call `hashCode` in any GC phase.

Property The property we checked is that *all calls of the* `hashCode` *method return the same value.* Because the hash code depends on the address where the object is placed when the first invocation of `hashCode` is made, we cannot determine a hash code before the method is first called on that object. Therefore, the property we checked is denoted by the linear temporal logic formula

[1] `https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#hashCode--`

Table 1: State transition in address based hashing.

	`hashCode()`		collector's copy
	return value	next state	*tospace* copy's state
`UNHASHED`	address	`HASHED`	`UNHASHED`
`HASHED`	address	`HASHED`	`MOVED`
`MOVED`	`hashcode` field	`MOVED`	`MOVED`

(**Generic Consistency**)
$\forall v \neq \texttt{INVALID}.\ \Box((\texttt{hash} = v) \implies \Box(\texttt{hash} = v))$,

where `hash` is the variable to which mutators write the return value of `hashCode`, and the initial value of `hash` is `INVALID`.

Our model uses a single object, with two addresses 0 and 1 for its *fromspace* and *tospace* replicas. Thus, the formula above is specialised as follows, which is the property we checked:

$$(\textbf{Consistency})\ \Box((\texttt{hash} = 0) \implies \Box(\texttt{hash} = 0)) \wedge \Box((\texttt{hash} = 1) \implies \Box(\texttt{hash} = 1)).$$

5.1 Address Based Hashing

In systems where objects do not move, we can use the addresses of objects for hash codes of them. This implementation is efficient because neither do we have to generate hash codes nor do objects require extra space to store hash codes. If objects move, however, this implementation does not work. The address of an object changes, and so `hashCode` on the same object would not yield the same hash code.

Instead, address-based hashing uses the address where an object is located when `hashCode` is called on the object for the first time as the hash code of the object. If the object moves after `hashCode` was called, the collector allocates an extra field in the *tospace* object to store its hash code.

In address-based hashing implementation, every object has one of three hashing states: `UNHASHED`, `HASHED`, and `MOVED`. Table 1 and Figure 3 show the state transition. Objects are initially in the `UNHASHED` state. When `hashCode` is called on an `UNHASHED` object, the address of the object is used for the hash code of the object, and the object transits to the `HASHED` state. When the collector moves a `HASHED` object, the collector allocates an extra word, the `hashcode` field, for the *tospace* object and stores the address of the *fromspace* object in that field. The *tospace* object is in the `MOVED` state. The `hashCode` method yields the value in the `hashcode` field if the object is in the `MOVED` state. When the collector moves a `MOVED` object, the collector copies the `hashcode` field as well as other words of the object.

In concurrent collectors, implementation of the address based hashing involves a subtle problem: the collector copying an `UNHASHED` object may race with

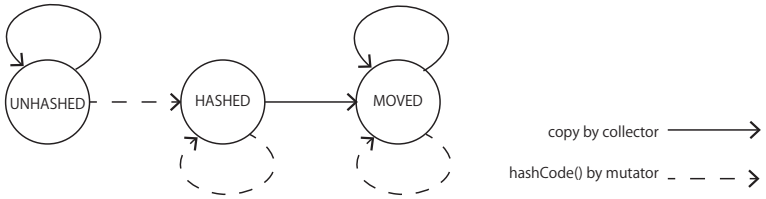

Figure 3: State transition in address based hashing.

a mutator executing the `hashCode` method. If the collector copies a `UNHASHED` object that the `hashCode` method is using its address as a hash code, `hashCode` returns the address of *fromspace* copy although the *tospace* object is in the `UNHASHED` state.

In our Sapphire implementation, object headers hold their hashing states. Sapphire uses an eager *tospace* invariant for object headers, i.e. the collector copies the object headers, and once the header of an object is copied, mutators access the header of the *tospace* copy. The `forwarded` bit in the object header of a *fromspace* object indicates that the header has been copied: the collector sets this bit when it copies the header.

To resolve races between the collector copying the header and mutators accessing and updating the hashing state, Sapphire uses a *meta-lock* mechanism. Whenever a mutator accesses an object header, it calls the `metaLockObject` method to obtain a *pseudo-reference* of the object. The mutator accesses the object header referred to by the pseudo-reference.

Algorithm 9 shows pseudocode for `metaLockObject`. A pseudo-reference of an object is actually a pointer to the copy of the object that has the up-to-date value of the object header. If the GC is not running or the object has already been copied, the object (in case the GC is not running) or the *tospace* copy (in the case that the object has been copied) holds the up-to-date value. If the object has not been copied, `metaLockObject` sets the `busy` bit to prevent the collector from copying the object. The `metaUnlockObject` method clears the `busy` bit.

5.2 Abstraction

Our model has a single object, with two addresses 0 and 1 for its *fromspace* and *tospace* copies. Because our model of the collector performs multiple GC cycles, which address is for *fromspace* changes during the execution. Our model keeps track of *fromspace*. The `IN_FROM_SPACE`(o) macro tells whether the copy of object o is in *fromspace* or not.

To obtain the address of the other copy of the object, the `FWD`(o) macro is useful. For the mutator, `FWD`(o) models the `getForwardingPointer` operation on o. For the collector, it gives the address to which o is copied.

An object is modelled with a combination of the contents of the hashing state (`hashState`), the busy bit (`busy`), the forwarded bit (`forwarded`), and the

Algorithm 9: Implementation of meta-locking

```
1  metaLockObject(o):
2      if not inGCCycle() || not inFromSpace(o)
3          return o
4      do
5          status ← o.statusWord
6          if (status & FORWARDED) ≠ 0
7              return o.forwardingPointer
8      while not CAS(&o.statusWord, status, status | BUSY)
9      return o
10
11 metaUnlockObject(o):
12     if not inGCCycle() || not inFromSpace(o)
13         return
14     o.statusWord ← o.statusWord & ~BUSY;
```

```
1  do
2  ::true -> obj = root[id];getObjectHashCode(obj, hc);hash = hc
3  ::atomic{m_phase_behind[id] -> m_phase_behind[id] = 0}
4  od
```

Model 15: Mutator model.

word used for the hash code (`hashcode`) if the object has been moved after its hashcode was obtained. The collector manipulates these variables directly to set up the *tospace* copy. The mutator accesses the hash state though macros `IS_UNHASHED`(o) and `IS_HASHED`(o), which tell if the hash state of o is `UNHASHED` and `HASHED`, respectively. Remark that `IS_HASHED`(o) yields false if the hash state of o is `MOVED`.

5.3 Mutator

This model has multiple mutator processes. We checked with up to three mutators (`N_MUTATORS` ≤ 3). All mutators share an object, referred to by `root`. Though the `roots` of mutators are local variables, they are represented by a global array whose index is the process ID in the model so that the collector can flip them.

Model 15 shows the mutator model. Each mutator repeatedly obtains the hash code of the object by calling `getObjectHashCode`, which models the `hashCode` method. The `getObjectHashCode` method returns the hash code through the second parameter `hc`. The mutator stores the hash code in a global variable `hash`. Thus, we can check that our implementation satisfies the requirements of `hashCode` by observing `hash`.

The mutator also advances its phase if it is behind the GC phase, as per

Algorithm 10: Hashing implementation

```
1  getObjectHashCode(o):
2    o ← hashByAddress(o)
3    if isHashed(o)
4      return (int) o
5    return readHashCode(o)                    /* HASHED_AND_MOVED */
6
7  hashByAddress(o):
8      if not isUnhashed(o)
9          return o
10     if not inGCCycle() || not inFromSpace(o)
11         setHashed(o)
12         return o
13     o ← metaLockObject(o)                  /* returns a pseudo−reference */
14     if isUnhashed(o)
15         setHashed(o)
16     metaUnlockObject(o)
17     return o
```

the mutator models in Sections 2 and 3. However, we modelled phase advancing mechanism in a different way from those models. The modelling of phase advancing is mentioned in Section 5.6.

5.4 The `hashCode` Method

Algorithm 10 shows pseudo code of the `hashCode` method. Our implementation of `hashCode` method consists of two parts. The `getObjectHashCode` function deals with the simple cases, where the object is in `HASHED` or `MOVED`. In these states, the hash code of the object has been fixed. Thus, `getObjectHashCode` can return the hash code regardless of the behaviour of mutators.

In the racy case where the object is `UNHASHED`, `hashByAddress` is called to make object `HASHED` atomically. The `hashByAddress` function takes meta-lock on the object if there is a risk of a race with the collector, i.e. the GC is running and the object is in *fromspace*, before changing the hashing state of the object to `HASHED`. Otherwise, the function makes the object `HASHED` without synchronisation.

We modelled these functions straightforwardly as shown in Model 16.

5.5 Meta-locking

Model 17 shows the model of meta-locking, whose algorithm is shown in Algorithm 9. The second parameter of `metaLockObject`, `oo`, is the out parameter, through which `metaLockObject` returns the pseudo-reference of object `o`.

The loop with the CAS in lines 4–8 in Algorithm 9 is not modelled straightforwardly. This loop sets the `busy` bit of the object if the `forwarded` bit is not set. Since the collector may set the `forwarded` bit while the mutator is executing

```
1  inline getObjectHashCode(o, r) {
2    int tmp;
3    hashByAddress(o, tmp);
4    if
5    ::IS_HASHED(tmp) -> r = tmp
6    ::else -> r = hashcode[tmp]
7    fi
8  }
9
10 inline hashByAddress(o, r) {
11   int pseudo;
12   do
13   ::if
14     ::!IS_UNHASHED(o) -> r = o; break /* return */
15     ::else -> skip
16     fi;
17     if
18     ::!IN_GC_CYCLE() || !IN_FROM_SPACE(o) ->
19       setHashed(o);
20       r = o; break /* return */
21     ::else -> skip
22     fi;
23     metaLockObject(o, pseudo);
24     if
25     ::IS_UNHASHED(pseudo) -> setHashed(pseudo)
26     ::else -> skip
27     fi;
28     metaUnlockObject(pseudo);
29     r = pseudo; break /* return */
30   od
31 }
```

Model 16: Hashing: the `hashCode` method.

```
1  inline metaLockObject(o, oo) {
2    if
3    ::!IN_GC_CYCLE() || !IN_FROM_SPACE(o) ->
4      oo = o
5    ::else ->
6      if                                        /* model of loop with CAS */
7      ::forwarded[o] -> oo = FWD(o)
8      ::atomic{
9          (!forwarded[o] && busy[o] == 0) -> busy[o] = 1
10       };
11       oo = o
12     fi
13   fi
14 }
15
16 inline metaUnlockObject(o) {
17   if
18   ::!IN_GC_CYCLE() || !IN_FROM_SPACE(o) -> skip
19   ::else -> busy[o] = 0
20   fi
21 }
```

Model 17: Hashing: meta-locking.

this loop, testing of the `forwarded` bit and setting the `busy` bit are performed atomically by using a CAS. In the model, we used the `atomic` block rather than iterating a loop.

5.6 Collector and Phase Advance

The collector performs GC continuously. The collector performs some actions in each GC phase. Model 18 shows the model of the collector.

In our model, GC consists of four phases.

- In the `MARK_ALLOC` phase, the collector copies the object header. First, the collector sets the `busy` bit. Then, it sets up the hashing state `hashState` depending on the hashing state of the *fromspace* object. Finally, it sets the `forwarded` bit and clears the `busy` bit.
- In the `COPY` phase, the collector copies the contents of the object. However, our model of hashing does not deal with the contents. Thus, the collector model does nothing in this phase.
- In the `FLIP` phase, the collector flips local variables of mutators.
- In the `RECLAIM` phase, the collector swaps the roles of spaces.

```
1  inline collection()
2  {
3    int i = 0;
4    int o = currentFromSpace;                          /* the live object */
5
6    advancePhase(MARK_ALLOC);
7    atomic{!busy[o] -> busy[o] = 1};                   /* assume CAS succeeds */
8    forwarded[FWD(o)] = 0;
9    if
10   ::(hashState[o] == UNHASHED)-> hashState[FWD(o)] = UNHASHED
11   ::(hashState[o] == HASHED) ->  hashState[FWD(o)] = MOVED;
12                                  hashcode[FWD(o)] = o
13   ::(hashState[o] == MOVED) ->   hashState[FWD(o)] = MOVED;
14                                  hashcode[FWD(o)] = hashcode[o]
15   fi;
16   forwarded[o] = 1;
17   busy[o] = 0;
18
19   advancePhase(COPY);
20
21   advancePhase(FLIP);
22   i = 0;
23   do
24   ::(i == N_MUTATORS) -> break
25   ::else ->
26     root[i] = FWD(root[i]);                          /* flip mutator's root */
27     i = i + 1
28   od;i = 0;
29
30   advancePhase(RECLAIM);
31   currentFromSpace = 1 - currentFromSpace;
32
33   advancePhase(NOGC)
34 }
```

Model 18: Hashing: the collector

```
1  inline advancePhase(newPhase) {
2    atomic {
3      g_phase = newPhase;
4      i = 0;
5      do
6      ::(i == N_MUTATORS) -> break
7      ::else -> m_phase_behind[i] = 1; i++
8      od;
9      i=0;
10   };
11   do
12   ::atomic{
13       i = 0;
14       do
15       ::(i == N_MUTATORS) -> i = 0; goto phase_changed
16       ::(i < N_MUTATORS && !m_phase_behind[i]) -> i++
17       ::(i < N_MUTATORS && m_phase_behind[i]) -> break
18       od;
19       i = 0
20     }
21   od
22 phase_changed:
23 }
```

Model 19: Hashing: advancing the GC phase

In addition to these phases, the `NOGC` phase represents the time when the GC is not running.

After the collector advances the GC phase, the collector waits for the mutators to change their mutator phases. The collector calls `advancePhase` in Model 19 to advance the GC phase. Because the collector consists of many phases, we did not model mutator phases directly. Each mutator model has a bit `m_phase_behind`[i] indicating that the phase of the mutator i is behind the GC phase. Each mutator changes its mutator phase with

```
3  ::atomic{m_phase_behind[id] -> m_phase_behind[id] = 0}
```

in Model 15.

5.7 Results

This model checking showed that our handling of `Object.hashCode` meets the requirement of the Java language. More precisely, we checked that our handling has the property that whenever and whichever mutator calls `Object.hashCode` on a particular object, the return values are the same; we did not find any error.

Although our setting of model checking was limited, it took into account those parts of the algorithm that we think subtle, listed below.

- Our model has a collector and multiple mutators so as to check there are no cases where different mutators have different views of the hash code of an object.
- Our model performs the entire GC cycle multiple times because mutators may call `hashCode` anytime. Especially, our model has ragged phase changes, and checking showed that calling `hashCode` during phase change did not cause any problems.

Most collector's routine that deals with the hash code and mutator's `hashCode` method are modelled straightforwardly and the correspondence between the algorithm and the model is clear. From this experience, we greatly increased our confidence in the correctness of our handling of `Object.hashCode`.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: a tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] Samuel Z. Guyer and David Grove, editors. *13th ACM SIGPLAN International Symposium on Memory Management*, Edinburgh, June 2014. ACM Press.

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

[4] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying GC without stopping the world. In *Joint ACM-ISCOPE Conference on Java Grande*, pages 48–57, Palo Alto, CA, June 2001. ACM Press.

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

[6] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, June 2013.

[7] Alexander Linden and Pierre Wolper. An automata-based symbolic approach for verifying programs on relaxed memory models. In *SPIN*, volume 6349 of *LNCS*, pages 212–226, 2010.

[8] Carl G. Ritson, Tomoharu Ugawa, and Richard Jones. Exploring garbage collection with Haswell hardware transactional memory. In Guyer and Grove [2], pages 105–115.

[9] Tomoharu Ugawa, Richard Jones, and Carl G. Ritson. Reference object processing in on-the-fly garbage collection. In Guyer and Grove [2], pages 59–69.

[10] Heike Wehrheim and Oleg Travkin. TSO to SC via symbolic execution. In *Proc. of HVC*, volume 9434 of *LNCS*, pages 104–119, 2015.