Transactional Sapphire: Lessons in High Performance, On-the-fly Garbage Collection

TOMOHARU UGAWA, Kochi University of Technology, Japan
CARL G. RITSON, University of Kent, UK
RICHARD JONES, University of Kent, UK

Constructing a high-performance garbage collector is hard. Constructing a fully concurrent 'on-the-fly', compacting collector is much more so. We describe our experience of implementing the Sapphire algorithm as the first on-the-fly, parallel, replication copying, garbage collector for the Jikes RVM Java virtual machine. In part, we explain our innovations such as copying with hardware and software transactions, on-the-fly management of Java's reference types and simple, yet correct, lock-free management of volatile fields in a replicating collector. We fully evaluate, for the first time, and using realistic benchmarks, Sapphire's performance and suitability as a low latency collector. An important contribution of this work is a detailed description of our experience of building an on-the-fly copying collector for a complete JVM with some assurance that it is correct. A key aspect of this is model checking of critical components of this complicated and highly concurrent system.

CCS Concepts: • **Software and its engineering → Runtime environments**; *Object oriented languages*;

Additional Key Words and Phrases: Concurrent garbage collection, Replicating garbage collection, Parallel garbage collection, Transactional memory, Model checking, Reference objects, Java

ACM Reference Format:
Tomoharu Ugawa, Carl G. Ritson, and Richard Jones. 2018. Transactional Sapphire: Lessons in High Performance, On-the-fly Garbage Collection. *ACM Trans. Program. Lang. Syst.* 1, 1 (June 2018), 57 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The last decade has seen two significant changes to the environments in which software is developed and deployed. First, application developers have increasingly turned[1] to managed languages running on top of virtual machine environments such as the Java Virtual Machine (JVM) or .NET. Managed runtimes offer developers advantages of easier deployment and increased security, not least through automatic memory management or 'garbage collection' (GC), which not only eliminates whole classes of bugs but also simplifies the construction of concurrent algorithms [Herlihy and Shavit 2008] and facilitates composition of software modules [Jones et al. 2012]. Second, constrained by physical limits and energy concerns, manufacturers' strategies have turned from uniprocessors with ever-increasing clock speeds to multi- and many-core processors. The consequences of these two changes are that application programmers have had to develop multi-threaded code in order to

[1]http://www.tiobe.com/index.php/content/paperinfo/tpci/

Authors' addresses: Tomoharu Ugawa, Kochi University of Technology, Japan, ugawa.tomoharu@kochi-tech.ac.jp; Carl G. Ritson, University of Kent, UK, carl.ritson@gmail.com; Richard Jones, University of Kent, UK, r.e.jones@kent.ac.uk.

0164-0925/2018/6-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

realise the performance gains potentially offered by this new hardware, and that virtual machines have had to make efficient and correct use of these parallel resources.

Many modern applications are sensitive to response time. A mobile device user will be dissatisfied unless her applications respond crisply. Enterprise applications must handle highly concurrent workloads without pausing transactions: delays may lead to a backlog of re-tried transactions or direct financial loss. Embedded systems may have hard real-time requirements: all operations must complete within a fixed time.

Stop-the-world garbage collectors stop all user program (*mutator*) threads in order to reclaim unused memory. Although performance can be improved further by deploying multiple collector threads (*parallel collection*), stop-the-world collection leads to unacceptable pause times: several seconds or more for very large heaps. Pause times can be reduced by allowing mutator and collector threads to execute in parallel (*concurrent collection*). However, almost all such collectors temporarily halt *all* mutator threads in order to scan their roots (in Java: statics, thread stacks, registers) or to change phases of a collection cycle; this can lead to significant delay in contexts where there are large numbers of threads [Jones and King 2005]. In contrast, an *on-the-fly* collector need stop only one mutator thread at a time.

Contributions

In this article, we explore new design choices and optimisations for *Sapphire* [Hudson and Moss 2001, 2003], a general purpose collection algorithm designed to support soft real-time applications running a large number of mutator threads on small- to medium-scale, shared memory, multiprocessors. To avoid fragmentation, Sapphire uses replication-based copying [Nettles et al. 1992], with collector and mutator threads collaborating to construct a compacted replica of live data structures. Building on and extending our earlier work [Ritson et al. 2014; Ugawa et al. 2014], we introduce a number of optimisations including the use of *transactional memory* (both software and hardware), *faster object copying* and a *simpler yet sound treatment of volatile fields*. We introduce a *general framework for on-the-fly, concurrent and parallel GC* for a widely used Java virtual machine, Jikes RVM/MMTk, on which we implemented the Sapphire algorithm.

Hudson and Moss built Sapphire on Intel's Open Run-time Platform [Cierniak et al. 2005] simply to validate their algorithm. However, their implementation was neither parallel nor tuned, nor was its performance evaluated thoroughly against a range of realistic benchmarks. We provide a *detailed account of the challenges* faced when implementing Sapphire in a new context — a high performance JVM — exposing both underlying (and not previously apparent) stop-the-world assumptions in Jikes RVM, and some subtleties of on-the-fly, copying collection in general and Sapphire in particular. Several conferences have introduced artifact evaluation to probe claims made in papers by *replicating* (repeating exactly the same) experiments in the same context (typically an operating system virtual machine). However, *reproduction* studies — made independently, in different contexts or using different data — are essential to validate or refute prior findings [Vitek and Kalibera 2011]. Their importance to computer science is now starting to be recognised, not least through the efforts of the Reproducible Research Planet project[2], the Evaluate Collaboratory[3] and the European Conference on Object-Oriented Programming which has included reproduction studies in its call for papers since 2014. We evaluate our implementation rigorously [Kalibera and Jones 2013] using the widely adopted DaCapo benchmark suites [Blackburn et al. 2006], and find that it offers high throughput and sub-millisecond responsiveness.

[2]http://www.rrplanet.com
[3]http://evaluate.inf.usi.ch

Ben-Ari [1984] described Dijkstra's simple concurrent, mark-sweep algorithm [1978] as "one of the most difficult concurrent programs ever studied". In contrast, our Sapphire implementation is a concurrent copying collector for a complete Java virtual machine that supports multiple threads, locking, hashing, finalisation, weak references, reflection, calls to native methods, and so on. Implementing and debugging any high performance collector is hard; it was clear from the outset that an ad hoc approach to designing, constructing and testing this particularly large and complex, concurrent system would not work. We demonstrate how a *systematic approach* using tricolour abstractions, specifying and enforcing invariants, and model checking critical/subtle parts of the system can give confidence in the correctness of the algorithms used.

Much of our approach is generic rather than Sapphire- or Jikes RVM-specific. We believe that our experience will help illuminate some of the challenges facing developers of highly concurrent garbage collectors, and provide guidance towards their solutions. In summary, our contributions are:

- We provide, for the first time within Jikes RVM's MMTk memory manager toolkit, a framework for on-the-fly GC (Section 5).
- Using this framework, we construct a complete, open source, on-the-fly collector that uses the Sapphire algorithm to manage replicated spaces (Section 6) and a mark-sweep algorithm for non-moving spaces. Our implementation uses parallel GC threads, in contrast to the original Sapphire. We focus mainly on the replicating collector here since the on-the-fly mark-sweep collector introduces few further complexities.
- We show how to deal with contention for object headers introduced by tracing, locking and hashing (Section 7), and introduce a simpler yet correct, lock-free way to handle accesses to volatile fields, thus making a step towards fully supporting programs that use fine-grain synchronisation. (Section 8).
- We implement and evaluate transactions for object copying, both in software and using the new hardware support for transactions provided by Intel's Haswell processors (Section 9).
- We extend Jikes RVM/MMTk to manage reference types in the context of on-the-fly GC in order to comply with the Java specification (Section 10).
- We show how to assure the correctness of a complex, concurrent system through specifications of abstractions and invariants, and model checking of critical components and actions, such as concurrent copying, phase changes, reference object processing and object hashing (Section 12).
- We thoroughly evaluate Sapphire's performance (Section 13). Our implementation combines sub-millisecond response times for periodic tasks with low execution time overheads.

2 BACKGROUND

2.1 Garbage collection

The goal of any collector is to preserve (at least) all objects that are *reachable* by following a chain of references from the program's *roots* (pointer variables in stacks and registers, global variables such as Java statics, etc), and to reclaim the space used by unreachable objects.

Many virtual machines segregate objects of different characteristics into different heap regions (*spaces*), managed with different allocation and collection algorithms. For example, MMTk uses a *Treadmill* collector [Baker 1992] to manage a *'large object' space*, and a mark-sweep collector to manage a non-moving space that holds internal VM data structures and compiled code. Collectors that never move objects are vulnerable to fragmentation: although the total free space available in the heap may be sufficient, insufficient contiguous free space to satisfy an allocation request can be found. The severity of this problem can be diminished but not eliminated by allocating objects from *segregated free-lists* [Jones et al. 2012, Chapter 7.4] or in separate large object spaces [Chapter 10.1]. A better solution is to have the collector move objects to 'squeeze out' unusable free space. *Copying* collectors divide the heap into two semi-spaces, *fromspace* and *tospace*. Objects are allocated linearly into *fromspace* until it is full. At that point, the collector is invoked to evacuate all live objects from *fromspace* to *tospace*, after which the roles of the two spaces are '*flipped*'. To preserve the topology of pointer structures in the heap, the collector records a *forwarding pointer*, typically in the header of each live *fromspace* object evacuated, indicating the address to which it has been moved.

2.2 Assuring complex concurrent systems

Any complex, concurrent system is challenging to construct correctly. Garbage collectors are notoriously difficult to debug as any error may not be observed until millions of instructions later. Small timing differences affect when collections occur and which objects survive. Timing-dependent races mean that testing alone cannot provide a guarantee of correctness of a new collector (and, in any case, any system as large and complex as the host Jikes RVM will be far from bug-free).

Abstraction is a successful way to deal with complexity. The *tricolour abstraction* [Dijkstra et al. 1978], which we explain below, characterises the collector's knowledge of objects in terms of three colours. Correctness is assured by enforcing invariants relating the colours of objects that reference each other. As a concurrent collector switches from one phase to another, the invariants to enforce may change. On-the-fly collection makes this particularly subtle since it does not halt all mutator threads simultaneously in order to switch phases. Instead, *ragged phase changes* allow each mutator to recognise the change independently. Thus, collection state changes do not appear atomic: different mutators may be enforcing different invariants which must not clash with each other. In Section 5.3 we introduce two novel design patterns that we believe are sufficiently general to support all types of ragged phase change that a collector might use.

Sapphire offers numerous opportunities for races as objects are copied. Both mutators and collectors access object header words for purposes of locking, hashing and copying; their interaction is fragile (Section 7). The handling of concurrent updates to object fields is delicate; in Section 9 we introduce new object copying mechanisms that are more efficient than those of the original Sapphire (which we outline in Section 3). Processing Java's reference types is also subtle (Section 10). In all these cases, we found informal reasoning to be inadequate to assure correctness of our implementation. However, bounded model checking with the SPIN model checker was straightforward to construct and quick to assure the correctness of our solutions (Section 12).

2.3 Coherency

It is essential that mutators and collectors share a view of the heap that is (eventually) coherent [Wilson 1994]. In a stop-the-world context this is trivial, but it is more difficult in a concurrent setting, whether objects are moved or not. For a non-moving collector, the requirement is that the mutator should not mutate the graph of objects behind the back of the collector. Moving collectors add the further constraint that the collector should not move objects without the mutator's knowledge [Pirinen 1998]. The solution to both problems is to have the compiler emit additional code when object fields are read (*read barriers*) or written (*write barriers*), in order that the system satisfy two properties: [*Safety*] the collector retains (at least) all reachable objects, and [*Liveness*] the collector eventually terminates.

Correctness of concurrent collectors is most easily reasoned about by considering invariants based on the tricolour abstraction that both collector and mutator must preserve. Here, tracing collection partitions the object graph into black (considered live) and white (dead) objects. At the start of a collection cycle, every object is *white*; when an object is first encountered during tracing it is coloured *grey*; when it has been scanned and all its children identified, it is coloured *black*. Conceptually, an object is black if the collector has finished processing it, and grey if the collector knows about it but has not yet finished processing it (or needs to process it again). The collector progresses by advancing through the heap a wavefront of grey objects that separates black from white objects, until all reachable objects have been traced black. At the end of the trace, no references from black to white (unreachable) objects remain, so white ones can safely be reclaimed.

Concurrent mutation of objects while the collector is advancing the grey wavefront may destroy this invariant. Objects can become lost only if two conditions hold at some point during tracing [Wilson 1994]: (i) the mutator stores a pointer to a white object (a 'white pointer') into a black object, and (ii) all paths from any grey objects to that white object are destroyed. Hence, safety requires that both conditions do not hold simultaneously. This leads to two alternative tricolour invariants: [*Strong invariant*] there are no pointers from black objects to white objects, or [*Weak invariant*] any white object pointed to by a black object is reachable from some grey object, either directly or through a chain of white objects. Colour can be extended to mutators [Pirinen 1998]. A *grey mutator*'s roots can refer to objects of any colour under either invariant. Under the strong invariant a *black mutator*'s roots cannot refer to white objects; it cannot allocate objects white under either invariant (since there are no other references to a new object). These properties have consequences for the termination of a garbage collection cycle, which we discuss below.

2.4 Barriers

Incremental update techniques preserve the strong invariant through a mutator *insertion write barrier* that prevents the insertion of a white pointer in a black object (thus bridging the wavefront) by colouring the white object grey (*shading*) or reverting the colour of either the target or the source of the pointer, depending on the algorithm [Jones et al. 2012]. The original Sapphire algorithm used an insertion barrier in its Mark phase. In contrast, *deletion write barrier* techniques preserve the weak invariant by shading the target. This protects against deletion of a pointer to a white object that may now be reachable only from a black object (and so would not be traced). In effect, deletion barriers capture a snapshot of the heap at the beginning of a collection cycle. Although deletion barriers may allow some garbage to 'float' to the next collection cycle, they lead to simpler termination.

All grey mutator techniques use an insertion write barrier. Black mutator techniques most commonly use a deletion write barrier to preserve the weak invariant; it is possible to use a read barrier to maintain the strong invariant, but these have higher overheads than write barriers [Yang et al. 2012].

2.5 Initialisation and termination

Initialisation and termination require care. We discuss termination first because it is simpler. With a black mutator, the collector terminates when no grey objects remain in its work list. At this point, even with the weak tricolour invariant, the mutator can hold only black references. Because no grey objects remain, all white objects must be unreachable. Because the mutator is black, there is no need to rescan its roots. Termination for a grey mutator is more complicated, since the mutator may acquire white pointers after its roots were initially scanned, and therefore must be rescanned before a collection cycle can terminate. If rescanning reveals any objects that are not black, these must be added to the collector's work list and the collection cycle must continue, and so on. Allocating new objects white may drag out termination; allocating black avoids this but at the cost of wasted space since it defers freeing any newly allocated object that becomes unreachable to the next collection cycle.

Initialising on-the-fly collection requires more care than for mostly-concurrent collection. A common approach for mostly-concurrent collectors, which stop all threads together at the start of a collection to scan their stacks, is to use a deletion barrier with a black mutator and to allocate new objects black. However, this is insufficient for on-the-fly collectors. Because they scan mutator stacks on the fly, some stacks may not yet be scanned (white) while others (black) have been scanned. For efficiency, barriers are not triggered on stack operations, leading to the possibility of supposedly black mutators loading references to white objects [Jones et al. 2012, Chapter 6.5]. The simplest solution is to use an insertion barrier with a grey mutator, despite the termination problems discussed above. In contrast, we combine both kinds of barrier: initially we use an insertion barrier to reduce the volume of floating garbage before switching to a deletion barrier for guaranteed termination. We give details in Section 6.2.

3 THE ORIGINAL SAPPHIRE ALGORITHM

Many applications need to complete tasks within a short period of time, and to do so with a high degree of predictability. For some 'hard' real-time applications, for example safety critical systems, any failure to meet a deadline is unacceptable. However, 'soft' real-time applications can tolerate some deviation from their responsiveness goal, provided it is rare [Printezis 2006]. Sapphire is a fully concurrent, replicating collector, designed to support soft real-time applications running a large number of mutator threads on small- to medium-scale, shared memory, multiprocessors. We first give an overview of the original algorithm here; full details can be found in Hudson and Moss [2001; 2003].

The original Sapphire algorithm assumes that data races occur only on volatile fields. Hudson and Moss [2003] argue that "data race accesses to non-volatile fields are not very useful, because they do not have strong enough synchronization and ordering properties". Worse, the Java Language Specification treats a single write to a non-volatile `long` or `double` value as two separate writes, one to each 32-bit half [Gosling et al. 2015, Section 17.7]. Thus, a thread may see the first 32 bits of a 64-bit value from one write, and the second 32 bits from another write. Data races are a rich source of bugs; Boehm and Adve [2008] insist that "good programming practice dictates programs be correctly synchronized or data-race-free", and indeed in all C/C++, Ada or Posix threads programs, data races are errors. Yu et al. [2005] believe that "data races almost always indicate a programming error and such errors are notoriously difficult to find and debug, due to the non-deterministic nature of multi-threaded programming". There is a large body of research on tools to discover and remove races (for example, Hong and Kim [2015]). Other than the Java language feature mentioned above, data races in programmes running over Sapphire are type-safe, and at worst may leave an old value in a non-volatile field.

To avoid unacceptable pauses, Sapphire has no global synchronisation phase in which all mutator threads must be stopped. Rather, it synchronises with each mutator thread independently, for example stopping it to scan its roots. It constructs a replica in *tospace* of the *fromspace* object graph while the mutator threads are running. The content of both pointer and non-pointer fields must be kept coherent, but synchronisation between mutator and collector is expensive. Sapphire places most of this overhead on the collector in order to avoid unpredictable mutator slowdowns as much as possible. Replicas are kept loosely coherent by relying on the Java memory model [Gosling et al. 2015]. Mutators update both *fromspace* and *tospace* copies of an object, whenever both exist. Read barriers are required only for pointer equality tests and reading volatile fields. Updates require neither expensive locks nor atomic instructions.

3.1 Collector phases

Sapphire operates in two main groups of phases (Algorithm 1), first to construct *tospace* replicas and then to 'flip' the roots incrementally from pointing to *fromspace* objects to pointing to *tospace*. Outside GC, mutators access *fromspace* objects. The Flip phases switch mutators to operating in

Algorithm 1. Phases in Hudson and Moss's Sapphire algorithm.

```
MarkCopy:
    PreMark                                       /* install Mark phase write barrier */
    RootMark                                          /* blacken global variables */
    HeapMark/StackMark                            /* process collector mark queue */

    Allocate                                            /* allocate tospace shells */

    PreCopy                                       /* install Copy phase write barrier */
    Copy                                  /* copy fromspace contents into tospace shells */

Flip:
    PreFlip                                       /* install Flip phase write barrier */
    HeapFlip                              /* flip all heap fromspace pointers to tospace */
    ThreadFlip                                                   /* flip threads */

Reclaim                                                       /* reclaim fromspace */
```

tospace. Thus, each phase requires different write barriers. For instance, during the Mark phases the write barrier queues a *fromspace* target for replication, but during the Copy phase it propagates the write to the *tospace* replica. We explain the basic operation of each phase here, deferring a discussion of the subtleties of phase transition to Section 5.

The MarkCopy phase group marks every reachable *fromspace* object, then allocates an empty 'shell' for it in *tospace*, and finally copies its contents to its *tospace* shell. In this phase group, new objects are allocated black, in *tospace*. The first, PreMark, phase installs the mark phase insertion barrier shown in Algorithm 2(a). With this barrier, a mutator storing a reference to an unmarked *fromspace* object adds that object to its own local queue without synchronisation. Note that MMTk provides each mutator or collector thread with its own queues in order to minimise synchronisation overheads; periodically these can be 'flushed' to global, i.e. shared, queues. Queued objects are implicitly grey. The RootMark phase scans and blackens global variables, enqueuing their referents with $\texttt{Write}_{\texttt{Mark}}$. The HeapMark/StackMark phases process the collector's queue and thread stacks. Any unmarked object is blackened by scanning its slots and enqueuing their unmarked referents for marking.

The original Sapphire algorithm used the usual termination process for incremental update write barriers: the mark queue and the set of grey objects must both be empty, and the collector must have scanned every thread's stack without finding any further pointers to white or grey objects before it can terminate. Termination relies on the write barrier keeping globals and newly-allocated objects black, and preventing mutators writing white references into the heap.

After the set of reachable objects has been determined, the Allocate phase creates an empty 'shell' in *tospace* for each reachable *fromspace* object, copying monitor lock information to the shell and placing a forwarding pointer to the shell in the *fromspace* object; we discuss object header formats and forwarding pointers for our revised algorithm in Section 7.1. Because it will also be necessary in the Flip phases to discover the *fromspace* object from its *tospace* replica, Allocate also constructs a hash table for this reverse mapping.

Once the *tospace* shells have been constructed, the contents of *fromspace* objects are copied to their *tospace* replicas in the Copy phase. It is necessary to ensure that *tospace* copies are up-to-date with respect to all writes and that *tospace* objects refer only to *tospace* objects; we call

Algorithm 2. Hudson and Moss's Sapphire write barriers.

```
Write$_{Mark}$(p, f, q):                          /* p.f = q */
    p[f] ← q
    if isFromSpace(q) && not marked(q)            /* white */
        enqueue(q)                                /* collector will mark later */
```

(a) Mark phase barrier.

```
Write$_{Copy}$(p, f, q):                          /* p.f = q */
    p[f] ← q                                      $
    pp ← toAddress(p)                             $
    if pp ≠ null                                  /* p is in fromspace */
        if isPointer(q)
            q ← forward(q)
        pp[f] ← q                                 $
```

(b) Copy phase barrier.

```
Write$_{Flip}$(p, f, q):                          /* p.f = q */
    if isPointer(q)
        q ← forward(q)
    p[f] ← q
    pp ← toAddress(p)
    if pp ≠ null                                  /* p is in fromspace */
        pp[f] ← q
        return
    pp ← fromAddress(p)
    if pp ≠ null                                  /* p is in tospace */
        pp[f] ← q
        return
```

(c) Flip phase barrier.

```
forward(p):                                       /* p is a non−null pointer */
    pp ← toAddress(p)                             /* pp is null if p is in tospace */
    if pp = null
        pp ← p
    return pp
```

(d) Pointer forwarding.

this a 'semantic copy'. To maintain this invariant, PreCopy installs a new write barrier, Write$_{Copy}$, Algorithm 2(b); here, toAddress(p) returns the forwarding address for p, or null if p is not in *fromspace*, and forward(q), defined in Algorithm 2(d), returns the forwarding address if q is in *fromspace* or q otherwise. Memory accesses marked with $ must be performed in the specified order — we assume that data dependencies imply ordering — but otherwise the barrier is unsynchronised because Sapphire assumes there are no mutator-mutator races on non-volatiles.

Algorithm 3. Hudson and Moss's Sapphire collector's word copying procedure. Note that they omitted the lines marked with a **#** although these are necessary to deal with some `StoreConditional` failures.

```
 1   copyWord(p, q):
 2       for tries ← 1 to MAX_RETRY do                         /* times to try non−atomic loop */
 3           toValue ← *p                                                                  $
 4           if isPointer(toValue)
 5               toValue ← forward(toValue)
 6           *q ← toValue                                                                  $
 7           fromValue ← *p                                                                $
 8           if toValue = fromValue
 9               return
10 #     loop
11           LoadLinked(q)                                                                 $
12           toValue ← *p                                                                  $
13           if isPointer(toValue)
14               toValue ← forward(toValue)
15           success ← StoreConditionally(q, toValue)                                      $
16 #         if success
17 #             return                                   /* StoreConditionally succeeded */
```

In the Copy phase, the collector copies the contents of each black *fromspace* object to *tospace*, using the lock-free synchronisation shown in Algorithm 3 to resolve races with the mutator. `copyWord` attempts to copy a word without synchronisation (up to some limit) before resorting to atomic operations. Hudson and Moss assumed that the `StoreConditional` could fail only if the value in the replica had changed during the copy. In this case, assuming no races between mutators, some mutator has written the up-to-date value into the replica. However, implementations of `StoreConditional` generally may fail for other reasons — for example, after a context switch on ARM. The solution is to repeat the atomic update in a loop until it succeeds (the lines 10–17 marked with a #). Of course, this gives no guarantee of progress.

Algorithm 4. Hudson and Moss's Sapphire pointer equality test.

```
flipPointerEQ(p, q):
    pp ← forward(p)
    qq ← forward(q)
    return pp = qq
```

Up to this point, mutators have been operating in *fromspace* (but replicating *fromspace* writes into *tospace*). The Flip phases flip all *fromspace* references to *tospace*. First, the PreFlip phase installs the $\texttt{Write}_{\texttt{Flip}}$ barrier, Algorithm 2(c). As roots are flipped one by one, this barrier must cope with mutators operating in both spaces, unlike other concurrent copying collectors which impose a *tospace* [Baker 1978] or *fromspace* [Nettles and O'Toole 1993] invariant. The HeapFlip phase flips all references held in global variables and new objects. The $\texttt{Write}_{\texttt{Flip}}$ barrier prevents this work being undone by ensuring that only *tospace* pointers are written. The ThreadFlip phase stops threads one by one, and flips any *fromspace* pointers in their stacks or registers. In Flip phases, mutators must still update both replicas, so must be able to discover *fromspace* copies using the hash table constructed in the Allocate phase. Note that, in Flip phases, pointer equality tests must cope with comparisons of *fromspace* and *tospace* pointers (Algorithm 4). Once the ThreadFlip phase has terminated, the Reclaim phase discards the write barrier and the hash table, and reclaims *fromspace*.

4 JIKES RVM AND THE MMTK MEMORY MANAGER

Jikes RVM [Alpern et al. 2002] is a metacircular virtual machine, written almost entirely in Java. Its classes are compiled by an ordinary Java compiler into bytecode, which the Jikes RVM compiler translates into native code, injecting code snippets, also written in Java, for barriers and checkpoints for garbage collection. It does not use an interpreter. Both its ahead-of-time compiler and its adaptive JIT compiler aggressively devirtualise and inline methods. Memory management is provided by MMTk [Blackburn et al. 2004], an independent memory management component that provides a framework for a variety of collectors using the key abstractions of spaces (discussed in Section 2), plans and phases. In MMTk, the set of spaces for a particular configuration is fixed statically. MMTk uses two levels of allocator for spaces other than the large object space. Each mutator thread allocates unsynchronised into its own per-space, thread-local allocation buffer (TLAB). When a thread exhausts its current TLAB, it acquires new one from the space's shared allocator, using a lock.

4.1 Plans

In MMTk, an implementation of a GC algorithm is called a *plan*. A plan is a Java package containing at least the following three main classes.

(1) A singleton **global class** holding global state, especially spaces. Whenever a thread requires a fresh page for new TLAB or for allocation of a large object, the framework asks this class whether to trigger a collection cycle.
(2) A **collector class** representing collector threads. Since MMTk supports parallel collection, multiple collector thread instances are typically created (at boot time, rather than dynamically).
(3) A **mutator class** representing mutator threads. Each algorithm implements an allocation method, using a thread-local allocator and write barriers, if necessary.

Plans form a hierarchy which an implementor can extend to implement a new collector, thereby reusing code. The most basic plan is `Simple` which the `StopTheWorld` plan extends. The `Concurrent` plan also extends `Simple` and provides a base for a concurrent mark-sweep collector. However, we found this plan to be insufficient for on-the-fly collection (see Section 5).

4.2 Phases

A collection cycle typically comprises multiple phases, as for example in Algorithm 1. In MMTk, phases are executed by collector threads. We call MMTK implementations of phases *stages*,[4] and an MMTk plan's global class specifies the list of stages to be executed in a collection cycle. The MMTk scheduler causes these stages to be executed one by one. A barrier synchronisation mechanism ensures that the execution of distinct stages does not overlap. MMTk uses three kinds of stage.

(1) A **global stage** is executed by a single collector thread, mainly to affect global data structures. A selected, master, collector thread will invoke the `collectionPhase` method of the plan's global class. Other collector threads wait for this to complete.
(2) A **collector stage** is executed as a set of parallel tasks, with each collector thread invoking the `collectionPhase` method of the plan's collector class. Typically most collection activities, such as graph traversal and heap scanning, are implemented as collector stages.

[4] MMTk calls a stage a "phase". In this article, we prefer the term "stage" to distinguish it from the broader "phases" used by general garbage collection terminology.

(3) A **mutator stage** manipulates a mutator's private data structures. For load balancing, parallel collector threads compete to claim a mutator and invoke its `collectionPhase`, until all mutators have been processed.

A *complex stage* can be used to bundle a series of stages into a list of sub-stages and a counter indicating the current sub-stage. The execution of the complex stage completes when the last sub-stage completes. The scheduler uses a stage stack to manage stages. However, since the current sub-stage counter is a field of each complex stage, care needs to be taken not to program complex stages recursively.

5 ON-THE-FLY COLLECTION FRAMEWORK

We extended Jikes RVM/MMTk with an on-the-fly *framework* that provides thread-by-thread stack scanning and ragged phase changes, and support for locking and hashcodes (Section 7), and reference type processing (Section 10) in an on-the-fly context. Our *implementation* builds on-the-fly versions of collectors for MMTk spaces over this framework. In addition, it provides the extensive barrier mechanisms needed for replication. Our framework is generic and should support on-the-fly collectors other than Sapphire. For example, our implementation collects non-moving spaces with an on-the-fly mark-sweep GC (although it takes a lock to release freed pages).

5.1 On-The-Fly Collector Framework

MMTk was originally designed as a framework for stop-the-world collection. Although it provides some support for concurrent collectors, we found it insufficient for on-the-fly collection as, for example, it blocks all mutators to change GC phases or to scan roots.

In contrast, our on-the-fly collection framework never blocks all mutators simultaneously. Rather, the controller thread awakens collectors while the mutators are running. Mutators run with barriers during a collection. The framework provides a mechanism to allow the collector to handshake with each mutator, one by one, in order to activate or deactivate these barriers. For the handshake, we introduced a new *on-the-fly mutator stage*. In this stage, collectors request each mutator to perform a task, such as changing the barrier, and wait for it to acknowledge that it has done so. It is important to recognise that on-the-fly collection implies ragged changes of collector phases. A collector developer must keep in mind at all times that some mutators may be running with one barrier and others with another, or none at all. It is the responsibility of the developer to ensure that the invariants required by their algorithm are preserved. We address this in more detail below.

5.2 Stack and global root scanning

Our framework provides new implementations of common phases to support on-the-fly collection. A StackMark phase has each collector thread compete to claim and halt a mutator, scan its stack and resume it. The number of mutators blocked at any time is at most the number of collector threads, typically significantly fewer than the number of CPU cores. We reused existing code from the `StopTheWorld` plan to scan mutators' stacks in parallel, with each GC thread stopping one mutator thread at a time, as it was less complex to implement than letting each mutator scan itself.

A RootMark phase has the collector scan global roots: vectors of references for static variables and Java Native Interface references [Oracle JNI 2015]. All writes to these vectors are expected to be protected by an insertion barrier so mutators do not need to be halted for scanning. RootMark comprises two stages. The first is a global stage in which the master collector measures each vector to balance loads in the following stage where all collectors scan the vectors in parallel. It is essential that the first stage is global since the size of a vector's live area may change as mutators run. If the range of a vector were to be computed asynchronously, some slots may be scanned by multiple collectors and others not at all. Protecting the vectors with a write barrier allows the live area to be enlarged or contracted safely (although contraction may mean that the collector traverses dead objects).

5.3 Ragged Phase Changes

Mostly-concurrent collectors stop all mutators when they change from one collection phase to another (see Figure 1). This ensures that,

- once the collector has entered a new phase B, no mutator observes a GC state S_A corresponding to the previous phase A; and
- at any time, all mutators have a coherent GC state, i.e. all observe S_A or all observe S_B.

In contrast, on-the-fly collectors stop mutators independently. A mutator will not recognise that the phase has been changed until it reaches a GC-safe point. This causes two problems.

(1) The collector cannot start collection work until it has determined that all mutators have recognised that the phase has been changed.
(2) There is a time window in which different mutators observe different GC states.

These are not new problems (see, for example, Doligez and Gonthier [1994]). Our contribution is the introduction of a general *design pattern* that we believe is sufficient to capture all types of ragged phase change. For problem (1), we add an intermediate GC phase between the main phases. In this GC phase, the collector does no collection work other than handshake with each mutator to ensure that it has recognised the change of main phases and changed its local state. A mutator responds to the handshake request at a GC-safe point. Once the collector has completed a handshake with every mutator, it is guaranteed that all mutators have recognised that the GC is going to move to the new phase. We call this type of ragged phase change *Type I*.

Problem (2) recognises the possibility of a mutator-mutator conflict. Those mutators that have not recognised the new phase run with an assumption based on an invariant for the previous phase. This may conflict with the invariant assumed by mutators that have recognised the change. We add two intermediate GC phases, I_1 and I_2, to solve this problem as follows. Before this ragged phase change, mutators respect the invariants of the current GC phase A. The purpose of phase I_1 is for mutators to change their local GC state to an intermediate state, S_I. Typically, a mutator in this intermediate state runs with a more complex barrier that respects the invariants of both main phases A and B, i.e. each mutator is prepared for another mutator acting according to the invariants of GC phase B, but none is doing so yet. For example, we shall see in Section 6 how the collector changes from phase A (no collection) to phase B (marking and creating empty shells). Here, in phase I_1 (PreMark1), a mutator is able to write to black objects safely but only allocates white. At the transition from I_1 to I_2 (PreMark2), the GC knows that all mutators are ready for black objects. In phase I_2, mutators start to allocate new objects black. At the end of phase I_2, the GC knows that all mutators are allocating black, and so can transition to state S_B in phase I_2. We call this type of ragged phase change *Type II*.

In our Sapphire implementation, all phase changes are Type I except for the transitions from ‘no GC’ to the Mark phase (Section 6.2), and from the Copy phase to the Flip phases (Section 6.4).

5.4 Barriers

Our Sapphire implementation uses barriers on write operations and pointer equality comparisons to maintain coherence. As is common practice [Blackburn and McKinley 2002], we implement barriers with an inline fast path (checking a *barrier flag*) and an out-of-line method call that does the actual work. Each mutator has a thread-local barrier flag, a single word with bits indicating which barriers are activated. We use a set of bits because in the Flip phase we activate both a write barrier and a

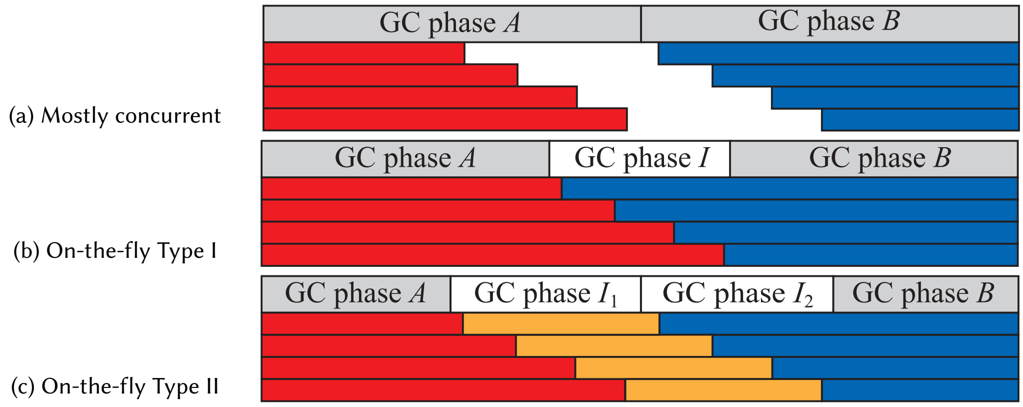


Fig. 1. Ragged phase changes.

pointer-equality barrier. The compiler emits inline code that tests if the barrier flag is non-zero (at least one barrier is activated) and, if so, calls the out-of-line method. Our implementation strategy is friendly to the compiler's inliner, and keeps the compiled code small and hence likely to fit in the code cache.

5.5 Termination Loop

On-the-fly collectors often need to repeat a sequence of tasks before a collection phase can terminate (see Section 2.5). Even with snapshot-at-the-beginning techniques, which keep the mutators' roots black, processing Java's reference types requires iteration to terminate (as we describe in Section 10). In principle, the number of iterations is unbounded although any practical collector will fall back to stop-the-world termination when necessary. Unfortunately, the Jikes RVM/MMTk scheduler — designed with stop-the-world, or at best mostly-concurrent, collection in mind — supports only a fixed sequence of stages. We introduced a construct to realise repetition by tricking the stage stack. We bundle a sequence of stages that need to be iterated plus a global stage into a single complex stage. In the global stage, the collector determines if we need to perform the complex stage again or not. If so, the complex stage is pushed back onto the stack again.

6 ON-THE-FLY SAPPHIRE PHASES: INVARIANTS, BARRIERS AND CORRECTNESS

Over the next five sections we describe our implementation of a fully concurrent, parallel Sapphire within our on-the-fly collector framework for Jikes RVM/MMTk. Construction of high performance garbage collectors is hard. Making a collector any one of copying, parallel, concurrent or on-the-fly increases the complexity enormously. Experience has shown us that an ad hoc approach to design, construction and testing is a recipe for frustration. Instead, we use a methodology based on discovering suitable *abstractions* to model the state to our system. Some of these abstractions are well-known, while others specific to Sapphire phases are new derivations. We identify *invariants* in terms of these abstractions: mutators or collectors must preserve these in each phase. Sometimes the invariants of adjacent phases are incompatible (see Section 5.3). This is problematic since threads may be running in different phases in an on-the-fly collector (we cannot stop all threads at once to change their phase state). Finding these invariant incompatibilities shows where new, intermediate phases need to be introduced to ensure that the invariants that mutators seek to enforce in any Phase n are compatible with those of adjacent Phases n-1 and n+1.

Understanding abstractions and invariants aids the construction of the *barriers* needed to enforce correctness in the context of multiple threads. In some cases, interactions between mutator and collector threads may be particularly complex, especially on relaxed-memory architectures, and here we found *bounded model checking* of our abstract algorithms to be invaluable. This was fairly straightforward to use, and quick to show when specifications do not hold. We discuss model checking in Section 12.

6.1 Collection Phases

Our Sapphire implementation comprises a similar set of phases to the original (Algorithm 1) but merges the Allocate phase into the marking phases. Our phases are shown in Algorithm 5. Most of the phase changes are Type I (Section 5.3), but changing from 'no collection' to the StackMark phase and from the Copy phase to the HeapFlip phase requires Type II phase changes (the two intermediate phases are prefixed with 'Pre' in the Algorithm.)

Algorithm 5. Our Sapphire phases.

```
Mark:
    PreMark1                     /* install Mark phase write barrier */
    PreMark2                     /* toggle allocation colour */
    StackMark                    /* process mutators' stacks */
    RootMark                     /* blacken global variables */
    HeapMark                     /* process collector mark queue */
    ReferenceProcess             /* mark reference types and terminate marking */

Copy:
    PreCopy                      /* install Copy phase write barrier */
    Copy                         /* copy fromspace objects to their tospace shells */

Flip:
    PreFlip1                     /* install the limited self−flip and equality barriers */
    PreFlip2                     /* install the full self−flip barrier */
    HeapFlip                     /* flip non−replicated spaces to point to tospace */
    RootFlip                     /* flip global roots to point to tospace */
    StackFlip                    /* flip mutator stacks to point to tospace */

Reclaim                          /* turn off barriers; reclaim fromspace */
```

Throughout, we impose the invariant that *tospace* slots never point to *fromspace*. This suffices for the Flip phases to scan only roots and non-replicated spaces, but not *tospace*, to flip pointers.

6.2 Mark Phases

The first main phase group is Mark, in which the collector marks (or blackens) all reachable objects. The representation of black objects may differ from one space to another. For mark-sweep spaces, where objects are not moved, the collector sets a mark in the object header. For replicating spaces, the collector creates an empty shell and installs a forwarding pointer (described in Section 7.1).

Throughout the Mark and Copy phases, we maintain the invariant that mutators access *fromspace* objects in the replicated heap. During the main marking phases, we allocate new objects black to ease termination. When allocating an object black, i.e. allocating both *fromspace* and *tospace* replicas, we always allocate the *fromspace* object first, checking if there is enough space for *both* objects. Thus, although a mutator may be blocked during the allocation in *fromspace* until the end of the GC cycle if there is insufficient memory, the subsequent allocation in *tospace* can *never* fail.

Up to the ReferenceProcess phase, we also maintain the strong tricolour invariant that there are no black-white pointers, which we enforce with an insertion barrier. The ragged transition from 'no collection' to the StackMark phase needs a Type II phase change, implemented by the PreMark1 and PreMark2 phases.

PreMark1 activates the insertion barrier shown in Algorithm 6. If the referent `q` of a pointer update `p.f = q` is in *fromspace*, and if it has not been forwarded, i.e. the collector has not created a shell for it, the barrier 'shades'[5] it (enqueues it for copying) so that the collector can create a shell for it. The barrier does not create the shell immediately, because that allocation path might include a GC-safe point and hence a potential phase change. Otherwise, an unmarked object in a non-replicating space is marked and added to a separate work list so that its children can be traced. Note that, to prevent a thread that has not yet activated the barrier from creating a black-white pointer, we do not allocate new objects black until all mutators are known to have activated this write barrier, i.e. until the PreMark2 phase.

Algorithm 6. New Mark phase write barrier.

```
checkAndEnqueue(q):                                        /* shade q */
    if inFromspace(q)
        if not forwarded(q)
            tobeCopiedQueue.enqueue(q)
    else
        if not testAndMark(q)
            worklist.enqueue(q)

Write_Mark(p, f, q):                                       /* p.f = q; */
      p[f] ← q
      checkAndEnqueue(q)
```

In the HeapMark phase, collectors traverse reachable objects, creating shells for them in *tospace*. Roots for the traversals include the `tobeCopiedQueue` filled by the mutator write barrier. Each shell's header (a status word and a back pointer to its *fromspace* replica) is created immediately (see Section 7.1). The back pointer is used in the Flip phases to propagate mutator updates to both copies of an object.

Because we use an insertion barrier, a grey mutator may acquire a reference to a white object after its roots were scanned. Although the collector could rescan each mutator's roots for white objects at the end of the HeapMark phase, we prefer to postpone this termination check until after the ReferenceProcess phase, since it requires the same check because mutators may `Reference.get` white targets of Java's reference type objects. We discuss reference processing in Section 10.

Up to the ReferenceProcess phase, we use the insertion barrier shown in Algorithm 6, not just for safe initialisation of an on-the-fly collection (see Section 2.5), but also because insertion barriers preserve less floating garbage than deletion barriers. However, in this phase, we switch on the deletion barrier shown in Algorithm 7 for quicker termination and a guarantee of progress. We explain why the deletion barrier guarantees progress in processing reference types in Section 10 and discuss performance in Section 13.8. As we switch the barrier, we scan the stack with both the

[5]The term 'shade' evokes the idea of changing white to grey but leaving black unchanged [Jones et al. 2012].

Algorithm 7. ReferenceProcess phase barrier.

```
Write_ReferenceProcess(p, f, q):                          /* p.f = q; */
    checkAndEnqueue(p[f])
    p[f] ← q
    if isFirstStackScan()
        checkAndEnqueue(q)            /* active only while switching barriers */
```

insertion and the deletion barriers active, as per Doligez and Gonthier [1994]. More precisely, a mutator uses both barriers during the first stack scan it performs in the ReferenceProcess phase.

6.3 Copy Phases

All live *fromspace* objects were marked — with *tospace* shells — in the marking phases. In the Copy phase, the collector copies the contents of each *fromspace* object into its shell; mutators continue to allocate new objects black. We choose to scan *tospace* rather than *fromspace* to avoid the overhead of skipping over dead objects. The collector first installs the Copy phase write barrier (shown in Algorithm 8) in the PreCopy phase. Whenever a mutator writes to a *fromspace* object, the write barrier writes the semantically equivalent value to the corresponding fields of its *tospace* copy. We designed the write barrier and the collector's copying procedure so that (i) an update by the mutator cannot be lost even when the collector copies the same object simultaneously, and (ii) the mutator's write will never fail, i.e. mutator writes are wait-free. We explain the design of the Copy phase barrier in Section 9.

Algorithm 8. New Copy phase barrier.

```
Write_Copy(p, f, q):                                      /* p.f = q */
    p[f] ← q
    if inFromspace(p)
        pp = p.forwardingPointer
        if inFromspace(q)
            pp[f] ← q.forwardingPointer
        else
            pp[f] ← q
```

6.4 Flip Phases

In the Flip phases, the collector flips references in non-replicated spaces, global roots and mutators' roots. Figure 2 shows the invariants to be preserved (the paler, broader bands) and how our barriers cause mutators to conservatively respect these invariants (darker, thinner bands). Until the PreFlip1 phase, roots and objects in all spaces (other than *tospace*) do not have *tospace* references: mutators never see *tospace* references. Without this invariant, a read would have to be forwarded to the latest copy of the object by read barriers. From the Heap/RootFlip phases onward (hatched paler and wider background band), mutators never write a *fromspace* reference into a slot of a non-replicated space but instead write its *tospace* counterpart to ensure termination. Without this invariant, the collector would have to scan the non-replicated space repeatedly for *fromspace* references. In consequence, mutators may now see *tospace* references, in conflict with the Copy phase invariant. Thus, once again a Type II phase transition is needed.

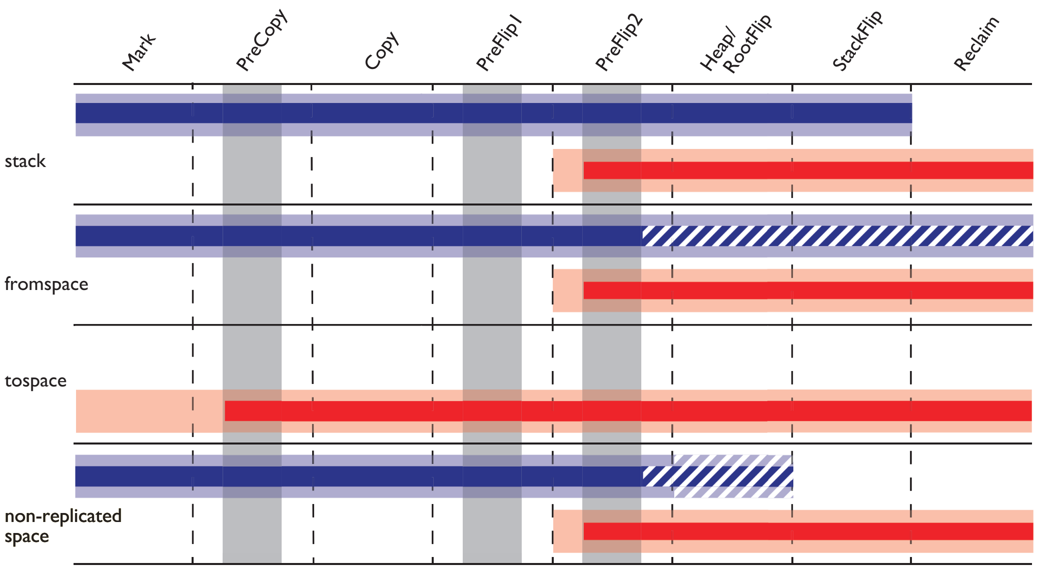


Fig. 2. Mutator barriers respect invariants. Horizontal bands indicate whether spaces can hold *fromspace* (upper, left-hand side bands) or *tospace* (lower, right-hand side bands) references. The hatched bands indicate when no new references can be stored in a space. The paler and wider background bands show the invariants that must be preserved. The darker and narrower foreground bands show how our barriers cause mutators to respect these invariants. Each vertical grey band represents the extent of a handshake.

By the end of the PreFlip1 phase, all mutators are ready to deal with *tospace* references, but they must not yet write *tospace* references. To deal with these, we require three barriers. First, we need a pointer-equality read barrier that yields true when a mutator tests the equivalence of references to the *fromspace* and *tospace* replicas of the same object. Second, we expand the double-update write barrier so that a mutator's write to a *tospace* object is propagated to its *fromspace* replica. Third, we use 'self-flip' write barriers that replace an attempt to write a *fromspace* reference with the corresponding *tospace* reference.

The self-flip barriers are necessary for termination. In the PreFlip1 phase, the mutator installs the pointer equivalence barrier, Algorithm 9(a), and the preliminary, limited self-flip write barrier, Algorithm 9(b), which flips a reference only when writing to *tospace*. Table 1 shows the behaviour of the limited self-flip write barrier. It never writes *tospace* references to any space other than *tospace* unless q is in *tospace*. In PreFlip1 phase, no mutator has access to *tospace*. Thus, the greyed cells in Table 1 never happen.

After all mutators have installed this barrier, the mutator installs the full self-flip write barrier, Algorithm 9(c), in the PreFlip2 phase; from now on as the invariant requires, this mutator never writes a reference to a *fromspace* object, as shown in Table 1(b). Note that we cannot install the full self-flip barrier in PreFlip1 as other mutators may still be in the Copy phase, expecting to see only *fromspace* references.

In the HeapFlip and RootFlip phases, the collector flips the non-replicated spaces and global roots, respectively. Most non-replicated spaces other than the large object space are small so our current implementation scans each one for *fromspace* references to flip. Hudson and Moss [2003] suggested an improvement of using a remembered set to keep track of locations containing references. Another option would be to have separate large object spaces for reference arrays and

Algorithm 9. New Flip phase barriers. Note that the forwardingPointer of a *tospace* object points to its *fromspace* replica.

```
pointerEQ(p, q)                                              /* p == q */
    if p = q
        return true
    if inFromspace(p)
        return p.forwardingPointer = q
    if inFromspace(q)
        return q.forwardingPointer = p
    return false;
```

(a) New pointer equivalence barrier.

```
Write_preFlip(p, f, q):                                      /* p.f = q */
    if inFromspace(q) && inTospace(p)
        q ← q.forwardingPointer
    p[f] ← q
    if inFromspace(p) || inTospace(p)
        pp ← p.forwardingPointer
        if inFromspace(q) || (inTospace(q) && inFromspace(pp))
            pp[f] ← q.forwardingPointer
        else
            pp[f] ← q
```

(b) PreFlip phase write barrier with limited self flipping.

```
Write_Flip(p, f, q):                                         /* p.f = q */
    if inFromspace(q)
        q = q.forwardingPointer
    p[f] ← q
    if inFromspace(p) || inTospace(p)
        pp ← p.forwardingPointer
        pp[f] ← q
```

(c) Flip phase write barrier.

for other objects. At the end of RootFlip, only the mutators' stacks contain *fromspace* references: the collector flips these in the StackFlip phase.

From the StackFlip phase onward, the allocator must return a *tospace* reference (for termination), while still allocating a replica in *fromspace*; we change the behaviour of the allocator to this in PreFlip2 phase to be ready to deal with *tospace* references.

6.5 Reclaim phase

After the StackFlip, mutators no longer hold *fromspace* references. The collector exchanges the roles of *fromspace* and *tospace* (as in any copying collector) so it can release the old *fromspace*. We implemented our replicating collector in a similar way to the existing semi-space copying collector, swapping the roles of *fromspace* and *tospace* at the end of each collection cycle; the collector has a *global flag* indicating which space is currently *fromspace*. In addition, each mutator caches pointers

Table 1. Behaviour of write barriers (a) `Write`$_{preFlip}$ and (b) `Write`$_{Flip}$ for `p.f=q`. These tables indicate the spaces, F(rom) for *fromspace*, T(o) for *tospace*, and N(on) for a non-replicated space, of the sources and destinations of references created by in each assignment in Algorithm 9. An entry "n: $X \rightsquigarrow Y$" indicates that a field of an object in space X points to an object in space Y after the assignment on line n. Column and row labels are spaces of destinations of references p and q. Note that neither p nor q can be *tospace* references in the PreFlip1 phase (indicated by the greyed column and row).

(a) PreFlip phase write barrier.

p \ q	From	To	Non
From	4: F$\rightsquigarrow$F 8: T$\rightsquigarrow$T	4: F$\rightsquigarrow$T 10: T$\rightsquigarrow$T	4: F$\rightsquigarrow$N
To	4: T$\rightsquigarrow$T 8: F$\rightsquigarrow$F	4: T$\rightsquigarrow$T 8: F$\rightsquigarrow$F	4: T$\rightsquigarrow$N
Non	4: N$\rightsquigarrow$F	4: N$\rightsquigarrow$T	4: N$\rightsquigarrow$N

(b) Flip phase write barrier.

p \ q	From	To	Non
From	4: F$\rightsquigarrow$T 7: T$\rightsquigarrow$T	4: F$\rightsquigarrow$T 7: T$\rightsquigarrow$T	4: F$\rightsquigarrow$N
To	4: T$\rightsquigarrow$T 7: F$\rightsquigarrow$T	4: T$\rightsquigarrow$T 7: F$\rightsquigarrow$T	4: T$\rightsquigarrow$N
Non	4: N$\rightsquigarrow$T	4: N$\rightsquigarrow$T	4: N$\rightsquigarrow$N

to its current TLABs for *fromspace* and *tospace*. One reason is for quick allocation. The other is more critical, and more subtle. If the space for *fromspace* were to change at an arbitrary point (especially during allocation), the mutator might be confused. To avoid this, each mutator uses its cached pointer, which is changed only at GC-safe points.

However, there is a time window between when the collector swaps the roles of *fromspace* and *tospace* by toggling the global flag, and when a mutator updates its cached TLABs. It is possible that a mutator thread might die inside this window. To take care of this, threads check the global flag while terminating, in order to give up their TLABs to the collector correctly. For regular allocation, we can always trust the cached pointer because the thread allocates in both spaces in the Flip phase, and the double-update barrier is turned off at the same time as the mutator updates the cached pointer to the TLABs.

7 RACES I: TRACING, LOCKING AND HASHING

Many of the challenges facing developers of on-the-fly collectors come down to dealing safely with races, whether between mutator threads, collector threads or both. We first address the problems of locking and hashing: it is essential that all threads share a consistent view of an object's lock or hash value. A concurrent replicating collector has two options. It must either ensure that locking or hashing information is stored in a single definitive location (not necessarily in either of the replicas), or that both replicas always hold identical information. In common with most JVMs, Jikes RVM uses headers when locking or hashing objects. In this section, we consider how to manage object headers using a hybrid policy. Often it is cheap to maintain identical hashing information in both replicas. Otherwise, and always for locks, we use a lazy *tospace* invariant.

7.1 Object Headers

Each Jikes RVM/MMTk object contains a header of at least two words. One is a type information block (TIB) pointer to a representation of the class of the object. The other is a status word, containing a hash code, monitor lock information and a few bits used by collectors. While the GC bits are independent, hash code and lock information are combined in a thin lock mechanism [Bacon et al. 1998]. Although production Jikes RVM/MMTk provides biased locking [Pizlo et al. 2011], our Sapphire implementation does not yet, but we foresee no problems in doing so. Since Sapphire copies objects while the mutators are running, we cannot overwrite any content of an object with a forwarding pointer as non-concurrent copying collectors do. For simplicity, our implementation adds a third word to each object header to hold the forwarding pointer. We could overwrite the TIB word with the forwarding pointer and use the TIB pointer in the *tospace* replica, but this would complicate method dispatch. Since we need a backward pointer from a *tospace* replica to the *fromspace* original, we need a distinct field (or some side table as per Hudson and Moss [2003]) anyway.

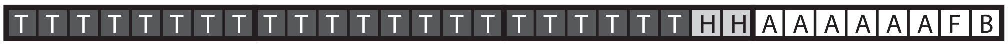

T T T T T T T T T T T T T T T T T T T T T T H H A A A A A A F B

Fig. 3. Jikes RVM object status word bits: thin lock (T), hash state (H), unused bits available (A) for use by the GC, forwarded (F) and busy bits (B).

Figure 3 shows the layout of the status word. The least significant eight bits are available for use by the GC. We use two bits to describe the status of the object; bit 0 is a *busy* bit and bit 1 is a *forwarded* bit. The busy bit is for mutual exclusion of actions that may modify the status or forwarding pointer words. The forwarded bit indicates that the object's *tospace* shell has been created and the forwarding pointer has been installed. Note that this bit is not set in *tospace* replicas so does not need to be cleared after the spaces are flipped. We enforce the following synchronisation policy on the status word.

Status word *tospace* invariant. Although Sapphire avoids read barriers on object fields, we do have to impose a read barrier on status word accesses. Since this word needs to be updated atomically, it would be costly to maintain multiple replicas. Thus, we transfer the status word to the empty shell in *tospace* when it is created. From this point on, the shell holds the up-to-date locking and hashing information. Thus, whereas we observe a *fromspace invariant* for object bodies for most collection phases, we observe a *lazy tospace invariant* for object headers, whereby mutators use the *tospace* copy of any status word if it has been copied, and the *fromspace* copy otherwise. An alternative might be to use transactional memory but the setup cost of Intel's Restricted Transactional Memory is around three times larger than that of a single compare-and-swap (CAS) instruction [Ritson and Barnes 2013]. Fortunately, only a limited set of actions access the status word, so it is feasible to apply a read barrier only to these accesses without overhead on other read operations.

Mutual exclusion while copying. We ensure that only a single empty shell is created for any *fromspace* object by *meta-locking* the busy bit, again as per Hudson and Moss [2003]. Figure 4 shows the transitions of the forwarded and busy bits. Whenever a collector attempts to create an empty shell for an object, it sets the object's busy bit with a CAS. After setting up the status word of the shell, the collector updates the *fromspace* object's status word with a bit pattern of `busy=0` and `forwarded=1`. We remark that a mutator will spin briefly if it attempts to take the meta-lock of an object for locking or hashing while the collector is allocating the empty shell. We discuss progress in Section 11.

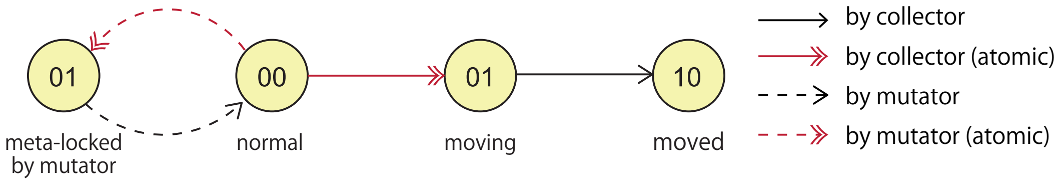


Fig. 4. Status word state transitions with the values of the forwarded (left) and busy (right) bits.

Algorithm 10. Implementation of meta-locking. Our convention is that CAS returns true to indicate that the word has been updated.

```
metaLockObject(o):
    if not inGCCycle() || not inFromSpace(o)
        return o
    do
        status ← o.statusWord
        if (status & FORWARDED) ≠ 0
            return o.forwardingPointer
    while not CAS(&o.statusWord, status, status | BUSY)
    return o

metaUnlockObject(o):
    if not inGCCycle() || not inFromSpace(o)
        return
    o.statusWord ← o.statusWord & ~BUSY;
```

Mutators always take the meta-lock when accessing *fromspace* locking information (but not when accessing the *tospace* lock) during a GC cycle to ensure that they recognise when lock information has been moved to *tospace* under our lazy *tospace* invariant. As an alternative, the mutator could detect that the lock information has been moved from the failure of the CAS at the end of the locking operation. However, we did not take this option because it complicates the interface between MMTk and its client (Jikes RVM). In contrast, we need to meta-lock to obtain the hash code only for unhashed *fromspace* objects (see Section 7.3).

7.2 Implementation of Meta-locking and the Tospace Invariant

Correct synchronisation requires locking and hashing operations to hold the meta-lock or to be forwarded to *tospace*, depending on whether a collection is in progress, the space of the object and the value of the status word. To maintain MMTk as an independent memory management module, we decided to separate these complicated decisions from the core locking and hashing mechanisms provided by the client VM (Jikes RVM). More specifically, we added two methods, `MemoryManager.metaLockObject` and `MemoryManager.metaUnlockObject`, to the MMTk interface. These methods are responsible for forwarding as well as meta-locking.

We designed these interfaces to support a variety of concurrent collectors. Intuitively, `metaLockObject` lends the client VM a *pseudo*-reference. The VM locks and hashes using the pseudo-reference and then hands it back with `metaUnlockObject`. The pseudo-reference points to a location from which the up-to-date status word can be found. However, there is no guarantee that the pseudo-reference refers to a valid object. For example, it may refer to *tospace* before the Flip phases. Therefore, the client VM must not store pseudo-pointers in the heap. Furthermore, before a GC-safe point, when the collector may scan the mutator's roots, the mutator must hand it back with `metaUnlockObject` and null any local variables that contain pseudo-references. We prevent the compiler from emitting a GC-safe point in any method where the mutator still holds a pseudo-reference. Thus, every `metaLockObject` has a matching `metaUnlockObject` in the same phase of GC.

Algorithm 10 shows our implementation of meta-locking. If a reference refers to *fromspace* inside a GC cycle, `metaLockObject` locks the object or forwards the reference depending on the forwarded bit. If the bit is set, `metaLockObject` returns a pseudo-reference to the *tospace* replica; otherwise, it

Table 2. Number of initial v. total number of calls of hashCode. The third columns show the percentage of initialising calls. For antlr, bloat, fop, and jython, the benchmark harness but not its workload called hashCode.

	all	initial	%		all	initial	%
antlr$_6$	646	14	2.2	avrora$_9$	700440	25	0.0
bloat$_6$	646	14	2.2	eclipse$_6$	921674	337920	36.7
fop$_6$	646	14	2.2	hsqldb$_6$	4071	93	2.3
jython$_6$	646	14	2.2	luindex$_9$	2330	55	2.4
lusearch$_9$	13527	1341	9.9	pmd$_9$	329739	104243	31.6
sunflow$_9$	1750	136	7.8	xalan$_9$	59018	558	0.9

returns the given reference as a pseudo-reference; this is the only case where the mutator sets the busy bit. `metaUnlockObject` clears the busy bit only if it was set by `metaLockObject`.

7.3 Locks and hash codes

We implement Java monitors with thin locks [Bacon et al. 1998] and use address-based hashing, i.e. we use the address of an object as its hash code. If the GC were not to move any object, this strategy would be simple and would require no additional space. However, a collector must preserve the hash code when it moves a previously hashed object.

MMTk's moving collectors do this by adding an extra hash code field when the collector first moves a hashed object. To realise this, two bits in the status word record the object's hash state: `UNHASHED`, `HASHED` or `HASHED_AND_MOVED`. An object is born `UNHASHED`. Calling `hashCode` on an `UNHASHED` object returns the address of the object and, at the same time, changes its state to `HASHED`. When the collector allocates a shell for a `HASHED` object (i.e. the first time that the object is copied after it has been hashed), the collector adds an extra word containing the object's hash code (its address at the time it was hashed) to the shell and sets its state to `HASHED_AND_MOVED`. At subsequent collections (when the object's state is `HASHED_AND_MOVED`), this word is copied to the new shell. For `HASHED_AND_MOVED` objects, `hashCode` returns the value stored in this extra field.

Although our default strategy accesses the status word with a CAS or by meta-locking, in common cases we can obtain the hash code without synchronisation. If an object has been forwarded, we can access the status word of its *tospace* replica under the *tospace* invariant as usual. Moreover, if its hash code has ever been observed, i.e., the state is `HASHED` or `HASHED_AND_MOVED`, we can also obtain the hash code without synchronisation (even from a *fromspace* object) because the hash code never changes. This observation can dramatically reduce the number of synchronisations. Table 2 shows the ratio of the number of initial calls of `hashCode` for each object versus the total number of `hashCode` calls, for benchmarks from DaCapo 2006 and 2009 (note that the values shown are characteristics of the benchmarks, independent of any virtual machine implementation). The ratio varies greatly but, generally, most calls are to `HASHED` or `HASHED_AND_MOVED` objects. In summary, mutators only need to meta-lock objects that have no replica and have not been hashed. Algorithm 11 shows our implementation.

As we have seen, treatment of the status word in concurrent collectors is delicate. We found the SPIN model checker (see Section 12.1) invaluable in confirming that our treatment of the status word was correct.

8 VOLATILES

The Java Memory Model [Gosling et al. 2015, Chapter 17] specifies legal executions of a program. Of particular interest are *synchronization actions*, which include reads and writes to volatile variables.

Algorithm 11. Hashing implementation.

```
getObjectHashCode(o):
    o ← MemoryManager.hashByAddress(o)
    if (o.statusWord & HASH_STATE) = HASHED
        return (int) o
    return readHashCode(o)                          /* HASHED_AND_MOVED */

isUnhashed(o):
    return (o.statusWord & HASH_STATE) = UNHASHED

setHashed(Oo):
    o.statusWord ← o.statusWord | HASHED;
```

(a) Hashing methods defined in Jikes RVM.

```
hashByAddress(o):
    if not isUnhashed(o)
        return o
    if not inGCCycle() || not inFromSpace(o)
        setHashed(o)
        return o
    o ← metaLockObject(o)                           /* returns a pseudo−reference */
    if isUnhashed(o)
        setHashed(o)
    metaUnlockObject(o)
    return o
```

(b) Hashing methods defined in MMTk.

Accesses to volatile fields must become visible to other threads essentially in the order given by the code. This presents a challenge to replicating collectors when there are two copies of a volatile field.

Replicating collectors must therefore take steps, often elaborate, to ensure that all accesses to volatile fields are made to a definitive, up-to-date copy. Hudson and Moss [2003, Section 5, pp. 244–254] proposed three fairly complicated designs to achieve this. Unfortunately, none of these support lock-free programs that employ fine-grain synchronisation as all block access to volatile fields while the collector thread is copying them (and thus rely on the operating system's thread scheduler). Furthermore, none of these solutions was implemented or evaluated.

We solve the problem of visibility of actions on volatiles — and take a step towards support for lock-free programs — in a much simpler way, by allocating any object that contains a volatile field in a non-replicated space. Jikes RVM already requires some objects to be allocated in a non-moving space so the addition of objects with volatile fields is trivial to do. One advantage of this approach is the flexibility it offers in support of atomic operations on data as the compiler can select to generate code that allocates in a non-moving space based on any property. We implemented our solution and measured benchmarks from the DaCapo 2006 and 2009 suites; we found the volume of objects with volatile fields to be negligible for all but two applications, *eclipse* and *hsqldb*. And here, even *eclipse*, the heavier user of volatiles, allocated only 1.6MiB of objects with volatile fields. We note in passing that, while this is an effective solution for the benchmarks adopted as a standard by the research community, their very light use of volatiles may raise the question of how representative these benchmarks are of modern, concurrent programs.

9 RACES II: TRANSACTIONAL COPYING

Any concurrent copying GC must deal with races between mutators updating, and collectors copying, an object. We investigate three mechanisms. Our primary target is x86 systems, for which most memory operations are guaranteed to be executed in order: only loads may be reordered with preceding store instructions. Throughout, we assume that all reference fields are word-aligned and sized (currently, 4 bytes as Jikes RVM uses 32-bit references). This allows us to improve performance by copying word by word rather than field by field, while still handling reference types and double-word types in accordance with the Java specification. All our copying mechanisms are oblivious to how mutators update multi-word fields.

9.1 Concurrent Copying with CAS

To copy object fields, we use a CAS in a loop, rather than Algorithm 3's `LoadLinked`/`StoreConditional`, copying *word by word* (Algorithm 12). This is more efficient because multiple non-reference fields within the same word can be processed in a single iteration of the loop; it also deals correctly with reference fields, which are word-aligned and sized. If the contents of the old and new words differ, the collector atomically updates the *tospace* word using a CAS. Failure means that a mutator has updated this word with a value newer than the one the collector attempted to write, so the collector does not need to copy this word.

Algorithm 12. The collector's word copying algorithm using CAS.

```
1  copyWord(p, q):
2      loop
3          currentValue ← *q;
4          toValue ← *p                                          $
5          if isPointer(toValue)
6              toValue ← forwardObject(toValue)
7          if toValue = currentValue
8              return
9          if not CAS(q, currentValue, toValue)                  $
10             return
```

Unfortunately, success does not mean that the collector has written the newest value as there is a risk of an ABA problem. If the mutator were to change the values held in `p` and `q` between lines 3 and 4, the test in line 7 would fail, so the collector would prepare to CAS `toValue` into `q`. In the absence of any further mutator action, this CAS would fail which is safe as noted above. But if the mutator were to update `p` and `q` again with their original value, the CAS would succeed, updating `q` with a stale `toValue`! To resolve this, our algorithm double checks in the next iteration of the loop. The risk to progress is very small: the mutator would have to continually create update the words to prevent progress. We verified that this copying scheme is correct under x86's TSO memory ordering using the SPIN model checker.

The overhead of using atomic instructions for almost all words in the heap is significant. Assuming that the overhead of CAS is dominated by the cost of flushing the store buffer, it could be reduced if a CAS could handle an arbitrary number of words (some processors provide a double-word compare-and-swap instruction, which could reduce the cost by half).

9.2 Transactional Memory

We can extend the idea of copying an arbitrary number of words atomically and efficiently by using *transactional memory* [Herlihy and Moss 1993]. Inspired by database systems, transactional memory allows a thread to execute a sequence of instructions as a transaction. If no other thread makes a conflicting access to the memory locations used by the first thread, the transaction commits. Otherwise, the transaction aborts and the state of the thread is rolled back to that before the transaction started. Transactional memory can be implemented in software or hardware. We start by exploring copying objects with hardware transactions before considering software transactional memory solutions. Note that large objects are allocated in Jikes RVM's separate large object space, which is managed outside this scheme by a non-moving collector.

9.2.1 Hardware Transactional Memory. The Restricted Transactional Memory (RTM) extensions to Intel's Haswell microarchitecture [Intel 2013] support atomic manipulation of arbitrary sized units of memory. With RTM, a transaction starts with an `XBEGIN` instruction and then proceeds normally, reading and writing memory with normal instructions, as well as branching and performing arithmetic. The transaction is committed with `XEND`. All loads and stores in the transaction are tracked in *read* and *write sets* at cache-line granularity. If any other hardware thread loads from or stores to locations in these sets in a manner that creates a conflict, or if the cache overflows, the transaction fails and the program is rewound to `XBEGIN`, which reports the cause of the failure.

The simplest HTM approach is to copy a single object in each transaction, word by word, dereferencing for forwarding where necessary, using normal load/store instructions. If `XBEGIN` reports that the transaction failed, then we fall back to the CAS version. Java objects are typically smaller than a single cache line (64 bytes) [Dieckmann and Hölzle 1999], but the expected read set of a transaction for a single object may be larger as semantic copying requires dereferencing each reference field of a source object.

In 32-bit execution mode, a single cache line can hold 16 references, each of which may refer to a distinct object. Assuming a typical object size of less than one cache line, the transaction's read-set will grow to a little over 1000 bytes at most. Our earlier work [Ritson and Barnes 2013] indicated that transactions up to 16KiB are possible on Haswell, so there is scope for copying multiple objects within a transaction. For example, with read and write sets of 1152 bytes and disregarding other overheads, 13 objects may fit in a transaction. This allows us to trade transaction size against the risk of failure.

We investigated two strategies for transactional copying. **Inline copying** starts a transaction and scans the heap as normal, attempting to add each visited object to the open transaction if it will fit within our limit. If not, the transaction is committed and a new transaction is started. Inline transaction construction is simple but has the disadvantage that scanning-related reads such as looking up object type information — which cannot cause a real conflict — will be included in the transaction.

Planned copying removes this disadvantage. Prior to each transaction, objects are visited as part of the heap scan and added to a 'to-be-copied' list. When this list reaches a limit, a transaction is initiated and all objects are copied before committing the transaction. Planning removes scanning traffic from the transaction and hence also reduces the risk of aborts, but it has a more complex implementation and associated overheads. Various other activities may need to be performed as part of planning the transaction, for example caching looked-up object type information (so that associated reads do not inflate the transaction).

We evaluate these strategies in Section 13.7: both showed a similar performance when transactions are small, while only planned copying maintained its high performance when transactions are large.

9.2.2 Software Transactional Memory. Rather than invent a general-purpose software transactional memory mechanism, we constructed a minimal solution just for the copying phase of our Sapphire implementation. This is facilitated by needing the *fromspace* and *tospace* replicas of the object graph to be only *eventually consistent.* As *tospace* is not read until after the Copy phase is complete, we do not need to maintain immediate consistency during the phase itself.

Our software transactional method (Algorithm 13) comprises a copying and a verification step to be performed for each object. The *copying step* semantically copies the object using normal load/store instructions. In the *verification step* the contents of the *tospace* replica are compared to the *fromspace* object. If the two are consistent, the object has been successfully replicated. If any word is found to be inconsistent with its replica, the object is copied again using the fallback CAS method. Since reference fields of the replicas may be semantically equivalent rather than bitwise identical, the values of *fromspace* reference fields are stored in a buffer during copying. Verification compares the current values of *fromspace* words against their *tospace* replicas or, for references, their buffered values. Note that we assume that reference fields are word-aligned and sized. A memory barrier (`MFENCE` on x86) separating the copying and verification steps is essential, but is the only fence needed. Without it, the store to the *tospace* replica could be reordered after the loads performed in the verification step, risking the loss of a mutator store. Again, we confirmed the correctness of our solution with SPIN.

Algorithm 13. Collector's code for copying an object using software transactional memory.

```
copyObjectTransactional(p, q):
    for i ← 0 to words(q)                                   /* copying step */
        toValue ← p[i]
        if isPointerField(p, i)
            buf[i] ← toValue
            toValue ← forward(toValue)
        q[i] ← toValue

    memoryBarrier

    for i ← 0 to words(q)                                   /* verification step */
        if isPointerField(p, i)
            if p[i] ≠ buf[i]
                goto FAIL
        else if p[i] ≠ q[i]
            goto FAIL

    return

FAIL:
    copyObject(p, q)                 /* fall back to copying word at a time with CAS */
```

Although the software transactional memory approach needs additional load/store instructions for verification, they are unlikely to cause L1 cache misses since all these addresses are used in copying. Alternatively, we could resolve the references again in the verification step rather than saving the unresolved references in the buffer, thus reducing the volume of data stored. However, in our Jikes RVM implementation resolving references is costly as it involves several instructions to identify the space of the referent, and we found that the buffered approach was faster.

Table 3. The number of object copies and the number of verification failures.

Benchmark	copies	failures	Benchmark	copies	failures
antlr$_6$	6,914,141	0	avrora$_9$	15,459,711	23
bloat$_6$	51,265,639	12	eclipse$_9$	277,139,667	9
fop$_6$	5,797,699	0	hsqldb$_6$	320,072,643	9
jython$_6$	66,562,921	27	luindex$_9$	2,844,425	0
lusearch$_9$	201,958,473	16	pmd$_9$	218,561,131	47
sunflow$_9$	174,138,725	104	xalan$_9$	81,518,412	60

We explored how transactional copying would cope with very large arrays, fearing that they may create too large a transaction. However, although HTM imposes a limit on the size of a transaction, we found that copying with software transactions coped well with large objects (see the response times in Figure 8). We fell back to allocating in Jikes RVM's large object space only for objects greater than 128KiB, several times larger than Jikes RVM's default.

We also measured how frequently the verification step fails and copying falls back to the CAS version by using benchmarks from DaCapo 2006 and 2009. Table 3 shows the number of object copies and the number of failures of verification. Failures are very few and unlikely to impact performance.

We demonstrate in Section 13.7 how software transactions offer performance similar to hardware transactions. As software transactions do not require hardware support, and so are portable, we adopt this approach hereafter (unless otherwise stated).

10 RACES III: 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. These can be used for a variety of purposes such as constructing caches or creating canonicalised mappings. Reference objects are widely used. Contrary to our expectations, we found that some DaCapo 2006 and 2009 programs used reference objects heavily (more than 1 million times per second), although this varied between programs (Figure 11). Most showed a small peak of usage by the Jikes RVM class loader at the start of execution. In addition, *lusearch* and *xalan* used reference objects throughout their execution, and *jython* used them heavily in particular phases. In contrast, other programs made little further use of them.

Correct handling of reference objects by a concurrent, let alone an on-the-fly, collector is complex. This task is not eased by the lack of any definition of the semantics of reference objects other than an informal prose account in the documentation for the `java.lang.ref` package. Here, we outline some of the problems and sketch our solution. For simplicity, we focus on strong and weak references. We provide a fuller account, including a formalisation of Java's reference types, in Ugawa et al. [2014].

Objects that are reachable from the roots by traversing only strong references are called *strongly-reachable*; they are not reclaimed. Objects that are not strongly-reachable but reachable from the roots by traversing strong and soft references are called *softly-reachable*. Objects that are neither strongly- nor softly-reachable but reachable from the roots by traversing one or more weak references are *weakly-reachable*. The collector must reclaim weakly-reachable objects, and may decide at its discretion to reclaim any softly-reachable object. In either case, it must clear the referent field of the reference type. Mutators follow 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.

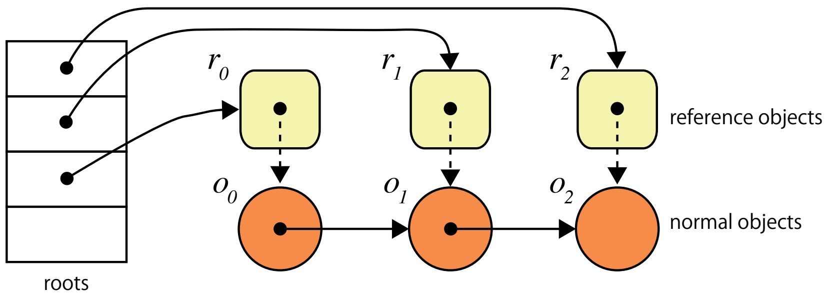


Fig. 5. Weak references held in strongly reachable objects r_0, r_1 and r_2 must be cleared atomically.

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 5), it must also clear *atomically*

(1) *all* weak references to o_2 (e.g. the reference from r_2 in Figure 5), 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: the collector traverses only strong references and marks strongly-reachable objects, after which it clears any weak reference with an unmarked referent.

However, in an on-the-fly context, a mutator may call `get` on a weak reference (e.g. r_2) whose referent o_2 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 became strongly reachable. A similar argument applies to any weak reference whose referent (e.g. o_2) became strongly-reachable because of calling `get` on another weak reference r_0. 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.

10.1 Processing weak references

Our solution is twofold. Our 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 6 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.

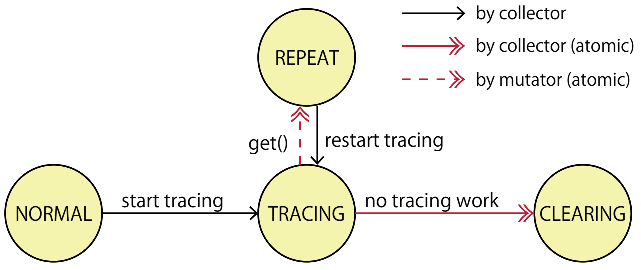


Fig. 6. Global reference state transitions.

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 the mutator 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 from 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 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 referent in the CLEARING state. This cannot 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.

10.2 Termination Loop

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. 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. Hence, get must shade the referent grey to preserve the invariant. 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.

Comparing these two styles of barriers, 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. As the `get` read barrier shades any white referent grey, and only a finite number of weak references with white referents can exist, the termination loop eventually completes. Thus, we decided to use the deletion barrier in the ReferenceProcess phase, as discussed in Section 6.2. Once again, model checking was useful. SPIN helped us discover the risk of non-progress with the insertion barrier and weak references, and confirmed the correctness of the deletion barrier version.

11 PROGRESS

The purpose of an on-the-fly collector is to avoid delaying mutator progress, for example by not stopping all mutator threads to collect. Several real-time collectors claim to be lock-free [Pizlo et al. 2007b], "mostly" or probabilistically lock-free [McCloskey et al. 2008; Pizlo et al. 2008b], or wait-free [Pizlo et al. 2008b, 2010] but provide few details (for example, of stack scanning or thread management). It may be that these properties apply to high-priority mutator tasks that can cause collector tasks to abort or retry. However Jikes RVM does not provide thread priorities. We do not claim to offer hard real-time guarantees, but rely on fair operating system scheduling and not saturating resources. With these caveats, we believe that our implementation provides a good platform for pause-sensitive applications. Assuming that the operating system provides certain guarantees (e.g. that every thread eventually leaves every critical section in a timely manner), the only events where our implementation may block a mutator are as follows. Otherwise, our implementation provides wait-free copying.

- While the collector enumerates live threads, mainly to determine the threads to handshake with the collector, new threads cannot be created.
- Installation of a forwarding pointer to an object's *tospace* shell meta-locks its header, blocking threads seeking to lock the object or hash it for the first time. Of course, a mutator that locks should expect to be blocked! In the worst case, the pre-emption of a thread holding the busy bit might delay rather than halt progress (there are no circular dependencies).
- Acquisition of a fresh TLAB page or allocation of a large object requires a lock.

All these actions are brief, infrequent or both. In addition, we halt mutator threads, one at a time, while we scan their stacks. This pause depends on the depth of the stack but could be bounded with stack barriers [Cheng and Blelloch 2001] or return barriers [Saiki et al. 2005]. Finally, the mutator could run out of space before a concurrent collection completes in poorly configured application. In this case, we fall back to stop-the-world collection. Addressing this issue is beyond the scope of this article, but we note that hard real-time collectors demand ahead of time schedulability analyses in order to meet pause time guarantees [Jones et al. 2012].

12 CORRECTNESS

In this section we describe our approach to assuring the correctness of our implementation. As we noted in Section 2.2, debugging garbage collectors is hard, and much more so for concurrent and on-the-fly collectors. The often long delay between an error (such as failing to trace a reference) and when that error becomes manifest makes it difficult to identify the cause. Concurrent interaction between mutator and collector threads hinders causal reasoning, and the exploration of possibilities is time consuming, error prone, and fails to provide confidence — it is common in our experience to revisit informal arguments repeatedly to reassure oneself of their correctness.

To overcome these problems, we adopted a principled design strategy from day one. We identified the invariants expressed in terms of tricolour abstractions that mutators and collectors should preserve in each phase of the algorithm (Section 6). This discipline was a considerable aid to understanding, and provided the basis for constructing the code. However, it is obviously not sufficient to eliminate all bugs. Thus, as is common practice in building complex systems, we programmed defensively, adding assertions liberally throughout our code to check whether these invariants were preserved. In addition to checking these assertions at run-time, we also used 'sanity checking' in testing, for example by halting the VM before the Reclaim phase and iterating through all references in order to check that no references in the rest of the heap pointed into the *fromspace* that was about to be reclaimed.

We tested the implementation extensively in order to obtain good coverage of possible interleavings. Each benchmark was executed 1000 times, running the benchmarks in parallel against one another in separate JVM instances. In this way, we placed a heavy load on the machine, thereby increasing the chance of different interleavings. Unfortunately, even an unmodified version of a system as complicated as JikesRVM (and the DaCapo benchmarks) contain their own bugs, and it was sometimes hard to determine the cause of a bug.

One good example was the hash code scenario. As we described in Section 7.3, both mutator and collector threads access an object's status word for hashing and locking. Address-based hashing uses an object's address as its hash code and this must be preserved by the collector if it moves the object. MMTk's moving collectors add an extra hash code field when a hashed object is first moved, with bits in the status word recording the hash state. This proved to be tricky to implement correctly: the cause of our error was not immediately apparent and testing did not always reveal the bug. We decided to model this scenario and check it with the SPIN model checker [Holzmann 2004]: the bug was discovered and corrected within a few hours.

This success convinced us to use model checking routinely as a design tool. Thereafter, before we developed code — for example, copying with software transactional memory (Section 9) or processing reference types (Section 10) — we model checked and revised our designs until they passed. We also retrospectively model checked other parts of our design (Type II phase changes, copying with CAS, etc.).

12.1 Model checking

Model checking is a verification technique for finite state systems. To verify some property such as an invariant, we translate the code to a state transition system that models it. 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.

We found bounded model checking to be a very effective aid for discovering and helping to eliminate bugs, and it gave us a high degree of confidence in the correctness of our design. We do not argue that it guarantees the correctness of an algorithm. But we found that exhaustively exploring a model for counter-examples, even in a simplified scenario, was of great practical use in eliminating bugs, such as missing memory barriers. However, model checking does not reduce the number of assertions needed in the code since the model is an abstraction of the Java code, and the code was not derived automatically from the model. Exhaustive testing and sanity checking were still essential. We believe that this principled approach to design — invariants expressed in terms of abstractions, the more complex algorithms model checked, and invariants tested at run-time with assertions — led to fewer failures and reduced development time. While we cannot prove this claim, the development of our design and implementation of Java's very subtle reference type processing did seem to proceed faster than the development of similarly complicated parts of the system where we had not used model checking.

We model checked the following more subtle parts of our algorithm.

- Concurrent copying, with either atomic CAS operations or software transactional memory, for reference fields, and single- and double-word scalar fields (six models in all).
- The changes of phase from 'no collection' to marking, and from copying to flipping.
- Reference object processing.
- Object hashing.

In all ten cases, model checking verified the correctness of our algorithms. It also allowed us to explore optimisations and termination properties, sometimes throwing up unexpected results. It demonstrated the need for a fence between copying and verification in the software transactional memory algorithm (Algorithm 13). It showed that Type II transitions (Section 5.3) are essential from the 'no collection' to the marking phase, and from the copying to the flip phase. And, model checking discovered that reference object processing was not guaranteed to terminate if an insertion barrier was used, but is guaranteed to terminate with a deletion barrier.

Below, we provide a detailed account of how we model checked concurrent copying and an outline, for reasons of space, of the other four cases. A full report on our models (10 in all) can be found at https://github.com/rejones/sapphire.

12.2 Model checking concurrent copying

The SPIN model checker accepts a model written in the Promela language, describing a state transition system as a form of sequential processes[6] communicating via channels. We express properties as assertions injected into the model. We developed six models of concurrent copying, for a reference field, a single-word scalar field and a double-word scalar field (thus covering all possible low-level types of field), each with copying by a CAS operation or with software transactional memory. Each model comprises a collector thread process, a mutator thread process, and a memory process that models the x86 relaxed memory architecture. A single mutator suffices since we are concerned here with verifying the interactions between mutator and collector, and because Sapphire assumes that mutators are data race free [Hudson and Moss 2003].

Modelling the memory model. In x86, each store issued from a CPU core is inserted into its store buffer, a FIFO queue, so stores are drained and affect the cache memory in order. Store-load forwarding allows a core to see any store still held in its store buffer. Only the core that issued a store instruction writing v to an address a can read the latest value v from the address a immediately, as the buffer is core-local. Note that we assume cache coherency, so once a store reaches cache memory, the change can be seen by other cores. For this reason, we did not include the cache hierarchy in our model.

We modelled the store buffer with channels between a thread process and the memory process (Model 1). When the mutator thread writes a value v to an address a, it uses the `MUTATOR_WRITE` macro to send a pair $\langle a, v\rangle$ to the `mutator_queue` channel. The `memory` process receives the store request and commits it to shared memory, modelled by the array `shared`. Similarly, the collector thread sends store requests though its `collector_queue` channel. At arbitrary times `memory` non-deterministically chooses the channel to drain.

To model store-load forwarding, we give each thread a copy of memory. When a thread writes to an address a, it writes at index a of its local memory at the same time as it sends the write instruction to the channel. In addition, it increments the appropriate counter (`mutator_queue_count[a]` or `collector_queue_count[a]`) for its local memory to indicate that the latest value is in the store buffer. The thread reads from the local memory if the corresponding counter is greater than zero. The memory process decrements the counter when it commits the write.

[6]Note that these are processes in the modelling language, and not to be confused with the operating systems concept of a process.

```
active proctype memory() {
  do
    ::atomic{COMMIT_WRITE(mutator_queue, mutator_queue_count)}
    ::atomic{COMMIT_WRITE(collector_queue, collector_queue_count)}
  od
}

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

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

#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 \
}
```

Model 1. The memory model; `COLLECTOR_WRITE` and `COLLECTOR_READ` are defined in a similar way.

Scenario. Our model-checking goal here is to verify that the mutator always reads the value that it last wrote, regardless of any action by the collector, such as copying objects or flipping the spaces. It therefore suffices to model a scenario consisting of a single object with a single field; here, we use a reference field but later we outline the case for single- or double-word scalar fields. The memory has only two addresses; each semi-space contains one object. Initially, the *fromspace* copy of the object has `NULL` in its field. This scenario and the mutator is common to both our collector models for concurrent copying.

Mutator model. Model 2 shows the model of the mutator. The `flipped` flag, whose initial value is 0, is a shared variable representing the Flip phase; setting it flips the mutator to read from *tospace*. In the first case of the **do** loop — the Promela construct for infinitely repeated, non-deterministic choice — the mutator may alter the value of the field, writing either `NULL` or a reference to the object itself; the write barrier causes it to write to *fromspace* and then *tospace*. Alternatively, in the second case, it may check the property that "the mutator reads the semantic equivalent of what it wrote." For reference fields, we convert the *tospace* address before the comparison. If the collector fails to copy, the mutator may read a stale value after flipping. In this way, the property is checked by actions scheduled infinitely often and at arbitrary times.

Models for scalar fields. The model for a single-word scalar field is almost identical to Model 2: it alternates writing `1` or `0` and omits the `FORWARD` action. For a double-word field, the mutator model writes `0`,`1` or `1`,`0` to two words in *fromspace* and in *tospace*, word at a time, i.e. it writes the first word to both spaces, and then the second word. Four separate `MUTATOR_WRITE`s give the collector a chance to act in between any two actions.

```
proctype mutator() {
  byte x = NULL, r;
  do
    ::true ->
      if
      ::true -> x = NULL
      ::true -> x = FROM_SPACE_OBJECT
      fi;
      r = x;
      MUTATOR_WRITE(FROM_SPACE_ADDR(0), r);
      FORWARD(r);
      MUTATOR_WRITE(TO_SPACE_ADDR(0), r);
      r = 0;

    ::true ->
      if
      ::!flipped ->
        MUTATOR_READ(FROM_SPACE_ADDR(0),r);
        assert(x == r);
      ::else ->
        MUTATOR_READ(TO_SPACE_ADDR(0),r);
        FORWARD(r);
        assert(x == r);
      fi;
      r = 0;
  od
}
```

Model 2. Model of the mutator.

Collector models. Models of the collector copying with CAS (Algorithm 12) are shown in Model 3, and with transactional copying (Algorithm 13) in Model 4. Both are straightforward. The **`atomic`** block in Model 3 represents the CAS instruction. Since CAS has the effect of a memory barrier, we inserted `COLLECTOR_MFENCE` into the block to force flushing of the store buffer.

Correctness. SPIN did not report any error with any of these six models, confirming that the mutator always reads a value semantically equivalent to the one that it has written most recently, regardless of collector's activity. However, note that if we remove the memory fence marked with # in Model 4, SPIN *did* report the assertion in Model 2 to be violated.

12.3 Modelling phase changes

We next consider Sapphire's two more complex changes of phase, from 'no collection' to marking, and from copying to flipping semi-spaces. Both of these are Type II phase changes, that is, they require two intermediate phases, rather than one (Section 5.3). We used model checking to assure both that our algorithms were correct, and that two intermediate phases are necessary. Here, we outline the scenarios for our models of phase change.

Modelling No GC to Marking. Mutators switch from 'no collection' to marking through the two intermediate phases, PreMark1 and PreMark2. In PreMark1, the insertion barrier is installed (Algorithm 6) but the mutator continues to allocate white objects; in PreMark2, mutators allocate

```
proctype collector() {
  byte currentValue, toValue, tmp, i;
  byte a, v;                               /* a and v are used in COLLECTOR_MFENCE */

  i = 0;
  do
    ::(i < N_WORDS) ->
       COLLECTOR_READ(TO_SPACE_ADDR(i), currentValue);
       COLLECTOR_READ(FROM_SPACE_ADDR(i), toValue);
#ifdef REFERENCE
       FORWARD(toValue);
#endif
       if
         ::(toValue == currentValue) -> i++
         ::else -> atomic {                                              /* CAS */
           COLLECTOR_MFENCE;
           COLLECTOR_READ(TO_SPACE_ADDR(i), tmp);
           if
             ::(currentValue == tmp) ->
               COLLECTOR_WRITE(TO_SPACE_ADDR(i), toValue)
             ::else -> i++
           fi;
           COLLECTOR_MFENCE;
         }
       fi
    ::else -> i = 0; break
  od;
  COLLECTOR_MFENCE;
  flipped = true;
```

Model 3. Model of the collector copying with CAS. N_WORDS is the number of words to copy, i.e. 1 for a reference or single-word scalar field, or 2 for a double-word scalar field.

black. Our heap model consists of three objects, each located at a distinct address represented by an integer. Each object has a colour and a single field. A colour is one of WHITE, GREY, BLACK or NOT_ALLOCATED. The field can hold an address of an object or NULL. Our mutator model emulates an arbitrary program that may read from a field, write to a field, allocate an object or advance the mutator's phase. 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 with one another, and the collector does not write to the variables that the mutator accesses in these phases. The collector simply advances the GC phase from 'no collection' through one or two intermediate phases (depending on the type of phase change being modelled) to marking, and waits for mutators to catch up at each step.

As Sapphire uses an insertion barrier with a grey mutator in these phases, an observer process checks the strong tricolour invariant that, if any object is BLACK and its field is not NULL, then the object it references is not WHITE. With two intermediate phases, the model checker confirmed that this invariant always holds. With only one intermediate phase, in which the mutator enables its insertion barrier and starts allocating black at the same time, the model checker found a counter-example, thus confirming that a Type II phase change is indeed necessary.

```
proctype collector() {
  byte currentValue, toValue, tmp, i;
  byte a, v;                          /* a and v are used in COLLECTOR_MFENCE */
  byte buf[N_WORDS];

  i = 0;
  do /* Copy */
    ::(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;

#        COLLECTOR_MFENCE

  do /* Verify */
    ::(i < N_WORDS) ->
#ifdef REFERENCE
       COLLECTOR_READ(FROM_SPACE_ADDR(i), currentValue);
       if
         ::(currentValue != buf[i]) -> goto FAIL              /* fall back to CAS */
         ::else -> skip
       fi;
#else
       COLLECTOR_READ(FROM_SPACE_ADDR(i), currentValue);
       COLLECTOR_READ(TO_SPACE_ADDR(i), toValue);
       if
         ::(currentValue != toValue) -> goto FAIL             /* fall back to CAS */
         ::else -> skip
       fi;
#endif
       i++
    ::else -> i = 0; break
  od;
  goto SUCCESS;
FAIL:
  /* copy with CAS here */
SUCCESS:
  COLLECTOR_MFENCE;
  flipped = true;
}
```

Model 4. Model of the transactional copying collector. N_WORDS is the number of words to copy, i.e. 1 for a reference or single-word scalar field, or 2 for a double-word scalar field.

Modelling Copying to Flipping. In the Copy and Flip phases, the mutator writes to both *fromspace* and *tospace* objects. In the Copy phase, the mutator assumes that *tospace* references are held only by *tospace* objects, and it never writes *tospace* reference to anywhere other than *tospace* (Algorithm 8). In contrast, in the Flip phase, the mutator never writes *fromspace* references. Mutators switch from copying to flipping through two intermediate phases, PreFlip1 and PreFlip2. The mutator uses the PreFlip barrier, Algorithm 9(b), in the PreFlip1 phase, and the Flip barrier, Algorithm 9(c), in the PreFlip2 and Flip phases. The PreFlip phase barrier carefully checks the destination of references so that it works as the Flip phase barrier if the mutator is writing a *tospace* reference, and as a Copy phase barrier in other cases.

Our model comprises two mutators and three spaces: *fromspace*, *tospace* and a non-replicated space. We consider two objects in *fromspace*, each of which has a replica in *tospace*, and a single object in the non-replicated space. Once again, addresses are represented by integers and each object contains a single field. Our mutator model emulates an arbitrary program that may read from or write to any object, and these actions are atomic. The collector model is the same as the one used above. This time our observer process checks that no *tospace* object holds a reference to a *fromspace* object. As before, the model checker verifies that this invariant holds if and only if a Type II phase change is used, thus confirming a bug in Hudson and Moss which did not use a Type II phase change.

12.4 Modelling Reference Type Processing

Java provides references of four levels of strength. 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 (Section 10). The semantics of reference classes require that, at the time that the collector decides to reclaim a weakly reachable object, it must also clear atomically *all* weak references to that object, and *all* weak references to other weakly-reachable objects from which that object is reachable through a chain of strong and/or soft references.

Our Sapphire implementation uses the state transition system shown in Figure 6. We added a fourth object colour, `RECLAIMED`, to model objects reclaimed by the collector. 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, no mutator will ever see the referent of that object.

Property P1 requires that, if a mutator loads a referent of a reference object, the referent has not been reclaimed; P2 implies the atomicity properties that the API definition requires. We checked the properties for the limited model shown in Figure 5. This model has three pairs of reference and normal objects, namely r_0, r_1, r_2 for reference type objects and o_0, o_1, o_2 for the corresponding normal objects. These normal objects are linked in a list, but there are no other strong references to them. We assumed that all reference objects remain directly strongly reachable from the root, and that the mutator can always call `get` on them.

Model checking confirmed that properties P1 and P2 held, regardless of whether an insertion or a deletion barrier was used in the reference processing phase. However, it also confirmed that tracing terminated only with the deletion barrier, but not with the insertion barrier.

12.5 Modelling Hashcodes

Finally, we modelled object hashing. Address-based hashing has a delicate interaction with concurrent copying garbage collection. Our model uses a single object. Two addresses represent the *fromspace* and *tospace* copies of the object. As the model of the collector performs many collection cycles, the address used for *fromspace* changes during the execution. To reflect Algorithm 11, the object is modelled with a combination of the contents of the hash state, the busy bit and the forwarded bit, and a hash code word if the object was moved after its hashcode was obtained. Our model of a single mutator comprises a single root pointing to the object. The mutator arbitrarily and repeatedly advances its phase and obtains the object's hashcode. The collector repeatedly iterates through the collection phases 'no collection', mark, copy, flip and reclaim, handshaking with the mutator as it changes phase. Model checking confirmed that our algorithm meets the Java language specification, i.e. that all calls of `hashCode` return the same value for an object.

12.6 Conclusions

We found bounded model checking with SPIN to be a practical approach to use. Although we were not familiar with model checking before, it took no more than a day to construct our first model and check it, including discovering and correcting some bugs. Although checking a particular scenario with bounded parameters (for example, a store buffer of size two) cannot give a 100% guarantee of correctness, we can be reasonably confident. Concurrent garbage collection is complex and it is easy to overlook corner cases. We believe that our approach to model checking offers reasonable confidence in an algorithm at a reasonable cost.

13 EVALUATION

Evaluating an on-the-fly or concurrent collector is fundamentally different from evaluating a stop-the-world collector. The design of a stop-the-world collector will typically focus on minimising application execution time, though reducing the extent of pauses for GC may also be important, depending on the application. The effect of the collector on overall execution time will depend on the number and duration of collections and any tax imposed on the mutator, for example due to write barriers. In contrast, concurrent collectors are designed primarily to maintain responsiveness, and this is necessarily achieved at some expense of mutator and collector throughput, and of memory footprint. Thus, to evaluate a concurrent collector, it is important to ask:

- How should concurrent collections be scheduled?
- Is the application responsive?
- What impact does concurrent collection have on overall execution time?
- By how much does concurrent collection increase heap size requirements?

13.1 Environment and Benchmarks

All results were obtained from a 2.1GHz, 64-core, 4-socket AMD Opteron 6272 system with 64GiB of RAM, running stock Ubuntu Linux 12.04.3 LTS (3.8.0-25-generic kernel). We disabled biased locking [Pizlo et al. 2011] and specialised scanning [Garner et al. 2011] only because we have yet to implement these in Sapphire. Hudson and Moss [2003] suggested that Mark and Copy phases can be merged into a single Replicate phase. We explored this, assuming that it might offer better locality, but were surprised to find that performance degraded when we used CAS for the Replicate phase. We believe that this is because interleaving CASes to copy slots interferes with the locality of marking by draining the processors' load-store buffers. In contrast, a separate Copy phase issues CASes in bursts. Inspection of CPU performance counters showed that a Replicate-with-CAS phase improved the number of executed instructions and cache misses at each level of the cache hierarchy, but introduced delays due to store-load forwarding. In contrast, using software transactional memory in the Replicate phase led to no degradation but gave only similar performance to that obtained by using separate phases for most benchmarks. As separate phases leads to a somewhat more straightforward implementation, we adopted this for our experiments.

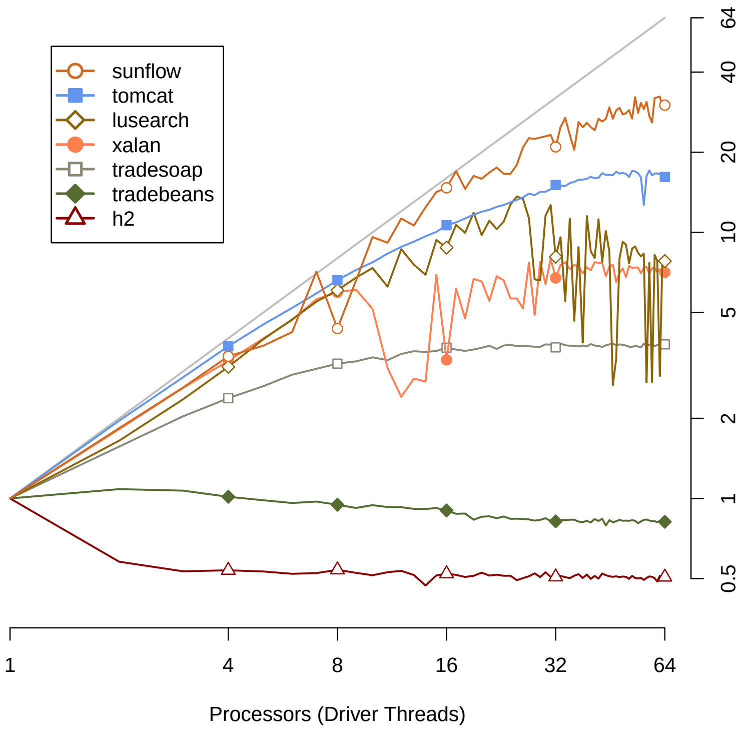


Fig. 7. Speedup v. number of driver threads for DaCapo 2009, from Kalibera et al. [2012]. These experiments ran with Oracle's HotSpot 1.7 JVM on the same 64-core, 4-node AMD machine as we use in this paper.

We used a set of benchmarks from the DaCapo 2006 and 2009 benchmark suites, preferring 2009 versions and excluding any that failed to execute in an unmodified Jikes RVM Semispace collector configuration (typically due to requiring libraries that Jikes RVM does not yet support). All of the programs used launch multiple threads. By default, the suites scale the work-loads to use the number of logical processors in the system, although seven of the DaCapo 2009 benchmarks allow scaling to an arbitrary number of driver threads. However, be warned that the number of threads created is a weak approximation of concurrency. Kalibera et al. [2012] wrote, "The DaCapo harness allows setting the number of threads that drives the workload, but this does not fully determine how many threads do a substantial amount of work concurrently. Work-loads often spawn threads of their own, either directly or indirectly through libraries. Some threads live for the entire execution of the benchmark, some only for one iteration and some only for short-term tasks within iterations. Some are active throughout whole execution of the benchmark, some only throughout one iteration (e.g. *avrora*$_9$) or some phase of it (e.g. *h2*$_9$[7]), and sometimes tens or hundreds of threads are created each for a single short-term task (e.g. *eclipse*$_9$). Moreover, the number of threads spawned may depend on the hardware — e.g. *tomcat*$_9$ spawns poller threads that service network connections depending on the number of logical processors available. On the other hand, even benchmarks that do not spawn any new threads can generate work for the VM's reference handler and finaliser threads." Of the benchmarks we used, *avrora*$_9$, *hsqldb*$_6$, *lusearch*$_9$, *pmd*$_9$, *sunflow*$_9$ and *xalan*$_9$ all have

[7]Neither *h2*$_9$ nor *tomcat*$_9$ run on Jikes RVM.

significant levels of concurrency, and *luindex*$_9$ has some meaningful concurrency, by Kalibera's definition.

The DaCapo 2009 benchmarks were limited to 32 threads, both to allow processor resources to be dedicated to the collector and because Kalibera et al. showed that most DaCapo benchmarks degrade if more than 32 threads are used — Figure 7 shows examples of scalability of a selection of DaCapo 2009 benchmarks. We found that the best performance was obtained by devoting 8 threads to the garbage collector.

13.2 Methodology

Users adopt on-the-fly collectors for just one reason: to improve the responsiveness an application. But there are unavoidable costs to on-the-fly collection: an application managed by any concurrent collector will have a longer execution time than if it were managed by a stop-the-world collector, and will have a larger memory footprint. Our primary measurement is therefore responsiveness (Section 13.5), and we compare our Sapphire implementation against Jikes RVM's only concurrent collector.

The Concurrent Mark-Sweep collector[8] is a mostly-concurrent collector that uses a deletion barrier, hence mutators are kept black. It stops all mutators together for thread-stack scanning, before allowing mutators to resume while the collector marks live objects. It then stops all mutators again to complete reference processing. The collector uses lazy sweeping to reclaim memory on demand [Hughes 1982]. Concurrent Mark-Sweep is not an entirely satisfactory point of comparison: it could not execute some benchmarks, either due to memory exhaustion or other implementation issues. It is also neither copying nor compacting, both of which give it some execution time advantages over Sapphire.

However, we also want to understand what the consequences of a design that permits on-the-fly collection will be on overall performance (e.g. execution time). To assess this, we need to compare our Sapphire implementation with the nearest structurally similar stop-the-world collector, in this case JikesRVM's Semispace collector. Both Sapphire and Semispace manage the heap primarily through a pair of copying semispaces, and neither are generational.[9] Semispace therefore provides an appropriate reference point. We stress that our methodology is designed to expose these overheads because we believe only this will provide researchers with an insight into the costs of building an on-the-fly replicating collector. In order to provide this insight, it is essential to provide both the Sapphire and Semispace configurations with the same environment and resources. Thus, we experimented with the configuration that gave Sapphire the best performance, e.g. heap trigger (Table 5) and number of GC threads (we know from Kalibera et al. [2012] that 32 mutator threads is optimal for most DaCapo benchmarks — see also Figure 7). We then ask, what is the cost of providing this configuration? Deliberately we do not ask, what is the best configuration for Semispace? Then, we give the mutators in both configurations the same number of threads (32), and we give the parallel collectors in both configurations the same number of threads (8).

In virtual machines research, performance changes are often small, so rigorous evaluation that takes system warmup and variance into account is essential. Unfortunately, as Kalibera and Jones [2013] have shown (and confirmed by Barrett et al. [2017]), it is often not possible to get managed language systems to a stable state. Unless otherwise stated, we use *compiler replay* [Georges et al. 2008]. For each selected benchmark, an initial set of ten warmup runs allows the optimising

[8]http://www.jikesrvm.org/JavaDoc/org/mmtk/plan/concurrent/marksweep/CMS.html

[9]It is certainly possible to build generational copying collectors (JikesRVM provides several varieties), and we do not believe there is any lack of generality in our approach. Our on-the-fly techniques could be adopted for generational collection which should reduce the time spent in garbage collection and hence the overheads incurred by mutators; the main challenges may lie in tuning a generational on-the-fly collector.

Benchmark	No barrier	Barrier	Overhead
antlr$_6$	1.94–1.95	2.54–2.55	31% ±0.3%
avrora$_9$	9.38–10.12	10.92–11.02	12% ±0.2%
bloat$_6$	8.49–8.54	10.33–10.37	22% ±0.2%
eclipse$_6$	56.98–58.20	56.86–57.59	-1% ±0.0%
fop$_6$	2.67–2.69	2.84–2.86	6% ±0.1%
hsqldb$_6$	2.83–2.85	2.49–2.50	-12% ±0.1%
jython$_6$	15.57–15.62	21.53–21.59	38% ±0.4%
luindex$_9$	2.67–2.72	3.21–3.24	20% ±0.2%
lusearch$_9$	3.76–3.83	4.35–4.43	16% ±0.2%
pmd$_9$	5.82–6.10	7.09–7.28	21% ±0.3%
sunflow$_9$	3.86–4.12	4.36–4.44	10% ±0.2%
xalan$_9$	3.05–3.10	4.44–4.57	46% ±0.5%

Table 4. Cost of Sapphire-style barriers: execution time (seconds) for Semispace with and without write barriers.

compiler to compile methods. The compiler's state is then recorded. Results are collected from 30 independent runs in which classes have been compiled and initialised using the previously recorded compiler state. We report results for benchmark execution only, and exclude time spent in VM and benchmark start-up or shutdown. 95% confidence intervals are computed as per Kalibera and Jones [2013].

13.3 Concurrency overheads

Using this methodology, we can determine in a fair manner the overheads incurred by Sapphire, e.g. in terms of the cost of barriers, GC time and overall execution time. In contrast, Semispace places no overhead on mutators. To estimate the cost of concurrency overheads, we adopted the approach of Yang et al. [2012], modifying a Semispace configuration to include Sapphire's write barriers but not to take any action, such as copying objects. We took care to ensure that the optimising compiler did not remove these barriers. The effect of the barriers on execution time is shown in Table 4 for execution with a fixed, and large, heap size of 384Mib, chosen to reduce the impact of collection on these overheads. It is clear that the barriers needed by *any* concurrent collector will impose a significant tax on the mutator. We were, however, surprised that the addition of barriers improves performance for the *hsqldb* benchmark. Examination of the hardware performance counters indicates that this improvement can be attributed to a 5% increase in instructions executed per cycle as a result of reduced pipeline stalls, cache misses and page faults when barriers are present. We assume that increasing the number of predictable branching instructions improves prefetching and caching. We were able to replicate this result on an Intel Haswell processor.

13.4 Triggering Collection

Running concurrent collections continuously would place an excessive barrier overhead on mutators, but determining an optimal schedule of collections is hard, even for a stop-the-world collector [Jacek et al. 2016]. An ideal concurrent collector must monitor the application's allocation rate and trigger collections so that they complete before the application runs out of memory. Such a trigger depends on estimates of allocation rate and collector throughput for given workloads. However, for many soft real-time applications, such estimates will not be available. Thus, to evaluate our Sapphire

Benchmark	Livesize (MiB)	Trigger (MiB)	Peak heap (×livesize)	Allocated during GC Min (MiB)	Geomean (MiB)	Max (MiB)
$antlr_6$	31.7	16	1.71	1.20	2.75	4.65
$avrora_9$	21.5	32	3.44	0.25	1.83	3.85
$bloat_6$	33.0	32	2.94	0.82	2.92	8.44
$eclipse_6$	59.1	48	3.60	0.26	4.68	16.67
fop_6	26.7	32	3.69	1.23	2.96	4.98
$hsqldb_6$	91.0	128	2.43	0.41	4.67	20.55
$jython_6$	32.5	48	3.73	0.27	3.73	10.07
$luindex_9$	21.5	32	3.38	0.69	1.95	4.00
$lusearch_9$	31.9	–	–			
pmd_9	90.8	96	3.18	1.44	5.98	51.81
$sunflow_9$	38.6	64	7.53*	1.53	28.85	79.73
$xalan_9$	45.5	–	–			

Table 5. Triggers for 100% on-the-fly collection with peak heap usage < 4× maximum livesize (*except for *sunflow* which needed 8× livesize to avoid stop-the-world pauses).

implementation, we use a simple trigger that initiates an on-the-fly collection after every fixed volume of allocation. This has the advantage of simplicity and stability. A more sophisticated solution might trigger collections by using predictions based on a history of allocation and collection rates.

Our assumption is that a user of an on-the-fly collector is prepared to pay some price in terms of elapsed time and heap footprint for responsiveness. An ideal trigger will minimise time spent in collection (additional mutator overhead) and heap footprint (further objects allocated during collection), while avoiding falling back to stop-the-world collection (application pauses). These are opposing objectives to be balanced: small (high frequency) trigger thresholds reduce the chance of a stop-the-world collection, while large (low frequency) thresholds minimise the total time spent in on-the-fly collection.

To explore the trigger selection space, we ran each benchmark with a range of allocation triggers (8–128MiB), allowing the heap to expand as necessary (up to 10× the maximum livesize of the benchmark), and observed the elapsed time for the benchmark and the peak heap usage. Note that we use such a large heap *only* to explore the triggers and *not for performance evaluation.* In common with other researchers, we ignore `System.gc` throughout as, in our current implementation, this call would trigger a stop-the-world collection even if an on-the-fly collection is already running. Note that *hsqldb* would otherwise call `System.gc` five times, degrading its performance by nearly 100%. Ignoring this request also improves performance with the Semispace collector configuration.

Given that Sapphire's priority is to remove application pauses, we first filtered out configurations with stop-the-world pauses. Any semispace copying collector needs a heap of at least 2× maximum livesize to *guarantee* that it will not run out of memory (although collection scheduling may lead to lower peak heap usage in practice). Some further headroom avoids thrashing the collector: 3× maximum livesize is reasonable for a stop-the-world collector but a concurrent copying collector needs further room as mutators continue to allocate during collections. We made a pragmatic and conservative choice to retain only trigger choices for which the heap footprint grows to no more than 4× maximum livesize, and then select the trigger that delivered the shortest elapsed time.

As expected, for most benchmarks, large triggers had to be rejected because they led to stop-the-world pauses as the concurrent collector could not keep up with allocation. Smaller triggers led to longer elapsed times, but these reduced sharply as trigger intervals increased until a point where increasing the trigger had diminishing effect on elapsed time. The triggers used in the remainder of our evaluation are shown in Table 5. Note that the 'Allocated during GC' columns do not include double-allocation in *tospace*. It is clear that Sapphire is able to eliminate stop-the-world pauses with modest peak heap usage. However, it is not suitable for applications with very high allocation or mutation rates. These scenarios may lead to more stop-the-world collection pauses. For example, *lusearch* allocates approximately 1GiB of objects each larger than 8KiB in one phase. Similarly, *xalan* has a sustained allocation rate of 500MiB/sec and a peak rate exceeding 3GiB/sec. This means it can achieve 90–100% of collections performed on-the-fly only at the expense of either a very large peak heap footprint (up to 12× maximum livesize) or a significant fraction of stop-the-world collections. We therefore exclude *lusearch* and *xalan* from the remainder of the evaluation. *sunflow* allocates a maximum of 80MiB (twice livesize) during collection but can avoid stop-the-world pauses if peak heap usage is allowed to grow to 8× maximum livesize.

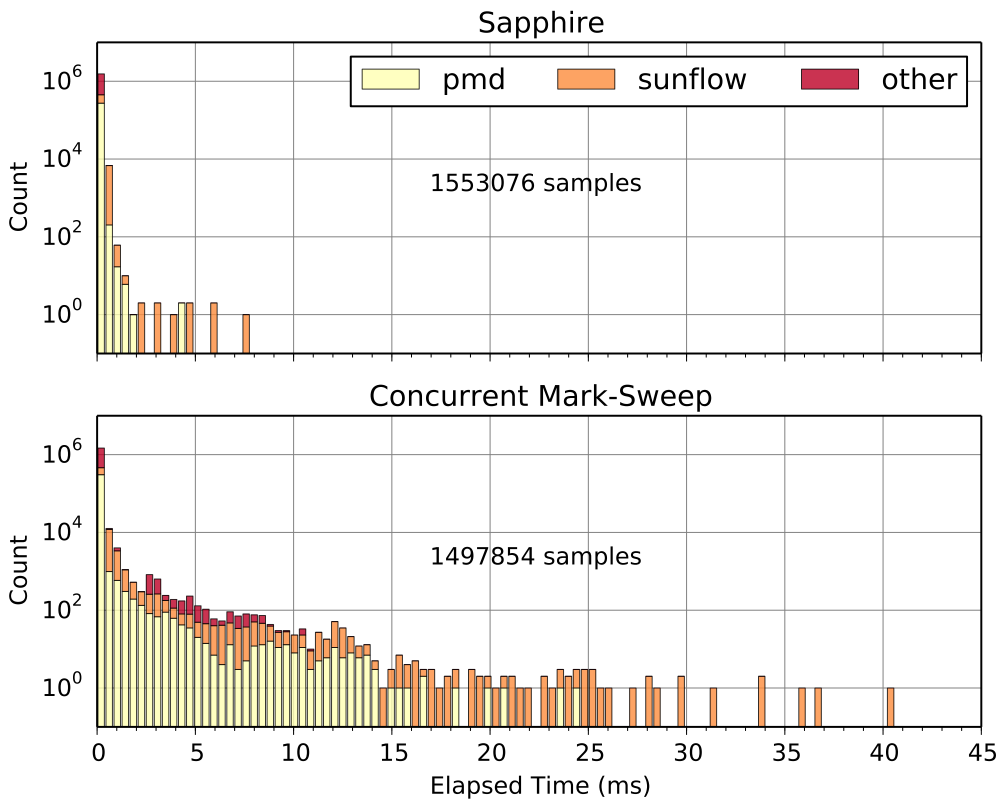
13.5 Responsiveness

The soft real-time scenario that Sapphire is designed to support requires that most tasks are completed within a given bound but infrequent deviations from this bound can be tolerated [Printezis 2006]. Responsiveness is the key metric for these applications. Worst case or average garbage collection pause time alone is not an adequate measure of responsiveness. Even in a stop-the-world collector, it is essential to understand pause time distribution as well as single metrics such as average or worst-case pauses. However, none of these are adequate measures of a concurrent collector. Even if individual pauses are very brief, many pauses in a short time window will impair application responsiveness. For this reason, some researchers have adopted *minimum mutator utilisation* (MMU), the minimum fraction of time spent in the mutator, for any given time window [Cheng and Blelloch 2001]. However, MMU is also an unsatisfactory measure. For example, outside stack scanning and construction of shells, Sapphire has 100% mutator utilisation in our configuration. MMU also raises the question of attribution of overheads. Barrier actions are likely to be counted towards mutator utilisation although they are really work on the behalf of the collector. Even with high MMU, these actions will increase the time to execute an application transaction.

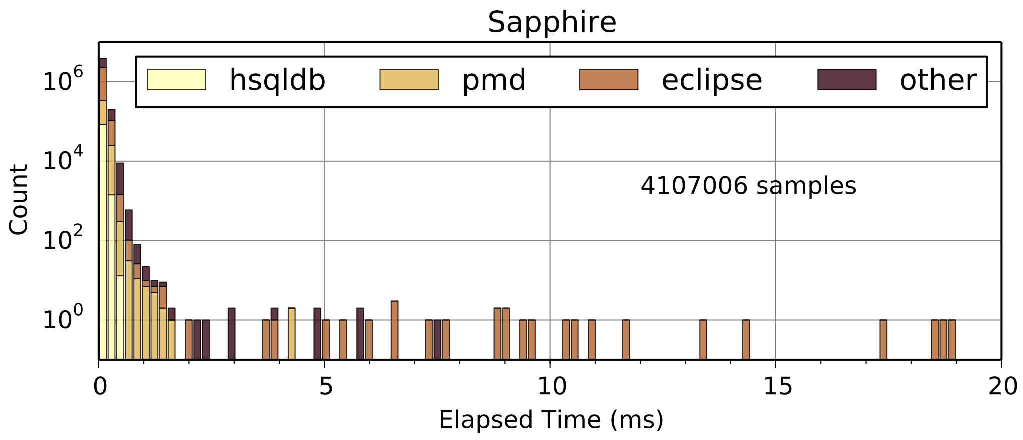
For these reasons, we assess the responsiveness of our implementation by measuring the response time of small transactions, as others have [Click et al. 2005; Pizlo et al. 2008b; Tene et al. 2011]. This is a measure more representative of the needs of real-world applications. To do this, we added a thread that starts when the VM starts and performs periodic transactions alongside the DaCapo benchmarks. These periodic tasks are affected by VM activity including garbage collection but do not interact directly with the DaCapo benchmarks, whose purpose is simply to keep the collector busy.

The periodic task has a small fixed workload that deletes and recreates 200 nodes in a balanced binary tree of 10,000 nodes at 1ms intervals. This workload is intended to exercise allocation and invoke write barriers, giving it a high probability of interference with a running collector. In isolation our periodic task typically takes $80\mu s$ to execute. The response time for each task execution was recorded in a statically allocated buffer. Between task executions, the test thread busy waits. The `getrusage` system call was used to detect context switches during task execution; any affected results were discarded.

We compared Sapphire and Concurrent Mark-Sweep configurations, excluding those benchmarks that cannot execute on the latter (*eclipse*, *jython*, *lusearch* and *xalan*). To minimise the time spent in collections, Concurrent Mark-Sweep was configured to trigger collection when 60% of the heap was used. Although Concurrent Mark-Sweep is a non-moving collector, we used the same heap sizes as in the Sapphire configurations, even though this gives Concurrent Mark-Sweep a significant advantage.



(a) Task times for Sapphire and Concurrent Mark-Sweep for applicable benchmarks (excluding *eclipse*, *jython*, *lusearch* and *xalan*). Each bar represents 500μs.



(b) Task times for Sapphire including *eclipse*, *jython*. 'Other' excludes *lusearch* and *xalan*. Note that the bars are stacked, giving more visual weight to *hsqldb* than to 'other'.

Fig. 8. Time to execute periodic tasks alongside DaCapo benchmarks. Note that the y-axes are logarithmic.

Benchmark	Semispace (s)	GC (s)	Sapphire (s)	OTF (s)	Overhead
antlr6	2.12–2.12	0.63–0.65	3.06–3.08	0.78–0.81	45% ±0.8%
avrora9	9.65–9.74	0.23–0.24	10.44–11.95	0.55–0.59	15% ±8.3%
bloat6	9.06–9.12	1.48–1.50	12.64–12.72	2.57–2.63	39% ±0.9%
eclipse6	54.18–55.22	3.57–3.60	60.76–73.30	10.86–13.44	23% ±13%
fop6	2.73–2.76	0.40–0.41	2.89–2.91	0.27–0.30	6% ±0.8%
hsqldb6	2.80–2.82	0.67–0.68	2.73–2.77	0.59–0.65	-2% ±1.0%
jython6	18.06–18.12	2.05–2.09	19.21–19.29	3.13–3.19	6% ±0.4%
luindex9	2.62–2.64	0.18–0.18	3.59–3.60	0.37–0.38	37% ±0.7%
pmd9	5.81–6.04	0.65–0.67	8.76–8.97	2.53–2.60	50% ±4.6%
sunflow9	4.06–4.17	0.46–0.48	5.22–5.30	3.54–3.61	28% ±2.7%

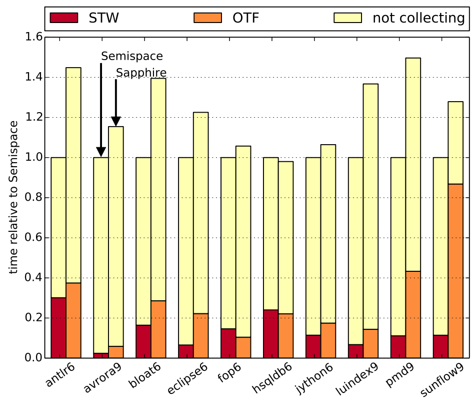
Table 6. Execution and GC times for Semispace and Sapphire configurations. In each case, 95% confidence intervals are shown [Kalibera and Jones 2013]. The OTF column shows the time spent in on-the-fly GC. The final column shows the overhead of Sapphire v. Semispace (ratio of geomeans).

Figure 8(a) shows the distributions of elapsed times for these tasks; note that the y-axis is logarithmic. It does *not* show pause times, which are negligible. Collections running during a task invocation imposes write barrier overhead on the task. We compare the collectors by considering how often a task deadline is missed. Sapphire reduced the rate of 1ms deadline misses by 100× compared to Concurrent Mark-Sweep, from 2.22 misses/second to 0.02 misses/second. Only 37 out of 1,553,076 tasks took longer than 1ms to complete, and none took longer than 7.5ms. For Sapphire, only tasks running against benchmarks with very high allocation rates (*pmd* and *sunflow*, highlighted in the figure) executed in more than 500μs. We conclude that Sapphire provides good responsiveness for applications other than those with very high allocation rates.

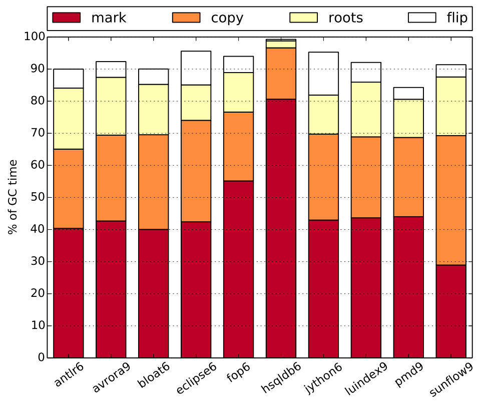
Figure 8(b) shows periodic task execution times across all but two of the benchmarks we used for the Sapphire; *lusearch* and *xalan* are again excluded as they could not avoid stop-the-world pauses. Results for *hsqldb*, *pmd* and *eclipse* are highlighted as they include significant outliers (task executions longer than 1ms) for Sapphire. *Eclipse* seems poorly suited to on-the-fly collection. However, many of the outliers are unrelated to GC — similar pauses outside GC time occur with the Semispace stop-the-world collector. We can reasonably conclude that these outliers are an artefact of Jikes RVM/*eclipse* beyond the scope of this paper. Excluding *eclipse*, we still see that only 37 tasks from over four million samples for Sapphire execute in 1–7.5ms; all other tasks (more than 99.999%) execute in less than 1ms. The geometric mean periodic task execution time outside of garbage collection across all benchmarks is 85μs (median 82μs). During garbage collection this rises to geometric mean 177μs (median 168μs), effectively doubling the time taken to execute a task. Naïvely we could interpret this as indicating that application code will run at half-speed during on-the-fly collection. However, it is important to recognise that our periodic task, composed of only allocation and pointer manipulations, which place most stress on the barriers, represents a worst-case. For example, Table 6 and Figure 9 show *sunflow* spends 67% or more of its execution in on-the-fly collection but suffers only a 28% slowdown, an improvement over the 35% we might naïvely expect. Note that the bars in Figure 9 omit the time spent in the sweep phase for non-moving spaces (as well as times in intermediate phases, none of which are significant).

13.6 Execution time

Any configuration using a concurrent collector will suffer throughput overhead compared with a collector that does not use barriers. Comparing the overall execution and garbage collection times of Sapphire and Semispace in Table 6, we observe some correlation with barrier overheads



(a) Visualisation of execution times from Table 6 where the first column per pair represents Semispace execution times and the second Sapphire.



(b) Breakdown of GC time. Sapphire GC activity showing *Mark*, *Copy* and *Flip* phase groups, and time spent scanning roots, but not sweeping non-moving spaces.

Fig. 9. Garbage collection time.

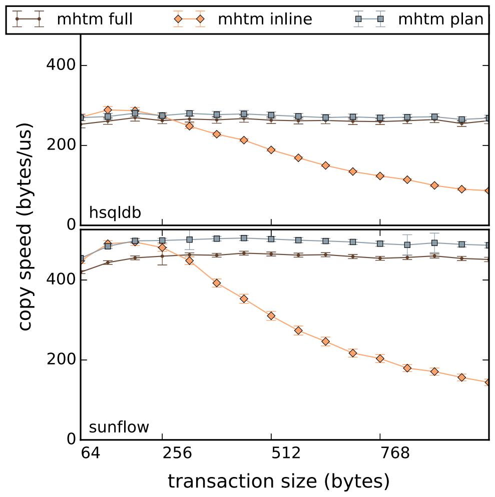
(Table 4) but also some inconsistencies. For most benchmarks where the GC times are comparable (*fop, hsqldb, jython*), Sapphire's overhead is small or negative. Here, the cost of write barriers is offset by running the GC concurrently (though not in the case of *antlr*). For example, *jython* has a stable livesize and steady allocation rate, making it an ideal candidate for on-the-fly collection. It also has a large number of weak references (over 30,000) which are well handled on-the-fly by Sapphire (see Section 13.8 for the details). In other cases, Sapphire spends significantly longer in GC than Semispace, since its collection mechanism is more complex. Here, the cost of mutator slowdown during GC is larger than the cost of Semispace's stop-the-world pauses, leading to an execution time overhead of 15–50%.

Other concurrent, moving collectors suffer similar slowdowns. Even with mutator performance optimised by hot-swapping code to install specialised barriers during collection phases [Arnold and Ryder 2001; Pizlo et al. 2008a], Pizlo et al found that *Chicken* [2008b] exhibited a three-fold slow-down during copying, *Clover* [2008b] five-fold and *Stopless* [2007a] ten-fold.

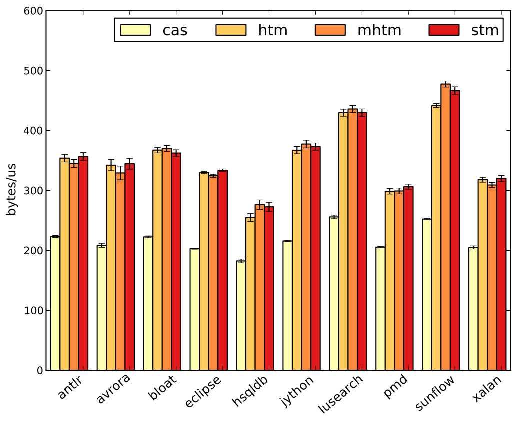
Concurrent collection necessarily requires some headroom for allocation during collection. However, as Table 5 shows, this overhead is modest in the common case (the geometric mean is never above 6MiB except for *sunflow*). For the worse cases, only *pmd* and *sunflow* allocate a substantial volume of memory during GC, 0.57× and 2.07× their maximum livesize respectively.

13.7 Transactions

We explored replacing copying with CAS with transactional copying, using either hardware or software transactions (Section 9). Full details can be found in Ritson et al. [2014]. First, we explored the effect of transaction size on raw copying speed, without mutators running, copying multiple objects in a single hardware transaction. Figure 10(a) shows the results for the fastest (*sunflow)* and slowest (*hsqldb)* copying benchmarks, but all benchmarks exhibited similar behaviour, with the performance of inline copying (*mhtm inline*) tailing off as the transaction size reaches 128 bytes (approximately 2–3 objects), presumably because the read set size becomes too large for most transactions to complete. On the other hand, copying with planned transactions (*mhtm plan*) shows little degradation, whether or not the plan pre-loads information (*mhtm full*) from the object's type



(a) Speed of different transaction size transactions with multi-object copying, using inline (*mhtm inline*), planned (*mhtm plan*) and planned with TIB pre-loading (*mhtm full*) transactions.



(b) Copying speed in Sapphire, using compare-and-swap (*cas*), hardware transactions (*htm*) and its planned multi-object variant (*mhtm plan*), and software transactions (*stm*).

Fig. 10. Transactional copying.

information block (TIB) in order to further reduce the number of cache lines accessed during the transaction.

Figure 10(b) compares the performance, *with* mutators running, of copying by a single collector thread using CAS (*cas*), hardware transactions with one object (*htm*) or multiple objects (*mhtm*) using a transaction size of 256 bytes, or software transactions (*stm*); the error bars represent 95% confidence intervals. Both transactional methods show substantial performance improvement over the original CAS technique. Both variants have comparable performance, suggesting that performance gains can be made without hardware support.

13.8 Reference types

Different applications vary widely in their use of reference types. Figure 11 shows the number of calls of a get method per second in each 10ms time window; the x-axis is the normalised elapsed time of the program. Contrary to our expectations, some programs used reference types heavily (more than 1 million times per second), but this varies between programs. Most showed a small peak at the beginning of execution; we found that these were due to the Jikes RVM class loader. Often programs made little further use of reference types but *jython*$_6$ made heavy use of them in particular phases. In contrast, *lusearch*$_6$, *lusearch*$_9$, *xalan*$_6$ and *xalan*$_9$ made substantial use of reference types for much of their execution.

We compared the performance of different techniques for managing reference types. These can be protected at collection time through a global mutual exclusion lock that prevents any mutator from turning a weakly reachable object into a strongly reachable one, or without blocking mutators as we discussed in Section 10. Either an insertion or a deletion barrier can be used. We provide a detailed evaluation in Ugawa et al. [2014]. Here, we consider the effect of different techniques on execution times.

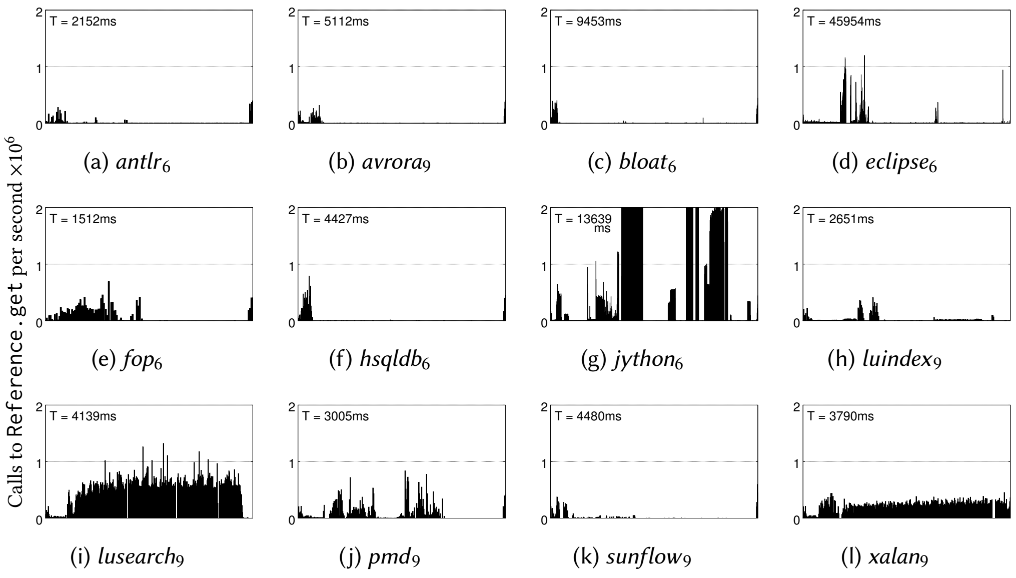


Fig. 11. Frequency of calls to Reference.get in DaCapo (10ms quanta); the x-axis is the normalised entire execution time, T.

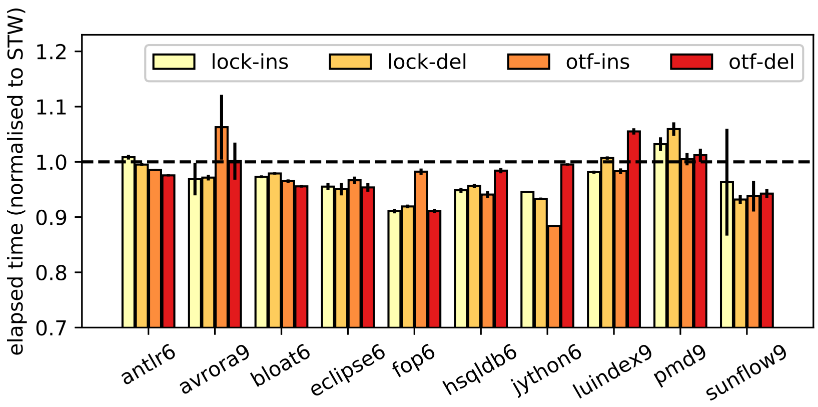


Fig. 12. Execution times with different reference processing approaches (either using a *lock* in get or processing on-the-fly, *otf*) and different barriers (insertion, *ins*, or deletion, *del*). The bars show geometric means and 95% confidence intervals, normalised against processing times with mutators stopped.

In summary, we found that execution using a mostly or fully concurrent technique tended to be shorter than if reference types were processed with the world stopped (Figure 12). This is because (most) mutators continue to run, in sharp contrast to the other approaches, provided they do not exhaust memory (admittedly this risk is increased as mutators run for longer between collection cycles). We also found that the on-the-fly reference processing phase was likely to converge fairly quickly if we use deletion barriers (Table 7). For most of the benchmarks, any of

	lock insertion				lock deletion				OTF ins'n		OTF del'n	
	active		block		active		block		iteration		iteration	
	90%	99%	90%	99%	90%	99%	90%	99%	90%	99%	90%	99%
antlr6	3	3	1	1	3	3	1	1	2	11	2	3
avrora9	6	8	1	1	5	8	1	1	11	20	2	6
bloat6	3	3	1	1	3	3	1	1	1	12	1	2
eclipse6	5	5	1	2	5	5	1	2	2	12	1	4
fop6	3	3	1	1	3	3	1	1	2	19	2	4
hsqldb6	3	3	0	0	3	3	1	1	1	2	1	2
jython6	3	3	1	1	3	3	1	1	2	13	1	4
luindex9	3	3	1	1	3	3	1	1	5	11	2	7
pmd9	6	11	1	5	6	17	2	9	3	7	2	4
sunflow9	34	34	0	1	34	34	0	1	1	9	1	4

Table 7. The number of threads that were actually runnable or would be runnable if they had not tried to take the lock for `Reference.get` (*active*), and the number of threads that waited for the lock (*block*) at the end of the ReferenceProcess phase (left). The count of iterations to terminate the ReferenceProcess phase using on-the-fly techniques(right). 90 and 99 percentiles are shown for each technique.

our reference processing techniques led to faster execution times than with stopping the world to process reference types. For all but *pmd*$_9$, some mostly or fully concurrent technique was significantly faster than stop-the-world processing. There was no clear overall winner among the locking and on-the-fly techniques. For example, although on-the-fly processing with a deletion barrier often performs well, it performed comparatively poorly for *jython*$_6$ (where some phases of execution use reference types very heavily) and *luindex*$_9$. For most benchmarks, only one or two mutators were blocked by the locking techniques, the exception being *pmd*$_9$: here the on-the-fly techniques worked much better. As expected, on-the-fly processing with deletion barriers needed substantially fewer iterations than with insertion barriers. We conclude that overall execution time is not increased significantly by processing references on-the-fly, and is often reduced.

14 RELATED WORK

In this section, we compare Sapphire with other approaches to incremental and concurrent garbage collection, especially those that move objects in order to defragment the heap.

14.1 Consistency

Object relocation presents a fundamental problem to both incremental and concurrent collectors: how to ensure that both collector and mutator threads see a consistent view of the heap. Incremental techniques compact the heap little by little in a stop-the-world fashion. Like Sapphire, Oracle's *Garbage-First* mostly-concurrent collector [Detlefs et al. 2004] and the incremental *'Train'* collector [Hudson and Moss 1992] abandon hard real-time guarantees for a softer, best-efforts approach in the expectation that it may yield throughput and space usage more appropriate for many applications. Each divides the heap into a number of much smaller regions. Although marking is performed mostly concurrently, mutators are blocked while objects in a victim region(s) are evacuated. In contrast, Sapphire evacuates *fromspace* as a whole, and has no stop-the-world phase.

If we allow mutators to run while objects are relocated, we need read barriers or replicating collection. Many real-time collectors use read barriers to ensure that, at any moment, only a single location holds the latest value of a field of an object. Bacon et al. [2003] discuss both an eager *tospace* invariant for *Metronome* (under which a thread always accesses *tospace* for the up-to-date version of an object) and a lazy read barrier (which does not forward a pointer until it is used). *Staccato* [McCloskey et al. 2008] and *Chicken* [Pizlo et al. 2008b] impose a lazy *tospace* invariant under which the latest value is the *tospace* one if the object has been copied. *Stopless* [Pizlo et al. 2007a] imposes a more complicated invariant, but still only one location holds the valid latest value of a field. As concurrent collectors move objects while the mutator is running, mutators and collectors must maintain a coherent view of the heap at each read or write operation, requiring frequent synchronisation.

Replicating collectors relax mutator-collector coherency requirements by allowing the mutator to access *fromspace* objects while the collector is copying them to *tospace*. Nettles et al. [1992] originally proposed replication as an incremental copying method that did not rely on expensive read barriers. To maintain consistency between replicas, the mutator's insertion write barrier logged *any* modification to previously replicated *fromspace* objects. The collector processed these logs to make semantically equivalent modifications to the *tospace* replicas before the mutator's roots were flipped to point to *tospace*. Collection was complete when the collector's work list and the mutation log were both empty. Unfortunately, the Nettles and O'Toole write logs are not bounded in size as any mutation must be logged. Thus, at some point, they must stop all mutator threads to complete the processing of these logs, including copying any previously undiscovered, reachable objects. *Sapphire* does not log updates in this way for two reasons. First, it is intended as an on-the-fly collector so it should not stop the world. Second, it was designed for Java, an object oriented language, where the number of mutations can be expected to be very much higher than in a mostly functional language like ML. Instead, Sapphire has the mutator write barrier apply changes immediately to both *fromspace* and *tospace* replicas.

All these consistency mechanisms rely on barriers, which impose an overhead on mutator threads and increase the size of mutator code, with possible effects on caching. However, there are opportunities to use static analysis or compiler optimisations to reduce barrier overheads. We follow the standard MMTk practice of separating barriers into an inline fast path and an out-of-line method call which, if necessary, does the work (see Section 5.4). Bacon et al. [2003] use compiler optimisations to reduce the cost of read barriers, in particular by reducing the number of null checks; similarly, we eliminate the pointer-equality barrier, Algorithm 9(a), if either operand is null. Adl-Tabatabai et al. [2009] similarly suggest opportunities for eliminating redundant operations across multiple barriers in the context of software transactional memory systems. We also rely on Jikes RVM's optimising compiler to reduce the cost of barriers. Pizlo et al. [2008a] propose *path specialization* to reduce barrier overhead, creating multiple copies of the code, modifying each copy to handle one or more phases of the collector. Code is then patched at run-time to call the copy corresponding to the current collector phase. This can remove phase-testing overhead in barriers and all overhead in phases where no barriers are active. We leave further barrier elimination for future work.

For on-the-fly collectors, how to change phases without stopping all mutators is another difficulty. The standard technique is that the collector handshakes with each mutator one by one. The Doligez and Leroy [1993] collector was the first to adopt ragged phase changes. Doligez and Gonthier [1994] corrected this algorithm by introducing two intermediate phases between 'no collection' and the mark phase in order to support multiple mutators. Domani [Domani et al. 2000] adapted their collector for Java. Our design patterns generalise the Doligez-Leroy-Gonthier approach to support all types of ragged phase change that might be used by a collector.

14.2 Virtual memory techniques

A number of concurrent and incremental collectors use page protection to provide synchronisation. The *Compressor* [Kermany and Petrank 2006] first constructs a compaction plan to identify the destination of each object to be moved. It then protects *tospace* pages from read and write access, and flips the mutators' roots. Whenever a mutator traps on a protected *tospace* page, the trap handler copies its objects from *fromspace* before resuming the mutator.

In contrast, *Pauseless* [Click et al. 2005] and its generational extension, *C4* [Tene et al. 2011], protect only those *fromspace* pages from which objects are being moved. Their read barrier repairs stale *fromspace* references. The original implementation for the Azul Vega platform implemented the barrier through a special load-reference instruction, but on stock hardware Pauseless compiles the barrier inline. Azul's proprietary hardware also supported fast user-mode trap handlers and added a GC-mode and larger pages to the translation lookaside buffer. More recently Azul has provided patches for the Linux kernel to support their algorithm on stock hardware [Azul 2010].

14.3 Real-time collection

Soft real-time goals that typically include a throughput requirement and an acceptable rate of failure to meet deadlines are sufficient for many applications [Printezis 2006]. However, hard real-time systems require all high-priority tasks to respond to events within a fixed time, and must schedule tasks so that their real-time constraints are met. This requres performing schedulability analysis ahead-of-time, assuming a particular (usually priority-based) run-time scheduling algorithm [Kalibera et al. 2009, 2011]. Scheduling may be work-based [Cheng and Blelloch 2001], slack-based [Henriksson 1998], time-based [Bacon et al. 2003] or a combination [Auerbach et al. 2008]. Kalibera et al. [2011] explore scheduling strategies in detail in a reimplementation of the Metronome collector on the Ovm platform [Armbruster et al. 2007]. They found that no one scheduling strategy was strictly preferable for all applications. Barrier overheads led to a mean of 69% slowdown on computational tasks (16% in a newer version of their virtual machine) but they comment that this could probably be improved with compiler optimisations. Our soft real-time Sapphire collector is oblivious to thread schedules, which are managed by the underlying Linux scheduler.

Metronome is one of the best-known real-time collectors. This incremental mark-sweep collector uses a deletion write barrier, Brooks-style forwarding pointers [1984], and time-based scheduling, and works best on uniprocessor or small multiprocessor systems. To support fine-grained, time-based work scheduling, Metronome's design allows it to do as much work, including evacuation, as possible within a given time quantum while mutators are stopped. Thus, it implements arrays with arraylets, which also helps avoid fragmentation. In contrast, Sapphire never stops all mutators. Metronome's *Tax-and-Spend* extension [Auerbach et al. 2008] never compacts. *Schism* [Pizlo et al. 2010] similarly adopts arraylets and relocates only array metadata.

Kalibera's replicating collector [2009] also uses a Brooks barrier but has the forwarding pointer held in the *tospace* replica point back to its *fromspace* original, thus admitting very simple and predictable mutator barriers. Kalibera and Jones [2011] showed how a handle-based VM offers the benefit of *immediate* reuse of space used by evacuated objects without a copy reserve overhead. They demonstrated, contrary to popular belief, that, with careful optimisations, this system achieves the same execution time overheads as a Brooks-style compacting collector. However, both collectors run only on uniprocessors with 'green' (VM-scheduled) thread scheduling.

Stopless [Pizlo et al. 2007a] was the first collector to claim lock-free relocation of objects. It ensures that all updates are made to the most up-to-date location of a field, thereby supporting mutators' use of atomic operations without locking. Its complicated protocol uses intermediate, 'wide' versions of objects and requires at least two double-word compare-and-swap atomic operations to copy a word. The write barrier also involves a double-word compare-and-swap operation.

Staccato [McCloskey et al. 2008] permits concurrent compaction without requiring locks. It attempts to move only a few objects at a time (for instance, to reclaim sparsely-occupied pages), with mutators aborting relocation if they write to an object being copied. A handshake between mutators and the collector ensures that reads find the latest location of a value without fine-grained synchronisation. *Chicken* [Pizlo et al. 2008b] is similar to Staccato but assumes the stronger memory model of x86/x86-64, and so only writes abort copying (because atomic operations order reads); it also uses few memory fences.

Clover [Pizlo et al. 2008b] relies on probabilistic detection of writes by the mutator to deliver guaranteed copying with lock-free mutator accesses except in rare cases, and lock-free copying by the collector. Instead of preventing races, the collector marks fields being copied by overwriting them atomically with a randomly chosen value α that Clover assumes the mutator will never write to the heap. The write barrier blocks the mutator whenever it attempts to write this value and causes the mutator to store into the up-to-date location of the object.

Of these three collectors, Stopless cannot guarantee collector progress, since repeated writes to a 'wide' copy may postpone copying indefinitely; Chicken guarantees progress, but at the expense of aborting some copies; and Clover is claimed to guarantee progress, though it may stall waiting to install α into a field that the mutator is repeatedly updating.

It is appealing to try to perform memory management in hardware. There has been little recent work here (see Jones [1996] for examples of earlier work). Notable exceptions include work by Bacon et al. with reconfigurable hardware [2012; 2014] and Puffitsch's use of a hardware copying unit [2011; 2013].

14.4 Transactions

To the best of our knowledge, McGachey et al. [2008] were the first to use software transactional memory to support GC. Their concurrent GC supported a version of Java extended with an **`atomic`** construct. Read and write barriers were used both for transactional code and to provide strong atomicity. In contrast to our Sapphire implementation which places no restrictions on mutators, McGachey et al. must put an object into exclusive mode before writing to it. The GC notes a version number stored in the object's header before copying it and checks that the number has not changed afterwards. If it has, the copy aborts and has to be retried. Only one object is copied in a transaction.

The *Collie* on-the-fly collector [Iyengar et al. 2012] uses a hardware-supported read barrier on Azul Systems Vega processors to ensure that mutators always access *tospace* replicas, unlike Sapphire which accesses *fromspace* until its RootFlip phase. To provide wait-freedom, an object referenced from a root or accessed by a mutator while Collie is trying to move it is not physically but virtually copied, by mapping its *fromspace* page to the same physical memory as its mirrored *tospace* page. During the compaction phase, read and write barriers mark objects as non-transportable, replacing references to them with references to their corresponding address on the mirrored to-space page. In contrast, our copying collectors never pin *fromspace* objects. During its mark phase, Collie constructs conservative 'referrer sets' of objects holding references to each object. To transplant an individual object, Collie copies its contents before it starts a transaction. Inside the transaction, it checks the references in the referrer set: if any point to the mirrored to-space, the transaction (and the copy) is aborted. Otherwise the transaction commits. Like our implementations, Collie tries to reduce the size of transactional read and write sets.

15 CONCLUSIONS

We have described our experiences building Sapphire, a fully concurrent copying garbage collector, for a widely used JVM. We provide for the first time within Jikes RVM a general framework for on-the-fly copying garbage collection, including a new and general design pattern for managing ragged changes of GC phases. Our implementation supports the complete Java language, including correct and on-the-fly handling of locking, hashing and reference types. Building any high performance collector is complex, and an on-the-fly copying collector particularly so. An ad hoc approach to construction would have been infeasible. We show how a methodology of defining abstractions, identifying invariants that can be enforced through write (and one read) barriers, and model checking leads to a reliable implementation. In contrast to Hudson and Moss's original Sapphire, our implementation uses parallel threads for GC. We introduce high performance object copying by using planned transactions: we find that hardware and software transactions offer similar levels of performance. We add simpler, yet lock-free, support for volatiles.

The source code of our collector, our SPIN models and a technical report describing the models in detail can be found at https://github.com/rejones/sapphire.

Sapphire is well suited to soft real-time applications. For the first time, we evaluate Sapphire against a set of realistic benchmarks. Our implementation achieves significant improvements in responsiveness over Jikes RVM's Concurrent Mark-Sweep collector, and is able to reduce the effect of the garbage collector on response times to less than 1ms for 99.999% of invocations of a periodic task.

ACKNOWLEDGMENTS

We are grateful to Rick Hudson and the Intel Corporation for providing us with a license to implement the Sapphire algorithm in Jikes RVM, and permitting others to access and use this code under the Eclipse Public License. We are also grateful for the support of the EPSRC through grants EP/H026975/1 (http://gow.epsrc.ac.uk/NGBOViewGrant.aspx?GrantRef=EP/H026975/1) and EP/M017176/1 (http://gow.epsrc.ac.uk/NGBOViewGrant.aspx?GrantRef=EP/M017176/1), the JSPS through KAKENHI grant numbers 25330080 and 16K00103, and Google through their Summer of Code (https://summerofcode.withgoogle.com) programme. We acknowledge the contribution of Laurence Hellyer, who designed and wrote much of the code for an earlier version of Sapphire in Jikes RVM/MMTk.

REFERENCES

Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. 2009. Optimizing Memory Transactions for Multicore Systems. In *Multicore Processors and Systems*. Springer, Chapter 5, 145–172. https://doi.org/10.1007/978-1-4419-0263-4_5

Bowen Alpern, Maria Butrico, Anthony Cocchi, Julian Dolby, Stephen Fink, David Grove, and Ton Ngo. 2002. Experiences Porting the Jikes RVM to Linux/IA32. In *Java Virtual Machine Research and Technology Symposium*. USENIX, San Francisco, CA, 51–64. http://www.research.ibm.com/people/d/dgrove/papers/jvm01.pdf

Austin Armbruster, Jason Baker, Antonio Cunei, Chapman Flack, David Holmes, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. 2007. A Real-Time Java Virtual Machine with Applications in Avionics. *ACM Transactions on Embedded Computer Systems* 7, 1, Article 5 (2007). https://doi.org/10.1145/1324969.1324974 Supersedes [Baker et al. 2006].

Matthew Arnold and Barbara G. Ryder. 2001. A Framework for Reducing the Cost of Instrumented Code, See [PLDI 2001 2001], 168–179. https://doi.org/10.1145/378795.378832

Joshua Auerbach, David F. Bacon, Perry Cheng, David Grove, Ben Biron, Charlie Gracie, Bill McCloskey, Aleksandar Micic, and Ryan Sciampacone. 2008. Tax-and-Spend: Democratic Scheduling for Real-Time Garbage Collection. In *ACM International Conference on Embedded Software*. ACM Press, Atlanta, GA, 245–254. https://doi.org/10.1145/1450058.1450092

Azul. 2010. *Comparison of Virtual Memory Manipulation Metrics*. White paper. Azul Systems Inc. http://www.managedruntime.org/files/downloads/AzulVmemMetricsMRI.pdf

David F. Bacon, Perry Cheng, and V.T. Rajan. 2003. A Real-Time Garbage Collector with Low Overhead and Consistent Utilization. In *ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, New Orleans, LA, 285–298.

https://doi.org/10.1145/604131.604155

David F. Bacon, Perry Cheng, and Sunil Shukla. 2012. And Then There Were None: a Stall-free Real-time Garbage Collector for Reconfigurable Hardware. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM Press, Beijing, China, 23–34. https://doi.org/10.1145/2254064.2254068

David F. Bacon, Perry Cheng, and Sunil Shukla. 2014. Parallel Real-Time Garbage Collection of Multiple Heaps in Reconfigurable Hardware, See [Guyer and Grove 2014], 117–127. https://doi.org/10.1145/2602988.2602996

David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. 1998. Thin Locks: Featherweight Synchronization for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM Press, 258–268. https://doi.org/10.1145/277650.277734

Henry G. Baker. 1978. List Processing in Real-Time on a Serial Computer. *Commun. ACM* 21, 4 (1978), 280–294. https://doi.org/10.1145/359460.359470 Also AI Laboratory Working Paper 139, 1977.

Henry G. Baker. 1992. The Treadmill, Real-time Garbage Collection without Motion Sickness. *ACM SIGPLAN Notices* 27, 3 (March 1992), 66–70. https://doi.org/10.1145/130854.130862

Jason Baker, Antonio Cunei, Chapman Flack, Filip Pizlo, Marek Prochazka, Jan Vitek, Austin Armbruster, Edward Pla, and David Holmes. 2006. A Real-Time Java Virtual Machine for Avionics — An Experience Report. In *IEEE Real-Time and Embedded Technology and Applications Symposium.* San Jose, CA, 384–396. https://doi.org/10.1109/RTAS.2006.7

Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications.* ACM Press, Vancouver, 52:1–52:27. https://doi.org/10.1145/3133876

Yves Bekkers and Jacques Cohen (Eds.). 1992. *International Workshop on Memory Management.* Lecture Notes in Computer Science, Vol. 637. Springer, St Malo, France. https://doi.org/10.1007/BFb0017181

Mordechai Ben-Ari. 1984. Algorithms for On-The-Fly Garbage Collection. *ACM Transactions on Programming Languages and Systems* 6, 3 (July 1984), 333–344. https://doi.org/10.1145/579.587

Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *International Conference on Software Engineering.* IEEE Computer Society Press, Edinburgh, 137–146. https://doi.org/10.1109/ICSE.2004.1317436

Stephen M. Blackburn, Robin Garner, Chriss Hoffman, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications.* ACM Press, Portland, OR, 169–190. https://doi.org/10.1145/1167473.1167488

Stephen M. Blackburn and Kathryn S. McKinley. 2002. In or Out? Putting Write Barriers in Their Place. In *ACM SIGPLAN International Symposium on Memory Management.* ACM Press, Berlin, Germany, 175–184. https://doi.org/10.1145/512429.512452

Hans Boehm and David Bacon (Eds.). 2011. *ACM SIGPLAN International Symposium on Memory Management.* ACM Press, San Jose, CA. https://doi.org/10.1145/1993478

Hans-Juergen Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model, See [Gupta and Amarasinghe 2008], 68–78. https://doi.org/10.1145/1375581.1375591

Rodney A. Brooks. 1984. Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware. In *ACM Conference on LISP and Functional Programming.* ACM Press, Austin, TX, 256–262. https://doi.org/10.1145/800055.802042

Perry Cheng and Guy Blelloch. 2001. A Parallel, Real-Time Garbage Collector, See [PLDI 2001 2001], 125–136. https://doi.org/10.1145/378795.378823

Michal Cierniak, Marsha Eng, Neal Glew, Brian Lewis, and James Stichnoth. 2005. The Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment. *Concurrency and Computation: Practice and Experience* 17, 5–6 (2005), 617–637. https://doi.org/10.1002/cpe.852

Cliff Click, Gil Tene, and Michael Wolf. 2005. The Pauseless GC Algorithm. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments.* ACM Press, Chicago, IL, 46–56. https://doi.org/10.1145/1064979.1064988

David Detlefs, Christine Flood, Steven Heller, and Tony Printezis. 2004. Garbage-First Garbage Collection. In *ACM SIGPLAN International Symposium on Memory Management.* ACM Press, Vancouver, Canada, 37–48. https://doi.org/10.1145/1029873.1029879

Sylvia Dieckmann and Urs Hölzle. 1999. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. In *European Conference on Object-Oriented Programming (Lecture Notes in Computer Science)*, Vol. 1628. Springer-Verlag, Lisbon, Portugal, 92–115. https://doi.org/10.1007/3-540-48743-3_5

Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. 1978. On-The-Fly Garbage Collection: An exercise in Cooperation. *Commun. ACM* 21, 11 (Nov. 1978), 965–975. https://doi.org/10.1145/359642.359655

Damien Doligez and Georges Gonthier. 1994. Portable, Unobtrusive Garbage Collection for Multiprocessor Systems. In *ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, Portland, OR, 70–83. https://doi.org/10.1145/174675.174673

Damien Doligez and Xavier Leroy. 1993. A Concurrent Generational Garbage Collector for a Multi-Threaded Implementation of ML. In *ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, Charleston, SC, 113–123. https://doi.org/10.1145/158511.158611

Tamar Domani, Elliot K. Kolodner, and Erez Petrank. 2000. A Generational On-the-fly Garbage Collector for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, Vancouver, Canada, 274–284. https://doi.org/10.1145/349299.349336

Robin J. Garner, Stephen M. Blackburn, and Daniel Frampton. 2011. A Comprehensive Evaluation of Object Scanning Techniques, See [Boehm and Bacon 2011], 33–42. https://doi.org/10.1145/1993478.1993484

Andy Georges, Lieven Eeckhout, and Dries Buytaert. 2008. Java Performance Evaluation through Rigorous Replay Compilation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, Nashville, TN, 367–384. https://doi.org/10.1145/1449764.1449794

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2015. *The Java Language Specification* (Java SE 8 ed.). Addison Wesley. https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

Rajiv Gupta and Saman P. Amarasinghe (Eds.). 2008. *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, Tucson, AZ. https://doi.org/10.1145/1375581

Samuel Z. Guyer and David Grove (Eds.). 2014. *ACM SIGPLAN International Symposium on Memory Management*. ACM Press, Edinburgh.

Roger Henriksson. 1998. *Scheduling Garbage Collection in Embedded Systems*. Ph.D. Dissertation. Lund Institute of Technology. http://www.dna.lth.se/home/Roger_Henriksson/

Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufman.

Maurice P. Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ACM/IEEE International Symposium on Computer Architecture*. IEEE Press, San Diego, CA, 289–300. https://doi.org/10.1145/165123.165164

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

Shin Hong and Moonzoo Kim. 2015. A Survey of Race Bug Detection Techniques for Multithreaded Programmes. *Software: Testing, Verification and Reliability* 25, 3 (May 2015), 191–217. https://doi.org/10.1002/stvr.1564

Richard L. Hudson and J. Eliot B. Moss. 1992. Incremental Collection of Mature Objects, See [Bekkers and Cohen 1992], 388–403. https://doi.org/10.1007/BFb0017203

Richard L. Hudson and J. Eliot B. Moss. 2001. Sapphire: Copying GC Without Stopping The World. In *Joint ACM-ISCOPE Java Grande Conference*. ACM Press, Palo Alto, CA, 48–57. https://doi.org/10.1145/376656.376810

Richard L. Hudson and J. Eliot B. Moss. 2003. Sapphire: Copying Garbage Collection Without Stopping the World. *Concurrency and Computation: Practice and Experience* 15, 3–5 (2003), 223–261. https://doi.org/10.1002/cpe.712

R. John M. Hughes. 1982. A Semi-Incremental Garbage Collection Algorithm. *Software: Practice and Experience* 12, 11 (Nov. 1982), 1081–1082. https://doi.org/10.1002/spe.4380121108

Intel. 2013. *Intel® 64 and IA-32 Architectures Software Developer's Manual*.

Balaji Iyengar, Gil Tene, Michael Wolf, and Edward Gehringer. 2012. The Collie: a Wait-Free Compacting Collector, See [McKinley and Vechev 2012], 85–96. https://doi.org/10.1145/2258996.2259009

Nicholas Jacek, Meng-Chieh Chiu, Benjamin Marlin, and Eliot Moss. 2016. Assessing the Limits of Program-specific Garbage Collection Performance. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, Santa Barbara, CA, 584–598. https://doi.org/10.1145/2908080.2908120

Richard Jones, Antony Hosking, and Eliot Moss. 2012. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall.

Richard Jones. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester. 403 pages. With a chapter on Distributed Garbage Collection by R. Lins.

Richard Jones and Andy C. King. 2005. A Fast Analysis for Thread-Local Garbage Collection with Dynamic Class Loading. In *IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society Press, Budapest, 129–138. https://doi.org/10.1109/SCAM.2005.1

Tomas Kalibera. 2009. Replicating Real-time Garbage Collector for Java. In *International Workshop on Java Technologies for Real-time and Embedded Systems*. ACM Press, Madrid, Spain, 100–109. https://doi.org/10.1145/1620405.1620420

Tomas Kalibera and Richard Jones. 2011. Handles Revisited: Optimising Performance and Memory Costs in a Real-Time Collector, See [Boehm and Bacon 2011], 89–98. https://doi.org/10.1145/1993478.1993492

Tomas Kalibera and Richard Jones. 2013. Rigorous Benchmarking in Reasonable Time. In *ACM SIGPLAN International Symposium on Memory Management*. ACM Press, Seattle, WA, 63–74. https://doi.org/10.1145/2464157.2464160

Tomas Kalibera, Matthew Mole, Richard Jones, and Jan Vitek. 2012. A Black-Box Approach to Understanding Concurrency in DaCapo. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, Tuscon, AZ, 335–354. http://www.cs.kent.ac.uk/pubs/2012/3246

Tomas Kalibera, Filip Pizlo, Antony L. Hosking, and Jan Vitek. 2009. Scheduling Hard Real-time Garbage Collection. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, Washington, DC, 81–92. https://doi.org/10.1109/RTSS.2009.40

Tomas Kalibera, Filip Pizlo, Antony L. Hosking, and Jan Vitek. 2011. Scheduling Real-time Garbage Collection on Uniprocessors. *ACM Transactions on Computer Systems* 3, 1 (Aug. 2011), 8:1–29. https://doi.org/10.1145/2003690.2003692

Haim Kermany and Erez Petrank. 2006. The Compressor: Concurrent, Incremental and Parallel Compaction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, Ottawa, Canada, 354–363. https://doi.org/10.1145/1133981.1134023

Bill McCloskey, David F. Bacon, Perry Cheng, and David Grove. 2008. *Staccato: A Parallel and Concurrent Real-time Compacting Garbage Collector for Multiprocessors*. IBM Research Report RC24505. IBM Research. http://researcher.watson.ibm.com/researcher/files/us-groved/rc24504.pdf

Phil McGachey, Ali-Reza Adl-Tabatabi, Richard L. Hudson, Vijay Menon, Bratin Saha, and Tatiana Shpeisman. 2008. Concurrent GC Leveraging Transactional Memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, Salt Lake City, UT, 217–226. https://doi.org/10.1145/1345206.1345238

Kathryn McKinley and Martin Vechev (Eds.). 2012. *ACM SIGPLAN International Symposium on Memory Management*. ACM Press, Beijing, China.

Scott Nettles and James O'Toole. 1993. Real-Time Replication-Based Garbage Collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, Albuquerque, NM, 217–226. https://doi.org/10.1145/155090.155111

Scott M. Nettles, James W. O'Toole, David Pierce, and Nicholas Haines. 1992. Replication-Based Incremental Copying Collection, See [Bekkers and Cohen 1992], 357–364. https://doi.org/10.1007/BFb0017201

Oracle JNI 2015. *Java Native Interface 6.0 API Specification*. http://docs.oracle.com/javase/8/docs/technotes/guides/jni/

Pekka P. Pirinen. 1998. Barrier Techniques for Incremental Tracing. In *ACM SIGPLAN International Symposium on Memory Management*. ACM Press, Vancouver, Canada, 20–25. https://doi.org/10.1145/286860.286863

Filip Pizlo, Daniel Frampton, and Antony L. Hosking. 2011. Fine-grained Adaptive Biased Locking. In *International Conference on Principles and Practice of Programming in Java*. ACM Press, 171–181. https://doi.org/10.1145/2093157.2093184

Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgard. 2007a. Stopless: A Real-Time Garbage Collector for Multiprocessors. In *ACM SIGPLAN International Symposium on Memory Management*. ACM Press, Montréal, Canada, 159–172. https://doi.org/10.1145/1296907.1296927

Filip Pizlo, Antony L. Hosking, and Jan Vitek. 2007b. Hierarchical Real-time Garbage Collection. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM Press, San Diego, CA, 123–133. https://doi.org/10.1145/1254766.1254784

Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. 2008a. Path Specialization: Reducing Phased Execution Overheads. In *ACM SIGPLAN International Symposium on Memory Management*. ACM Press, Tucson, AZ, 81–90. https://doi.org/10.1145/1375634.1375647

Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. 2008b. A Study of Concurrent Real-Time Garbage Collectors, See [Gupta and Amarasinghe 2008], 33–44. https://doi.org/10.1145/1379022.1375587

Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. 2010. Schism: Fragmentation-Tolerant Real-Time Garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, Toronto, Canada, 146–159. https://doi.org/10.1145/1806596.1806615

PLDI 2001 2001. *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, Snowbird, UT. https://doi.org/10.1145/378795

Tony Printezis. 2006. On Measuring Garbage Collection Responsiveness. *Science of Computer Programming* 62, 2 (Oct. 2006), 164–183. https://doi.org/10.1016/j.scico.2006.02.004

Wolfgang Puffitsch. 2011. Hard Real-Time Garbage Collection for a Java Chip Multi-Processor. In *International Workshop on Java Technologies for Real-time and Embedded Systems*. ACM Press, York, 64–73. https://doi.org/10.1145/2043910.2043921

Wolfgang Puffitsch. 2013. Design and Analysis of a Hard Real-time Garbage Collector for a Java Chip Multi-processor. *Concurrency and Computation: Practice and Experience* 25, 16 (2013), 2269–2289. https://doi.org/10.1002/cpe.2921

Carl G. Ritson and Frederick R.M. Barnes. 2013. An Evaluation of Intel's Restricted Transactional Memory for CPAs. In *Communicating Process Architectures*. Open Channel Publishing Ltd, 271–291.

Carl G. Ritson, Tomoharu Ugawa, and Richard Jones. 2014. Exploring Garbage Collection with Haswell Hardware Transactional Memory, See [Guyer and Grove 2014], 105–115. https://doi.org/10.1145/2602988.2602992

Hideaki Saiki, Yoshiharu Konaka, Tsuneyasu Komiya, Masahiro Yasugi, and Taiichi Yuasa. 2005. Real-time GC in JeRTy^{TM}VM Using the Return-Barrier Method. In *IEEE International Symposium on Object-oriented Real-time distributed Computing*.

IEEE, 140–148. https://doi.org/10.1109/ISORC.2005.45

Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: The Continuously Concurrent Compacting Collector, See [Boehm and Bacon 2011], 79–88. https://doi.org/10.1145/1993478.1993491

Tomoharu Ugawa, Richard Jones, and Carl G. Ritson. 2014. Reference Object Processing in On-The-Fly Garbage Collection, See [Guyer and Grove 2014], 59–69. https://doi.org/10.1145/2602988.2602991

Jan Vitek and Tomas Kalibera. 2011. Repeatability, Reproducibility, and Rigor in Systems Research. In *ACM International Conference on Embedded Software*. ACM Press, New York, NY, USA, 33–38. https://doi.org/10.1145/2038642.2038650

Paul R. Wilson. 1994. *Uniprocessor Garbage Collection Techniques*. Technical Report. University of Texas. ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps Expanded version of the IWMM92 paper.

Xi Yang, Stephen M. Blackburn, Daniel Frampton, and Antony L. Hosking. 2012. Barriers Reconsidered, Friendlier Still!, See [McKinley and Vechev 2012], 37–48. https://doi.org/10.1145/2258996.2259004

Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *Symposium on Operating Systems Principles*. ACM Press, 221–234. https://doi.org/10.1145/1095810.1095832