Incremental Garbage Collection Using Method Specialisation
Final Report

Luke Terry
ljt06@doc.ic.ac.uk

Supervisor: Dr Tony Field

June 15, 2010

Abstract

This report documents the development of a prototype incremental garbage collector in the Jikes RVM 3.1 using a novel alternative to standard read barriers known as specialised self-scavenging. Self-scavenging collectors encode scavenge state on a per-object level, which allows individual objects to conditionally activate scavenge code during collection, eliminating the always-on nature of traditional barriers. We develop and evaluate an incremental collector framework coupled with the first traditional incremental Baker-style garbage collector for Jikes. We then modify the Baker collector to use self-scavenging, where objects encode their scavenge state in a header word. We finally eliminate the cost of explicitly encoding state in the object header by using specialised method variants, introduced through an updated method specialisation framework from the Jikes RVM 3.0.1. This results in a collector that has *no* conditional state checks when the collector is off or when an object has been scavenged.

Acknowledgements

I would like to thank my beautiful and understanding fianceé for putting up with me over the last four years, I know at times this was not easy! I extend immense gratitude and respect to my supervisor Tony Field for remaining confident in my ability, for ideas and inspiration as to how to attack the inherent problems I encountered and for proof reading everything an uncountable number of times! I also want to thank Andrew Cheadle and Will Deacon for imparting their knowledge of garbage collection, Jikes and method specialisation to me and for providing invaluable support in the early hours. Finally I want to acknowledge the unrelenting support, wisdom and understanding of my parents who have guided, fed and shaped me into who I am today.

Contents

1 Introduction **4**

2 Background **6**
2.1 Garbage Collection . . . 6
2.1.1 Copying Collectors . . . 8
2.1.1.1 Semi-Space Collector . . . 9
2.1.1.2 Generational Collectors . . . 9
2.1.2 Real-Time Collectors . . . 10
2.1.2.1 Incremental Collectors . . . 11
2.1.2.2 The Tri-Colour Abstraction . . . 11
2.1.2.3 Scheduling . . . 14
2.1.2.4 State-of-the-Art . . . 15
2.1.3 Baker's Incremental Garbage Collection . . . 17
2.2 Jikes RVM . . . 20
2.2.1 MMTk . . . 21
2.2.1.1 Spaces . . . 21
2.2.1.2 Allocation Policies . . . 21
2.2.1.3 Plans . . . 22
2.2.1.4 Collection Policies . . . 23
2.2.1.5 Concurrency . . . 23
2.2.1.6 Testing and Visualisation . . . 24
2.2.1.7 Semi-Space Collector . . . 24
2.2.1.8 Concurrent Mark-Sweep Collector . . . 28
2.3 Method Specialisation . . . 28
2.3.1 Class Transformations . . . 33
2.3.2 Beanification . . . 33
2.4 Related Work . . . 38

3 Design and Implementation **40**
3.1 MMTk Incremental Framework . . . 40
3.1.1 Incremental Closure . . . 41
3.1.2 Work-Based Triggers . . . 45
3.1.2.1 Threading Issues . . . 46
3.1.3 Mutator Object Tracing . . . 47
3.1.4 High-Level Plan . . . 49

3.2 Incremental Baker Collector Implementation . . . 50
3.2.1 Incremental Trigger . . . 50
3.2.2 Forwarding Pointers . . . 51
3.2.3 Read Barrier . . . 51
3.2.4 Cross-Space Write Barrier . . . 53
3.2.5 Calculating Required Incremental Work . . . 54
3.2.6 Missing Objects . . . 56
3.2.6.1 Sanity Checker Confirmation . . . 56
3.2.6.2 Instrumentation . . . 56
3.2.6.3 Missed Barriers . . . 59
3.2.6.4 Race Conditions . . . 62
3.2.6.5 Context Switching . . . 63
3.2.7 Aside: Execution Issues . . . 63
3.3 Method Specialisation . . . 64
3.3.1 Method Specialisation Framework . . . 64
3.3.2 Garbage Collection Specialisation . . . 65
3.3.3 Extending The Beanification Transform . . . 66
3.3.4 Fixing Method Specialisation . . . 69
3.3.4.1 Obsolete Methods . . . 69
3.3.4.2 Specialised Superclass Methods . . . 69
3.3.4.3 Interface Methods . . . 70
3.3.4.4 Final Classes . . . 72
3.4 'Free' Read Barrier . . . 73
3.4.1 Cross-Space Write Barriers . . . 73
3.4.2 Global Beanification . . . 73
3.4.3 Transforming Methods . . . 73
3.4.4 Explicit Self-Scavenging . . . 74
3.4.4.1 First Implementation . . . 76
3.4.4.2 Revised Implementation . . . 78
3.4.4.3 Lazy vs Eager Self-Scavenging . . . 78
3.4.4.4 Early Scavenging . . . 80
3.4.4.5 Problem Classes . . . 80
3.4.4.6 Array Handling . . . 81
3.4.5 Implicit Self-Scavenging . . . 81
3.4.5.1 Flipping Specialisations . . . 81
3.4.5.2 Specialised Self-Scavenge Transform . . . 82
3.4.5.3 Interacting With Collection . . . 85
3.4.6 Issues . . . 88

4 Evaluation **89**
4.1 Beanification . . . 89
4.2 Incremental Baker Collector . . . 91
4.3 Explicit Self-Scavenging . . . 99
4.4 'Free' Read Barrier . . . 99

5 Conclusion **100**
5.1 Future Work . . . 102
5.1.1 Baker Collector Improvements . . . 102
5.1.2 'Free' Read Barrier . . . 103
5.1.3 Path Specialising Collector . . . 103
5.1.4 Instrumentation . . . 104
5.1.5 Specialising JNI and Reflection . . . 105

1 Introduction

All programming language run-times need to support the *allocation* and *reclamation* of memory. Manually managing memory operations can be a cumbersome task for a programmer and inspired the development of automatic memory reclamation policies (*garbage collection*). The problem with garbage collection is that the user application (the *mutator*) cannot perform useful work while collection is in progress, manifested as periods of unresponsiveness. This is made more apparent when a collection cycle runs uninterrupted, to completion, with the mutator paused - the most commonly implemented type of collection algorithm.

Concurrent garbage collectors for real-time and interactive applications have been developed to bound, or minimize, mutator pauses respectively. These collectors run either in parallel (pure *concurrent collection*), or in short interleaved bursts (*incremental collection*). Concurrent collection introduces the possibility for the mutator to gain access to an object that has been marked for collection. Allowing access to a possibly collected object may result in erroneous behaviour. For example, the memory may have been returned to the operating system, therefore, attempted access would violate memory protection. To solve this problem *barriers* are introduced; these intercept object reads or writes and ensure they are to objects that will survive collection. However, especially for reads, these barriers are invoked frequently and can prove expensive. To reduce their processing cost they must be implemented efficiently.

The first incremental garbage collection algorithm, the classical Baker collector, was designed for list processing languages and is a pure *copying collector* [7]. The algorithm interleaves mutator work with increments of object copying (*evacuation*) from an area of memory that contains garbage (*from-space*), to an area that will contain only objects surviving this collection cycle (*to-space*). Using a *read barrier*, the algorithm tests whether the referenced object has already been copied and, if not, copies it. This ensures that the mutator only 'sees' objects in to-space. Although this approach reduces pause times, due to the test-and-branch the read barrier implementation often incurs a very high cost.

This project details the development of an incremental garbage collector called *SSB* (Self-Scavenging Baker). SSB is based on an incremental Baker collector using *method specialisation* [18, 17, 22] in place of the expensive read-barrier usually associated with Baker's scheme. Method specialisation supports the transparent switching between multiple variations of the same virtual methods. We aim to remove the cost of the expensive read barrier test, resulting in a "free" read barrier.

The key to achieving a "free" read barrier is to virtualise all field accesses by generating getter and setter methods, a process known as *beanification*. This means every reference to an object requires a virtual method call. We can then implement a variant of each

method that first evacuates all of its reference-type fields (*scavenging*) before "flipping" to the default variant. We only execute the self-scavenging variant when the object has been copied, but not yet been scavenged. This means we do not have the expensive read barrier test on every read, instead incurring the cost of a single virtualised method call plus a read barrier test, only when an object has not yet been scavenged. We will investigate this approach in the Java Jikes RVM following results obtained using a similar system in the Glasgow Haskell Compiler [16, 18].

We make the following contributions to the Jikes RVM 3.1:

- An incremental garbage collection framework to support the development of work-based incremental collectors (Section 3.1).
- The first implementation of an incremental Baker collector for Jikes (Section 3.2).
- A port of a previous developed method specialisation framework from Jikes 3.0.1 (Section 3.3).
- The discovery and fixing of previously unknown issues within the method specialisation framework (Section 3.3.4).
- A new style of incremental collector that replaces the expensive Baker read barrier with self-scavenging objects. Each object tracks its scavenge status (*scavenged* or *unscavenged*) in a header word and forwards all of its reference-type fields the first time it is accessed in the unscavenged state. When an object is forwarded it sets its status to unscavenged and reverts back to the scavenged state once it self-scavenges (Section 3.4.4).
- A prototype incremental self-scavenging collector that replaces the explicit per-object scavenge status word with an implicitly encoded state using method specialisation. Specialised method variants perform the self-scavenging and revert back to the original method variant when self-scavenging completes (Section 3.4.4).
- The evaluation of the updated method specialisation framework, revealing average overheads of 8.4%, supporting the results obtained in preceding research [18, 17, 22] (Section 4.1).
- The evaluation of the incremental Baker collector on DaCapo and SPECjvm2008 benchmarks that confirms average overheads of 42%, consistent with previous literature [43]. This indicates that implementation in the meta-circular Jikes RVM is no better or worse than implementations in other runtimes or languages (Section 4.2).

2 Background

2.1 Garbage Collection

Garbage Collection removes the programmer's responsibility for managing heap object deallocation. In languages that do not have a garbage collector, such as C, C++ and Ada[1], it is up to the programmer to ensure that objects no longer in use have their memory reclaimed. This poses several issues for the programmer; not reclaiming unused memory leaves memory leaks that may lead to out-of-memory errors. On the other hand reclaiming memory that is still in-use causes memory access errors. There is no easy way to fix memory issues in such situations and this can result in hours of debugging for simple causes.

To free the programmer from this requirement new languages were developed in which the memory used by dead objects was automatically freed and reused. The first of these languages was LISP, designed by John McCarthy in April 1960 [33]. LISP's "reclamation" process means the programmer no longer needs to explicitly allocate and deallocate memory, as it will be done automatically for them. Garbage collected languages can still have memory leaks, but the scope for introducing such leaks is substantially reduced. Garbage collection also increases program reliability and enables programmers to focus on design rather than memory management.

Garbage collection algorithms can be broadly separated into five categories[2]:

- *Reference Counting* - Using a reference counter for each object [20], every new reference to an object increments the object's counter and every dereference decrements the object's counter. When an object's counter reaches zero it means it is no longer being referenced and can subsequently be collected. The memory that used to be occupied by a newly collected object is added to a free list, used for allocating new objects. The space overhead incurred with a counter for each object and the need to synchronise on updating reference counts, means that it is often replaced by alternative algorithms that do not require a storage overhead, or tight synchronisation. Novel approaches to reference counting have seen it successfully applied to multi-processor [29, 30] and generational [12] collectors.

- *Mark-Sweep* - Mark-Sweep algorithms require a mark bit per object that indicates whether an object needs to be collected or not [33]. The algorithm operates in two

[1] Although the language specification allows for automatic memory management it is often not implemented

[2] An excellent survey of garbage collection algorithms is given in [27].

phases: (1) Mark - search the entire heap, setting the mark bit on every object that is live, (2) Sweep - scan the heap again, adding all memory addresses of unmarked objects to the free list. These algorithms are more space-efficient than reference counting, but the two-phase implementation makes it less time-efficient. Real-time mark-sweep algorithms, such as those described in [42, 24], explore different approaches to improving mark-sweep time-efficiency.

- *Mark-Compact* - Mark-Compact algorithms also require a mark bit per object. The algorithm uses the same mark phase as a mark-sweep algorithm, but instead of adding to a free list the live objects are *compacted* (moved to a contiguous area of memory). This leaves a contiguous block of free memory that can be more efficiently allocated using an incrementing bump pointer. The cost of compaction means that these algorithms are not as time-efficient during collection as mark-sweep algorithms, but result in faster allocation and do not suffer from fragmentation issues. There are several approaches to mark-compact algorithms, broadly defined in [27] as (1) arbitrary - objects are moved in no particular order, (2) linearising - related objects are moved together and (3) sliding - all live objects are shifted to one end of the heap. Objects that are related are, in general, accessed at the same time and thus the number of memory cache misses can be reduced by grouping them together in memory (*spacial locality*) [26]. While arbitrary compaction is fast, it means that you cannot benefit from spacial locality achieved by linearising or sliding algorithms.

- *Copying* - Copying algorithms segregate the heap into several partitions. When a partition becomes full the contained live objects are copied to another partition. This method, in general, requires more memory than mark-compaction algorithms, but uses a more efficient method for moving live objects. More complex copying collectors, such as generational collectors (see Section 2.1.1.2), attempt to address the issue of space-efficiency by using a mark-sweep, mark-compact or reference counting algorithm for one or more of the partitions. Initially all objects are stored in a *nursery* partition. When the nursery is full, selected objects are copied into a different partition based on some criteria, for example objects that have survived a specified number of collection cycles.

- *Region-Based* - Region based algorithms separate the heap into regions, which may be independently managed. The idea is that an entire region can be deallocated along with all of its objects, rather than deallocating individual objects. Recent work in region-based collectors produced Immix [13], now the default collector in the Jikes RVM.

Garbage collection algorithms can also be separated into classes according to how and when collection is performed:

- *Stop-the-world* - Stop-the-world collectors perform garbage collection uninterrupted, to completion, with the user application paused. This results in, sometimes long, periods of unresponsiveness. These collectors are the most widely implemented, but they are undesirable for interactive and real-time applications that must have guaranteed and short pause times.

- *Parallel* - Like stop-the-world collectors, parallel collectors perform garbage collection in one go. To reduce the pause times encountered during collection several threads are used in parallel to perform collection. All threads participating in garbage collection need to synchronise to ensure all collection work is performed. Simple parallel collectors are not desirable for interactive and real-time applications, because they perform all work in one-go, which means the time taken is proportional to the number of live objects.

- *Incremental* - Incremental collectors perform garbage collection work in frequent, but short bursts. During these bursts the mutator is paused, resulting in short and frequent pauses. As the mutator is performing work interleaved with the collector the heap will be in an inconsistent state. The collector and mutator need to cooperate, so that, before a mutator reads or writes to the heap the collector is given an opportunity to ensure the modification is not invalid. These collectors are ideal for interactive applications, because the short pauses do not result in application unresponsiveness. They cannot always guarantee the mutator will perform useful work during each time-slice and so are not ideal for real-time applications.

- *Concurrent* - Concurrent collectors have their own threads that run in parallel with the mutator. When garbage collection is started the collector threads will work to clear garbage without pausing the mutator. This makes them highly desirable for real-time and interactive applications. As with incremental collectors, the heap may be in an inconsistent state and so concurrent collectors also require cooperation from the mutator.

This project specifically targets an incremental copying garbage collector known as the classical Baker garbage collector [7].

2.1.1 Copying Collectors

Copying collectors are the dual to compacting collectors[3]. Essentially the heap is separated into two or more partitions, depending on the underlying algorithm. When one of the partitions can no longer allocate new objects, the objects that are still reachable are

[3]Compacting collectors use a single heap space. Once the garbage is found and the space reclaimed the remaining live words are compacted. This leaves only contiguous free space that can be allocated using a bump-pointer.

copied to another partition. This method is repeated whenever a partition becomes full; out of memory errors are thrown when all candidate partitions are full. The simplest of these algorithms is the semi-space collector.

2.1.1.1 Semi-Space Collector

A semi-space collector splits the heap exactly into two equally sized areas of memory. One of these is called the *to-space*, and is where objects will be allocated using a bump pointer, the second is called *from-space*. When to-space becomes full a garbage collection cycle is started, to-space becomes from-space and vice versa. The set of all objects that are unconditionally referenced, such as those referenced in the stack or registers, the *root set*, are immediately copied into the new to-space. Depending on the algorithm all remaining reachable objects are copied into to-space. Once all of the live-objects are copied, from-space only contains garbage. The process is then repeated at the next collection.

2.1.1.2 Generational Collectors

Generational collectors are a more complex form of copying collector, relying on heuristics that predict a certain pattern to object reachability based on the lifespan of an object. *Infant mortality* is a heuristic that observes the most recently allocated objects are also the ones likely to die quickly [8]. This gives rise to a partitioned heap, where one partition (the *nursery*) deals with recently allocated objects and uses an algorithm efficient at reclaiming lots of object's memory in quick succession (such as a mark-compaction collector). The other partitions contain objects that have survived several collection cycles (which are considered *mature* and unlikely to die) and use a simpler collector (like mark-sweep).

Because generational collectors use heuristics they are not guaranteed to collect all unreachable objects during a cycle. Indeed, if an object is promoted to a higher generation and immediately becomes unreachable it may not be collected until the mature space is traced much later. Generational collectors perform a collection cycle over the mature space when it becomes full or after a certain amount of time, to remove those unreachable objects. This is referred to as a *major* collection cycle; the collection of the nursery is known as a *minor* collection.

A key issue associated with generational collectors is the tracing of the root set of reachable objects. For collection of the nursery, any objects that are referenced by objects in mature spaces must be part of the root set for that collection. This means determining inter-generational pointers, which requires the mutator to communicate such information to the collector. The most common method of determining inter-generational pointers is to record them as they are created in *remembered sets*. Each generation, except for the youngest, has a remembered set that tracks those objects in the generation which refer to objects in a lower generation. When a generation is garbage collected, the remem-

bered sets of higher generations are scanned to determine which objects in the collected generation should be added to the root set, i.e. those objects referenced by objects in a higher generation. The alternative is to search the entire heap for such pointers on each minor collection, but this is very expensive.

2.1.2 Real-Time Collectors

A *real-time* collector ensures that the mutator is paused for a bounded fixed amount of time. In *soft* real-time environments, missing the deadline does not result in a system failure (like *hard* real-time), but often results in degraded performance. How "real-time" a collector is should be measured using the minimum mutator utilisation (MMU) [19]. The MMU calculates the minimum amount of mutator progress (mutator execution time), within a fixed time-slice, over the lifetime of an application. A low MMU indicates that the collector is not performing in real-time, because the mutator is not always running for the majority of each time-slice.

While stop-the-world collectors prevent the mutator from running during a collection cycle, real-time collectors do not. Instead they must ensure that they maintain a consistent view of the heap. In order to achieve this they must synchronise with the mutator. Synchronisation can occur whenever a pointer is dereferenced (a read-barrier) or whenever a pointer is written (a write-barrier). The two barriers can be of different granularity, but ensure that any reads or writes will be made to objects that will survive garbage collection.

There are two major forms of real-time collector, concurrent and incremental. Concurrent collectors have independent mutator and collector threads. They therefore require a synchronisation protocol so that neither thread is starved of processor resources. Incremental collectors often only have a single thread that handles mutator and collector work. A visual comparison of mutator and collector execution for concurrent collectors can be seen in Figure 2.1.

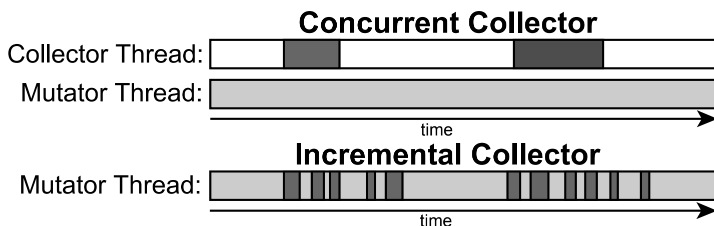


Figure 2.1: Thread view of mutator/collector work for concurrent and incremental collectors. A concurrent collector is constantly executing the mutator and executes collection work on a separate thread when required. An incremental collector executes small bursts of collection work when collection is in progress, during these bursts the mutator is paused.

2.1.2.1 Incremental Collectors

Incremental collectors perform increments of a collection cycle interleaved with mutator execution. In this way they only lead to short, but frequent, pauses of the mutator, during which a small part of the collection cycle is performed. This makes them highly desirable for interactive applications.

Despite the potential benefits, implementing a read-barrier can be very expensive, because a test needs to be executed for every pointer load. This is covered in more detail in Section 2.1.3.

Incremental collectors that use work-based scheduling (see Section 2.1.2.3) are not guaranteed to be real-time. When allocation rates are high so are collection rates and this can result in the violation of real-time deadlines. Current research has moved into more complex time-based (see Section 2.1.2.3) and priority-based scheduling to avoid linking to specific mutator work.

2.1.2.2 The Tri-Colour Abstraction

In order to reason about the state of the heap more easily we consider an abstraction [24] that describes the three different object states encountered during garbage collection: white, grey and black objects.

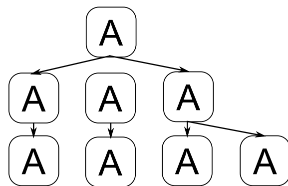
Initially all objects are coloured white as illustrated in Figure 2.2a. Starting from the root set, the current objects are marked grey (Figure 2.2b). The newly greyed objects are traced and all of its children are coloured grey (Figure 2.2c). After this the parent is coloured black and the children are recursively traced (Figures 2.2d and 2.2e).

When tracing has finished only white and black objects are left; the white objects are those that have not been traced and are thus garbage, while the black objects are live objects.

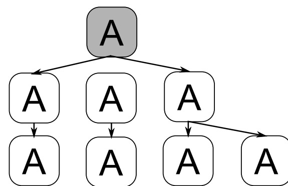
A black object can be assigned a white child in a real-time garbage collector, where the heap is concurrently being traced alongside the mutator. In this situation the white object will never be traced (Figure 2.3), because the black object has already traced its children, and will subsequently be garbage collected. Undefined behaviour occurs when the black object tries to access it (potentially a memory access error).

Preventing a black object from seeing a white object is known as the *strong tri-colour invariant* [18]. In order to prevent undefined behaviour the collector needs to know when a black object has been assigned a white child and recolour either the black object grey, as shown in Figure 2.4a, or the white object grey (Figure 2.4b). To maintain the strong tri-colour invariant a synchronization barrier is used, which can either be a read or a write barrier.

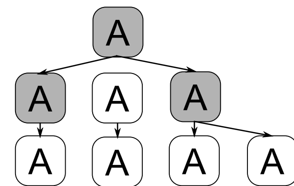
A read barrier is executed whenever the mutator tries to dereference a pointer to an object. The collector determines if the object being dereferenced is white and, if so, recolours it grey before returning it to the mutator. As such, the mutator will never see



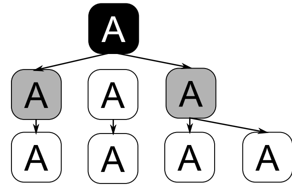
(a) All objects are coloured white before collection starts.



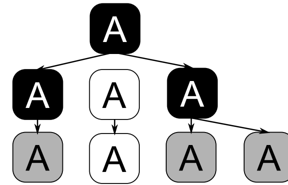
(b) The root objects are coloured grey.



(c) The children of newly greyed objects are coloured grey.

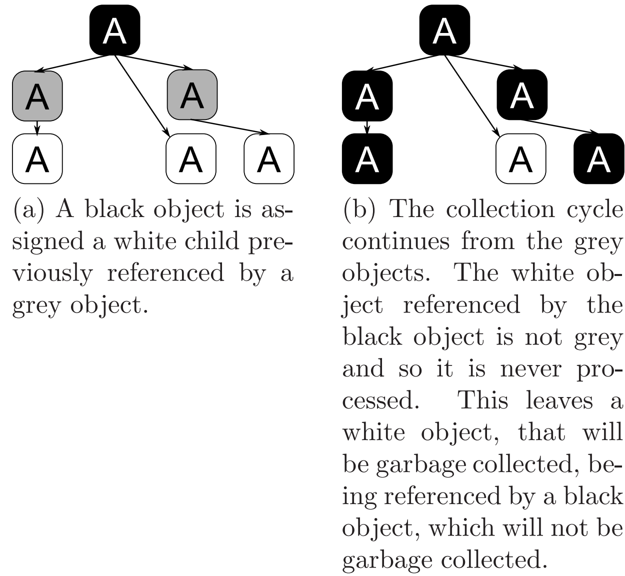


(d) Objects that only have grey children are coloured black.



(e) The process repeats with the children of newly greyed objects being coloured grey.

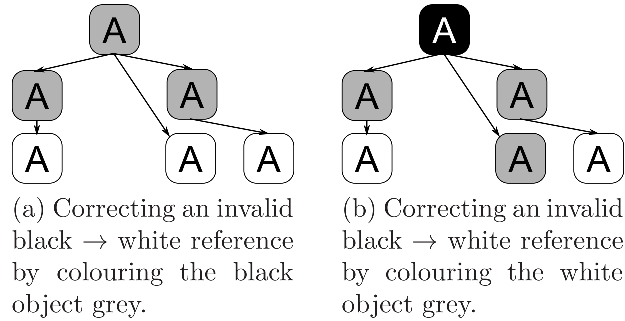
Figure 2.2: Tricolour abstraction applied to normal heap operation i.e. no concurrent mutator operations. Objects begin coloured white and are recursively traced starting from the root objects. Child objects are coloured grey and parent objects coloured black, until the entire heap is made up of black and white objects.



(a) A black object is assigned a white child previously referenced by a grey object.

(b) The collection cycle continues from the grey objects. The white object referenced by the black object is not grey and so it is never processed. This leaves a white object, that will be garbage collected, being referenced by a black object, which will not be garbage collected.

Figure 2.3: Mutator assignment causing a black object to reference a white object. This occurs only when the mutator is executing concurrently with the collector, for example a new object (which will be coloured white) may be assigned to a field in a black object. This newly allocated white object will never be coloured grey and will subsequently be collected, despite it being referenced by a black (uncollected) object.



(a) Correcting an invalid black → white reference by colouring the black object grey.

(b) Correcting an invalid black → white reference by colouring the white object grey.

Figure 2.4: The black → white invalid reference can be fixed by colouring the black object grey or the white object grey.

a white object and the invariant is upheld.

A write barrier presents an alternative to the read barrier that instead hijacks pointer writes into objects. If the object is black it may be immediately coloured grey and therefore rescanned later, or may colour the new referent grey. Either way it means there will never be an edge from a black to a white object; instead it may be grey to black, grey to white or black to grey. Again this ensures that the tri-colour invariant is never violated.

The two types of barrier differ in how often they are called by the mutator. The read-barrier needs to be called for every pointer load, which is a more common operation. The write-barrier is only concerned with pointer writes, which are less frequent, however, it requires more work to maintain the strong tri-colour invariant.

2.1.2.3 Scheduling

Real-time collectors need to ensure that enough garbage is collected for all new allocations to succeed. As real-time collectors are usually running irrespective of the state of the mutator, they need to perform collection asynchronously and, to do so, need to employ some scheduling of collection work.

Work-Based Work-based scheduling causes collection to occur when the mutator allocates a new object. This link between allocation and collection can be quantified such that unreferenced objects are collected from the heap before memory is exhausted. For a tracing collector, if there are L live words in a heap of size H at collector initiation, then the lower bound on the tracing rate, m, is determined by:

$$m = (H - L)/L$$

Unfortunately the number of live words in the heap is unknown; it is only known at the end of the collection, so results from previous collection cycles are used to estimate L. To ensure all garbage is collected before memory is exhausted $k \geq 1/m$ words must be traced at each allocation. Due to the use of live object estimation it is not guaranteed to keep up with mutator allocation and may require a pre-emptive stop-the-world collection to prevent early out of memory errors.

Time-Based Time based scheduling is completely decoupled from the mutator state while running. A time based scheduler is invoked periodically, at intervals that are calculated as a function of mutator utilisation rates. This scheme is far more complex than work-based, because how many allocations will occur during collection is unknown. If there is no memory available for new allocations while the collector is running, the size of the heap is expanded to cope with the new allocation requests. If the heap cannot be expanded then the mutator is paused and a complete collection is performed. Clearly,

if new allocation requests still cannot be managed an out of memory exception must be thrown.

2.1.2.4 State-of-the-Art

Due to the popularity of interactive and real-time applications, real-time collectors have been the focus of garbage collection research. The state-of-the-art included here represent the most efficient and novel approaches to real-time collection:

- *The Sliding-View Recycler* - The Recycler [6] is based on the work in [31, 32], which describes an approach to collecting cyclic structures using reference counting. To collect cycles, the Recycler determines candidate objects, which may be in a cyclic structure, by tracking objects whose reference counts remain non-zero. Using a tri-colour marking algorithm the candidate objects that can be collected are found. Additionally, a snapshot of the heap before collection is used alongside a write-barrier in order to accurately track reference counts during cyclic traversal [29]. The snapshot is taken one thread at-a-time, pausing only one mutator thread, a process known as *sliding-view*. This unique approach has resulted in a collector that achieves extremely low mutator pause times.

- *The Metronome* - The Metronome [4] is a hybrid mark-sweep and copying collector. The heap is segmented into blocks. Collection is normally performed using an incremental mark-sweep collector. When a block becomes highly fragmented an incremental copying collection is performed, to move the objects to other blocks that are mostly full. Completely free blocks are then returned to a free-list. The Metronome is designed specifically for embedded real-time Java applications and achieves high mutator utilisation rates and space-efficiency.

- *STOPLESS, CHICKEN,* and *CLOVER* - CHICKEN and CLOVER [37] are based on STOPLESS [35] and share the same infrastructure. Collector threads run on their own processors, with mutator threads executing on any unused processors. This removes the complexities associate with scheduling collection. Fundamental to the algorithm is the support for lock-free allocation and collection. This means that the mutator always makes progress after a finite number of steps. The collection algorithm performs mark-sweep and copying compaction concurrently on separate threads. All three algorithms use the same mark-sweep collector. STOPLESS uses a compaction algorithm called CoCo, which processes a list of marked objects. For each object to be copied, a wider object is created in to-space that allows for the storage of the object fields, plus a status word. The status word is used to determine which copy of the object has the up-to-date value. When a wide object is completed, a normal-sized to-space copy is created that is then filled with up-to-date values. A read and write barrier are used to ensure that the most up-to-date

value is read and stored in the correct object copy, respectively. CHICKEN does not use an intermediate wide object, it assumes that an object will not be modified during copying. If an object is modified during copying then the copy is aborted and attempted later. An unconditional read-barrier ensures that the correct object copy is accessed. A conditional write barrier is used to modify the correct object copy and, if necessary, abort a copy if it is in progress. CLOVER uses a random value to mark fields as obsolete (instead of a status word) when they have been copied to to-space. A write barrier is used to ensure that the mutator does not write this value, if it does then the mutator is paused (and hence may not be lock-free) until the copying phase completes. All three algorithms are designed to achieve very high mutator utilisation rates and very short pause times through their lock-free approach.

- *Garbage First* - The garbage first algorithm [23] mixes concurrent, parallel and stop-the-world phases, with the intention of reducing stop-the-world pause times to below a user-defined threshold. The heap is partitioned into regions that can be collected independently. Each region requires a remembered set that is implemented using the card table mechanism [21]. The heap is divided into 512 byte *cards* that map to a card table entry. The remembered sets are hash tables of these cards. Every thread maintains a log of all modified cards and a write-barrier implementation adds modified references to the log. A dedicated thread scans the logs and determines which references may have modified each card. If a reference from outside the heap region owning the card is found, then it is added to the remembered set for that region. Allocations are thread-local and the collector maintains a cost associated with each region. Using the estimated region cost, the collector decides which regions should be collected in the next time-slice. The first phase of collection is the concurrent mark phase. The mutator is paused while the objects referenced by the root set for each region are marked. After marking the root set, the remaining marking occurs concurrently with the mutator. The collector maintains an estimate of the amount of live data in each region and uses it in conjunction with expected pause times to decide which regions to collect next. Priority is given to the regions with the least number of live objects and most amount of garbage (*garbage first*). The final phase of collection is evacuation and occurs in parallel (with the mutator paused). Live objects are moved and compacted into another region in one atomic phase using a load-balancing technique to minimize pause time.

- *Non-Stop Haskell* - Non-Stop Haskell [18, 16] is an incremental *soft* real-time uniprocessor garbage collector implemented in the Haskell Spineless Tagless G-Machine. It implements an incremental time-based collector. For Haskell's lazy evaluation model, all object are accessed via an "info table" pointer. This means that all object accesses, including fields, are via dynamic despatch. The key to achieving soft real-

time performance is in the implementation of the, usually expensive, read-barrier. The idea is that the info table pointer of an object that must be scavenged is hijacked, so that it first self-scavenges. The self-scavenging is enforced by changing the info table pointer when an object has been evacuated (but not yet scavenged), so that it points to self-scavenging code (returning the original pointer after scavenging). To support fast self-scavenging, specialised variants of every method are created that inline self-scavenging code and execute the original method body. As shown in figure 2.5, the specialised variants are stored in duplicate info tables, with each info table containing a reference to the other, in order to perform switching of the info table pointer. This results in no read-barrier overheads when objects are referenced when the garbage collector is off, or the object has already been scavenged.

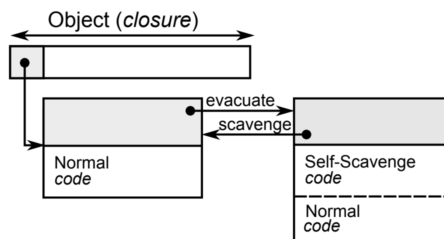


Figure 2.5: Object Specialisation in Non-Stop Haskell.

To further extend its performance, the same approach is taken to incrementally collect stacks (by hijacking stack activation records). By modifying the soft-scheduler in the Glasgow Haskell Compiler (GHC), every time a thread returns to the scheduler, the time spent in the mutator is checked and the collector is scheduled if the mutator quantum has expired. This decouples the scheduling of the collector from mutator work, allowing it to achieve soft real-time deadlines. The results obtained in [18] show very low overheads compared to the, highly optimised, stop-and-copy collector that is default in GHC. The key to Non-Stop Haskell's performance is the use of its "free" read barrier, which can be applied to other collection schemes, such as generational collectors. Furthermore, this approach can be extended to other object oriented languages, such as Java, as discussed in [18, 16] and is the basis for this project.

2.1.3 Baker's Incremental Garbage Collection

In 1978 Henry G. Baker published a paper describing a novel approach to garbage collection in the programming language LISP [7]. At that time the main garbage collection approaches considered were stop-the-world, mark-sweep and mark-compact algorithms.

These resulted in pauses in execution when processing a collection cycle, which prevented the wide-spread adoption of list processing languages in real-time and interactive applications. Baker went so far as to say that his approach could allow for operating systems and database management software to utilise these higher-level languages without a performance hit.

Baker's approach involves splitting the heap into two semispaces, known as *to-space* and *from-space*. New objects are allocated into to-space. When garbage collection is invoked, to-space and from-space are flipped. During garbage collection all accessible objects are traced and moved to the new to-space. A forwarding address is inserted in place of the object. Whenever an edge is traced that points to a forwarding address it is updated to the new location in to-space. At the end of the cycle all accessible objects exist in to-space, whilst from-space contains only garbage (see Figure 2.6). The spaces are then flipped.

This simple method means that only a single pass is performed over the heap. Furthermore, the free-list is avoided, as allocation in to-space can be implemented using a bump pointer.

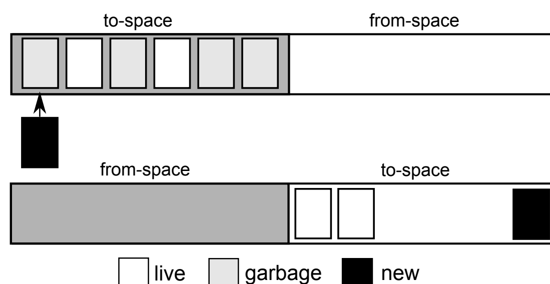


Figure 2.6: A semispace collector has two spaces, during a collection cycle new objects are assigned to the top of the to-space while objects copied from from-space are assigned to the bottom. Objects that are not copied into to-space are garbage and will be overwritten when the spaces flip again.

In this form the collector is still stop-the-world. In order to make it incremental the cycle is split into multiple iterations so that, for each allocation of a new object, k iterations are performed, after which the mutator continues. Initially, the root objects must be evacuated, i.e. those references that exist in program registers (the program stack and globally accessible values). An object is evacuated by copying it into to-space, inserting a forwarding pointer to the copied object and placing it on the scavenge queue, see Algorithm 2. Each iteration then scavenges k objects. The process of scavenging an object involves removing it from a scavenge queue and evacuating all of the objects it references, as illustrated in Algorithm 1. The value of k is important as it must ensure that all of the accessible objects are moved into to-space before allocation uses up the remaining memory (see Section 2.1.2.3).

Algorithm 1 Pseudo-Code for depth-first scavenging of an object. Every field within the object reference is evacuated.

```
scavenge(reference):
for field in reference do
   if field is reference type then
      field := evacuate(field)
   end if
end for
```

Algorithm 2 Pseudo-Code for evacuating an object. First the reference is checked to see if it has already been evacuated. If it has not then it is copied into to-space and the new reference is used to install a forwarding pointer where the old reference is. The new reference is returned to the mutator.

```
evacuate(reference):
if reference is in toSpace then
   return reference
end if
newReference = copy(toSpace, reference)
installForwardingPointer(reference, newReference)
return newReference

copy(space, reference):
{Copies the object pointed to by reference into virtual memory space space}

installForwardingPointer(reference, newReference):
{Copies the address of newReference to the old reference.}
```

Now there is an issue, in that from-space will still contain reachable objects. The mutator must not see this partial heap state and the strong tri-colour invariant (see Section 2.1.2.2) must be maintained.

To uphold the tri-colour invariant a read barrier is used: whenever an attempt is made to load a field that points to an object, the object is tested to see if it has been scavenged. If not, it is scavenged.

The read barrier represents a significant performance bottleneck in many applications. As discussed in [43], the CPU overhead incurred by an incremental read barrier can be as high as 60%.

2.2 Jikes RVM

The Jikes Research Virtual Machine (RVM) is primarily designed as a platform for academic research in Java, virtual machines and memory management. This project concentrates on the Jikes RVM, because it is open-source, under active development and allows us to compare results against other implementations in Jikes. The configuration options available in the Jikes RVM, and the use of the memory management toolkit (MMTk) to separate memory management concerns, makes it an ideal platform to create new garbage collection strategies and test them.

The RVM is itself written in Java. It is known as a *meta-circular machine*, in that it evaluates the language in which it is written. In order for this to be achieved the RVM is *bootstrapped* at run-time, setting up the Java runtime environment for use by user applications. The bootstrapping process creates a memory snapshot of the RVM object state using another JVM and reflection and stores it as a boot image. The boot image is a binary file that contains (1) platform specific boot code and data, (2) the RVM image, its initial code and data, and (3) the RVM heap. A small C application is tasked with loading the boot image into memory when the RVM is invoked; it also installs low-level functions used by the RVM to interact with the underlying operating system.

Java does not allow access to low-level machine functions such as hardware traps and memory addressing. For this reason it is much harder to implement a virtual machine entirely in Java. To overcome this problem the RVM uses a special package known as "vmmagic". This system uses stub code to enable compilation, which is then replaced with machine code that interacts directly with the operating system and hardware. This allows the RVM to access low-level functions while maintaining a Java-in-Java environment. Vmmagic is used extensively within the memory management toolkit to gain access to raw memory operations.

At runtime the Jikes RVM actively monitors and optimises code based on performance measures, such as how often methods are invoked, using an adaptive optimisation system (AOS) [1]. This results in methods being compiled and recompiled, perhaps several times during execution, in order to benefit from more complex and expensive optimisations.

This happens in parallel with the application execution. Newly compiled methods replace existing methods when optimisations have been completed. These optimisations are performed by the optimising compiler on request. Initially all methods are compiled by the simpler baseline compiler [1].

2.2.1 MMTk

The Jikes RVM separates its memory management from the rest of the virtual machine. The Memory Management Toolkit (MMTk) provides the tools required to develop a garbage collector independently from the virtual machine, complete with scripted testing framework and debugger.

The MMTk already implements many of the required functions for a garbage collector, including tracing operations over the heap and root discovery. Furthermore, it includes many commonly used memory allocation policies such as mark-sweep, mark-compact and copy *spaces*.

2.2.1.1 Spaces

In order to take advantage of multi-processor environments the MMTk is split into *global* and *local* data structures. In the simplest example, an entire memory region may be managed by a `Space` object (the global data structure) and each processor will have a `SpaceLocal` object. The `SpaceLocal` object will request chunks of memory from the global `Space` object and object allocation will be performed on the thread-local chunk (with further chunks being requested as required). This method benefits from fast thread-local access and promotes a discontiguous memory model.

Each space is defined with specific allocation policies. For example, the `CopyLocal` class, the `SpaceLocal` synonym for copying regions, extends `BumpPointer`, which performs allocation using a bump-pointer. Along with a thread-local class there is a global class that manages the entire virtual memory space. These classes perform the specific tracing operations required by the storage policy. For example a `CopySpace` requires forwarding pointers that need to be managed during tracing.

2.2.1.2 Allocation Policies

The MMTk defines many of the commonly used allocation policies:

- BumpPointer - allocates memory to objects by incrementing a pointer into memory.
- BlockAllocator - allocates blocks of memory in powers of two. This is used to allocate chunks to thread-local objects, which use another allocation policy to allocate memory to objects.
- SegregatedFreeList - allocates memory based on a free-list of memory addresses.

These allocation policies are used by the different types of virtual memory space to manage collection. For instance, a mark-sweep collector will use a free list allocator, while a mark-compact will use a bump pointer.

2.2.1.3 Plans

The MMTk is divided into layers. At the top-level is a *plan*. A plan consists of several classes that provide memory layout, allocation, garbage collection and statistical gathering functions. The plan combines schemes for collection and memory management to create a consistent heap allocation/deallocation system.

For each plan the virtual memory is split into a collection of spaces, each of which may have a separate allocation and collection policy. Depending on the memory management model, these spaces will be coordinated to perform specific tasks. For instance an immortal space can be used to allocate objects that will be in use throughout the lifetime of the runtime environment (i.e. never collected). A copy space and a mark-sweep space may be used in combination to create a generational collector. The toolkit has much flexibility for how memory is managed and this allows for any stop-the-world collector to be implemented with ease.

The most basic plan has a VM space, an immortal space, a raw memory space for meta-data and a large object space. Further spaces are defined in different plans to implement different collectors.

A garbage collector is implemented in the MMTk via several classes:

- `Plan` - implements the global functionality for all memory management operations, such as setting the order and phases of garbage collection.

- `Mutator` - implements per-thread mutator operations, such as memory allocation and read/write barriers.

- `Collector` - implements per-thread collection operations, for example, thread-local tracing of live objects.

- `TraceLocal` - implements the transitive closure over the heap. This class follows object references to find all live objects.

- `Constraints` - indicates (1) whether the plan requires barriers, (2) whether the plan is concurrent and (3) whether the collector requires object header bits/words.

Garbage collection occurs in different *phases* that identify the tasks that should be carried out during the collection cycle. For instance, there is a closure phase that calculates the transitive closure. Phases are combined into a complex phase that determines ordering; after one phase completes the next is started. This repeats until there are no phases left, at which point garbage collection is complete. Phases can be either a (1)

`SimplePhase` - executes with the mutator paused, (2) `ConcurrentPhase` - executes concurrently with mutator execution or (3) `ComplexPhase` - executes a series of contained simple and concurrent phases. Each phase is named and given a unique identifier. The identifier is used during collection to determine which phase is currently in progress and therefore what work should be executed.

The phases are managed using a static stack within the `Phase` class and corresponding static methods to manage the processing of the stack. During processing the phase on the top of the stack is executed and then popped from the stack, at which point the next phase at the top of the stack is executed. This repeats until no phases remain on the stack. A `ComplexPhase` is expanded when processed so that all immediate child phases are added to the stack to be processed.

A phase is associated with an integer that stores the phase type in the most significant two bytes and the unique phase identifier in the least significant two bytes. The phase stack is limited to sixty four phases. There are five existing phase types that schedule global, collector, mutator, complex and place-holder phases. The addition of a new phase type requires that the identifier is unused and has a `short` value.

The identifier associated with each scheduled phase allows an object to distinguish between the purpose of each phase. For instance there are root scanning, stack scanning and closure phases associated with collectors and these are distinguishable to the collector using the unique identifiers.

2.2.1.4 Collection Policies

The collection policies are managed at a high level within a plan. The way in which objects are collected is determined by the segregation of the heap into spaces. Each space may have a different policy for collecting objects. Within each virtual memory space, specific tracing actions are performed, such as installing a forwarding pointer or marking an object. While a plan will decide when to trace objects, the tracing is governed using a `TransitiveClosure` object passed to the `Space`. The `Space` object prepares for collection either through switching to a new space (for copying collectors) to service new allocation requests, or clearing meta-data from a previous collection. The `Space` object will also perform post-collection clean-up, such as resetting counters and clearing mark bits. The `TransitiveClosure` determines the *edges* (references) to other objects.

2.2.1.5 Concurrency

In order to develop concurrent collectors the MMTk offers support for both read and write barriers that can be implemented in the `Mutator` objects representing the user application. Concurrent collectors then use `ConcurrentPhase` objects that are handled differently to normal phases.

Concurrent phases require all threads performing GC to synchronize. Any concurrent

phase must be implemented so that only one thread will cause the next phase to begin. Unlike normal phases, concurrent phases do not immediately cause the next phase to start. Rather, they continue until all work has been completed (while the mutator is still running) and then notify the plan. The next phase will then begin at the next convenient *yield point*, when the mutator is not performing critical work.

By using phases, a collector can combine concurrent and stop-the-world operations in order to, for instance, perform a concurrent mark followed by a stop-the-world sweep.

2.2.1.6 Testing and Visualisation

The MMTk includes a set of tools that enables the testing of garbage collectors within a virtualised heap. This supports the debugging of collectors and enables collector implementers to test their collector without recompiling the RVM. Included with the test harness is a scripting engine that allows for specific scenarios to be tested, for instance, cyclic structure collection. Unfortunately the test harness is not entirely stable and cannot be used to ensure that a collector will work correctly within the RVM. It remains a useful tool for debugging common mistakes, as an indication that a collector implementation is stable and for reasoning about collector execution.

GCSpy is another tool that monitors a garbage collector during application execution, within the RVM, and can display a visual representation of the heap. The collector implementer needs to insert calls to GCSpy methods in order for it to correctly display the heap state, which has resulted in few collectors supporting it. However, it can be used to determine whether all objects are being correctly collected and offers an alternative method of reasoning about a collector's operation. For this reason it can be valuable to introduce GCSpy support.

2.2.1.7 Semi-Space Collector

It is much easier to understand the MMTk by looking at existing implementations. As we are investigating an incremental semi-space collector we shall look at the stop-the-world semi-space collector and the concurrent mark-sweep collector.

All plans implemented in the MMTk extend the `Plan` class. This class (and its supporting classes) includes the implementation for the common functionality found in all MMTk collectors. The simplest memory management system contains a handful of memory spaces:

- *VM Space* - used only for VM-specific objects, like the boot image.
- *Large Object Space* - used for storing large objects, because such objects require infrequent, but large, allocation requests they need to be efficiently collected. Therefore they are stored separately so that the general allocation algorithm can operate independently.

- *Immortal Space* - used for storing objects that will remain live throughout the operation of the VM. These objects therefore will never be collected.

- *Meta Data Space* - used by the MMTk to store meta data, this includes information on allocated and unallocated space in the rest of the heap.

- *Non-Moving Space* - used for storing objects that should be pinned in memory; they will remain at the same address throughout their lifetime.

- *Small/Large Code Space* - used for storing class code.

These spaces are all dynamically sized. The remaining space in memory is left for all *small* objects, less than 10^{13} bytes. This can have any number of policies governing its allocation and reclamation. It is this small object space that is managed by subclasses of `Plan`.

The semi-space plan (`SS`) implemented in the MMTk is one of the simplest memory management plans. On top of the spaces defined within `Plan`, `SS` defines two more copy spaces. As per the semi-space paradigm, these represent from- and to-space. To manage the flipping of spaces and to access the correct space, supporting methods are defined along with a boolean field to determine which of the two spaces is to-space. Figure 2.7 illustrates the semi-space memory management object structure. The `ActivePlan` class is global and remains constant, providing access to plan objects that have been instantiated based on compile-time selection. For the semi-space collector this includes an `SSMutator` and `SSCollector` object per thread and a global `SS` object. The `SSMutator` and `SSCollector` objects each have a `CopyLocal` object that manages thread-local allocation blocks from a global `CopySpace`. The flipping of the spaces results in the *rebinding* of each `CopyLocal` object to the `CopySpace` representing to-space. The `SS` object manages the flipping of the spaces. The order of phase-execution is controlled via static methods of the `Phase` class. Each phase is represented by a `Phase` instance, with one top-level `ComplexPhase` instance that determines global collector execution.

The `SS` class represents global allocation and reclamation. For specific phases of collection the global responsibilities are handled by this class:

- *prepare* - flips the two global `CopySpace` instances and prepares the global `Trace` instance for this collection phase by clearing the completion flag and resetting the scavenge queue.

- *closure* - prepares the global `Trace` instance for this collection phase, as with the prepare phase. There may be several closure phases, but only a single prepare phase.

- *release* - releases from-space of allocated data by resetting the bump pointer and freeing all unused pages.

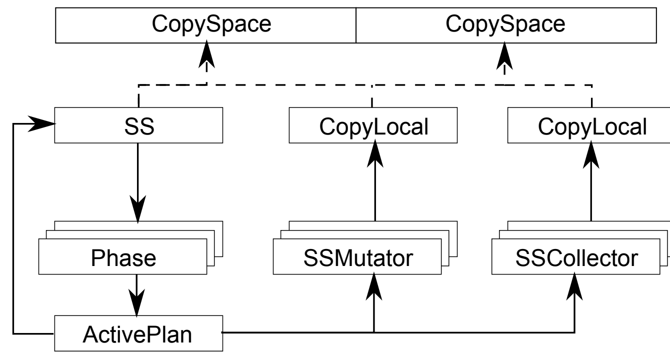


Figure 2.7: The semi-space memory management object structure. The structure consists of two `CopySpace`'s, a global `SS` instance that instantiates several `Phase` instances, an `SSMutator` and `SSCollector` instance for each thread and an `ActivePlan` class that ensures the correct semi-space objects are instantiated together at run-time.

The other major task performed by the `SS` class is the specification and ordering of collection phases. Figure 2.8 demonstrates how a collection is executed in phases. When a collection is started through either a user call to `System.gc()`, or a failed allocation request, a new *phase stack* is started. The phase stack contains all phases of execution (as defined by the plan) and executes each phase in turn. The type of the current phase, which can be one of `SCHEDULE_GLOBAL`, `SCHEDULE_MUTATOR`, `SCHEDULE_COLLECTOR` or `SCHEDULE_CONCURRENT`, determines the object that will process the phase; either `SS`, `SSMutator` or `SSCollector` respectively. When the phase stack has been exhausted the collection is complete.

In addition to handling collection, the `SS` class also implements some accounting methods that are used to determine the current status of memory allocation, such as the number of pages available. Furthermore it registers a specialised scan method to enforce the use of the `SSTraceLocal` class when scanning the heap.

The global operations performed within the `SS` class do not collect any garbage; they merely prepare the groundwork for the per-thread collector objects to work on their local heap chunks. The `SSCollector` class is used to create thread-local collectors. These thread-local collectors contain a local `TraceLocal` class, which performs all of the heap tracing operations, as well as a `CopyLocal` bump pointer that maintains allocation on thread-local memory chunks.

As a semi-space collector is a copying collector, it implements special methods that allocate memory for copies of objects i.e. when collection requires an object to be copied from one space to another. It is handled differently from normal allocation, because it is an internal function of a copying collector.

The thread-local responsibilities of different collection phases are handled by the `SSCollector` class:

- *prepare* - binds the thread-local bump pointer to the new to-space (for future allocations)

- *closure* - performs a complete trace over the heap, processing the entire live object graph and scavenge queue (the actual collection)

- *release* - release the `TraceLocal` object, resetting the root-set and scavenge queue

These plan classes are simply a means to synchronize allocation and collection. The core of the collection work is performed during tracing with `TraceLocal` instances that scan the heap and `Space` instances that implement forwarding or marking functionality.

The thread-local dual to `SSCollector` is the `SSMutator` class. This class handles object allocation, forwarding allocation requests onto the thread-local bump pointer whenever an allocation is requested to the managed space. At the release phase the `SSMutator` object rebinds the bump-pointer to to-space, which ensures that all future allocation requests will be made to the correct `CopySpace` instance.

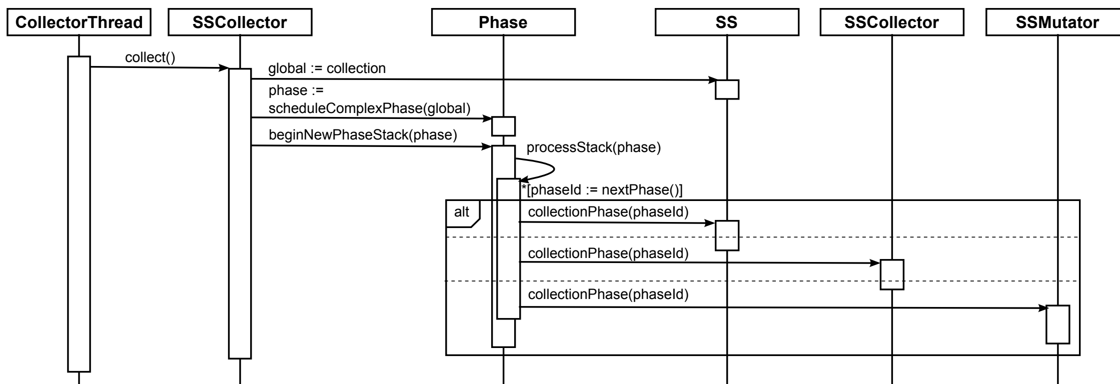


Figure 2.8: A sequence diagram showing how the semi-space collector executes different phases of collection. The different phases are scheduled by superclasses of `SS` and are processed, in scheduled order, when a collection is requested. The possible phases relate to either global, thread-local collector or thread-local mutator operations handled by the `SS`, `SSCollector` and `SSMutator` classes respectively.

The final part of the semi-space collector is the `SSTraceLocal` class. This extends from the `TraceLocal` class that performs all of the tracing functions. To manage two new spaces the `SSTraceLocal` class overrides several methods:

- `isLive(ObjectReference ref)` - determines if the object reference is for a live object, based on whether it resides in the current to-space.

- `traceObject(ObjectReference ref)` - traces an object by forwarding the trace request onto the semi-space in which the reference resides.

- `precopyObject(ObjectReference ref)` - to prevent the collector from collecting itself all objects critical to MMTk operation are pre-copied. This method performs the pre-copying in the same way as normal object tracing, but may be modified by subclasses.

- `willNotMoveInCurrentCollection(ObjectReference ref)` - returns true iff the reference does not exist in the current from-space, i.e. it does not exist in memory that is currently being collected. This is primarily used to ensure that thread stacks will not move.

2.2.1.8 Concurrent Mark-Sweep Collector

The MMTk is especially suited to developing stop-the-world collectors, as is evident in the semi-space collector example. Very little work is needed to implement this system and this is a similar story with many simple collection strategies. A concurrent collector, on the other hand, needs barrier implementations and support for scheduling collector threads.

A concurrent mark-sweep plan (`CMS`) is also implemented in the MMTk. Like the semi-space implementation, very little additional work is performed in the `CMS`, `CMSCollector`, `CMSMutator` and `CMSTraceLocal` classes defining the plan. Instead, most of the concurrent infrastructure is embedded into the core plan classes, along with the additional abstract `ConcurrentPlan`, `ConcurrentCollector` and `ConcurrentMutator` classes, to manage the concurrent work.

The differences in stop-the-world and concurrent collectors in the MMTk lies in the implementation of the phase structure. Using the phase infrastructure the thread-local collection is performed in increments, until the mark phase is completed. The number of references to trace in each increment is chosen so that, for large amounts of work, the collector thread yields to the mutator thread, so that it does not starve the mutator. The sweep is not performed concurrently.

At the thread-level a concurrent collector has one collector thread per processor and many mutator threads (one adaptive optimising thread, one timing thread, one main thread and one thread per `java/lang/Thread` instantiation). These threads communicate through handshaking, where the mutator makes a request to the collector to perform some garbage collection. The main `run` method in both the mutator and collector executes indefinitely, suspending when there is no work to be done. In this way the collector thread does not require processor time until a collection is in progress.

2.3 Method Specialisation

Method specialisation is an approach to storing object state through variations of methods, instead of variable testing and branching instructions. The usual method for deter-

mining which state an object is in involves a test followed by a conditional branch, which is cumbersome. Listing 2.1 shows a class, `A`, which conditionally loads data into a variable and then, unconditionally, prints what it has loaded. Using a variable it remembers whether it has already loaded the data, hence it has two states: data loaded and data unloaded. For complex state representations the *state pattern* may be used, but if the states share data this becomes complex. Furthermore, implementation details exist in different classes, which can be difficult to manage. Listing 2.2 demonstrates the state pattern using a class, `B`, an interface, `State`, and another class, `DefaultState`, that implements `State`. Whenever a particular method is called in class `B`, the execution will be deferred to a method of the `State` instance, assigned to its `state` member variable. Different implementations of the `State` interface can be used to achieve different behaviours and thus represent object states. The state-specific tasks are therefore executed purely using dynamic despatch and state transitions must be explicitly handled by the programmer. The programmer must also ensure that the dynamic type of the `state` member variable is always up-to-date.

These issues are addressed by method specialisation by employing transparent variants of methods that are executed based on the state of an object. For dynamically despatched languages, such as Java, object methods are stored in a virtual method lookup table. When a method is invoked the lookup table is used to determine where in memory the code for the method can be found. Every object has a pointer to its type's virtual method lookup table, method specialisation allows this pointer to be modified during execution. Object state can be achieved by switching the pointer between several method tables. Static methods can be managed in a similar way, by modifying the pointers to static methods directly (rather than modifying the pointer to an entirely different static method table). Listing 2.3 shows conceptually how method specialisation can be used to switch between object state without the need for state tests. Two different variants of method `foo` are created. The default method `foo()` performs some data loading and then switches to specialised variant one, using a call to `specialise(1)`. The specialise method changes the virtual method table pointer to the specified variant. Next time `foo` is called it will execute the specialised method `foo_1()`, which prints the loaded data. This is synonymous to the state pattern, in that it involves switching a pointer to represent object state. The difference between method specialisation and the state pattern is that it needs to be supported by the underlying VM and it does not require using separate state classes or interfaces. Furthermore, method specialisation does not affect the programmer's view of the type system i.e. multiple types do not need to be defined to handle different object states. Method specialisations can be applied retrospectively to applications, this allows stateful behaviour to be introduced generically across every application without modifying the Java source code.

Method specialisation was added to the Jikes RVM through the modification of object type information blocks (*TIB*s). Every class within the Jikes RVM has an associated TIB.

```
class A{
  enum DataState{
    LOADED, UNLOADED
  }

  private DataState state = DataState.UNLOADED;
  private String data;

  public void foo(){
    if(state == DataState.UNLOADED){
      data = loadData();
      state = DataState.LOADED;
    }
    print(data);
  }
}
```

Listing 2.1: Code using explicit state tests to determine which code to execute. Explicit variable state checks can be cumbersome when there are many possible states.

```
class B{
  private State state = new DefaultState();

  public void foo(){
    state.bar();
  }

  public void changeState(State state){
    this.state = state;
  }
}

interface State{
  void bar();
}

class DefaultState implements State{
  public void bar(){
    print(``DefaultState'');
  }
}
```

Listing 2.2: Code using the state pattern to change object behaviour. The state pattern fragments the code and is difficult to manage when data needs to be shared between states

```
class C{
  public String data;
  //The default method variant
  public void foo(){
    //Load data from disk
    data = loadData();
    //Switch to specialisation one
    specialise(1);
  }
  //Specialisation one of method foo
  public void foo_1(){
    print(data);
  }
}
```

Listing 2.3: A conceptual example of how method specialisation works. The default method will load the data and switch to a different variant (specialisation) that prints the loaded data.

This is used to store type information about the class, for example a pointer to an `RVMType` instance representing the class. More importantly it stores pointers to all virtual methods as an embedded virtual method table *VMT* (see Figure 2.9). To implement method specialisation a class has several TIBs, one for each specialisation. Each TIB contains pointers to the virtual method variants that are associated with the TIB's specialisation. As shown in Figure 2.9 every object has a pointer to their respective TIB, based on their class. In order to switch an object's specialisation the TIB pointer in the object's header is modified to point to the TIB associated with the specialisation. When a method is invoked on an object the TIB is used to find the method code. For a specialised TIB this will result in the specialised method variant being executed.

It is also possible to apply generic method specialisations automatically to all classes, supporting the implementation of advanced system-wide functionality, for example:

- Orthogonally Persistent Java - This allows for objects to be automatically persisted across VM executions. By using method specialisation an object can be loaded from available resources when it is accessed and saved when it is modified. This means that there is no explicit test to see whether the data has already been loaded every time it is accessed.

- Distributed Virtual Machines - Distributed virtual machines create a distributed application execution environment across a network. Using method specialisation the VM can silently execute system functions located on different machines through specialised method variants. To enable the distribution of objects the VM needs to explicitly determine on which machine the object is located, in particular, whether

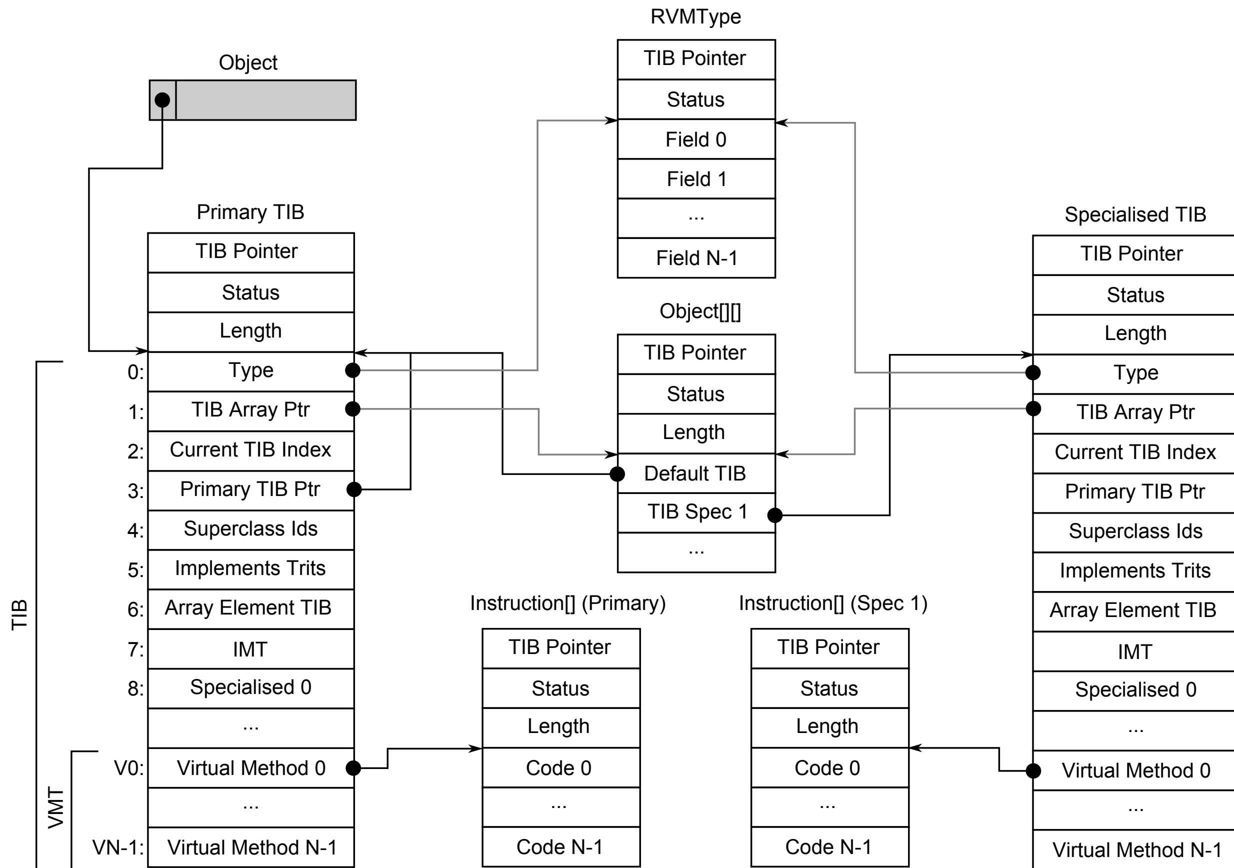


Figure 2.9: The TIB structure used for method specialisation. Every object contains a pointer to their type's TIB in their header. When a method is invoked on an object the TIB pointer is used to find the code for the method by looking at the TIB's virtual method table. A specialised method variant can be executed by changing an object's TIB pointer to point to a class' specialised TIB.

the object is local and can be accessed directly. Method specialisation can be used to replace the explicit state checks, potentially improving performance.

- Incremental Garbage Collection - The read-barrier of an incremental garbage collector has an explicit test to determine whether an object needs to be scavenged (see Section 2.1.3). Using method specialisation an object can be specialised so that it exists in two states: scavenged and unscavenged. In the unscavenged state all of an object's methods first self-scavenge, execute the method body and then switch to the scavenged state. The scavenged state only executes the method body. This behaviour means that there is no need for a read-barrier test, as the switch in specialisations implicitly encodes its state, only self-scavenging when required.

2.3.1 Class Transformations

The method specialisation framework in the Jikes RVM [18, 17, 22] adds method specialisations using a series of user-defined class *transformations*. Class transformations modify Java class bytecode, using a bytecode engineering library called ASM [15], to change the behaviour of a class at load-time. This means that a class is transformed before it is used in the Java run-time. This allows transforms to be applied to any user application classes without modifying the existing code. For example, you can make any application persistent using a single persistence transform that specialises methods performing field access.

2.3.2 Beanification

In order for method specialisation to be of value to a broad range of applications, it is necessary to allow object interaction only via virtual methods. For example, to implement orthogonally persistent Java, the processes of reading and writing a field need to be identifiable so that they can be modified to load or save the field respectively.

While it is good programming style to ensure that field access is encapsulated, often local fields are directly accessed and sometimes fields are made public. Method specialisation requires that *all* field accesses are via methods (except when already within a getter/setter method), a process known as beanification. To enforce this, the method specialisation framework automatically replaces any field accesses with a generated getter/setter method call using a class transform known as a beanifier.

The current beanifier implementation cannot guarantee unique getter and setter method names for classes and this causes problems with internal classes (such as `java.lang` classes) and certain user applications. This is because the names of fields in the entire class hierarchy must be known in order to generate unique method names. Loading class information directly from the VM invokes the beanifier. This means that when a class hierarchy with circular references is explored, an infinite loop is inadvertently tripped.

A naive implementation may try to avoid exploring the class hierarchy and use the dynamic type of the receiver to determine the method name. The problem with this is that the dynamic type may not be the class that declares the field, in fact it could be any subclass of the declaring class. This would lead to the incorrect method name being generated and either result in the wrong field being accessed or a `MethodNotFound` exception. Therefore the exploration of the class hierarchy to find the declaring class is a necessity.

Figure 2.10 shows two classes, in class `A` there is a method, `foo()`, that accesses the field `y` in an instance of class `B`. Class `B` also has a method, `bar()`, that accesses the field `x` in an instance of class `A`. This means that when class `A` is loaded and method `foo()` is inspected, an access to field `y` in class `B` is found. This reference needs to be replaced by an accessor method. To generate the correct method accessor name the loading of class `B` is required, so all fields within the class hierarchy can be explored. The loading of class `B` invokes the beanifier. This time method `bar()` is inspected, in which an access to field `x` in class `A` is found. Again, the field access needs to be replaced with an accessor method and thus class `A` must be loaded to explore its class hierarchy. At this point the Jikes RVM is still in the process of loading class `A`, but it does not know this and so starts loading the class again, resulting in an infinite loop. This problem is illustrated in Figure 2.11.

```
1 class A{
2   public int x;
3   public void foo(){
4     B b = new B();
5     b.y = 14;
6   }
7 }
```

```
1 class B{
2   public int y;
3   public void bar(){
4     A a = new A();
5     a.x = 4;
6   }
7 }
```

Figure 2.10: An example of circular class references that lead to an infinite class loading loop when attempting beanification.

Once the class hierarchy can be explored the correct getter/setter method names can be generated by appending the class name of the declaring class, this is known as *name mangling*. This will guarantee that the correct field is being accessed whenever a virtual method is replacing a direct field access.

Without name mangling it is not possible to beanify internal classes (such as `java.lang` classes), which is a requirement for the garbage collector with the “free” read barrier to be able to collect all objects.

To allow the class hierarchy to be explored we need a way to load class information without loading it into the VM, therefore avoiding the beanifier. We call this loading of only class information, *partial loading*, and its use will result in a two-pass class loader.

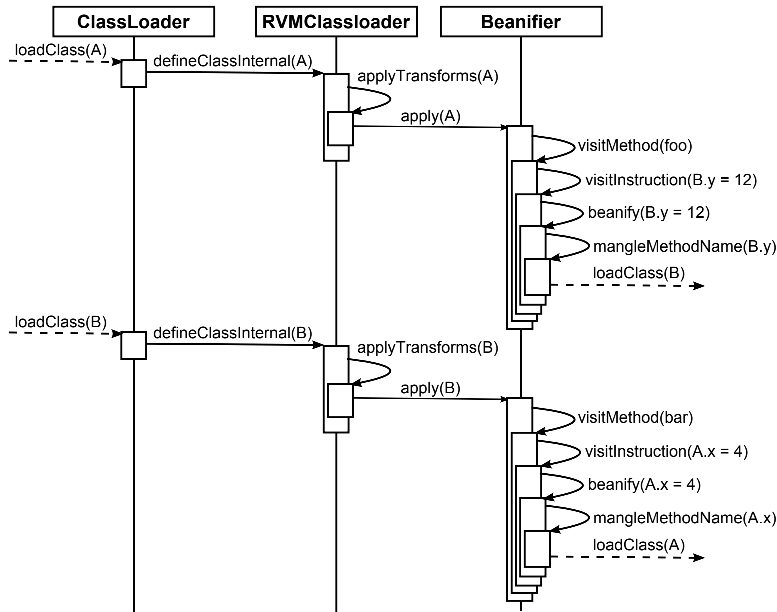


Figure 2.11: A sequence diagram showing how an infinite class loading loop can arise from cross-referencing classes during beanification. First class `A` is loaded, which has a field access instruction to a field in class `B`. To apply beanification, the unique accessor method name to use to replace the direct field access requires the exploration of class `B`'s class hierarchy. This causes class `B` to be loaded. During loading a field access instruction to a field in class `A` is found. Again, to apply beanification the exploration of class `A`'s class hierarchy is required. This results in class `A` to be loaded again and thus an infinite loop arises.

As illustrated in Algorithm 3, when a class is being loaded for use in the VM we search through all of its methods for instructions that load or store fields. These instructions are replaced by a virtual method call to an accessor method. To ensure that the right accessor method is called, we search through the class hierarchy of the object whose field is being accessed. We loop through the class hierarchy using our partial class loader, until we find the field that we are accessing. We then use the name of the class that declares the field to generate the unique method name. For every class that is loaded we also loop through its own fields and create the accessor methods using its class name. The name of the generated methods will therefore match the name found through searching the class hierarchy.

Two-pass class loading will load all of the field information for a class in the first pass. This will require extra methods in the class loaders to ensure that during the first pass an `RVMType` object is not created, which is used internally to represent a completely loaded class (avoiding issues related to loading a class multiple times). Unfortunately the `Classloader` itself is not accessible, as it resides within the Classpath implementation, and so we cannot prevent the class being loaded from disk twice[4] (once for partial extraction and again for full loading). The second pass will fully load a class, during which, field information will be requested for any classes that it interacts with. If these classes are not fully loaded then the information will be partially loaded, otherwise we can find the field information using reflection.

When a class is fully loaded its entire hierarchy is then loaded. As such, when we are searching for field information we have two class exploration stages. The first stage explores only partially loaded classes, but once we hit a fully loaded class we can explore the remaining hierarchy using reflection. With concurrent class loaders we may run into an issue where the entire hierarchy is not yet completely loaded and may still result in a class being resolved twice.

The partial loading will result in extra data being stored in memory, but it only needs to be retained until a class is fully loaded, at which point it can be removed. The extra costs due to this double loading of classes will be paid only initially; converged execution times, representing long-running application performance, should not be affected.

[4]In Linux the *buffer cache* will prevent the second load from disk if the second pass is sufficiently temporally close to the first. This may not be the case if the class being loaded requires the loading of several other classes, which may then overwrite the buffer cache.

Algorithm 3 Pseudo-Code for executing a two-pass instance field beanifier. The beanify method illustrated here is executed when a class is being fully loaded. When a field access instruction is found we search the class hierarchy of the receiver using partial class loading in order to find the declaring class name. This lets us generate a unique getter/setter method name by concatenating the declaring class name. The getter/setter methods themselves are generated in the final stages of full class loading by iterating over the class' fields.

```
beanify(cls):
for mthd in cls.methods do
  for instr in mthd.instructions do
    if instr.opcode == GETFIELD || instr.opcode == PUTFIELD then
      instr ← replaceInstruction(instr)
    end if
  end for
end for
for fld in cls.fields do
  if !isStatic(fld) then
    addGetMethod(cls, fld)
    addSetMethod(cls, fld)
  end if
end for

replaceInstruction(instr):
declarer ← getFieldDeclarer(instr.fieldName, instr.owner)
if declarer == NULL then
  return instr
end if
return InvokeVirtualInstruction(getMethodName(instr, declarer))

getFieldDeclarer(instr,owner):
if owner == NULL then
  return NULL
end if
cls ← loadPartialClass(owner)
for all fld in cls.fields do
  if fld.name == fldName then
    return owner
  end if
end for
return getFieldDeclarer(fldName, owner.superClass)
```

where *loadPartialClass*(*owner*) retrieves the class information from the class file or an already loaded class

2.4 Related Work

A similar approach to method specialisation, known as *path specialisation*, has been investigated in [36]. The system involves generating two specialised method variants for every method. Each variant contains barrier code required for different phases of collection. For a classical Baker collector one variant would contain no barrier code and the other would contain the traditional Baker read-barrier. During normal execution the unspecialised variant is used. When garbage collection enters a phase in which the barriers are required (such as during incremental collection) a conditional branch instruction before every barriered access diverts execution to the required specialised variant. Points in code known as GC *safe-points* are included, where threads can be safely stopped and the garbage collection phase switched. Just before these safe-points the specialised method variant jumps back to the unspecialised variant. Optimisations are used to remove dead code, reducing code bloat. Figure 2.12 shows the use of path specialisation to switch between barriered and unbarriered code during garbage collection phases and comparatively depicts execution without specialisation.

This scheme means that the read barrier overhead is only encountered during garbage collection phases that require a barrier. During phases in which no barriers are required there is an overhead of one conditional branch for each barriered memory access.

The problem with path specialisation is that the barriers are active for longer than they need to be. Once a field has been forwarded the barrier code will still remain until the end of the garbage collection phase. Our approach eliminates this overhead by implicitly tracking which objects have been scavenged and hence only incurring the self-scavenging overhead once per object per complete collection cycle. The self-scavenging technique does incur the overhead of beanifying all reference-type field accesses and so when the collector is not running path specialisation is more efficient. Some of the beanification overhead may be reclaimed when the compiler inlines the generated methods. It is therefore a matter of considering how long collection is running for and whether the additional overhead during collection in [36] is greater than the additional overhead outside of collection in our approach.

A direct comparison is not possible at this stage, since the work in [36] is implemented in the C# Bartok runtime environment. Furthermore [36] describes the use of this approach for STOPLESS, a Brooks-style unconditional read barrier and a concurrent mark-sweep write barrier, omitting a Baker-style read-barrier.

Path specialisation is different to method specialisation since it requires cross-specialisation execution mid-method. To introduce path specialisation into the Jikes RVM the compilers would need to be heavily modified in order to add the branching instructions to other method variants. The method variants themselves could be handled by the method specialisation framework, although dead code elimination would not be supported.

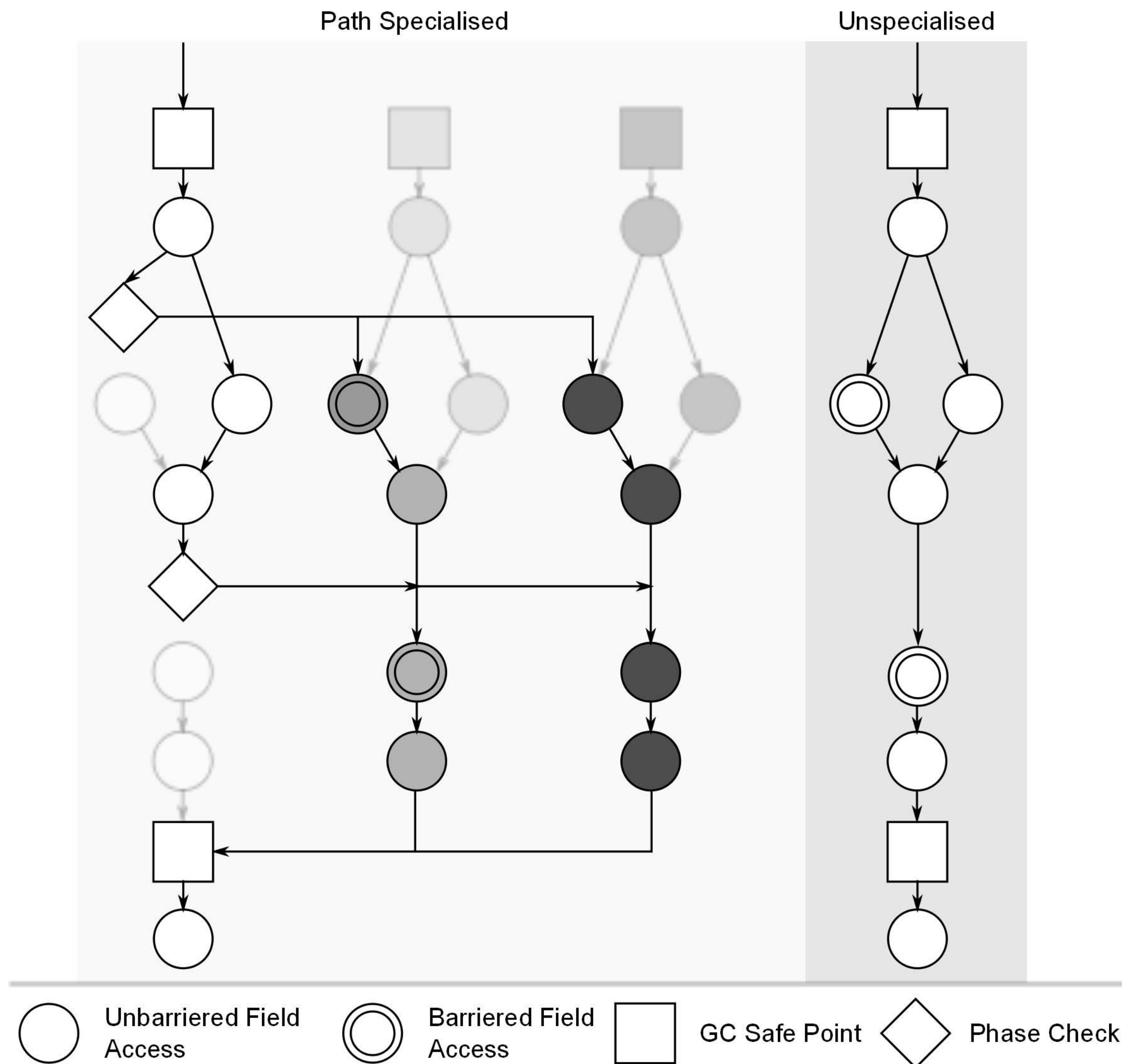


Figure 2.12: Visualisation of path specialisation applied to a code snippet (adapted from [36]). The path specialised code consists of three method variants. The first is the 'main' variant that delegates to the other variants depending on the current garbage collection phase. The second variant contains barriered field accesses used during incremental heap closure generation. The third variant contains no barriers and is used when garbage collection is not in progress. For comparison the unspecialised code is shown, which includes barriers and does not have any alternative method variants.

3 Design and Implementation

The development of the SSB collector on top of the Jikes RVM requires the use of two separate components, the MMTk and the method specialisation framework. The MMTk handles the definition of collectors within the Jikes RVM, but lacks the support for incremental collection schemes. To enable effective use of the MMTk for incremental schemes it has been augmented with two additional features:

1. An incremental framework within the MMTk to support the development of work-based incremental collectors (Section 3.1).
2. An incremental Baker collector built on top of the incremental framework to evaluate the correctness of the framework, the state of the barrier support within the RVM and for comparison with SSB (Section 3.2).

Further to the development of incremental collector support this project also requires modifications to the existing method specialisation and class transformation components to enable the application of method specialisation and beanfication across the entire RVM:

1. Updating the method specialisation framework to Jikes RVM 3.1 (Section 3.3).
2. Correcting the beanification transform to support name mangling and the application to internal RVM classes (Section 3.3.3).

The SSB collector is the combination of an incremental collection scheme based upon the incremental Baker collector *without* read-barriers, the corrected beanification transform and a self-scavenging method specialisation transform.

3.1 MMTk Incremental Framework

In a memory management scheme governed by incremental collection mutator threads are prioritised. This means that collector threads are blocked until a mutator thread triggers an event that leads to an incremental step. As depicted in Figure 3.1a mutator threads yield to collector threads when work is required, determined by the incremental trigger condition.

Conversely, when collector threads have priority mutator threads are blocked during collection. Collector threads will yield to mutator threads when some condition is met e.g. every hundred or so object traces as shown in Figure 3.1b. The mutator threads

will unconditionally yield to the collector threads at the next *safe* yieldpoint. Yielding at every safe yieldpoint means that the collector thread decides whether to continue collection. While it is possible to implement an incremental paradigm using this approach (i.e. immediately yield back to the mutator thread if a trigger condition is not met), it is extremely inefficient, as there will be numerous unnecessary context switches between mutator and collector threads.

The existing MMTk incorporated in the Jikes RVM 3.1 includes a concurrent collector framework ported from Jikes 3.0.1. It must be noted that this existing framework is specifically tailored towards prioritising collector threads rather than mutator threads.

Since the MMTk only currently offers collector prioritisation, and this is an inefficient means for implementing an incremental collector, it must be augmented to support mutator thread priority. This requires the introduction of the following constructs in the MMTk:

1. Incremental closure phases to perform heap closure generation incrementally (Section 3.1.1).

2. Work-based triggers to enable mutator threads to trigger an incremental collection step on object allocation (Section 3.1.2).[1]

3. An abstract incremental `Plan` and supporting classes to structure and simplify the development of incremental collectors (Section 3.1.4).

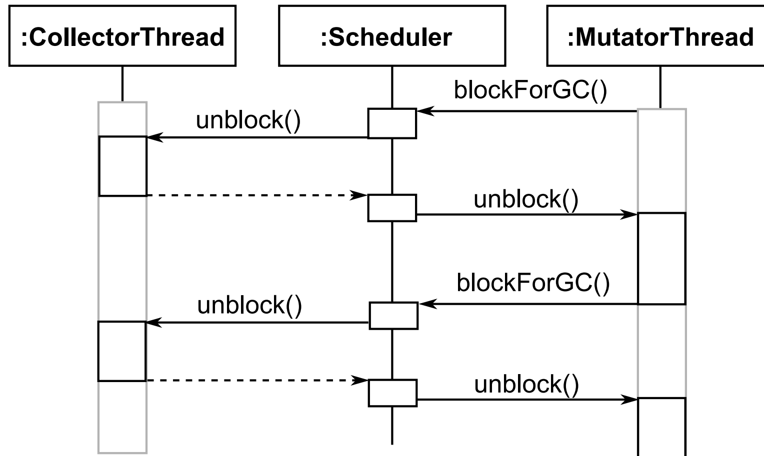
3.1.1 Incremental Closure

The stop-the-world nature of the MMTk is immediately clear in the implementation of the collection process. The operation of a collector is split into phases that are concerned with specific aspects of collection. These phases are placed on a phase stack from which phases are popped once they begin execution. This only allows for a phase to execute once, after which the next phase on the top of the stack will execute. There exists no way for an incremental process, which executes distinct iterations, to remain on the stack.

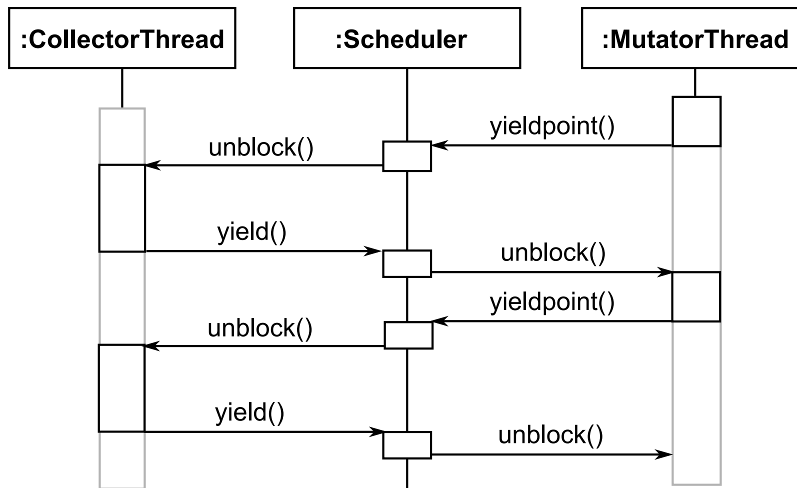
The implementation of the phase stack and associated management routines is a static array of `Phase` objects, a phase pointer indicating the next phase to execute and static methods within the `Phase` class to manage the execution of phases and the modification of the stack.

The existing concurrent collection support in the MMTk is able to utilise the existing phase structure since the concurrent phase will not return to the phase stack manager until after the concurrent closure has completed. An incremental phase must return to the phase stack manager after *every* incremental step is completed. Since the heap closure

[1] Time-based triggers have also been inserted into thread yieldpoints based on an existing timer thread that maintains a configurable thread quantum count. This allows the collector thread to be controlled by partial soft real-time bounds.



(a) Depiction of mutator priority, where a mutator thread dictates when more incremental work will be executed.



(b) Depiction of collector priority, a collector thread will perform work and periodically yield for a mutator thread to execute.

Figure 3.1: A visual representation of mutator thread priority and collector thread priority for scheduling concurrent collection.

may not have been completely generated after an incremental step, the removal of the phase will result in the incorrect assumption that the closure has been completed.

To enable incremental execution an additional phase type is introduced that remains on the phase stack until an arbitrary condition is met. An incremental phase is tied directly to the collector, since it is the collector's responsibility to perform the heap closure.

The traversal of the phase stack is performed on a phase-by-phase, case-by-case basis, every phase being processed is handled within a switch statement on the phase type. The addition of a new phase type simply needs the introduction of a case to handle it. Listing 3.1 shows the code used to execute an incremental phase, the phase execution method is called on the collector and is removed from the stack only if the collector has indicated that the incremental work is complete. Figure 3.2 shows the phase execution of an incremental collector. The only difference between the incremental and stop-the-world phase stack is in the generation of the heap closure, with the incremental collector using many incremental steps.

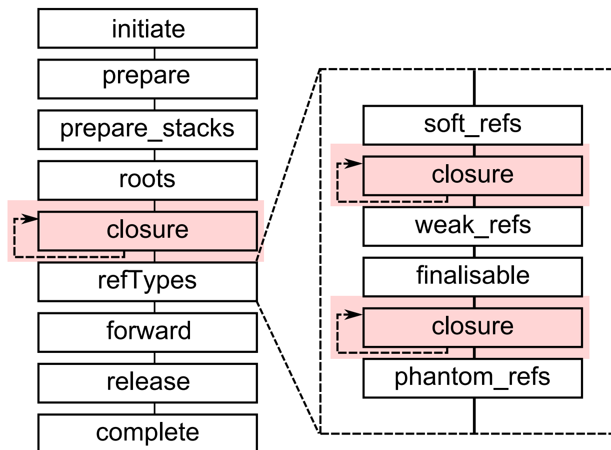


Figure 3.2: Outline of the phases that form an incremental collection.

An incremental closure phase relies on the collector being notified of modifications made by the mutator threads to the heap graph. To support the notification of mutations the incremental phase consults the mutators before calling the collector. This allows the mutators to flush all of the objects in their *remembered set* buffers into the shared closure queue. These buffers simply maintain a record of objects that have been updated by the mutator. It is necessary for this to happen before the collector continues processing the closure as these changes could cause a reachable object to be left unvisited.

```
1  public void processStack(){
2    ...
3    switch(state){
4      ...
5      case SCHEDULE_INCREMENTAL:
6      {
7        if( primary ){
8          MutatorContext mutator;
9          /* Flush mutator object bufferes */
10         while( ( mutator = VM.activePlan.getNextMutator() ) != null ){
11           mutator.collectionPhase( getPhaseId(FLUSH_MUTATOR),
12                                    primary );
13         }
14         VM.activePlan.resetMutatorIterator();
15       }
16       /* Sync collector threads as new objects may need scanning */
17       collector.rendezvous();
18       /* Perform incremental collection */
19       collector.incrementalPhase(INCREMENTAL, primary);
20       /* Sync collector threads before testing completion */
21       collectorContext.rendezvous();
22
23       if(!incrementalClosureCompleted){
24         if( primary ){
25           /* Ensure the mutators think collection is complete */
26           Plan.setGCStatus(Plan.NOT_IN_GC);
27           pauseCollectionTimers();
28         }
29         return;
30       }else{
31         if( primary ){
32           /* We're done, so pop the phase */
33           popScheduledPhase();
34           incrementalClosureCompleted = false;
35         }
36         /* Sync collector threads so they continue from new phase */
37         collector.rendezvous();
38       }
39       break;
40     }
41     ...
42   }
43   ...
44 }
```

Listing 3.1: Code used to process an incremental phase. First the mutators are flushed so that the collector has the latest mutated object queue. Next the collector performs some incremental work. The collectors synchronise and if incremental work is not completed return to the mutator thread. Otherwise the remaining collector phases are executed.

3.1.2 Work-Based Triggers

In order to support incremental collection the mutator threads need to be able to trigger more collector work to be done. Without a triggering mechanism the heap closure would never be completely generated and the collector would not reach the release phase where memory is reclaimed. Since mutator threads have priority in an incremental scheme the absence of a trigger will prevent garbage collection.

While there is a trigger system for initial collection there is no method that allows collection to be triggered on an allocation (*work-based*) or time slice (*time-based*).

A simple modification to the mutator allocation method can trigger incremental collection on every object allocation. However the MMTk uses a block allocation method to reduce thread contention for allocation. This means that instead of a thread asking for space to allocate each object, it asks for a number of fixed blocks that can serve several allocation requests. Therefore, triggering collection on thread-local allocation results in wasted attempts, since the collector already believes the block is allocated. Unless a new block is requested the thread-local allocation does not affect the amount of free memory (according to the collector).

Instead we exploit the block allocation system and use the request for a new block to trigger incremental collection. In fact, initial collection is triggered when a request for a block cannot be served; this is designated a safe-point for garbage collection.

Listing 3.2 shows the trigger code for incremental collection alongside trigger code for initial collection, allowing the addition of incremental collection without affecting existing collectors. When a mutator requests more pages for allocation the initial collection is triggered if there are not enough pages (`reservePages()` returns False). This causes the request to be rejected and the mutator will retry for allocation. For incremental work, collection is triggered if there is still work to do regardless of whether the request can be served or not. The trigger code works by consulting the collector to check whether incremental work needs to be performed. If it does then the mutator thread yields for collection. The advantage of this is that a work-based collector can use whatever triggering condition it wants and it is completely encapsulated in the collector class. By default a stop-the-world collector never yields for incremental work.

A minor issue can arise from incremental collection steps being mistaken for complete collections, since any garbage collection work, partial or not, causes the allocation to fail and retry. The number of attempts at allocation are maintained and used to indicate when there is a possible out of memory error. To prevent the incremental collections from contributing to the number of allocation attempts the process of copying an object uses different collection routines. These routines perform allocation as normal, but they do not trigger collection (hence prevent an infinite loop caused by acquiring space during a collection) and they do not affect the number of allocation attempts.

The MMTk originally ensured that only collector attempts at allocation are not recorded. However, the introduction of object forwarding in mutator threads means

```
1  public Address acquire(int pages, boolean gcRequest){
2      ActivePlan currentContext = VM.activePlan;
3      //handle the case where a request cannot be served
4      if( !reservePages(pages) ){
5        if( currentContext.isMutator() &&
6            Plan.isInitialised() &&
7            currentContext.global().poll() ){
8                clearRequest(pages);
9                return Address.Zero;
10           }
11     }
12     ...
13     //allow collection to resume if required
14     currentContext.global().yieldForIncrementalWork(pages, gcRequest);
15     //allocate pages
16     ...
17 }
```

Listing 3.2: The modified block acquisition method that provides an opportunity for the collection scheme to block mutator threads for incremental work

that an object could be copied for collection by a mutator. The MMTk would treat this as a normal allocation and subsequently record it. The use of separate paths for allocations due to copying allows all collection allocation to be safely ignored.

3.1.2.1 Threading Issues

The code in Listing 3.2 is flawed, it can incorrectly allocate objects to the wrong `Space`. This was one of the stability issues in the Baker collector original believed to be caused by missed barriers (see Section 3.2.6). The problem is that a thread may be suspended at `yieldForIncrementalWork` during one cycle and not get resumed until the next cycle has started. The suspended thread would be trying to acquire space in cycle i. The to-space pointer for cycle i is different to the to-space pointer for cycle $i+1$. When the thread is resumed in cycle $i+1$ it will be acquiring space in the current from-space. This may be granted, since the trigger condition does not necessarily result from cycle i's to-space becoming full (now from-space). The mutator thread would then have assigned a new object to from-space and may have set it as a field in an object. A read of the same field before the cycle terminates would copy it into the correct to-space, but the object reference can still be stored on the stack unmodified. This leads to null pointer exceptions as a reachable object is collected with the rest of from-space when the cycle terminates.

To fix this issue a modification is made to `yieldForIncrementalWork` so that it now indicates whether the thread did actually yield. Listing 3.3 shows the modified `acquire`

method. If a thread did yield for collection then the `Space` is consulted to see if it is still actively accepting allocation requests. If the `Space` is no longer valid for new allocations then the thread must try to acquire pages from a different `Space` (returns `Address.zero()` to indicate failure). A `Space` can be set to accept new allocations by the collector, for a copy space this is indicated by the internal `boolean fromSpace` field (if it is representing from-space it does not accept allocations).

It is not very easy to detect that this scenario is the culprit, since the errors caused by it are exhibited perhaps one or two collection cycles in the future. Fortunately, the explicit self-scavenging barrier development uncovered the problem due to a fail-fast test on self scavenging. If an object is self-scavenging then it should exist in to-space (copied over by a parent who has self-scavenged), therefore an assertion is placed in self-scavenging code that ensures this is the case. Coupled with stack traces, the failing of this assertion revealed that some threads are blocked waiting on a brand new collection while other threads are waiting on a currently active incremental collection. This indicates that some threads have not been resumed before a new collection has been initiated (otherwise they would be waiting on a new collection). Furthermore, the fact that the assertion failed on a newly allocated object led to the investigation into how allocation is handled and subsequently the discovery of the problem.

3.1.3 Mutator Object Tracing

Since the MMTk primarily focusses on stop the world collection it assumes that the collector threads are the only threads that handle the tracing and forwarding of objects. With incremental copying collection we need the mutator threads to contribute to the forwarding of objects in the read or write barriers.

To support mutator object forwarding a new `SharedContext` class representing all thread contexts is used that contains the interface for forwarding objects[2]. The `SharedContext` is used in place of the `CollectorContext` in memory management routines that perform object copying.

This modification allows a mutator thread to perform object forwarding through the modified memory management copy operations.

[2]The `SharedContext` class is abstract, but cannot be made into an interface as the `INVOKEINTERFACE` bytecode cannot be used within uninterruptible regions (such as many memory operations).

```
1  public Address acquire(int pages, boolean gcRequest){
2      ActivePlan currentContext = VM.activePlan;
3      //handle the case where a request cannot be served
4      if( !reservePages(pages) ){
5        if( currentContext.isMutator() &&
6            Plan.isInitialised() &&
7            currentContext.global().poll() ){
8                clearRequest(pages);
9                return Address.zero();
10           }
11     }
12     ...
13     //allow collection to resume if required
14     //return failure if we can no longer allocate in this space
15     if( currentContext.global()
16                       .yieldForIncrementalWork(pages, gcRequest)
17         && !isAllocationAccepted() ){
18          clearRequest(pages);
19          return Address.zero();
20     }
21     //allocate pages
22     ...
23 }
```

Listing 3.3: The corrected block acquisition method that fixes a threading issue from the first implementation

3.1.4 High-Level Plan

Many incremental collectors will have exactly the same sequence of phases and so a generic high level plan has been developed that uses the incremental heap closure phase. This allows for an incremental collector implementation to be concerned only with the implementation of the closure and barriers rather than the entire plan sequence and the subtleties associated with phase ordering.

The high-level plan includes the order of phase execution. As with stop the world collectors execution starts with a preparation phase that resets counters and queues ready for a new collection. This is followed by preparing the stacks for root scanning, ensuring that all mutator threads are in a state that will allow stack inspection. The roots, including the thread stacks, are then scanned and this adds all root objects to the closure queue for processing. There are three closure phases. The first traces all objects that are reachable from the roots; this is where objects are marked or forwarded as required. Weak references (and soft/phantom[3] references) are then processed and the closure of these computed. All weak references are then processed, as they are not directly referenced, the weak references that are included in the closure remain and are forwarded. Any weak references not marked during the closure are removed. Finally the memory occupied by untraced objects is released and the collection cycle terminates.

The modifications to phases, triggers and object tracing enable support for incremental collectors. The implementation of the proposed extensions are evaluated by building an incremental Baker collector.

[3]Object reachability is defined in the Java specification as: "An object is strongly reachable if it can be reached by some thread without traversing any reference objects. A newly-created object is strongly reachable by the thread that created it. An object is softly reachable if it is not strongly reachable but can be reached by traversing a soft reference. An object is weakly reachable if it is neither strongly nor softly reachable but can be reached by traversing a weak reference. When the weak references to a weakly-reachable object are cleared, the object becomes eligible for finalization. An object is phantom reachable if it is neither strongly, softly, nor weakly reachable, it has been finalized, and some phantom reference refers to it." - from `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ref/package-summary.html#reachability`

3.2 Incremental Baker Collector Implementation

An incremental Baker collector is essentially a semi-space collector than performs an incremental work-based closure of the heap. Using the existing semi-space stop-the-world collector as a base the new Baker collector modifies the class hierarchies that it inherits from (`Incremental` over `StopTheWorld`) and the closure type used. The incremental framework is used to handle the common elements of incremental collectors, such as switching the closure phase type and providing the interface methods that need to be implemented by any incremental collector.

For the Baker collector the order of some maintenance events need to be altered. For example the stop-the-world semi-space collector flips to-space and from-space at the end of collection, just before re-enabling the mutator threads. The Baker collector must do this at collection initialisation, otherwise when the mutator threads are re-enabled after the first incremental step they will be allocating to the already full from-space.

Additional elements need to be incorporated into the Baker collector so that it can function correctly. These include:

1. An incremental trigger condition so that work-based incremental phases can be triggered when required (Section 3.2.1).

2. Forwarding pointer construction for objects that have been copied to to-space, but are still referenced by objects that have yet to be scanned (Section 3.2.2).

3. Baker-style conditional read barrier implementation to uphold the strong tri-colour invariant (Section 3.2.3).

4. Yuasa style write barrier to support the incremental marking of mark-sweep spaces (Section 3.2.4).

3.2.1 Incremental Trigger

The Baker collector implementation uses a work-based trigger condition that is determined by whether collection is currently in progress and whether there are any more objects left to process in the heap closure. Once all objects have been processed the trigger condition fails, this also causes the incremental closure phase to terminate after which collection finalisation occurs.

Listing 3.4 contains the code used to trigger an incremental collection. A shared `Trace` object contains the global queue of objects waiting to be processed, once this queue is empty the incremental phase can terminate. If incremental work is required then all mutator threads are blocked for garbage collection, which resumes from the incremental phase at the top of the phase stack. Otherwise there is no active incremental phase and the request is served as usual.

3.2.2 Forwarding Pointers

To support the forwarding of objects a header word is appended to every object that stores the forwarding address. As forwarding is not atomic the word also handles forwarding state. When any contention arises one thread will perform the forwarding while any contending threads will spin-wait on the forwarding state until it is forwarded. This does not cause any significant issues since forwarding is a relatively fast process.

Once an object is forwarded future reads of the old object will go through the read barrier, which will return the forwarding address in to-space as the object reference.

Listing 3.5 shows the use of forwarding pointers during read barrier execution. When an object needs forwarding the state of the object is first determined; if it has already been forwarded then the forward reference is returned. If another thread is currently forwarding the object then it waits until forwarding is complete and then returns the forward reference. Otherwise the current thread needs to forward the object, it sets the object state to forwarding and forwards it using memory management operations. Once the forwarding is complete the new address is returned and copied into the forwarding word, at which point the state is set to forwarded. The object is then added to the scavenge queue (`trace.processNode`).

The existing semi-space collector in the MMTk makes use of a forwarding bits in the status word of each object. However, according to the core Jikes development team, the status word is used for other mutator operations and could result in the corrupting of the forwarding pointer. The forwarding word implementation given here is logically identical to the semi-space implementation, save for the use of different memory management functions to access the correct memory location.

3.2.3 Read Barrier

The read and write barriers are one of the most important aspects of any concurrent collector. They ensure that the collector's view of the heap is consistent with mutator operations. The Baker collector only requires a conditional read-barrier that forwards an object that is about to be read if it is not already in to-space. Unfortunately this is not the whole story.

The MMTk splits the heap into several areas of memory, known as `spaces`. Each `Space` handles a different kind of object based on how it is accessed and its life cycle. There are five main spaces, *Immortal*, *Non-Moving*, *Large-Object*, *Meta* and *Small-Object*. A collector implementation is usually focussed only on small-object space as this is where the majority of objects will reside. However, the remaining spaces are used for internal objects (Non-Moving and Immortal), objects that are considered large (Large-Object) and collector data structures (Meta). Immortal space uses no collection, as all the objects in it are live for the entire life of the RVM. Meta data space employs explicit memory management as it is used by the garbage collector. The remaining spaces use slightly

```
public void yieldForIncrementalWork(int pages, boolean gcRequest){
  /* Only block for GC if work is required
   * and this is not a request by GC
   */
  if( !gcRequest && incrementalWorkRequired() ){
  /* Notify the collector of how much is being allocated */
  setPagesRequested(pages);
  /* Trigger garbage collection */
  triggerInternalCollectionRequest();
  VM.collection.blockForGC();
  }
}

/* Incremental work is always required
 * when incremental closure is active
 */
public boolean incrementalWorkRequired(){
  return incrementalCollectionInProgress;
}
```

Listing 3.4: The code used to trigger an incremental trace

```
public ObjectReference traceObject(TraceLocal trace,
                                   ObjectReference object){
  //No need to trace an object that has already been copied
  if( !fromSpace ) return object;
  //We retrieve the previous forwarding state of the object atomically
  Word forwardingWord = attemptToForward(object);
  //If we need to perform forwarding then the state will be unforwarded
  if( stateIsForwardedOrBeingForwarded(forwardingWord) ){
    //Busy wait until the object has been forwarded
    while( stateIsBeingForwarded(forwardingWord) )
      forwardingWord = readForwardingWord(object);
    return extractForwardingPointer(forwardingWord);
  }
  //We perform the copy
  ObjectReference newObject = copy(object, ALLOC_DEFAULT);
  //First ensure that the new object has no forwarding state
  clearForwardingPointer(newObject);
  //Update the forwarding pointer of the object
  //(also sets state to forwarded)
  setForwardingPointer(object, newObject);
  //Add the new object to the scan queue
  trace.processNode(newObject);
  return newObject;
}
```

Listing 3.5: The code used to forward an object during read barrier execution

different variants of mark-sweep collection. This poses a serious problem for the Baker collector, since it assumes one unified heap. It is not possible to traverse the heap graph of just a single space, since objects may have references across spaces. When performing incremental collection over small-object space you are in fact performing a global incremental collection. Therefore the mark-sweep spaces need to have an incremental mark-sweep algorithm to handle them. One of the benefits of the Baker collector is that it only affected object reference reads; however to handle incremental mark-sweep an additional write barrier is required. Thus, it is not currently possible to have a pure Baker collector in the Jikes RVM - it becomes a hybrid.

The read barrier is the bottleneck associated with the Baker collector and to make significant performance gains it requires detailed optimisations. Originally the collector incurred overhead greater than 100% and this was eventually reduced to an average of 40-60% through switching from dis-contiguous to contiguous space allocation.

Further optimisations are known to be available, for instance an optimisation that should be added is the inlining of the fast path of the barrier. The fast path handles the case in which no object forwarding is required. When this is attempted in the Jikes RVM segmentation faults arise, despite the non-inlined implementation working correctly. This is due to the optimising compiler, which is not guaranteed to leave inlined code untouched. As a result optimisations may be applied to the inlined barrier that break the assumptions of the barrier code (such as uninterruptibility, execution order and execution contract). By not inlining the barrier code the optimising compiler is restricted to the contract of the barrier method, preventing some of the more advanced optimisations that break it, but limiting the efficiency of the barrier.

3.2.4 Cross-Space Write Barrier

The mark-sweep spaces are managed by a Yuasa snapshot-at-the-beginning write barrier. Coupled with the Baker style read barrier all object accesses end up passing through the collector. This is not desirable, but it would be far less effective to switch all spaces to a semi-space copying algorithm. Although the use of two different barrier mechanisms should not cause any issues the Jikes RVM is not a perfect implementation and subtle problems can arise from using multiple styles simultaneously.

Listing 3.6 contains the common code that is used for all write-barriers in order to trace object references being overwritten. The basic structure delegates the object tracing to the `Space` in which the object currently resides. Each type of `Space` has a different method for tracing objects, therefore a unified method for tracing cannot be used.

The Jikes RVM roadmap indicates that in future releases they aim to separate memory management concerns more explicitly so that application level objects can be managed independently of internal RVM objects. This should make the implementation of an incremental copying collector easier, however for a real-time incremental collector the entire heap needs to be incrementally handled (as well as the stack).

```
public void checkAndEnqueueReference(ObjectReference ref){
  if( barrierActive && !ref.isNull() ){
      if( isInSpace( NON_MOVING, ref ) )
          nonMovingSpace.traceObject(remset, ref);
      if( isInSpace( LOS, ref ) )
          loSpace.traceObject(remset, ref);
      if( isInSpace( SMALL_CODE, ref  ) )
          smallCodeSpace.traceObject(remset, ref);
      ...
  }
}
```

Listing 3.6: The code used to enqueue references for scanning. Used by the write barriers to record overwritten references as these may become unreachable in the unscanned heap graph while still being reachable from an already scanned object.

3.2.5 Calculating Required Incremental Work

Originally the incremental framework introduced the ability for mutators to maintain counts of bytes and objects allocated. Coupled with the counts of traced objects from the previous collection this enabled the collector to make collection decisions based on heap liveness.

The first implementation of the Baker collector calculated the number of objects that it needs to trace at each allocation using the report of heap liveness from the previous cycle (initially assuming the entire heap is live). This value, known as k, amounts to:

```
[(liveBytes / (totalBytes-liveBytes))/ bytesPerObject] * pagesRequested + remsetsAdded
```

The calculation uses the current number of live bytes (`liveBytes`) and the total number of bytes in the heap (`totalBytes`) to form a ratio of bytes used to the number of bytes that should remain after collection. If we assume that `liveBytes` bytes will be live after collection, then `totalBytes-liveBytes` should be the number of free bytes when collection terminates. The current number of live bytes is equal to the number of bytes we estimate will require tracing and therefore how many bytes will be visited as part of this collection. Therefore per byte allocated we must trace enough live bytes to complete tracing before the number of free bytes is below the number of bytes that are currently live. If we do not ensure this than by the end of collection there would not be enough bytes available to copy the remaining live bytes from the start of collection. Since collection is in terms of the number of objects collected we estimate the number of bytes per object (`bytesPerObject`) to convert our byte-based collection requirement to object-based. This is calculated once at the start of collection. We introduce a catchup factor that ensures the collector will trace enough objects to stay ahead of mutator alloca-

tions by using the count of objects traced by the mutators during incremental collection (`remsetsAdded`). Since allocation is made in terms of pages rather than per object, we multiply the result by the number of pages that are requested when an incremental step is triggered.

The final number of objects collected during each iteration can be altered by changing the `kmodifier` parameter (usually fixed at 1). This parameter allows comparisons of incremental work vs total execution time to be generated. For instance increasing the amount of incremental work increases the incremental pause time, but reduces the time spent during garbage collection and therefore the number of read barrier calls. This demonstrates the trade-off between total execution time and incremental pause times with respect to the Baker read barrier. It also allows the incremental collector to exhibit stop-the-world behaviour by setting the parameter high enough to force a complete collection on the first iteration.

If the value for k is too small then the collector starts reporting live ratio warnings. These warnings indicate that the size of the heap is larger than its maximum allowed size and will eventually lead to an out of memory error. As there are no warnings if you are collecting more than you need to, knowing whether the collector is running with the most efficient value for k is obtainable only through modifying the value of k and comparing the results.

The original calculation for k does not guarantee that the heap will not run out of space, it is based on an instantaneous interpretation of the live objects in the heap and does not take into account floating garbage. To overcome this problem the live object reporting mechanism was replaced with a method that ensures that per byte allocated, m bytes are collected. Where m is calculated such that the entire object graph is traversed before the remaining memory is used up. To achieve this we need to ensure that collection terminates before the amount of used memory, `C`, is greater than the size of the heap, `H`, plus the amount of used memory at collection start, `CT`, divided by two:

```
C > (H+CT)/2
```

Intuitively, if the whole heap was managed by the Baker collector this would mean that `CT = H/2`. Therefore `(H+CT)/2 = (H+H/2)/2 = 3H/4`, so we want collection to terminate before we're using three quarters of the heap. Ideally you would want termination to occur just before the heap is full again, to maximise mutator utilisation. However, we need to take into account *floating garbage* (objects that become garbage during collection). So in cycle i, objects that immediately become garbage may not be collected until the end of cycle $i+1$, therefore we need two complete cycles to collect floating garbage. This can be translated as collecting m bytes per byte allocated:

```
m = bytesAllocated * (CT/(targetUsedBytes-CT)) where targetUsedBytes = (H+CT)/2
```

This calculation amounts to the ratio of bytes currently used (i.e. the bytes that need tracing) to the number of bytes remaining until the target number of bytes is used. This does not use an estimation of the number of live objects in the heap and thus collects

more objects per increment, however, it does reduce the time the barriers are active and guarantees that collection will complete before the heap is out of memory.

3.2.6 Missing Objects

It is evident from testing that the collector is stable but not robust. Many different, but related error messages occur after tens to hundreds of complete cycles of collection. The fatal errors point to the collection of objects that are still live and accessible, while the remaining errors indicate that from-space objects are being used where the forwarded to-space copies should be.

3.2.6.1 Sanity Checker Confirmation

The sanity checker is a debugging tool that uses a reference counting algorithm across the entire heap graph to determine which objects are still referenced and therefore still live. When used in conjunction with another collector it highlights any objects that it believes are still accessible, but are condemned for collection.

Running the sanity checker against the Baker collector revealed that the collection cycle completed last fails on tens to hundreds of objects against the sanity checker, just before any error message is thrown. This confirms that the Baker collector is not maintaining a correct representation of the heap. Since this happens after several complete cycles it cannot be caused by a fundamental issue in the incremental steps, as this would be manifested on all cycles. It is therefore much more likely that the read/write barriers are failing to capture all mutations.

3.2.6.2 Instrumentation

To confirm that there are objects being used that remain in from-space i.e. objects that break the tri-colour invariant, a simple transform was generated that checks all object reference memory locations. If they still reside in from-space then clearly a barrier has been missed and did not forward the object before it is accessed. As expected many instances of local variables residing in from-space were reported. The instrumentation transform also confirmed that all of the common bytecode instructions such as `GETFIELD` and `PUTFIELD` do trigger barriers correctly.

Listing 3.7 shows the instrumentation transform operating over the `ALOAD` bytecode. Every `ALOAD` is hijacked and a utility method (Listing 3.8) is called that checks whether the object has been forwarded. It is not possible to set the return type of the utility method so that the bytecode verifier accepts the return value in place of all `ALOAD`'s, since this would require an upcast to the expected type. Instead the `DUP` bytecode is used so that the value returned by the `ALOAD` is duplicated and the first reference is

swallowed by the utility method, the duplicated reference is still typed on the operand stack correctly and can be verified.

```
 1 public class ALoadVerifier extends LowLevelTransform {
 2
 3   public boolean isExempt(String cname){
 4     //Do not apply to classes used during verification
 5     //Or we'll get an infinite loop on every aload
 6     return super.isExempt(cname) || ...;
 7   }
 8
 9   public InputStream apply (InputStream is, final String cname) {
10     final ClassReader reader;
11
12     if (isExempt(cname))
13       return is;
14
15     try {
16       reader = new ClassReader(is);
17     } catch (IOException e) {
18       e.printStackTrace();
19       return null;
20     }
21
22     final ClassWriter writer =
23         new ClassWriter(reader, ClassWriter.COMPUTE_MAXS);
24     ClassAdapter adapter = new ClassAdapter(writer) {
25
26       public MethodVisitor visitMethod(final int access,
27                                        final String mname,
28                                        final String desc,
29                                        final String signature,
30                                        final String[] exceptions) {
31         //Allow the method to be handled as usual
32         MethodVisitor mv = super.visitMethod(access, mname, desc,
33                                              signature, exceptions);
34         //Don't include methods that are called by Log.write
35         if(mname.equals("resolveInternal")
36             || mname.equals("<init>")
37             || mname.equals("<clinit>")
38             || mname.equals("toString")){
39           return mv;
40         }
41         return new MethodBarrierVerifer(cname,mname,mv);
42       }
43     };
44
45     reader.accept(adapter, ClassReader.SKIP_FRAMES);
46     return new ByteArrayInputStream(writer.toByteArray());
```

```
47   }
48
49
50   class MethodBarrierVerifer extends MethodAdapter{
51
52     private final String cname;
53     private final String mname;
54
55     public MethodBarrierVerifer(String owner,
56                                 String method, MethodVisitor mv) {
57       super(mv);
58       this.cname = owner;
59       this.mname = method;
60     }
61
62     public void visitVarInsn(int opcode, int var){
63       super.visitVarInsn(opcode, var);
64       if(opcode == Opcodes.ALOAD){
65         //Duplicate the value just loaded
66         super.visitInsn(Opcodes.DUP);
67         //we loaded a local variable top of the stack is now ObjectRef
68         //let's push the class and method name onto the stack
69         super.visitLdcInsn(cname);
70         super.visitLdcInsn(mname);
71         //Now we call our check
72         //Being a local variable this should have been forwarded
73         //when processing the stacks OR added later, in which case
74         //should have been forwarded by read barrier
75         super.visitMethodInsn(Opcodes.INVOKESTATIC,
76           Type.getInternalName(ObjectModel.class),
77           "validateReference",
78           Type.getDescriptor(Type.VOID_TYPE,
79             new Type[]{
80               Type.getType(ObjectReference.class),
81               Type.getType(String.class),
82               Type.getType(String.class)
83               });
84         //Method is void so our duplicated ObjectRef
85         //is now on top of the stack
86       }
87     }
88   }
89 }
```

Listing 3.7: The object verification transform used to add logic over `ALOAD` bytecodes to determine whether the loaded object is forwarded.

```
1   public static void validateReference(ObjectReference ref,
2                                         String cName, String mName){
3     if(VM.fullyBooted && MemoryManager.incrementalGCInProgress()){
4       if(!ref.isNull() && !MemoryManager.isObjectForwarded(ref)){
5         VM.sysWrite("Object Read Missed Barrier: ");
6         VM.sysWrite(cName); VM.sysWrite(" - "); VM.sysWriteln(mName);
7       }
8     }
9   }
```

Listing 3.8: The object reference verification utility method used to verify that an object reference has been forwarded.

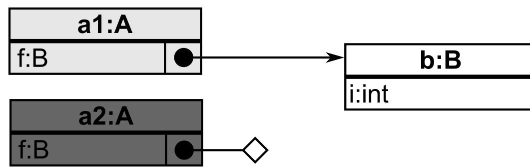
3.2.6.3 Missed Barriers

A possible reason for not tracing objects is if a mutation occurs that does not notify the collector. This would mean that the heap graph has changed and the collector may no longer trace all the objects that are accessible. Figure 3.3 depicts a simple example where one object that is yet to be scavenged copies a reference to an object that has already been scavenged and then removes the reference from itself. This would mean that the copied object is never traced, unless it is caught by a barrier (as depicted in Figure 3.4), because its new parent has already been scavenged.

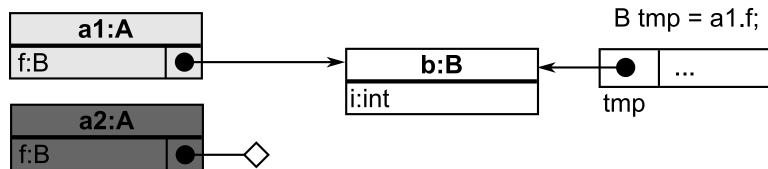
The Jikes RVM includes some poisoning tests that are used to confirm that all object reference writes are either root references or are executed by the write barriers. Therefore the example in Figure 3.3 cannot cause an issue, since the assignment will trigger the write barrier. However these poisoning tests are flawed, since they actually only test whether references that pass through the write barriers also pass through the read barriers. Object references that have no read or write barrier applied to them will be validated. Furthermore these barriers are very basic, they are likely to survive optimising compiler applications without breaking.

Magic One possible source of missed barriers is *Magic*, which is the collective name for methods within the Jikes RVM that perform low-level functions that are not usually possible in Java. The method bodies for these special methods are replaced in the compilers with machine level code. These methods are used in many places throughout the VM to perform memory functions and bypass the collector to modify memory contents. As such they can be used to change object references directly and are a likely source of inadvertent barrier bypasses.

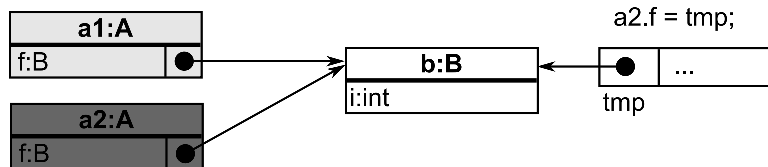
Unfortunately they are implemented at such a low level that it is very difficult to determine which, if any, magic calls are bypassing the barriers. Furthermore, it is possible that these bypasses are at a valid point in execution, such as the collector executing an object reference swap.



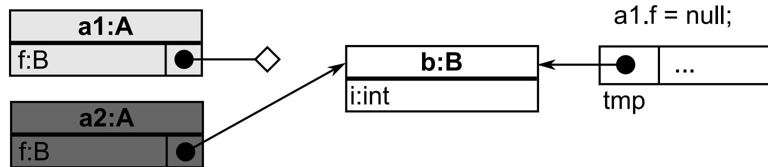
(a) Initially `a2` is scavenged and `a1` is forwarded but not yet scavenged.



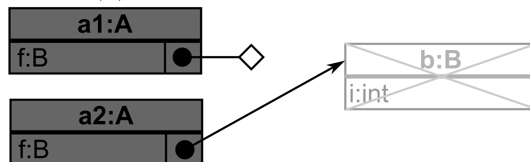
(b) Field `a1.f` is read, no barrier is tripped and the object reference is copied into the local variables.



(c) Field `a2.f` is written with the value in local variable `tmp`.

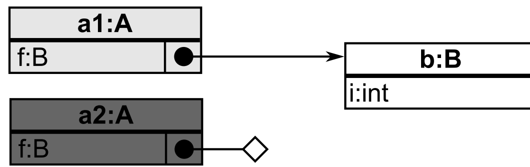


(d) Field `a1.f` is assigned null.

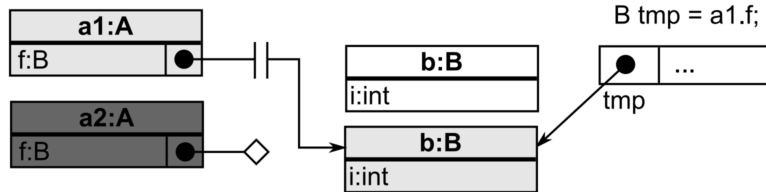


(e) Collection completes by scavenging `a1`. Object `b` is left unforwarded and the memory it occupied is subsequently reclaimed.

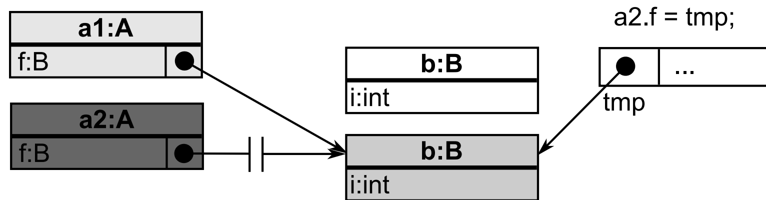
Figure 3.3: Demonstration of how missed read and write barriers can cause objects to be collected that are still reachable.



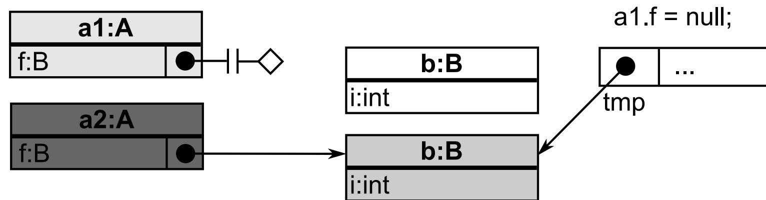
(a) Initially `a2` is scavenged and `a1` is forwarded but not yet scavenged.



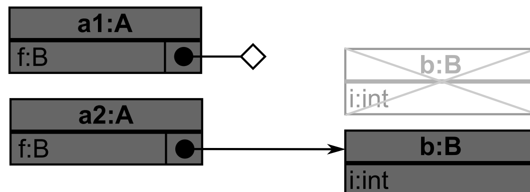
(b) Field `a1.f` is read, a read barrier is tripped and the referenced object is forwarded. The forwarded reference overwrites field `a1.f` and is copied into the local variables.



(c) Field `a2.f` is written with the value in local variable `tmp`. A write barrier is tripped which does nothing since the object being written is already forwarded and is replacing a null value.



(d) Field `a1.f` is assigned null. A write barrier is tripped that does nothing since the value being overwritten has already been forwarded.



(e) Collection completes by scavenging `a1` and the new copy of object `b`. The old copy of object `b` is left unforwarded and the memory it occupied is subsequently reclaimed.

Figure 3.4: Demonstration of how read and write barriers ensure all possibly reachable objects are not reclaimed.

Root Mutations + Scanning Another opportunity for object references to be modified without triggering a barrier is through root mutations. Since root objects are not accessed via other objects they need to be specially managed and scanned. This means that any mutations to the roots may result in a root being missed, and thus any objects that are dominated by the missed root will also be missed. While objects that become roots should be added to the tracing queue before they become a root, it is possible that mutations to root objects are handled differently by the RVM.

According to the Jikes RVM core researchers root scanning is heavily and frequently tested, while they do not rule out the possibility of an issue the likelihood is that the problem resides elsewhere. A modification to the Baker collector enforced root scanning on every incremental step, this drastically slowed the runtime, but when left running for long enough errors still occurred. This indicates that at the very least that there are other causes.

Adding to the Optimising Compiler To attempt to find and eradicate the cause of any missed barriers the optimising compiler needs to be modified in order to add barrier code where it perhaps should already exist. This includes adding barriers to magic calls one by one. Barrier calls are used in so many different scenarios that adding a barrier usually results in breaking RVM execution, making this an impractical approach to a solution.

3.2.6.4 Race Conditions

A more likely reason for objects being missed is the existence of a race condition within the barrier code. This better explains the higher rate of failure in multi-threaded benchmarks and why failures do not occur very frequently in single threaded benchmarks. A race condition may lead to two threads forwarding the same object or trying to append to the shared queue at the same time. In both scenarios some objects that should be traced are not, leading to the collector believing that they are unreachable.

Confirming that race conditions are the cause requires a scenario where no race conditions exist, for instance by forcing sequential access via synchronisation. By guaranteeing there are no race conditions, if the collector still fails then race conditions are at best not the only problem. Within the barrier code synchronisation is not an option and it is up to the programmer to ensure that there are no race conditions. Race conditions for forwarding objects are managed through a tri-state forwarding scheme as explained in Section 3.2.2. Close scrutiny of the algorithm confirmed that it was not possible for two threads to both forward an object. This means that there are no race conditions in the read barrier.

The write barriers are thread-local and monotonically increasing, so there is no way for any object references to get lost through adding to the local queue. Adding a local queue to the global queue is performed through a special locking mechanism on the global

queue. Therefore there is no way for two local queues to be added to the global queue concurrently. This means that there does not exist a race condition in the write barrier.

Given that both the read and write barriers contain no unmanaged race conditions the problems with the Baker collector cannot be attributed to thread contention in the barriers.

3.2.6.5 Context Switching

The act of switching between running thread contexts involves saving the stack and registers of the previous context and restoring the registers of the next context. These operations are performed solely through Magic and as such evade the invocation of any barriers. This should not cause an issue, however it is possible that a thread's registers or stack contain from-space references that are not valid.

A clear indication that the problems are thread-related is the fact that multi-threaded benchmarks fail more often than single-threaded benchmarks. Moreover, single-threaded benchmarks that disable the use of the AOS thread fail even less often.

To determine if context-switching is a cause the yieldpoint mechanism of Jikes RVM threads is hijacked so that the stack of a newly activated context is scanned before being allowed to continue. Unfortunately this approach leads to the RVM appearing to grind to a halt and makes it an impractical method for analysis, since a test that may usually take several seconds can be extended to tens of minutes or even hours, depending on the number of context switches.

3.2.7 Aside: Execution Issues

A major disadvantage to developing collectors over any other application is that it is next to impossible to test parts of the collector individually, specifically there is no way of ensuring that barriers are working correctly without an incremental collector that relies upon them. This makes it especially difficult to find the causes of problems. During early development it became clear that there is an issue with Jikes 3.0.1 running with an incremental copying collector, a major problem related to soft-thread scheduling. While several incremental collection cycles would run the entire runtime would suddenly pause. The system process monitor indicated that Jikes used no CPU hence immediately striking off a busy-wait or loop scenario. Unfortunately the only way to find out what each thread is doing is to request the thread dump from the debugging thread within Jikes. Of course the Jikes scheduler in 3.0.1 is a soft-scheduler and given that it was blocked it could not switch to the debugging thread. The only threading data retrievable was during execution and forcing a dump of thread call graphs and locks whenever they changed. Although painstaking this did provide some insight into the problem, threads were contending for locks and not being notified when the lock was released. This is attributable to one thread blocking on an old reference to an object that is forwarded. Thin locks in Jikes

are implemented in the headers of objects, hence a forwarded object will not notify a blocked thread that is waiting on an old copy. This indicates that the barriers are not catching every case i.e. correct barriers should never allow an old copy to be referenced.

Finding out exactly when and why the barriers are being missed is also an extremely challenging task, since when a problem caused by a missed barrier manifests itself it is too late to know when and where the barrier was missed. The implementation of the forwarding word corrupted the header of objects according to the Jikes RVM maintainers and needed to be reimplemented to consume a header word on its own.

With the fixed header word the Baker collector is able to run a full incremental collection, however it cannot handle any subsequent collection. The problem of being unable to get a thread dump became a major issue with trying to work out what was going wrong. The fact that the soft-scheduler was unable to switch to the debugging thread made finding the causes to problems much more time consuming.

To overcome this problem the porting over to Jikes RVM 3.1 was brought forwards as a necessary step in order to aid development. While this causes a fairly large set back in the project time-line it was part of the original project plan and would allow the use of the debugging thread, now seen as an invaluable tool since without it every problem becomes a trial and error investigation. The major problem with being unable to use the debugging thread is that building Jikes takes around six minutes for any change. Therefore if the only option is to make small changes to see what affect it has or glean some more output during execution then the time to find a cause is exacerbated.

3.3 Method Specialisation

The original method specialisation framework was built upon Jikes RVM 3.0.1. To take immediate advantage of the specialisation framework, development originally aimed to form a complete incremental Baker collector and 'free' read barrier alternative in this version of the RVM. Development shifted to Jikes RVM 3.1 after encountering several threading issues that were extremely time consuming to debug in 3.0.1 due to the soft-scheduler (See Section 3.2.7).

3.3.1 Method Specialisation Framework

The move from Jikes RVM 3.0.1 to 3.1 meant that the method specialisation framework developed in [18, 17, 22] had to be ported in order to support a 'free' read barrier implementation. The process involved careful merging of many compiler classes. However the majority of the port involved no changes as the framework only touches object structure and code optimisations. Several issues resulted from badly merged classes, where a missing one line statement caused multiple low-level problems.

One important issue encountered revolved around the inlining of type information block (*TIB*) accesses. As reported in [18, 17, 22] one major problem with introducing method specialisation is that classes with specialised variants have multiple TIBs (see Section 2.3) and the default inlining behaviour will cause the primary TIB reference to be inlined. This causes any method call to go via the primary TIB regardless of the currently active variant.

To confirm the problem a simple test case with two specialised method variants is used. The application executes one method variant and then flips to execute the other and flips back again repeatedly. This infinite loop is required to determine whether the problem exists and if so whether it affects the baseline compiler, optimising compiler or both. Initially the code compiled by the baseline compiler will execute until the adaptive optimisation system determines that the code is frequently executed, at which point the optimising compiler will recompile the code and replace the existing baseline compiled version. If the specialised variant correctly executes on the baseline compiled code and then stops executing once the optimising compiler version takes over then the issue relates only to the optimising compiler.

The merge caused the flip test to fail for the optimising compiler, requiring a closer look at what the optimising compiler actually inlined. The Jikes RVM enables the output of each optimising compiler phase to be echoed to the console and this can be used to confirm that the problem is related to inlining. As the TIB is accessed indirectly in the method specialisation framework the output of the optimising compiler should include two address resolution instructions, one for resolving the TIB array and another for resolving the element in the array. In the case of inlining the TIB the resolution is replaced with a direct memory address offset, leading to only a single address resolution instruction.

To solve the inlining issue the `Simplifier` class in the merged optimising compiler was corrected with reference to the working specialisation implementation. With the code reintroduced the method specialisation test passed and Jikes 3.1 now supports the method specialisation framework.

3.3.2 Garbage Collection Specialisation

The Jikes RVM includes an unfortunately named method specialisation framework for inserting scavenging methods that apply to certain object types. These types are identified with *patterns* that indicate, for types with six or fewer fields, which fields hold object references. This is used to reduce the use of the slow path, which must iterate through all fields of a type to identify reference entries.

When a type is resolved the `SpecializedScanMethod` object is notified and installs the specialised scavenge methods into the type's TIB. When an object is later scavenged the specialised scavenge method is looked up in the TIB of the object and executed directly. This lookup is encoded in the optimising compiler using the offset of the method in the TIB.

The original method specialisation framework did not handle the installation of specialised scavenge methods in all TIBs. Most likely because the existence of the specialised scavenging methods is only clear when dealing with garbage collection. This means that when an object using a specialised TIB is scavenged the virtual machine encounters a hardware trap, since the specialised scavenge method is not installed.

The framework has now been updated to install specialised scavenge methods into all TIBs for a type.

3.3.3 Extending The Beanification Transform

To enable a free read-barrier implementation all object interactions need to be hijacked so that self-scavenging can be triggered. Interaction with an object can occur through method calls and field accesses. Method calls can be hijacked through method specialisation, but field accesses occur directly through object references and do not execute any object owner code. This means that the object to which a field belongs is not 'aware' of the field access and as such cannot perform the self-scavenging process.

This can be counteracted by an object oriented paradigm known as beanification - wrapping all field accesses in method calls. So that field accesses are made indirect through method calls on the owner of the field. With such a mechanism in place all object interactions can be hijacked using method specialisation.

The existing beanification transform cannot handle cyclic dependencies (see Section 2.3.2). In general this is not an issue with application-level code that contains no cyclic dependencies. However, the internals of the Jikes RVM and the Java classpath do have instances of cyclic dependencies. A pure free read barrier implementation that uses method specialisation needs to operate across every object of every class, requiring a fix to the beanification problem.

The updated beanification transform postpones fully loading classes during class beanification by loading any referenced classes directly (bypassing the JikesRVM class loading call graph that invokes beanification). This process leads to the loading of classes twice, once for extracting field information and again for performing actual class loading. The overhead of such a process is considerable. However it only happens during initial class loading and so does not affect amortised results that are of interest in this project.

Listing 3.9 shows part of the `FieldCache` class used to store and retrieve field information. There is more overhead concerned with the updated process due to the addition of field caching, used to prevent previously explored class hierarchies from being revisited. The key difference between the two processes is the way classes are loaded for field exploration. The original process causes a class to be resolved through type reflection and the updated process uses the class loader directly to read the class file (bypassing actual class loading and resolution).

```
1 public class FieldCache {
2   /* Store class name > field information */
3   private static Map<String, ClassInfo> loadedClasses;
4
5   static{
6     loadedClasses = new HashMap<->();
7   }
8   /* Private class storing class information
9    * This means we don't need to use the tree-api */
10  class ClassInfo{
11    /* instance fields */
12    public final Set<String> fields = new HashSet<->();
13    /* static fields */
14    public final Set<String> staticFields = new HashSet<->();
15    /* static final fields */
16    public final Set<String> staticFinalFields = new HashSet<->();
17    /* super class */
18    public String superClass = null;
19    /* interfaces */
20    public String[] interfaces = new String[0];
21  }
22
23  /**
24   * Gets the name of the class for which the given field is an
        instance
25   * searching using only fully loaded classes
26   * @param root the class to start looking from
27   * @param fieldName the name of the field to find
28   * @return the name of the class which declares the field
29   *         or null if the field was not found
30   */
31  private String getInstanceFieldOwnerFromClasses(Class root,
32                                                   String fieldName){
33    while(root != null){
34      try{
35        Field f = root.getField(fieldName);
36        return f.getDeclaringClass().getName();
37      }catch(NoSuchFieldException nsfe) { }
38      root = root.getSuperclass();
39    }
40    return null;
41  }
42
43  /**
44   * Gets the name of the class that declared the given field
45   * @param root the class to start searching from
46   * @param fieldName the name of the instance field to find
47   * @return the name of the declaring class
```

```
48     */
49    public String getInstanceFieldOwner(String root, String fieldName){
50      try{
51        String curr = root;
52        String result;
53        boolean fullyLoaded;
54        while(curr != null){
55          fullyLoaded = false;
56          /* If the class is fully loaded
57           * then the action of loadPartialClass will
58           * return false, because there will be no input stream found
59           */
60          if(!loadedClasses.containsKey(curr)){
61            fullyLoaded = !loadPartialClass(curr);
62          }
63          if(fullyLoaded){
64            //This means that from now on we search using full class
65            result = getInstanceFieldOwnerFromClasses(
66                         loadFullClass(curr),
67                         fieldName );
68            /* If the result is not null return the class name found
69             * Otherwise we return the root classname
70             */
71            if(result != null)
72              return result.replace('.','/');
73            else
74              return root.replace('.','/');
75          }else{
76            /* Continue our search using partial data */
77            if(loadedClasses.get(curr).fields.contains(fieldName))
78              return curr.replace('.','/');
79          }
80          curr = loadedClasses.get(curr).superClass;
81        }
82      }catch(Exception ex){
83        ex.printStackTrace();
84      }
85      /* we got here, means we didn't find it
86       * we own it (probably)
87       */
88      return root.replace('.','/');
89    }
90    ...
91  }
```

Listing 3.9: An extract from the new `FieldCache` class showing the code used to retrieve field information from classes.

The modification to the transform successfully enables internal Jikes RVM classes to be beanified. Since the original transform is unable to do this [18, 17, 22] underestimates the cost of beanification. Section 4.1 details benchmarking results that compare the original and updated beanification transforms.

3.3.4 Fixing Method Specialisation

The use of method specialisation to implement a 'free' read barrier resulted in the discovery of several flaws in the method specialisation framework that prevented the SSB collector from functioning correctly. The use of the framework over every method of every class was the most widespread application of method specialisation since its development. This process revealed that the original testing did not sufficiently cover common specialisation cases, including the specialisation of methods:

1. In a class that are not overwritten in a subclass.
2. Implementing an interface method.
3. Declared or inherited in a final class.

3.3.4.1 Obsolete Methods

The original method specialisation framework failed to record the specialisations in which a method exists. Each method maintains a map to determine which specialisation the method resides. This map is used to reverse look-up a method by name. If the method does not have an entry for the currently active type information block (*TIB*[4]) in its map then it is not the correct version of the method to execute. For methods that straddle specialisations they must have their map updated for each specialisation. The original framework did not ensure this and so erroneous `MethodNotFound` exceptions resulted. A simple modification to class resolution ensured that all methods recorded every TIB in which they existed.

3.3.4.2 Specialised Superclass Methods

When specialising a method in a class that is not overwritten by a subclass and the subclass has fewer specialisations, some specialised variants are not included in the subclass. Specifically if a class has i specialised method variants and a superclass has j specialised variants and $j > i$, then all specialised variants $> i$ in the superclass will be ignored by the class.

[4] A Type Information Block contains information about an object's type, it also includes a virtual method table used during dynamic invocation. The method specialisation framework switches an object between different TIBs to change which version of a method a dynamic call will execute. All TIBs store method variants at the same offset as the original method in the primary (default) TIB.

This problem arises because during class resolution the number of alternative TIBs was determined by the class' immediate specialisations i.e. the maximum number of specialised variants declared by the class. Therefore, if the superclass has more specialised variants no TIBs would be created for them. The number of TIBs is then used to copy specialised methods that are not overridden from the superclass.

To fix this issue the number of TIBs to generate is calculated as the maximum of the number of TIBs in the superclass and the number of specialised method variants declared in the immediate class. This calculation is then used when copying specialised superclass methods.

A basic test case for this issue has been written consisting of a class that contains several methods and a specialised variant for each and a subclass that overrides no methods. This would result in the subclass having no specialised TIBs (only the primary TIB), which can be checked at runtime.

3.3.4.3 Interface Methods

The `INVOKEINTERFACE` bytecode is handled differently in the Jikes RVM from normal virtual method invocation. Specifically interface methods are looked up in an interface method table (*IMT*) in the TIB, rather than through the virtual method table (*VMT*) embedded in the TIB. The reason for this is that any virtual method will be stored in the same TIB offset in the defining class and every sub-class, but interface methods can belong to different class hierarchies and so the TIB offset may be different between different implementing classes.

The IMT is identical in structure to a VMT, storing pointers directly to method code. However, as shown in Figure 3.5 the VMT is part of the TIB, whereas the IMT is referenced in the TIB, but stored separately in memory. For efficient interface invocation the IMT ensures that the offset of each interface method in the IMT is identical for any class implementing the same interface.

The method specialisation framework did not take this into account. The same IMT table used by the primary TIB was duplicated in all other TIBs. However, if a method that implements an interface is specialised then the specialised variant will not be invoked when the corresponding specialised TIB is in use (the method in the primary TIB will be invoked instead).

An update to the framework now provides each specialised TIB a specialised IMT. The IMT is generated using the methods in the TIB that will reference it. For example, the associated IMT for TIB 1 will reference only the methods that exist in TIB 1, as shown in Figure 3.5.

A further modification needs to be made to handle updates to compiled methods by the optimising compiler. When a *hot* method is identified by the AOS (see Section 2.2) it is added to a queue to be optimised. The optimising compiler will generate a new, optimised set of instructions for the method. This newly generated optimised version

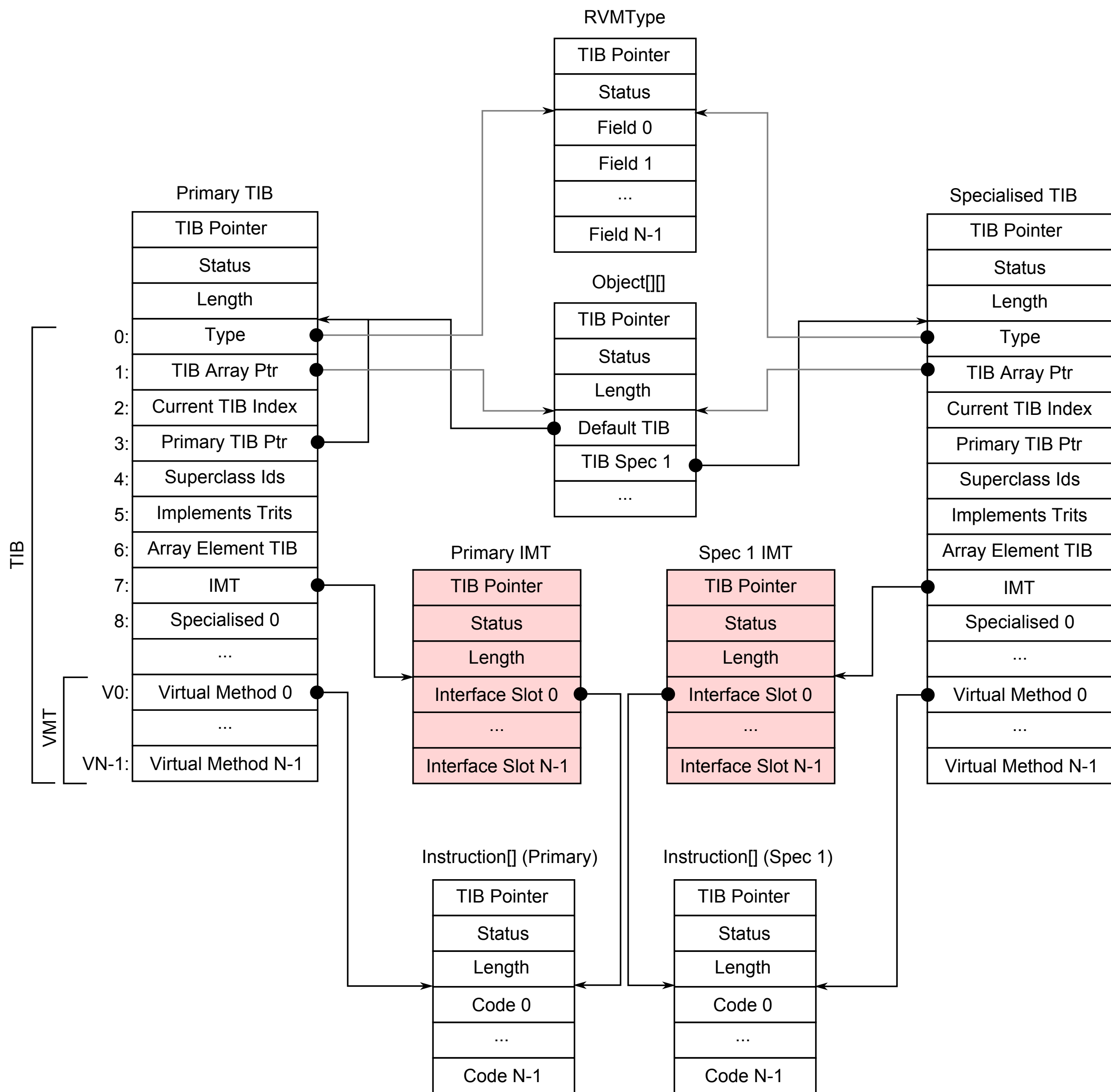


Figure 3.5: The modified TIB structure used for method specialisation. The key change is the use of separate interface method tables for each TIB. The TIB-specific IMTs reference only the methods that the referencing TIB points to, supporting the `INVOKEINTERFACE` bytecode for specialised method variants.

replaces the existing compiled method for the corresponding `RVMMethod` instance. At this point the newly optimised code will not be executed as the TIBs of the class to which the `RVMMethod` instance is associated have not been updated (they point directly to the existing unoptimised instructions).

The declaring class of the method is notified that a newly compiled version is available. This results in the process of updating the TIB entries for the method so that the newly compiled instructions are referenced. Subsequently the IMT of each updated TIB must be updated. To perform the IMT update a map of `RVMMethod` instances that implement interface methods is used to find the slot in the IMT that needs to be updated. This search is performed using direct address comparisons between the map entries and the updated method instance. For specialised methods the updated method instance will not be identical to that stored in the map (since the map is generated using the primary TIB) and so will not be updated. To overcome this issue the search checks to see if the updated method instance is a specialised instance, if it is then the unspecialised variant stored in the primary TIB is found and used for the comparison instead.

Jikes treats TIBs as primitive `word` arrays and this means that any references within a TIB are not treated as references by the garbage collector. By specifying an IMT per TIB, only the TIB holds a reference to the corresponding IMT. To prevent the IMT from being garbage collected it is added to the declaring class' *object cache*. The object cache is an object array used to ensure that objects referenced by untraced structures (such as the TIB and IMT) survive garbage collection.

3.3.4.4 Final Classes

Methods declared in a final class usually cannot be overridden and the optimising compiler in the Jikes RVM takes advantage of this by inlining the method call. However, method specialisation allows a final method to be specialised and so inlining the method call would result in the specialised variant never being executed.

The specialisation framework handles the general case of method inlining to ensure that the inlining of specialised methods is guarded. This checks whether the current TIB of the receiver matches the inlined method version, if not then a non-inlined call is made. However, for Jikes 3.1 this happens only for methods that are declared final and not methods within final classes. The implementation in Jikes 3.0.1 does handle final classes correctly, indicating that this issue results from changes to the optimising compiler between Jikes 3.0.1 and Jikes 3.1.

The optimising compiler uses symbolic method references during code generation. These symbolic references can be marked *precise*, meaning that there is only a single possible method that can be called. Precise methods can therefore be inlined. The existing inline guard placed on methods by the specialisation framework only affects methods that are not marked precise. A method is marked precise if the class in which it is declared is marked final, although it is not marked precise if the method itself is final.

Therefore a method in a final class will be marked precise and not have a specialised inline guard added, resulting in any specialised variant not being executed.

This issue is easily resolved by marking a method as precise only if the class is final *and* the method has no specialised variants.

3.4 'Free' Read Barrier

Any issues pertaining to the Jikes RVM read barriers do not have an effect on the free read barrier implementation as self scavenging is used in place of explicit read barriers. However, additional problems arise through the use of method specialisation across all classes (including internal classes).

To complete a transition from Baker conditional read barriers to implicit stateful behaviour via method specialisation, several variants were implemented in order to simplify the task of debugging. First, the conditional read barrier was replaced by an explicit conditional self-scavenging block appended to the front of every method call. After this, the same explicit self-scavenging block was appended to a specialised version of each method and the specialisation of objects flipped when an object is first traced and after it self-scavenges. Finally, the explicit self-scavenging block was removed from the unspecialised method variant and the specialised variant is switched to execute unconditional self-scavenging.

3.4.1 Cross-Space Write Barriers

The free read barrier replaces the Baker style read barrier, however the incremental mark-sweep Yuasa snapshot-at-the-beginning write barrier is still required to manage the mark-sweep spaces. It is possible to replace the write barriers with a method transform, however it will essentially have the same job as the existing barrier since it must remain active for the entire garbage collection closure phase and cannot be removed by specialisation.

3.4.2 Global Beanification

In order to implement a free read barrier successfully every class needs to be beanified using the beanification transform. The corrected beanification transform supports this (see section 3.3.3).

3.4.3 Transforming Methods

The method transformation framework developed in [22] was used to create a transform that adds a method call to a scavenging method for every non-static and non-initialising method of every class. We do not add scavenging code to static methods as they are not

```
1  a.someCall()
2  -> MemoryManager.scavenge(a)
3     -> objectType = ObjectModel.getType(a)
4     -> objectType.getReferenceOffsets()
5        -> MemoryManager.scavenge(objectType)
6           -> rvmTypeType = ObjectModel.getType(objectType)
7           -> rvmTypeType.getReferenceOffsets()
8              -> MemoryManager.scavenge(rvmTypeType)
9                 -> rvmTypeType' = ObjectModel.getType(rvmTypeType)
10                -> rvmTypeType'.getReferenceOffsets()
11                   -> MemoryManager.scavenge(rvmTypeType')
12                   ...
```

Listing 3.10: The infinite callgraph for self-scavenging methods when applying self-scavenging to every method of every class.

associated with any object to scavenge. We also ignore initialisation methods since they are executed on object creation and new objects will always be allocated into to-space (or marked).

The transform cannot be applied to methods that are used during scavenging since an infinite loop will arise. Figure 3.6 illustrates an example of type representation for an object. Each object header contains a pointer to a type information block (*TIB*), which includes a pointer to an `RVMType` object that represents the type. For scavenging an object the associated `RVMType` instance is used to retrieve the reference field offsets for the object. If we include self-scavenging code on `RVMType.getReferenceOffsets()` then we will end up with an infinite trace as in Listing 3.10. Fortunately all methods on the self-scavenging call graph are either static, belong to immortal objects (also not self-scavenging) or do not access reference-type fields.

Detecting which methods/classes should not have the self-scavenging transform applied to them is a process of investigating the call graph for self-scavenging. When the transform is being applied to methods that are used during self-scavenging the RVM throws a segmentation fault. While there are many causes of segmentation faults, without changing the RVM in any other way, the introduction of self-scavenging can easily lead to infinite loops early in execution (before exception handling is initiated). Errors during garbage collection suffer a further setback, the debugging thread is unable to dump the stacks of collector threads, therefore if a collector thread gets stuck it's hard to find where and why (an issue similar to that detailed in Section 3.2.7).

3.4.4 Explicit Self-Scavenging

Before implementing a full-blown free read barrier the collector used explicit tests to determine whether it should self-scavenge or not; this code existed in a static method

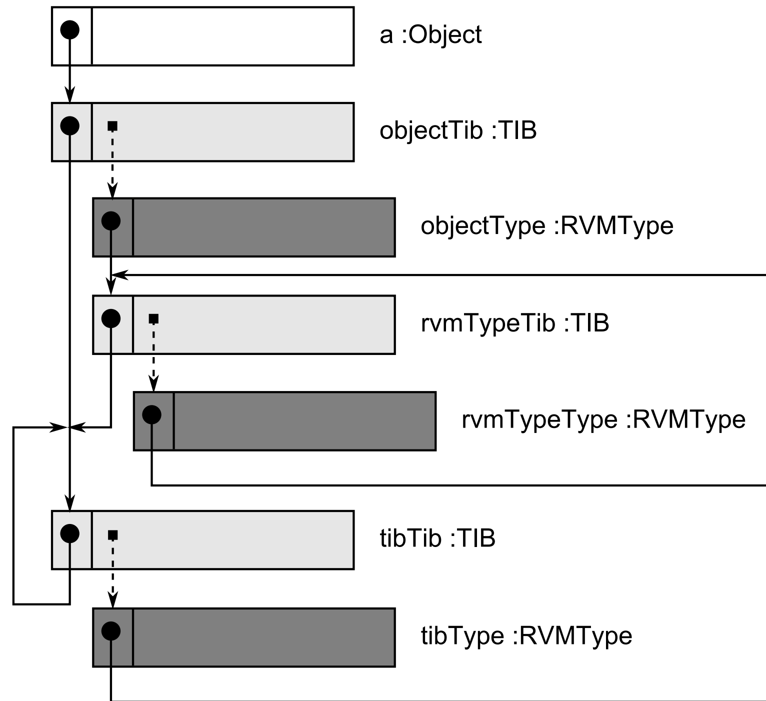


Figure 3.6: Example of object type representation in the Jikes RVM. Each object header contains a pointer to a type information block (TIB) that includes a pointer to an `RVMType` object representing the type.

called by every method of every class.

3.4.4.1 First Implementation

An additional two bits were assigned in the header of each object to record the scavenging state (scavenged, unscavenged and scavenging). Initially all objects are in the scavenged state.

When an object is first traced its scavenge state is set to unscavenged. The first time the object is activated (i.e. through a method call) the self-scavenging code executes, which starts the scavenging process. Any other threads that execute the self-scavenging code perform a busy-wait on the scavenge status to prevent multiple scavenges of the same object. Self-scavenging ensures that any accessed object is in to-space (the to-space invariant). We can thus safely ignore introduced setter methods, since they never read and overwrite a value that will no longer be accessible from this object. The write-barrier still exists over all writes as part of the Yuasa-style scheme. Furthermore we can safely ignore static methods as they cannot perform non-static field reads. Ideally we would ignore any methods that do not read a field that is potentially an object reference, since whether the fields are scavenged or not does not affect execution. It is possible to write such a transform, however this would introduce considerable overhead. Each method would need to be examined for the presence of an instruction `ALOAD 0` followed by `INVOKEVIRTUAL _getv_⟨className⟩!⟨fieldName⟩`. Furthermore the type of the `className.fieldName` field would need to be looked up in order to determine whether it is a non-primitive type.

Scavenging involves tracing all of the objects directly referenced by the current object. Jikes handles this through a class called `SpecializedScanMethod` where the notion of scanning is synonymous to scavenging. A special scanning mechanism is used for objects with six or fewer fields[5], where a pattern is matched against the object structure. For instance the pattern `NNRRNN` represents an object whose third and fourth fields are reference fields and should be traced. Since fields have fixed offsets from the start of an object any object matching one of the predefined patterns is scanned using constant offsets. Objects that match no pattern are handled generically with an inspection of the object reference offsets using the class definition. This generic approach is used to implement self-scavenging (see Listing 3.11). As there is a specific contract for interaction with the `SpecializedScanMethod` it is not possible to reuse it in the self-scavenging context. Once all referenced objects have been traced the owner's scavenge state is changed to scavenged and the thread continues as normal.

Two bits in the object header to indicate scavenge state is not sufficient. We cannot reset the scavenge state for the next collection. Once an object is scavenged it will always believe it is scavenged, since there is no way of resetting the scavenge state of all objects at the start of collection.

[5]Six is chosen as it is considered the point at which the trade-off between scanning execution time and space requirements for specialised scanning data-structures is optimal.

```
 1 public static void scavenge(ObjectReference objectRef){
 2   if( booted && needsObjectReferenceSelfScavenge()
 3             && isBarrierActive() ){
 4     Word scavengeWord = ScavengeWord.attemptToScavenge( objectRef );
 5     /* Spin-wait if we are not the scavenger */
 6     if( ScavengeWord.stateIsScavengedOrBeingScavenged( scavengeWord ) )
         {
 7       ScavengeWord.spinAndWaitForScavenge( objectRef );
 8     }else{
 9       RVMType type = ObjectModel.getObjectType( objectRef.toObject() );
10       /* Only class-type objects get here, arrays have no methods */
11       RVMClass klass = type.asClass();
12       /* Get the offsets for all reference-type fields */
13       int[] offsets = klass.getReferenceOffsets();
14       for(int i=0; i < offsets.length; i++) {
15         /* Tell the current mutator context to forward each field */
16         Selected.Mutator.get().processEdge(
17             objectRef,
18             objectRef.toAddress().plus( offsets[i] ) );
19       }
20       /* Notify any waiting threads the object is scavenged */
21       ScavengeWord.setScavenged(objectRef);
22     }
23   }
24 }
```

Listing 3.11: The code used to scavenge an object when no pattern can be applied. Used to self-scavenge objects.

3.4.4.2 Revised Implementation

An extra word overhead is added to each object to record the scavenge state so that an object can store scavenge state on a per-collection basis. Four bits of the word are used to encode the scavenge state, two bits are used for each alternating cycle. Naming each cycle as odd or even, with the first cycle being even, a cycle is named the opposite of the previous cycle. When an object is scavenged in an even cycle it resets the odd cycle state and vice versa. Therefore an object knows whether it has been scavenged in the current collection cycle since, for any cycle, the bits representing the scavenge state would have been reset by the previous cycle. Ideally this would be limited to only four bits. However the Jikes RVM makes use of header words since memory layout will be word-aligned.

While it is not strictly necessary to indicate self-scavenge state, i.e. there's no harm in self-scavenging multiple times as scavenging is idempotent, such redundancy would result in a considerable performance hit. The introduction of the tri-state mechanism reduces this overhead, as an object will not need to loop through all of its reference fields after it is scavenged.

Enabling self-scavenging code on every method causes additional issues when collection is started mid-method execution. To initiate collection correctly roots cannot simply be evacuated, but must also be scavenged. To see why this is so, consider the case where collection is started mid-method execution in which the next instructions to be executed are `ALOAD 0, GETFIELD <field_id>`. Evacuation of the roots will only forward the object reference corresponding to `this` and not the field that is about to be directly accessed. Without a read barrier in place the `GETFIELD` instruction will result in an object reference being returned that is possibly in from-space. By scavenging all roots, not only the object reference for `this` is forwarded, but also all fields of `this`, hence the `GETFIELD` instruction will return a reference to the forwarded to-space object.

3.4.4.3 Lazy vs Eager Self-Scavenging

Initially, to avoid scavenging the roots only the generated getter methods have scavenging code added to them. This leads to a *lazy* scavenging approach, where objects only self-scavenge on the first attempt to access a field, if an object has no fields, or if methods are called that do not access fields, then the object will not be self-scavenged. In fact we only need to self-scavenge on fields that are reference-type fields and whose value may reside in from-space. Many internal classes are declared as non-moving and these will reside in a mark-sweep space that do not need to be forwarded and hence any getter methods that access non-moving fields also need not have self-scavenging code applied.

The alternative to lazy scavenging is *eager* scavenging, where all methods have self-scavenging code and so even if a field is not accessed they will still be scavenged.

To gain an advantage from only self-scavenging on getter methods the methods must be declared *uninterruptible*, since we do not want collection to start after the scavenging

code has been executed but before the field is actually read.

There are several benefits to lazily self scavenging that make it the preferred approach to eager scavenging, these include:

- Easier Debugging - The introduction of self-scavenging code can have adverse affects to methods of particular classes due to the actions of the optimising compiler. By using lazy self-scavenging the only method that are affected are the generated getter methods, limiting the scope of any issues only to those methods. This makes it far easier to pinpoint which field of which class is causing the problem, since any getter method only deals with a single field.

- Less Code Bloat - By only self-scavenging on a subset of methods there are fewer methods to modify and specialise. This reduces code bloat and class loading time (when compared to applying self-scavenging to all methods).

- Improved Efficiency - Although lazy self-scavenging requires private field access also to be beanified, there are many methods that associate no overhead to lazy self-scavenging. Any method that does not access an object's fields does not invoke any getter method and hence does not execute any self-scavenging code. In an eager scavenger these methods would still have self-scavenging code, despite no fields being accessed.

Lazy self-scavenging also has drawbacks when compared to eager scavenging:

- Longer average incremental pauses - More work is left up to the collector, since an object may contain fields that need forwarding, but no method that accesses a field is executed before the collector processes it on its queue. This can extend incremental pause times, since more object forwarding will be required.

- Increased class loading cost - The conditional application of self-scavenging code makes the self-scavenging transform more complex and expensive, as each method needs to be examined to determine whether self-scavenging is required. This can be ignored in long-running applications since class loading only happens once per class.

- Increased beanification cost - Lazy self-scavenging requires access to private fields to also be beanified, incurring the penalty of virtual method invocation on every field access, not just external accesses. This is because a method that accesses a private field will not have scavenging code, the field may still be in from-space. Without beanifying the private field access the scavenging code will not be called before the field is read. This breaks the to-space invariant.

- Harder Debugging! - The lazy scavenging of objects leaves the runtime more open to objects being accessed without going via a self-scavenging method call and hence

leading to cascading unscavenged objects and violation of the tri-colour invariant. The Jikes RVM is a complex application with many alternatives for invoking methods and complex optimisations. Lazy scavenging will only identify issues if the beanified getter code is executed. An eager self-scavenger will be able to fail-fast on any assertions as they are applied across all methods. A fail-fast architecture makes debugging easier, as virtual machine information will be closer to the origin of the problem.

3.4.4.4 Early Scavenging

The explicit self-scavenger is applied to as many classes as possible, including internal VM classes. This means that virtual machine initialisation and set-up code calls self-scavenging getter methods before the virtual machine has booted all necessary classes to support self-scavenging. This leads to faults extremely early in the VM booting process. To prevent this from happening the first condition on self-scavenging code is that the VM has fully booted. This is safe since during booting the VM does not start a collection, however it becomes an unnecessary check after the VM has fully booted (as it will never cease to be fully booted).

An implicit self-scavenging scheme does not have to consider boot-time checks, since the self-scavenging code will only be activated once the first collection has started, at which point the VM is already fully booted.

3.4.4.5 Problem Classes

Some internal RVM classes were unable to self-scavenge on every method causing segmentation faults to arise with no useful error information. This is a very big issue for a self-scavenging collector, since it relies on every object being able to self-scavenge. If there is an object that does not self-scavenge then its children may not be forwarded when required, allowing access to from-space copies. Fortunately many of the objects that cannot be self-scavenged have only non-reference type children or immortal/non-moving children that will eventually be traced by the collector. To what extent these excluded classes affect the correctness of the collector is unknown, however it is highly likely that they are a source of some missed objects.

An excellent example of such a problem is the adaptive optimisation system (AOS) within Jikes. Most classes within the AOS subsystem are complex and contain some fields that need to be self-scavenged. Manually identifying these fields is difficult and so a more complex transform has been developed that determines the type of field a getter is accessing; if this field is a reference type then it needs to have scavenging code added. This adds class loading overhead, but it minimises where self-scavenging code needs to be placed.

3.4.4.6 Array Handling

Arrays pose an interesting and specific problem. Since arrays have no methods and are accessed using direct address offsets how can they self-scavenge?

The simple answer is that they cannot. It would be infeasible to wrap all array access in method calls, since this would slow-down all array accesses. However we can instead force arrays to be scavenged when their owners are scavenged, or alternatively use a standard read barrier across `AALOAD` bytecodes.

Reference arrays that are also roots pose a further issue, since only the array will be evacuated. The references contained within the array may still point to from-space objects, this includes multi-dimensional arrays, which are implemented as arrays within arrays. Static references can be scavenged through the static getter methods generated by the beanification transform, but this does no handle roots that are already on the stack.

This problem means that lazily self-scavenging objects requires root scavenging and therefore loses one of its advantages over eagerly self-scavenging, which necessarily requires root scavenging.

3.4.5 Implicit Self-Scavenging

Implicit self scavenging occurs the first time an object's method is called while it is in its unscavenged state. It executes the same scavenging code as an explicit self-scavenger, except it is transparent with respect to standard method execution, i.e. the receiver will not know whether the scavenging code has been executed.

Instead of using bits in the header to record the scavenge state the current object specialisation encodes the state of the object. If it is unscavenged then the self-scavenging method variants will be activated, after scavenging the unmodified method variant will be active.

3.4.5.1 Flipping Specialisations

Specialisations are flipped when an object is forwarded and flipped back when it self-scavenges. This means that objects only execute scavenging code during collection when they have been forwarded and their fields have not. This results in a mechanism similar to a barrier, but much cheaper.

Flipping an object into its specialised self-scavenge state occurs when the object is traced by the collector. To ensure that this will only happen once per collection cycle, the object's specialisation is only flipped if it has not already been traced. For objects living in a copy `Space` the first trace forwards the object. Hence subsequent traces are identified because the object has already been forwarded. For objects living in a mark-sweep/mark-compact `Space` the first trace marks the object, subsequent traces are identified since the

object is already marked. These are the only types of `Space` that the MMTk uses in the Baker collector.

Flipping an object back into its original state occurs when the object is scavenged. An object may be scavenged more than once, if it is residing on the scavenge queue, but has already been self-scavenged. This is not an issue, since flipping an object into a state that it is already in has no effect. We must guarantee that an object always ends up in its original state before collection terminates and so it is only necessary to ensure that it will never flip into the unscavenged state after it has been scavenged. As it is only ever flipped into the unscavenged state once and this occurs before it is scavenged by the collector (since it is only added to the scavenge queue when first traced) we guarantee that the last state change before collection terminates will be from unscavenged to scavenged.

Listing 3.12 shows the code used to flip an object's specialisation. Flipping to a specialised variant requires the specialisation number for that variant. Flipping to the default specialisation always loads the primary TIB for the type of an object.

```
1  /* Perform the TIB switch */
2  public static void hijackTIB(Object ref, int tibSpecNum) {
3    TIB[] tibs = getTIB(ref).getTibArray();
4    /* Fetch the TIB that will now be active */
5    TIB newTIB = (TIB)((getTIB(ref).getTibArray())[tibSpecNum]);
6    /* Set the object's TIB */
7    setTIB(ref, newTIB);
8  }
9  /* Flip to self-scavenging specialisation */
10 public static void flipToSelfScavengingTIB(ObjectReference objectRef){
11   hijackTIBIfPossible(objectRef.toObject(), 1);
12 }
13 /* Flip to default specialisation */
14 public static void restoreTIB(ObjectReference objectRef){
15   hijackTIBIfPossible(objectRef.toObject(), PRIMARY_TIB_INDEX);
16 }
```

Listing 3.12: The code used to flip the specialisation for objects.

3.4.5.2 Specialised Self-Scavenge Transform

Developing a specialised transform is different from a normal transform. New methods are added for each specialised variant required on a per-method basis. When the original method is visited by the transformation framework the method variants are generated, these are empty method shells.

For self-scavenging specialised methods the original methods are left untouched and a single specialised variant is generated. Listing 3.13 details the code for the self-scavenging transform, which generates the following code:

1. A call to the self-scavenging method, passing the receiver ('this') using `ALOAD 0`.
2. A further call to restore the receiver's TIB to its primary TIB, which points to the original unspecialised methods.
3. The construction of the method arguments to pass to the call to the original method. This needs to take into account the type of each argument so that correct operand stack offsets can be generated (since doubles and longs consume two operand stack entries).
4. A call to the original method, using the arguments constructed previously.
5. The return instruction, which must also be typed correctly according to the return type of the original method. This ensures that the result from the original method is cascaded to the caller.

```
1 public class SelfScavengeTransform extends Transform {
2   //Returns true iff the method should not be specialised
3   public boolean isExempt(String cname, String mname){
4       //we ignore immortal classes, treated as roots
5       return cname.contains( "org/mmtk" ) || ...;
6
7   }
8   //Returns a bitmap indicating how many
9   //specialised variants should be generated.
10  public int getSpecialisationMap(String cname,
11                                  String mname,
12                                  String mdesc, int access) {
13    //Static and native methods cannot be specialised
14    //Also don't specialise any exempt methods
15    if (    (access & Opcodes.ACC_STATIC) == 0
16         && (access & Opcodes.ACC_NATIVE) == 0
17         && !isExempt(cname, mname) )
18      return addToBitmap( 1, 0 );
19    else
20      return 0;
21  }
22  //Visiting the original method, which we want to keep
23  public int visitMethodStart(String className, String methodName,
24                              String methodDesc, MethodVisitor
                                    methodVisitor){
25      return APPEND_ORIGINAL_METHOD;
26  }
27
28  //Visiting a specialisation
29  //Since we only have a single specialisation we don't consider the id
30  public void visitSpecialisation(String cname,
```

```
31                                   String mname, String mdesc,
32                                   MethodVisitor specMethodVisitor,
33                                   int id) {
34      /* Invoke the scavenger method */
35      specMethodVisitor.visitVarInsn( Opcodes.ALOAD, 0 );
36      specMethodVisitor.visitMethodInsn(
37          Opcodes.INVOKESTATIC,
38          Type.getInternalName( MemoryManager.class ),
39          "scavenge",
40          Type.getMethodDescriptor(
41              Type.VOID_TYPE,
42              new Type[]{ Type.getType( ObjectReference.class ) } ) );
43      /* Restore the object's TIB */
44      restoreObject( specMethodVisitor );
45      /* Load <this> reference */
46      specMethodVisitor.visitVarInsn( Opcodes.ALOAD, 0 );
47      /* Construct the call to the non-specialised version */
48      callHelper( cname, mname, mdesc, specMethodVisitor );
49
50      /* Return the result from the original method */
51      specMethodVisitor.visitInsn( Type.getReturnType( mdesc )
52                                     .getOpcode( Opcodes.IRETURN ) );
53    }
54
55    //Construct a method call, taking into account argument types
56    private void callHelper(String cname,
57                            String mname, String mdesc,
58                            MethodVisitor mv){
59      //Construct loading of arguments to stack
60      Type[] argTypes = Type.getArgumentTypes( mdesc );
61      int opcode;
62      for ( int i = 0, opstackpos = 1;
63            i < argTypes.length;
64            ++i, ++opstackpos ) {
65        opcode = argTypes[i].getOpcode( Opcodes.ILOAD );
66        mv.visitVarInsn( opcode, opstackpos );
67        //Add an additional local offset for each double operand stack
              entry
68        if( opcode == Opcodes.DLOAD || opcode == Opcodes.LLOAD ){
69            opstackpos++;
70        }
71      }
72      //Make the call
73      mv.visitMethodInsn( Opcodes.INVOKEVIRTUAL, cname, mname, mdesc );
74    }
75  }
```

Listing 3.13: The method specialised self-scavenging transform used to generate self-scavenging method variants.

3.4.5.3 Interacting With Collection

The use of method specialisation to encode object scavenge state requires that the state is kept up-to-date with respect to collector progress. For instance, all objects immediately accessible from the roots should be flipped into an unscavenged state at the start of collection. This can be achieved through flipping specialisations when they are added to the scavenge queue. However, there are many intricacies concerned with the processing of the heap live graph and problems encountered with object state not being maintained caused several issues during development. There are three main possible causes for objects not correctly self scavenging: (1) objects are not flipped into the correct specialisation before they are first accessed, (2) there exists some unspecialised method that accesses a field and (3) methods are being invoked for the wrong specialised variant.

Incorrectly Flipping Specialisations The flipping of specialisations occurs when an object is added to the scavenge queue, by virtue of the collection algorithm all roots are flipped at the start of collection. Since the only accessible objects are from the roots, when the roots are first scavenged all of their reference fields are added to the scavenge queue and therefore flipped. This should guarantee that the current reachable set of objects at a program point are made up of already scavenged objects and those that are on the scavenge queue (and are therefore in the unscavenged state).

Missing Specialisations Lazy self-scavenging only requires the specialisation of generated getters for all reference type fields. Since explicit self-scavenging also had this requirement and works with the same set of self-scavenging methods this is unlikely to be a problem for the lazy scavenger. However, the eager self-scavenger needs specialisation to be applied across every method of every class. The specialisation framework has been modified to output which methods are being specialised and has been used to confirm that specialisation is being applied to all methods.

Invoking Incorrect Variants The method specialisation framework has not been exhaustively tested. There were some invocation paths that did not correctly identify the method variant that should be executed. Even if all required methods were specialised, if invoking the method does not consult the currently active TIB then the specialised method variant will not be executed. The specialisation framework only searches for the correct variant if it is invoked via the dynamic linker i.e. a normal lazily invoked method. However, if the method is called using reflection the TIB is not consulted and the primary TIB is used to find the method's code. In this case the specialised variant will not be called. The generated getter methods will all be invoked using the dynamic linker, since they are invisible to any developer no reflection code could pre-exist to call these methods, hence it should not be an issue for a lazy self-scavenging scheme.

Issues encountered with the method specialisation framework invoking incorrect method variants are discussed in Section 3.3.4. These issues were all uncovered during the implementation of the 'free' read barrier using two method transforms. The first transform is identical to the specialised self-scavenge transform (Listing 3.13). Listing 3.14 shows the second transform which is applied to all original method variants. The `TIBCheck` method (Listing 3.15) determines whether the currently active TIB index of the object matches the expected TIB index. If the TIB check fails then the VM is shut-down, which dumps the stack of the executing thread. The stack dump is used to determine which method failed. By comparing the results of several stack dumps from different applications the common elements between the failed methods can be identified (such as implementing an interface).

```
public class TIBCheckTransform ... {
  ...
  /* MethodAdapter visitCode method
   * Executed when a method's code is visited */
  public void visitCode(){
    /* Default method should be in the primary TIB (index 0)
     * All methods are specialised, so will only exist in primary TIB
     */
    visitInsn(Opcodes.ICONST_0);
    /* Load 'this' as the receiver */
    visitVarInsn(Opcodes.ALOAD, 0);
    /* Call checkTIB method */
    visitMethodInsn( Opcodes.INVOKESTATIC,
                     Type.getInternalName( MemoryManager.class ),
                     "checkTIB",
                     Type.getMethodDescriptor(
                       Type.VOID_TYPE,
                       new Type[]{
                         Type.INT_TYPE,
                         Type.getType( ObjectReference.class ) }
                       ) );
    /* Generate the existing method code */
    super.visitCode();
  }
}
```

Listing 3.14: Part of the transform used to check that the correct method is being executed for an object's currently active TIB.

```
public final class MemoryManager ... {
  ...
  public static void checkTIB(int specNo, ObjectReference object){
    if( booted ){
      TIB tib = ObjectModel.getTIB(object);
      if( tib != null && specNo != tib.getTibIndex()[0] ){
        RVMType type = tib.getType();
        if( type.isArrayType() )
          return;
        VM.sysFail("Checking method TIB failed!");
      }
    }
  }
}
```

Listing 3.15: Method used to check the expected TIB index against the receiver's actual TIB index.

3.4.6 Issues

While explicit self-scavenging executes correctly across most benchmarks the implicit self-scavenger does not. One of the associated problems with the implicit self-scavenger is the interaction of method specialisation with the core Jikes RVM classes, specifically the optimising compiler. To reduce the application of self-scavenging and avoid the complications of self-scavenging the virtual machine, the allocation profile is modified so that all RVM objects are allocated into immortal space. This leaves only application-level objects that need to self-scavenging.

By immortalising RVM objects self-scavenging requires a modification to the definition of the root set. Since all immortal objects are not self-scavenging they must be considered roots. This means that the immortal space must be scanned and every object in immortal space must be scavenged. To achieve this a linear scan of the immortal space is added to the root scanning phase. The MMTk already contains utility methods to achieve this that require the implementation of the abstract `LinearScan` class. The `MutatorContext`s must perform the linear scan based on the thread-local bump-pointers that are used to allocate into immortal space. The scan itself involves searching the locally allocated pages for object references. Since immortal space is allocated with a bump pointer scanning uses a *cursor* address that starts from the first memory address of a page (which will contain an object reference). Subsequent objects are found by incrementing the cursor by the size of the current object. This process retraces the action of the bump pointer. The last object is identified as the next reference will evaluate to null. For every object that is found a call is made to a delegate `LinearScan` instance. The implementation of the `LinearScan` is very simple, adding any object references found in the scan to a local root object queue. The root object queue is flushed to the collector's shared root queue, which already exists to process roots found during the root scanning phase.

Currently the implicit self-scavenger is not stable. A couple of benchmarks are able to run for a handful of iterations, but fail before useful results can be obtained. The issues with the collector are related to those discovered in the specialisation framework and the inability to apply specialisation across the entire virtual machine. This is a specific problem with a meta-circular machine. A runtime that is not meta-circular would not have to apply specialisation or self-scavenging to itself, since memory would be managed either explicitly or via another runtime environment.

Development is still in progress to reduce the number of immortalised RVM objects and fix the remaining specialisation issues. Useful results will be obtainable if the number of immortalised objects are reduced. Fixing the specialisation issues will allow for more benchmarks to run successfully and provide a wider range of data for evaluation.

4 Evaluation

All benchmarking results were obtained on an Intel Petium D 3.40 GHz dual core system with a 2048 KB L2 cache and 2GB RAM. The operating system is a 32-bit build of Ubuntu 9.10 (Karmic) running a standard 2.6.31-14-generic kernel.

4.1 Beanification

To analyse the cost of beanifying all classes rather than only application classes the benchmarks are executed against the updated beanification transform applied across all classes and the subset used in [22]. We consider the *converged* execution overhead obtained from running each benchmark 5 distinct times for 50 iterations in the same instance of the RVM and taking the average from the last 10 iterations of each. Table 4.1 and Figure 4.1 show the *converged* execution overhead of the method specialisation framework (NB), full beanification (FB) and application class beanification (AB) against the vanilla Jikes RVM 3.1 (O). There is a clear increase in overhead by beanifying all classes (8.4% geo. mean), since any use of in-built java types such as `String`s and `Collection`s are now beanified. While the overheads are slightly higher than originally expected they are acceptable. The results obtained from application class beanification show a geometric average of 5%, correlating with those found in [22] (where MBS denotes the equivalent to AB), despite the overhead appearing to be ~3% lower. As reasoned in [22] the use of name mangling allows the beanified methods to be declared as final and inlined by the optimising compiler, reducing the overhead after runtime compilation.

In depth tests were performed that analyse the execution of consecutive benchmark iterations within the same RVM instance (executed with the command `./rvm -jar dacapo.jar -n <iterations> <benchmark>`). Figures 4.2a-4.2d show graphs detailing the results of execution of full beanification (FB) against the vanilla RVM (O). All of the benchmarks exhibit the same pattern of execution, with the execution times converging from the initial exaggerated iteration. The addition of class hierarchy exploration indicates that the new beanification process is similar in cost to the original, however this is most likely due to the use of Jikes RVM 3.1 rather than 3.0.1. As an example, the vanilla RVM executes antlr in 3.18 seconds, while FB executes in 4.20 seconds, a 32% overhead (see Figure 4.2a). While still undesirable, there is no easy way of rectifying this without incorporating beanification as a permanent fixture.

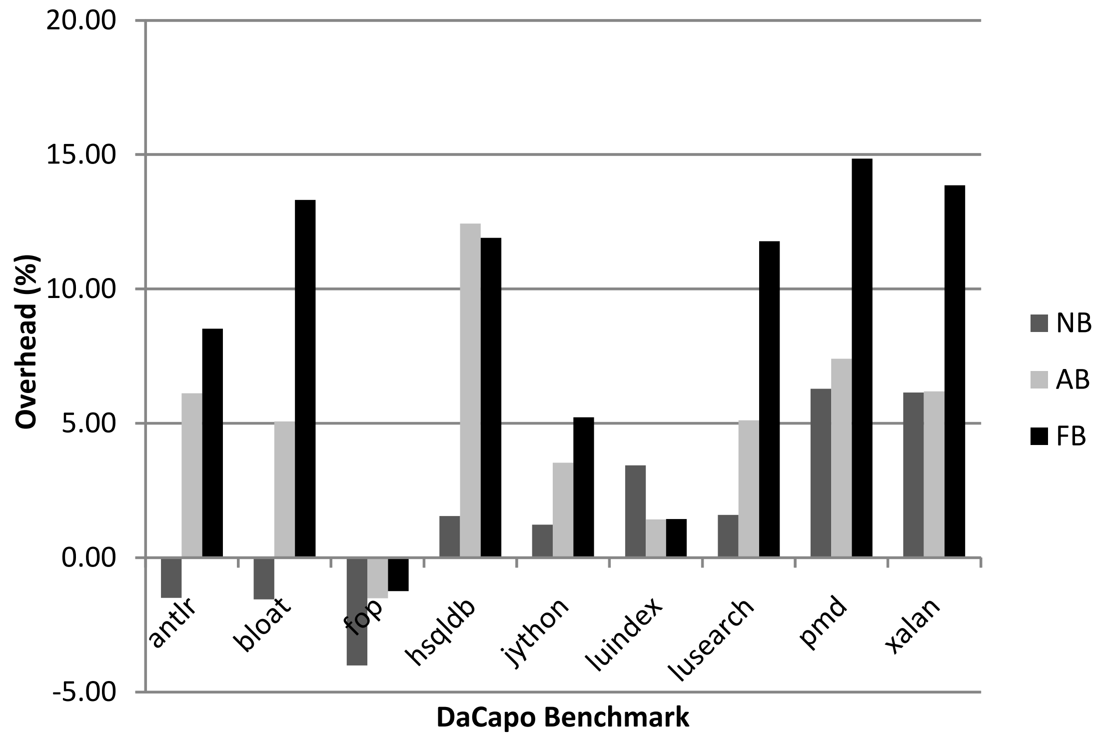


Figure 4.1: Graph showing the converged execution overhead for DaCapo benchmarks of beanifying all classes (FB), beanifying only application classes (AB) and beanifying no classes (NB) measured against the vanilla Jikes RVM 3.1 running the GenImmix collector.

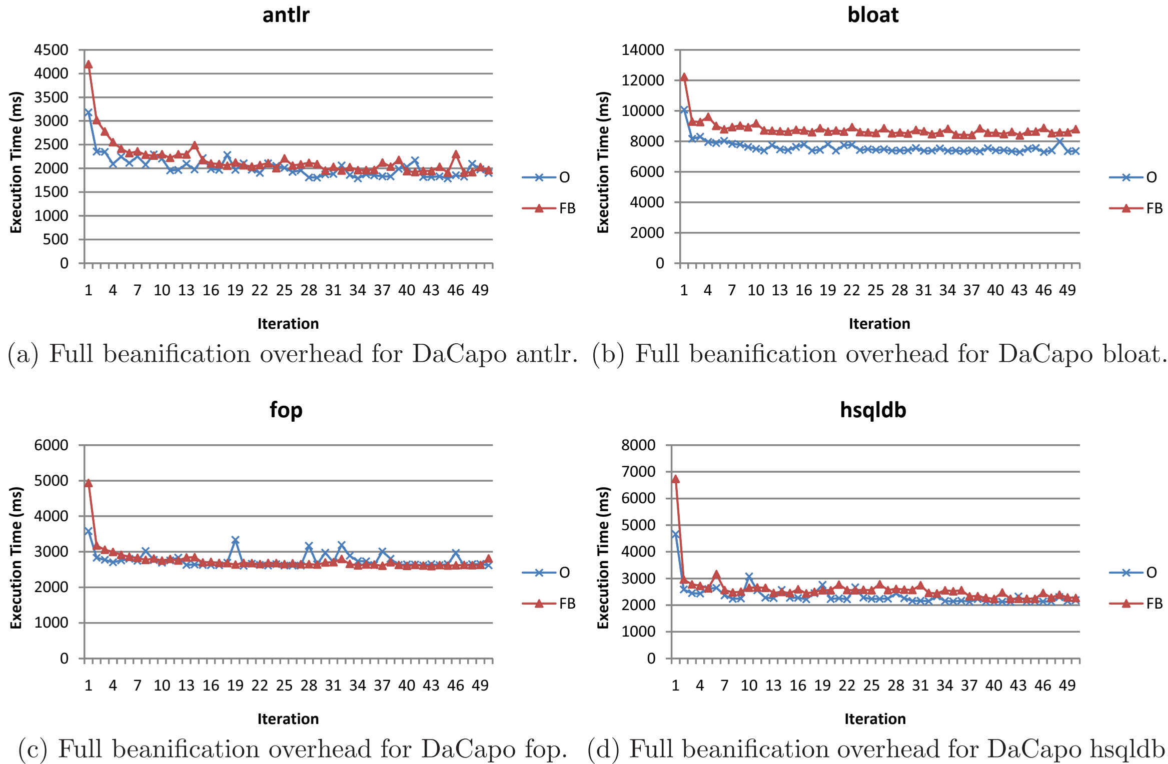
Benchmark	O (total ms)	% Overhead		
		FB	AB	NB
antlr	1797	8.5	6.1	(-1.5)
bloat	7466	13.3	5.1	(-1.6)
fop	2641	(-1.2)	(-1.5)	(-4.0)
hsqldb	2075	11.9	12.4	1.5
jython	5030	5.2	3.5	1.2
luindex	9916	1.4	1.4	7.1
lusearch	4600	11.8	5.1	1.6
pmd	4471	14.9	7.4	6.3
xalan	5571	13.9	6.2	6.1
Min	n/a	(-1.2)	(-1.5)	(-4.0)
Max	n/a	14.9	12.4	7.1
Geo. Mean	n/a	8.4	5.0	3.1
Ari. Mean	n/a	8.9	5.1	1.9

Table 4.1: Table showing the comparison of converged execution time, in terms of the vanilla RVM 3.1 with GenImmix (O), across the DaCapo benchmarks for the beanification of all classes (FB), beanification of application classes (AB) and the method specialisation framework with no beanification (NB).

4.2 Incremental Baker Collector

To analyse the performance of the incremental Baker collector benchmarks were run against the existing semi-space stop-the-world collector on which the Baker collector was based. Table 4.2 shows the overhead of the Baker collector (SSInc) against the semi-space stop-the-world collector (SS) for the DaCapo benchmark suite. While the results indicate varying overheads of 16-73% the geometric mean of 41.2% (arithmetic mean of 44.3%) is towards the top-end of expected overhead. The detailed analysis of the time spent in garbage collection (see Table 4.3) shows that most of the overhead is attributable to the barriers, as these are logged in mutator execution. The average ratio of mutator execution to collector execution tells us how much time is spent in the mutator for every 1 ms of collector time. It is evident from the results that, although total execution time is greater, the proportion of time spent in the collector is notably lower. It does not indicate how much progress is made in the mutator however and so is merely useful as an indication of the barrier cost.

Determining how much time is actually spent in the barriers is difficult as the addition of measurement code will increase the cost of the barriers. A hardware-based timing mechanism would definitely be required, since the introduction of a software timing mechanism led to exacerbated readings.



(a) Full beanification overhead for DaCapo antlr. (b) Full beanification overhead for DaCapo bloat.

(c) Full beanification overhead for DaCapo fop. (d) Full beanification overhead for DaCapo hsqldb.

Figure 4.2: Graphs showing the difference in execution time between the vanilla RVM (O) and fully beanified RVM (FB) over 50 iterations for four of the DaCapo benchmarks.

The average overhead is higher than predicted, however predictions were based on a pure Baker collector. Due to the design of the RVM virtual memory a pure Baker implementation is not possible and write barriers were required to manage the mark-sweep spaces. This adds further overhead as the collector is more conservative than it needs to be. The snapshot-at-the-beginning Yuasa write barrier adds more objects to be scanned during incremental phases that may be managed differently with a pure Baker collector.

An interesting artefact in the multi-threaded benchmarks arises after around 20 iterations. The collector becomes constantly active and is unable to sufficiently keep ahead of mutator allocation. Figure 4.3 shows the issue for the jython benchmark. It is clear that this is a result of incorrectly calculating required collector work, however it is notable that this does not affect single-threaded benchmarks or the first iterations of multi-threaded benchmarks. Once the collector starts overworking it remains in this state. This artefact could be related to the remaining issues with the Baker collector, there may exist a race-condition within the incremental framework. The results detailing average overhead discount the runs after which multi-threaded benchmarks over-collect, since the initial

iterations illustrate correct collector operation.

Benchmark	SS (total ms)	SSInc (% Overhead)
antlr	2289	37.0
bloat	9648	60.7
fop	2892	36.8
hsqldb	5340	39.4
jython	8983	51.1
luindex	10834	34.1
lusearch	6844	16.3
pmd	5766	72.7
xalan	10118	50.4
Min	n/a	16.3
Max	n/a	72.7
Geo. Mean	n/a	41.2
Ari. Mean	n/a	44.3

Table 4.2: Table showing the comparison of converged execution time, in terms of the vanilla RVM 3.1 with SS (SS), across the DaCapo benchmarks for the incremental Baker collector (SSInc).

The results of the multi-threaded benchmarks (jython, lusearch, pmd and xalan) are based on partial completion i.e. they ran for 20-30 iterations. This means that they are likely to report higher overheads since there is less time for optimisations to occur. The reason that these benchmarks could not run to completion is due to their multi-threaded nature and the missing object issue detailed in Section 3.2.6. It is therefore likely that the real overhead is slightly less than the reported overhead.

Table 4.4 shows the statistics for individual mutator pause times for the incremental Baker collector over 24000 steps. The results indicate that the collector does produce small pause times (average 1.45 ms). However it also indicates some extreme fluctuations, as much as 80.04 ms, due to the initial root scanning phases. The low standard deviation of 2.31 ms is a clear indication that incremental steps are generally consistent. Figure 4.4 shows the distribution of the increments over the DaCapo antlr benchmark. There are clearly three garbage collection cycles starting at ~1450 ms, ~2850 ms and ~4000 ms into execution. Figure 4.5 shows a more detailed view of the cycle starting at ~1450 ms, and the results indicate that the work-based approach does cause significant bunching of incremental steps (average 2.5 ms gap between increments). This bunching leads to lots of short pauses in quick succession giving the impression of a longer pause, in general pause times were below 3 ms, except for the initial root-scanning phase which cause 30+ ms pauses. With an incremental stack scanning algorithm this situation could be improved. A time based incremental trigger would alleviate the bunching of steps significantly which

Benchmark	time in mu (ms)		time in gc (ms)	
	SS	SSInc	SS	SSInc
antlr	1713	2849	438	669
bloat	7310	13181	2340	2354
fop	2546	3448	0	148
hsqldb	1371	4535	3535	1862
jython	5037	11782	10094	3145
luindex	9913	13417	766	859
lusearch	5447	6773	1572	1336
pmd	4061	6521	1388	1495
xalan	5296	7992	5344	2802

Benchmark	mu/gc	
	SS	SSInc
Min	0.4	2.4
Max	inf	23.4
Geo. Mean	2.1	5.6
Ari. Mean	3.5	7.5

Table 4.3: Table showing the comparison of converged execution time for the stop-the-world semi-space collection (SS) and the incremental Baker collector (SSInc). From left to right the columns contain the name of the benchmark, the time spent in mutator execution in milliseconds for SS and SSInc and the time spent in garbage collector execution in milliseconds for SS and SSInc.

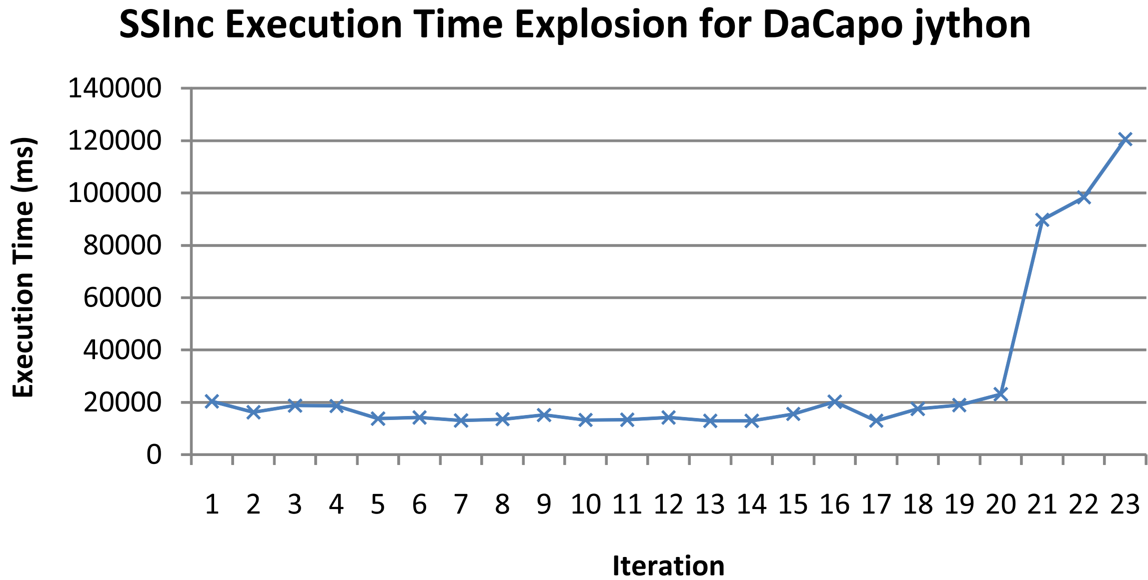


Figure 4.3: Graph showing the issue with multi-threaded benchmark execution time rapidly increasing for the jython DaCapo benchmark.

would likely result in reduced overhead.

Stat. (ms)	SSInc
Min	0.24
LQ	0.65
Med	0.78
UQ	1.65
Max	80.08
Ari. Mean	1.45
St. Dev.	2.31

Table 4.4: Table showing the statistics for incremental garbage collection pause times in milliseconds.

Figure 4.6 shows a comparison of the incremental Baker collector (SSInc) execution time against the semi-space stop the world collector (SS) for the DaCapo antlr benchmark. The graph shows erratic timings between consecutive runs for the incremental Baker collector. This may seem very strange, but when compared to a modified stop-the-world collector that has a permanently active read barrier (W) performing a much simpler operation we see exactly the same behaviour. The sudden jumps in execution time can be attributed to the read-barrier, even though it was active permanently the spikes are still present. These could be areas of intense field access, but since each run performs the same operations this seems unlikely. It appears as though when the collector is performing copying while the barriers are active the cost of the barriers explodes. For the incremental Baker collector small collections will be happening almost all of the time, which is likely the reason for the spikes being much smaller for the Baker collector.

Additionally the SPECjvm2008 benchmarks were executed in 5 separate VM instances for a single iteration with the default heap size of 50 MB. Some of the benchmarks were unable to run due to configuration issues with the benchmarking machine. The small heap size prevented the large data-set versions of the scimark benchmark from running, although Table 4.5 shows the results of these benchmarks with a heap size of 300 MB.

The SPECjvm2008 benchmarks used cover a wide range of scenarios, from heavy numerical calculations to image rendering:

- scimark - widely used as a floating point benchmark. There are small and large variations to test the JVM and memory management subsystem respectively.
- sunflow - tests graphical visualisation by rendering images using the open-source sunflow global illumination rendering system.
- mpegaudio - floating point heavy, tests mp3 decoding.

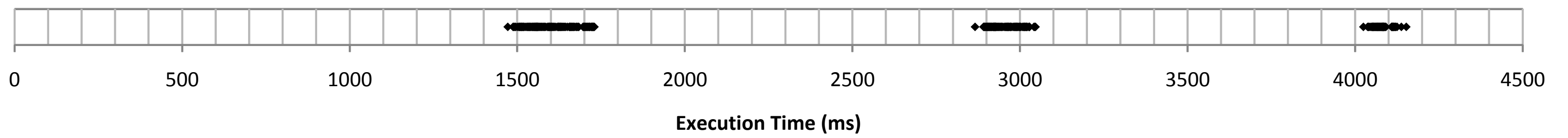


Figure 4.4: Graph showing the distribution of incremental steps over a single run of the antlr DaCapo benchmark.

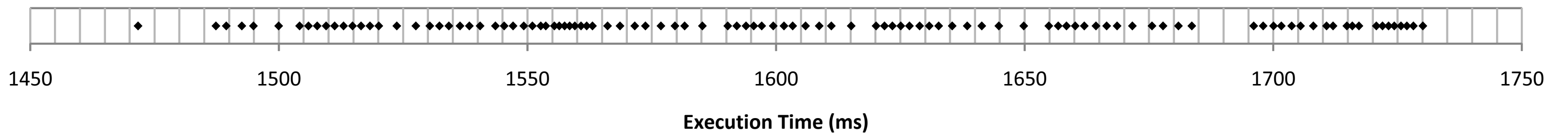


Figure 4.5: Graph showing a detailed view of the distribution of incremental steps for the collection at ~1450 ms into the execution of a single run of the antlr DaCapo benchmark (as seen in 4.4).

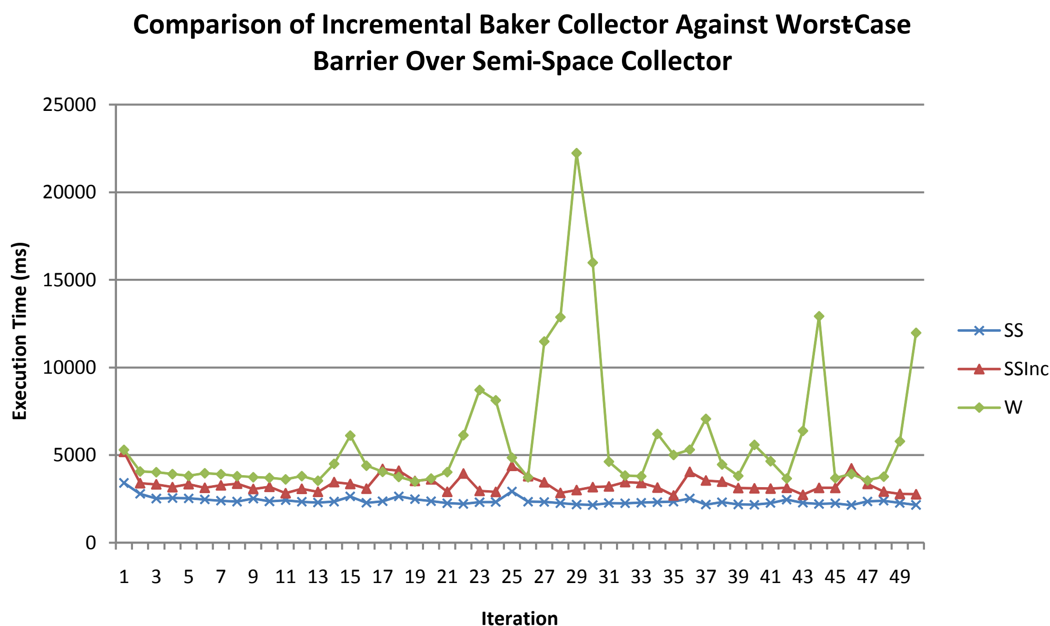


Figure 4.6: Graph showing the comparison of execution time for the incremental Baker collector (SSInc), the standard stop-the-world semi-space collector (SS) and a permanently barriered stop-the-world semi-space collector (W). The permanently barriered collector is used to determine whether the erratic differences in execution time for consecutive benchmarks is attributable to the barriers. It is clear from this comparison that this is the case.

Benchmark	SS (total ops/m)	SSInc (% Overhead)
scimark.fft.small	3.26	(-16.9)
sunflow	2.97	4.4
mpegaudio	6.66	44.3
compress	16.64	283.4
crypto.signverify	8.01	(-4.1)
scimark.fft.large	2.24	16.4
scimark.lu.large	1.57	6.9
scimark.monte_carlo	4.18	1.9
Min	1.57	(-16.9)
Max	16.64	283.5
Geo. Mean	n/a	15.1
Ari. Mean	n/a	42.0

Table 4.5: Table showing the comparison of operations per minute (ops/m), in terms of the vanilla RVM 3.1 with SS (SS), across the SPECjvm2008 benchmarks for the incremental Baker collector (SSInc).

- cryto.signverify - signs and verifies signatures using MD5, SHA1 and RSA with input sizes of 1 KB, 65 KB and 1 MB.
- compress - compresses data using a modified Lempel-Ziv method (LZW).

What is noticeable from the SPEC benchmarks is that slightly lower overheads than the DaCapo benchmarks are reported (geo. mean 15%). The DaCapo benchmarks are much shorter and therefore the results are less accurate at representing incremental collector benefits over application execution, since there is more variation in the number of collection cycles from run-to-run. This is easily observable in the DaCapo benchmarks as some iterations may take twice as long as others due to additional collection cycles. Because the SPEC benchmarks are longer running there is less variation in the number of collection cycles.

The exception to this is the compress benchmark which reports a ~280% overhead. The likely reason for this is that incremental collection is active for long periods in which many field access operations occur. This will push the barrier overhead much higher. However, the SPECjvm benchmarking suite is limited in its verbose output and the overhead could be caused by something more sinister, such as the overworking of the collector as seen in the DaCapo benchmarks.

The larger datasets are designed to test the memory management subsystem and these benchmarks tell a similar story to the DaCapo benchmarks. The scimark.*.large benchmarks use a dataset of 32 MB, whereas the better performing scimark.*.small benchmarks only use a dataset of 512 KB. The smaller datasets are designed to stress the JVM and perform heavy floating-point calculations. The use of the larger datasets causes garbage collection to play a more active role and hence reduces the number of operations per minute.

In general the number of operations per minute is very low and this does not seem correct when compared to results reported in other papers (specifically [22]). This could be for a number of reasons, the configuration of the benchmark could be different and the specification of the machine will also have an affect. Having said this the comparison between the incremental Baker and stop-the-world semi-space collector is still valid, as the benchmarking configuration was the same for both.

A couple of benchmarks actually reported better throughput (scimark.fft.small and crypto.signverify) for the incremental Baker collector than the stop-the-world semi-space collector. This could be due to a number of factors. The incremental nature of the collector will allow for operations to continue to execute while garbage collection is in progress, whereas the stop-the-world collector will not. These benchmarks are also more heavily focussed on floating-point calculations and therefore they may not perform as many object allocations. In benchmarks where object allocation and field read rates are low and the incremental collector is likely to perform better, since it may not complete every collection cycle. A stop-the-world collector *must* perform a complete collection.

4.3 Explicit Self-Scavenging

While no serious benchmarking results were taken for the intermediate explicit self-scavenging implementation the preliminary results obtained during development are worth mentioning. The unconditional explicit self-scavenging routine resulted in an unsurprising overhead exceeding 500% in some cases. This overhead is attributable to the very expensive scavenge barrier, which executes on every getter method execution. The scavenge routine includes three basic boolean checks as shown in Listing 4.1, which amount to three loads and three comparisons. Additionally the actual scavenging is executed, which in the worst case amounts to forwarding all reference type fields. In the common case (when the incremental closure is being generated) the scavenging will loop through all reference type fields and perform several comparisons to determine the forwarding state. Clearly in the case where the incremental closure is not in progress the cost will amount to the four state checks only.

The explicit self-scavenging routine with a state check has the same three basic checks as in listing 4.1 and a scavenge state check, which is a single load followed by a comparison. This leads to a worst case equivalent to the unconditional self-scavenger's worst case plus the additional scavenge state check. However, the common case will be only the four basic checks plus the scavenge state check, as in the case where the incremental closure is not in progress. This results in a reduced overhead ranging from between 80%-200%.

```
1 public static void scavenge(ObjectReference objectRef){
2   if( booted && needsObjectReferenceSelfScavenge()
3               && isBarrierActive() ){
4     /* Find and forward the reference-type fields for objectRef */
5   }
6 }
```

Listing 4.1: The code used for the explicit self-scavenging collector.

4.4 'Free' Read Barrier

The 'free' read barrier implementation for the SSB collector is not developed enough to yield meaningful results. A handful of benchmarks are able to run in a partially collected Jikes RVM, where some of the Jikes classes are stored in immortal space. However, the immortal space must be linearly scanned as root objects and this adds considerable overhead (500 ms per collection). Moreover the additional root objects lead to rapid heap growth and a higher base heap usage. This results in more collections being triggered and more frequently than the vanilla Jikes RVM. The combined initial overhead and increased number of collections makes the results incomparable.

5 Conclusion

The implementation of an incremental Baker collector in the Jikes RVM consumed far more time than expected, leaving little time to implement a 'free' read barrier. The problems encountered and those still left unresolved required a significant investment of time to debug and yet the solutions are relatively straightforward. It is the cost of debugging that makes implementing garbage collectors so time consuming, with better debugging tools this is not such an issue. The soft-scheduling problems encountered in Jikes 3.0.1 show clearly the difference in time required to find a solution between having no useful tools at your disposal and having just what you need. Without the debugging thread being available to provide meaningful thread dumps trial and error led to weeks of frustration and lack of progress, while as soon as the debugging thread could be used it took less than a day to resolve the issue.

This project has managed to produce the first stable incremental Baker collector in Jikes RVM 3.1 that can run over many benchmarks and produce useful results. It is not robust or efficient enough for a production release, but many of the roadblocks to building an incremental collector have been overcome. The results discussed in Chapter 4 are consistent with previously published overheads [43] and show just how expensive a Baker conditional read barrier can be, with overheads between 30%-70%. Without optimisations this overhead exceeded 100%.

It is significant that no incremental collector exists in the Jikes RVM as most other mainstream virtual machines contain at least one implementation, including Sun's Java HotSpot and many embedded JVMs. There have been several papers that report on barrier overheads without building a supporting collector. This is most likely due to the development cost of building an entire incremental framework to evaluate barriers. The results obtained from these implementations are usually collected after heavy optimisations and not a true reflection upon how a working implementation may function. In reality these are best-case analyses of collector performance, if everything worked perfectly.

The state of the 'free' read barrier is disappointing considering that the original project outline aimed to produce a complete working variation on the incremental Baker collector. Further unexpected issues with the method specialisation framework hindered development, but have led to a more robust implementation of specialisation within Jikes 3.1. There are two known issues still left to overcome in order to have a working SSB collector:

1. Specialising all internal RVM classes. The use of immortal space for internal classes that cannot be specialised prevents an efficient SSB collector from being developed. The optimising compiler does not function when method specialisation is applied

to it for an unknown reason. It is the application of specialisation that causes the issue and prevents the Jikes RVM from booting and is therefore likely the result of mangled code generation. Issues within the optimising compiler are very difficult to debug and without the virtual machine even booting there is little information to aid investigation.

2. Execution of incorrect method variants is still an issue despite the fixes detailed in Section 3.3.4. One source seems to be in the declaration of static member classes, however, according to the Java specification these are compiled as top-level classes and treated no differently. On closer inspection of optimising compiler output the failing methods are called in exactly the same way as any other. This indicates that the problem resides with the specialisation flipping *after* the TIB is loaded but *before* the method is called. The code pointed to be the loaded TIB will be incorrect. Unfortunately, to ensure that the loaded TIB is the same when the method is called would require combining the TIB load and method call in an uninterruptible region. Every method call would then require uninterruptibility and that will have dire consequences for the running of the RVM, reducing benefits for multi-threaded applications and internal instrumentation.

The issues with the specialisation of internal RVM classes are specific to the meta-circular nature of the Jikes RVM. The proposal of the Jikes researchers to segregate internal RVM and application-level memory would remove this requirement. Such a modification would be considerable and is still in the conception phase of development, an investigation and implementation of memory segregation in the Jikes RVM alone would be valuable to memory management research. Specifically, determining whether segregation is even possible and how it affects the performance of the virtual machine.

There are alternative Java virtual machine implementations, such as OpenJDK, which are not meta-circular that have well developed incremental collector support and could be an alternative for implementing a 'free' read barrier. However, none of these include a method specialisation framework that is an essential part of the project. A port of the specialisation framework would not be trivial, since each virtual machine implementation handles dynamic method invocation slightly differently. Despite the issues with the meta-circular nature of the RVM it still remains the most applicable virtual machine for this project, since alternatives would need considerable work to reach the same stage of development.

The use of method specialisation to implement a garbage collection barrier is not limited to Java. In fact the preceding work in [16, 18] is developed for Haskell and it can be applied to any language that employs dynamic despatch. The C# Bartok compiler and runtime environment for Microsoft .Net languages is a popularly used research platform for modern garbage collection. It may be advantageous to introduce method specialisation to the Bartok compiler as an alternative to Java as it appears to have more developed garbage collector support to leverage.

Despite the underdeveloped SSB collector this project has made considerable contributions to the Jikes RVM, including an incremental collection framework and stable incremental Baker collector implementation. The method specialisation framework has been ported to the latest version of Jikes and previously unknown issues have been discovered and fixed, specifically interface method invocation, superclass specialisations and final class declarations. Furthermore the beanification transform has been improved, applied and evaluated across the entire RVM. Additionally transforms have been developed to aid debugging, namely the read barrier and specialisation verifiers (see Sections 3.2.6.2 and 3.4.5.3 respectively). While the final SSB collector is not functioning to an evaluation standard there is an indication that there are only a couple more issues to overcome.

5.1 Future Work

5.1.1 Baker Collector Improvements

There is clearly plenty of work left to produce a stable Baker collector, the problems that remain are all difficult to debug and time consuming to investigate. Towards the latter stages of the project the Jikes RVM researchers were more active and conversed more regularly, providing helpful insight into possible issues. This really came too late in the project timeline to have a major impact, but it did lead to some late stability improvements. It is clear that the Jikes RVM is extremely complex, many of the permanent Jikes researchers are familiar with only a subset of the virtual machine with clear boundaries between MMTk developers and core developers. It is for this reason that it would not be surprising if edge cases were left unchecked for concurrent collectors, especially since nobody has yet to completely implement a copying concurrent collector.

The optimisation of the read barrier is a definite area worth investigating, specifically enabling safe barrier inlining (as discussed in Section 3.2.3).

As an extension to the Baker collector a time-based trigger could be implemented and compared to the work-based variant to better understand the effect of bunching requests in the Jikes RVM. This would need to start an incremental collection on a time slice. The Jikes RVM uses a timer thread to maintain a software based thread timer interrupt. By default every 10 ms the timer thread signals to all other threads that a quantum has passed. When a thread enters a yieldpoint (on a *backedge*, *epilogue* or *prologue*) it checks whether a quantum has passed, if so it records some adaptive optimisation statistics. It is at this point that the garbage collector could stop all mutator threads and execute for a fixed quantum, using the same timer interrupt to determine when to return to the mutators.

A crude implementation of a time-based collection trigger was implemented during this project using the timer thread as a basis. The timer interrupts are not particularly reliable for controlling collector execution, incremental pause times ranged from 10 ms

to 30 ms (or 1 to 3 quanta), which only provides soft-real-time bounds. The observed overhead from the experiment matched that of a work-based incremental collector. The pause times of the time-based collector were much higher than the work-based, but there were fewer and they were not bunched together.

5.1.2 'Free' Read Barrier

Further work needs to be concentrated on the 'free' read barrier implementation, which is lacking in both stability and coverage. The intricacies associated with method specialisation mean that more work is required to investigate the affect of specialisation on core RVM classes. From the preliminary work in this project it is clear that specialisation has adverse effects on the core operation of the virtual machine. This should hardly come as a surprise since the VM itself is implemented in Java, there is sure to be some issue with specialising classes that may have special meaning to the VM, that should be left untouched.

As discovered while implementing the explicit self-scavenging collector there are many classes and methods that can be safely excluded from self-scavenging and therefore method specialisation. Developing an efficient way to automatically determine whether specialisation and self-scavenging transforms need to be applied could improve stability and reduce overheads.

A further investigation into the comparison of lazy and eager self-scavenging would be an interesting avenue for research. Although there are theoretical benefits to both scavenging methods, reasoning about which approach would be practically more efficient is impossible without understanding their performance overheads over real benchmarks.

The state of the incremental Baker collector is sufficient to support the investigation into stabilising the Baker collector and continuing the implementation of the 'free' read barrier simultaneously. This would allow for much more efficient use of time, since there would be no separation of concerns during development as encountered with this project.

5.1.3 Path Specialising Collector

The work described in [36] defines an alternative use for method specialisation in garbage collection algorithms that would be very interesting to compare against the self-scavenging approach developed in this project. Since [36] is implemented in C# it would be necessary to either re-implement their work in Jikes or our work in C#. This would allow the analysis of the key differences between intuitive read barriers and self-scavenging. Where traditional barriers deal with each field access as a separate entity there is no way to easily determine if a barrier is required after it has been hit once. So the barriers have to remain active until they are no longer needed. Self-scavenging benefits from being able to switch of scavenging on a per-object basis after it has been first scavenged. However, self-scavenging needs to forward multiple objects when it is tripped, whereas traditional

barriers only need to forward a single object. Therefore a self-scavenging algorithm will be more efficient if the same fields are accessed many times, since a traditional barrier will be tripped on every access. If on the other hand fields are only read at most once within a collection then the self-scavenger loses, because it will have forwarded more objects than necessary.

This could be achieved through introducing a `NoBarrier` annotation and making the required modifications to the optimising compiler such that a method with a `NoBarrier` annotation has no barriers inserted. A transform could then be written that generates specialised variants of all methods, adding `NoBarrier` annotations to the original methods. This would maintain backwards compatibility with all other collectors, while adding the ability to selectively turn off barrier generation. Additionally the optimising compiler would need to be modified so that methods can jump to the same point in code in alternative specialisations. The yieldpoints used in Jikes to collect optimisation statistics and yield for garbage collection will also need to be hijacked so that just before the yieldpoint (a garbage collection safe-point) all specialised method code paths return to the unbarriered method. For the unbarriered methods all field accesses where barrier code would usually be placed need to have a conditional branch inserted that will jump to the specialised method variant for the active garbage collection phase.

This extension would require significant modifications to the optimising compiler and an excellent understanding of how code is generated within Jikes. The main issue will be to introduce cross-method invocations, being able to track which code is related in other specialisations and reference those points directly. Since the optimising compiler will recompile hot-methods these cross-specialisation references will not be constant and tracking the changes is non-trivial.

5.1.4 Instrumentation

Another interesting avenue to explore would be the use of transformations to instrument the VM, providing an alternative method for gathering statistics, perhaps even enabling some additional debugging support through self-reporting objects. While limited bytecode instrumentation was added through a special transform in this project it did not have the coverage that would be required for a useful instrumentation framework. Instrumentation is still an active area of research within Jikes, with numerous projects targeting different ways of getting instrumentation into the RVM to aid debugging, currently the only method is via the complex optimising compiler.

Introducing instrumentation through bytecode transforms is inefficient compared to the hardware-based instrumentation techniques that have been developed for Jikes. Hardware-based instrumentation is far more complex and the benefit of having an alternative transform-based mechanism would be in its ease of use. A transform can be written and applied to the VM without having to deal with any Jikes RVM internals, however, the transformed bytecode will be optimised and this could lead to invalid instrumentation

output. For debugging purposes, even during this project, transform-based instrumentation is a quick and useful method.

5.1.5 Specialising JNI and Reflection

The method specialisation framework still lacks support for Java Native Interface method specialisation. This would require a more substantial effort, since the specialised native methods would need to be implemented in native code. This would need to be developed in order to support a full 'free' read barrier implementation.

Calling methods using reflection also bypasses method specialisation. While not as complex as JNI, specialising reflection is non-trivial. Although the same dynamic dispatch methods can be used, specialised reflection-based invocation has some edge-case issues that are difficult to determine.

Bibliography

[1] Jikes RVM Architecture. `http://jikesrvm.org/Architecture`.

[2] Jikes RVM Research Mailing List. jikesrvm-researchers@lists.sourceforge.net, jikesrvm-core@lists.sourceforge.net.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Architecture and policy for adaptive optimization in virtual machines (ibm research report rc23429). Technical report, 2004.

[4] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 2003. ACM.

[5] David F. Bacon, Perry Cheng, and V. T. Rajan. A unified theory of garbage collection. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 50–68, New York, NY, USA, 2004. ACM.

[6] David F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 207–235, London, UK, 2001. Springer-Verlag.

[7] Henry G. Baker, Jr. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, 1978.

[8] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 187–196, New York, NY, USA, 1993. ACM.

[9] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: the performance impact of garbage collection. In *SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 25–36, New York, NY, USA, 2004. ACM.

[10] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? high performance garbage collection in java with mmtk. In *ICSE '04: Proceedings of the*

26th International Conference on Software Engineering, pages 137–146, Washington, DC, USA, 2004. IEEE Computer Society.

[11] Stephen M. Blackburn and Antony L. Hosking. Barriers: friend or foe? In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 143–151, New York, NY, USA, 2004. ACM.

[12] Stephen M. Blackburn and Kathryn S. McKinley. Ulterior reference counting: fast garbage collection without a long wait. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 344–358, New York, NY, USA, 2003. ACM.

[13] Stephen M. Blackburn and Kathryn S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 22–32, New York, NY, USA, 2008. ACM.

[14] Steve Blackburn, Robin Garner, and Daniel Frampton. *MMTk - The Memory Management Toolkit*. `http://cs.anu.edu.au/~Robin.Garner/mmtk-guide.pdf`.

[15] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: A code manipulation tool to implement adaptable systems. Technical report, 2002.

[16] A. M. Cheadle, A. J. Field, S. Marlow, S. L. Peyton Jones, and R. L. While. Non-stop haskell. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 257–267, New York, NY, USA, 2000. ACM.

[17] A. M. Cheadle, A. J. Field, and J. Nystrom-Persson. A method specialisation and virtualised execution environment for java. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 51–60, New York, NY, USA, 2008. ACM.

[18] Andrew Cheadle. *Real-time Garbage Collection for Dynamic Dispatch Languages*. PhD thesis, Imperial College London, 2009.

[19] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 125–136, New York, NY, USA, 2001. ACM.

[20] George E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, 1960.

[21] Urs Hlzle Computer and Urs Hlzle. A fast write barrier for generational garbage collectors. In *OOPSLA/ECOOP 93 Workshop on Garbage Collection in Object-Oriented Systems*, 1993.

[22] Will Deacon. A Generic Approach to Object Migration using Specialised Methods. Master's thesis, Imperial College London, 6 2009.

[23] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 37–48, New York, NY, USA, 2004. ACM.

[24] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[25] Samuel Z. Guyer, Kathryn S. McKinley, and Daniel Frampton. Free-me: a static analysis for automatic individual object reclamation. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 364–375, New York, NY, USA, 2006. ACM.

[26] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: improving program locality. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 69–80, New York, NY, USA, 2004. ACM.

[27] Richard Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons Ltd., 7 1996.

[28] Martin Kero, Johan Nordlander, and Per Lindgren. A correct and useful incremental copying garbage collector. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 129–140, New York, NY, USA, 2007. ACM.

[29] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for java. *SIGPLAN Not.*, 36(11):367–380, 2001.

[30] Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for java. *ACM Trans. Program. Lang. Syst.*, 28(1):1–69, 2006.

[31] Rafael D. Lins. Cyclic reference counting with lazy mark-scan. *Inf. Process. Lett.*, 44(4):215–220, 1992.

[32] A. D. Martínez, R. Wachenchauzer, and R. D. Lins. Cyclic reference counting with local mark-scan. *Inf. Process. Lett.*, 34(1):31–35, 1990.

[33] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960.

[34] Microsoft. *.NET Framework Developer's Guide : Garbage Collection*. `http://msdn.microsoft.com/en-us/library/0xy59wtx.aspx`.

[35] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 159–172, New York, NY, USA, 2007. ACM.

[36] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. Path specialization: reducing phased execution overheads. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 81–90, New York, NY, USA, 2008. ACM.

[37] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 33–44, New York, NY, USA, 2008. ACM.

[38] Wolfgang Puffitsch and Martin Schoeberl. Non-blocking root scanning for real-time garbage collection. In *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 68–76, New York, NY, USA, 2008. ACM.

[39] Ian Rogers, Jisheng Zhao, and Ian Watson. Boot image layout for jikes rvm. In *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS)*, 2008.

[40] Darko Stefanović, Matthew Hertz, Stephen M. Blackburn, Kathryn S. McKinley, and J. Eliot B. Moss. Older-first garbage collection in practice: evaluation in a java virtual machine. In *MSP '02: Proceedings of the 2002 workshop on Memory system performance*, pages 25–36, New York, NY, USA, 2002. ACM.

[41] Steve Wilson and Jeff Kesselman. *Java Platform Performance: Strategies and Tactics*. Prentice Hall PTR, 6 2000.

[42] T. Yuasa. Real-time garbage collection on general-purpose machines. *J. Syst. Softw.*, 11(3):181–198, 1990.

[43] Benjamin G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, 3 1989.