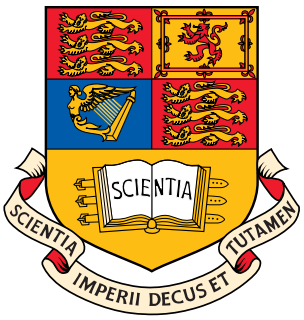
University of London
Imperial College London of Science, Technology and Medicine
Department of Computing

# Soft Real-time Garbage Collection for Dynamic Dispatch Languages

Andrew Cheadle




Submitted in part fulfilment of the requirements for the
Doctor of Philosophy of the University of London, and for
the Diploma of Imperial College London

*To my family and in loving memory of my grandparents,*
*Eric and Pamela Cheadle*

**Abstract**

Common to many high-level, declarative and object oriented languages are the use of *dynamic dispatch* and *garbage collection*. Dynamic dispatch is the mechanism by which the selection of a method to run, based on both its signature, and the type of one or more of its arguments, is made. A garbage collector is responsible for efficient 'recycling' of memory that is no longer needed, termed 'garbage', via a process of automatic deallocation that returns the memory to the program's allocation area. In this work we investigate various schemes for the *efficient* implementation of soft real-time multi-generation garbage collection algorithms within dynamic dispatch languages.

Our focus is on techniques that reduce and eliminate the synchronisation costs between interleaved user program and collector execution. Furthermore, we work to bound the amount of time for which the user program is paused, while the collector runs, so that real-time constraints can be met.

This dissertation describes the implementation of a production uniprocessor collector for Haskell, and evaluates its performance against implementations of existing algorithms. An important focus of our analysis is the effect that the various memory overheads of each scheme have on the performance of the collector. Our experiments demonstrate that closure code specialisation improves performance when it is used to remove dynamic space overheads. A 30% code bloat on top of the baseline generational collector buys us a time-based incremental generational collector whose *mutator* overhead is 5% when averaged across the `nofib` benchmark suite. Furthermore, not only does the collector achieve a consistent minimum mutator utilisation of 50% at around 10 to 15 milliseconds, but results, on average, in a performance increase of around 5% at the expense of a small additional amount of memory.

The remainder of the dissertation explores the applicability of our collector to other dynamic dispatch languages with a prototype implementation for Java.

## Acknowledgements

I owe so much to so many people for the encouragement and patience that has been extended me during the period of this thesis.

First and foremost to my supervisor, Tony Field, who has been an exceptional mentor and guide and whose perceptiveness, knowledge and subtle whip-cracking has made this an enjoyable learning process. I must thank both Tony and Paul Kelly for comments on the contents and structure of the thesis and for helping me to pull it together into a single coherent body of work.

I am especially grateful to Simon Marlow for his tutorage in the internals of The Glasgow Haskell Compiler and in its debugging – without his support I would never have managed to keep my implementations current nor have determined the tradeoffs that make them practical and production-worthy.

The development effort underpinning this dissertation and the thesis investigation has been enormous. I am therefore indebted to those that have alleviated this burden somewhat. To John Zigman, for his Jikes RVM BCEL patch. To Steve Blackburn and Tony Hosking for their Jikes RVM conditional read barrier patch. To Johan Nystrom-Persson for the inlining optimisations that unleash the full potential of the Jikes RVM TIB specialisation.

I'd like to thank the coauthors of our publications, Tony Field, Simon Marlow, Simon Peyton Jones and Lyndon While, for their helpful, honest and sometimes brutal comments during the drafting process. It is however unlikely that I have refined my often verbose style to one that now satisfies!

I am most grateful to the following funding sources that have made this work possible – *IC-PARC* and the *Department of Computing, Imperial College London*; *Telcordia Technologies Inc.*, New Jersey, USA; *Microsoft Research*, Cambridge, UK and finally, *The Engineering and Physical Sciences Research Council* under funding granted to *The UK Memory Management Network*.

Special thanks goes to Richard Jones for making his all encompassing garbage collection bibliography publically available and for his role as coordinator of The UK Memory Management Network. This consortium has organised many highly educative and interesting workshops and has provided a most valuable forum for UK researchers to meet and discuss their ideas.

Finally, I'd like to thank my family and friends whose support, patience and understanding, especially in times of self-imposed solitude has helped me achieve this goal. In particular, I must thank Andrew Sadler who as a colleague and long-time friend "stared blankly at me and occasionally grunted, whilst I explained problems and sought solutions without his intervention" – well, that's his take on it!

# Contents

# List of Tables

# List of Figures

# List of Algorithms

# Listings

# Chapter 1

# Introduction

Programming languages have evolved to place emphasis on the modelling and specification of software components and their hierarchical composition. These components are designed to encapsulate separation of concerns, promote portability and re-use and provide clean and well-defined interfaces that often implement dynamic behaviour. High-level languages, usually declarative or object oriented, achieve this through the provision of programming constructs which are heavily abstracted from both hardware architecture and operating system. These language features obviate such concerns as explicit thread scheduling, asynchronous event handling, low-level input-output and dynamic memory management. The responsibility for such services lies with a language's *runtime system* of which the *allocator* and *garbage collector* are two core components. The allocator is responsible for the efficient allocation of memory from the program's *heap*, the area of memory from which run-time data structures are allocated. The garbage collector is responsible for efficient 'recycling' of memory that is no longer needed, termed 'garbage', via a process of automatic deallocation, termed *garbage collection*, that returns the memory to the heap.

The low cost of random access memory means that gigabyte heaps are increasingly affordable. Simultaneously, large object inheritance hierarchies, encouraged by object oriented programming paradigms, drive up the memory footprint of a program. The combined effect is that programs often work with heaps in excess of four gigabytes that stress traditional allocation and garbage collection techniques, and furthermore, render many unsuitable. Many garbage collection algorithms require that the program be stopped for a period of time while the garbage collector runs. The pauses that result can last for many seconds. Whilst acceptable for batch processing, these pauses are unacceptable for interactive or real-time applications.

In this work we investigate garbage collection techniques that are suitable for large heaps with the primary goal of limiting the period of time a program must wait between activities.

## 1.1 Motivation and Thesis Contributions

The introduction of multicore and multiprocessor architectures into desktop computers is increasingly commonplace and a roadmap adopted by the majority of chip manufacturers. In response, language implementers are looking to exploit the concurrency and parallelism that these architectures provide. The focus of this exploitation is in the virtual execution environment, the virtual machine, and language's runtime — primarily within the garbage collector. By multi-threading the collector, parts of "stop-the-world" collection algorithms can be parallelised, and with appropriate synchronisation, can run concurrently with the mutator. These techniques reduce the length of the stop-the-world pause and increase both the responsiveness of the mutator and its utilisation. However, as application developers begin to exploit the concurrency and parallelism that these multiprocessor environments provide, so the collector threads will be forced to contend for processor time with the application threads. These collector threads will no longer have the luxury of an entire processor or core to themselves. As a result, state-of-the-art collectors must be capable of running incrementally, as they have within uniprocessor environments. In such an environment, the traditional way to reduce the pause times is to perform a small amount of garbage collection periodically, rather than collect the whole heap at once. In this way the activities of the mutator and garbage collector are interleaved.

In this work, we explore, develop and evaluate, mutator-collector barrier synchronisation techniques for uniprocessor architectures, with the primary goal of reducing their associated run-time overhead. The use of a uniprocessor environment simplifies our study whilst the techniques we develop are equally applicable to multiprocessor environments.

In the context of *copying* garbage collectors [Che70], there are three general approaches to achieving incremental collection. The first, Baker's algorithm [Bak78], uses a *read barrier* to prevent the mutator from accessing (live) objects that have yet to be copied (in *from-space*). In the second, the *replicating* scheme of Nettles et al. [NOPH92], the reverse is true: the mutator is forbidden to access the objects that have been copied, until the current garbage collection cycle has completed. Instead, a *write barrier* is used to trap updates to from-space objects. These are logged separately and the log is used later to update the objects that have been copied to *to-space*. In the third, Baker's read barrier is replaced by Brooks' unconditional read barrier and is employed in conjunction with a write barrier. Brooks' read barrier imposes a one word overhead on all objects and redirects the mutator to the to-space object once it has been copied. The write barrier is then used to ensure that all writes occur to the to-space copy once it exists. Fundamental to the performance of the two latter algorithms is the substitution of the expensive read barrier in Baker's scheme with a rather cheaper write barrier, but at the expense of some additional work in maintaining the write log or in eagerly applying the updates.

Baker's scheme has been explored extensively for a range of computing platforms and runtime systems, see for example [Zor90]. Replicating collection has been explored by Nettles

and O'Toole for SML/NJ [AM87] and by Blelloch and Cheng [BC99] for TIL/ML [TMC+96] and has been shown to incur overheads that are substantially smaller than those typically reported for Baker's scheme.

Brooks' approach has been adopted in the work of Henriksson et al. [Hen94, MH95, Hen96b, Hen96a, Hen97, Hen98] and also by Bacon et al. in the Metronome collector [BCR03b]. However, neither bodies of work are evaluated against a comparable Baker implementation, nor is the true cost that the additional word on each object imposes determined.

In this thesis, we evaluate Baker's scheme against an alternative *barrierless* implementation, in the context of the Haskell GHC compiler with generational garbage collection. Experiments within our evaluation allow us to measure some of the overheads that both replicating and Brooks-style collectors would incur when implemented within GHC's runtime system. Haskell differs from ML in quite significant ways when it comes to garbage collection. In particular it is a lazy language, so the heap is typically heavily populated with closures. Closure updates, in which a closure is overwritten with its value, are thus the main form of update and are commonplace. For some applications we would therefore expect the balance of reads and writes to vary substantially from that of ML. While we expect the mutation log to be longer in Haskell, all closure accesses equate to pointer loads which still significantly outnumber closure updates. Intuitively one might expect this to mitigate replicating collection and more so by combination of the replicating and generational write barriers when replication is employed for a generational collector. However, our experience in implementing such a collector for GHC demonstrates that this is not so. GHC's implementation of lazy evaluation uses dynamic dispatch. This allows us to develop a technique for the removal of both Baker's read barrier and the inter-generational write barrier, yielding a barrierless collector that operates both incrementally and generationally. A key objective is to explore these issues for a range of benchmark applications in order that the tradeoffs of read barrier, write barrier, and barrierless schemes can be better understood in dynamic dispatch languages. Armed with such understanding, our main objective is to develop a soft real-time collector that can be scheduled to meet soft real-time constraints on the order of 10 milliseconds.

The thesis of this dissertation is:

*Devirtualisation and compile-time dynamic dispatch elimination are assumed optimisations. They should not be considered so. The presence of dynamic dispatch enables a class of runtime system optimisations, particularly, but not exclusively, suited to efficient garbage collector implementation, that outweigh the benefits of their indiscriminate removal.*

We make the following contributions:

- We discuss techniques that exploit the underlying dynamic dispatch mechanism to eliminate unnecessary read/write barriers in incremental and generational garbage collectors. We investigate the use of *object specialisation* to eliminate run-time memoisation overheads, and analyse the effect on code size and instruction and data cache performance,

in addition to the run-time space and time savings.

- We describe detailed implementations of incremental semi-space and incremental generational garbage collectors for Haskell, that utilise the read/write barrier elimination techniques developed. In particular, we detail the variation in their designs which are required to support the *Push/Enter* and *Eval/Apply* closure evaluation models that a lazy functional language can be built upon.

- We compare our collectors with reference implementations of Baker's algorithm. Our analysis focuses on the space and time overheads associated with enforcing the collectors' to- and from-space invariants.

- Finally, we discuss the issues surrounding the use of our collectors, Baker's standard collector and replicating collectors in languages other than Haskell or ML, focusing on the Java Virtual Machine. We describe a prototype implementation that incorporates our barrier elimination techniques.

The basic idea behind the barrierless collector, and its performance in comparison with a traditional implementation of the Baker scheme, was reported in [CFS+00] for a single generation (i.e. a semi-space) collector. In [CFS+04] we resolved deficiencies, in the semi-space collector that manifested primarily as a run-time space overhead; described the incorporation of the barrierless collector within a generational context; and applied the read barrier elimination technique to the generational write barrier. Finally, in [CFNP08, CFNP07] we presented a modified JVM providing a fully virtualised execution environment for Java and a framework in which method specialisation could be effected. Using this framework, we investigated the application of the read barrier elimination technique to this rather less pure dynamic dispatch language.

## 1.2 Structure of Dissertation

This chapter has provided an introduction to automatic dynamic memory management and motivation for the thesis.

In Chapter 2 we discuss the classic garbage collection algorithms which lie at the heart of the majority of today's collectors.

In Chapter 3 real-time garbage collection algorithms, strategies and techniques are described and the current 'state-of-the-art' surveyed.

Chapter 4 discusses the basic implementation techniques that span (lazy) functional programming language implementations, with particular focus on the abstract machines that underpin them. Finally, the *Non-stop Spineless Tagless G-machine* is introduced, that augments the Spineless Tagless G-machine — GHC's abstract machine, with very basic modifications to add support for efficient incremental garbage collection of the machine's heap allocated closures. The *While and Field Collector* [WF92] that results has never been implemented,

and is in no way complete, but is fundamental to our general approach and subsequent implementations.

Chapter 5 initiates our main contributions, starting with a prototype scheme and implementation for an incremental semi-space collector for Haskell. This collector optimises Baker's read barrier, completely eliminating it when it is off, at the expense of an extra word for each *copied* object.

Chapter 6 investigates the exploitation of GHC's dynamic dispatch mechanism for the write barrier elimination of a generational copying collector. In Chapter 7 we eliminate the extra-word overhead of our prototype collector, migrate it from a semi-space to a generational context, and work to bound the pause times, focusing on thread stacks and inter-generational remembered sets. We detail the implementation of both work-based and time-based collectors, the latter of which targets consistent utilisation at time quanta of the order of 10 milliseconds. We also briefly describe the challenges of implementing a replicating collector for GHC, concluding that the overheads are prohibitive and such a collector impractical, at least for Haskell.

Chapter 8 presents a detailed evaluation of the collectors and various trade-offs that must be made in the resulting production collector. We complete the chapter with a study of the (soft) real-time capabilities of our various collector implementations.

In Chapter 9 we investigate the applicability of our techniques to other dynamic dispatch languages, focusing on Java, and evaluating a prototype implementation.

Each of the major chapters provides an independent conclusion where appropriate. Collectively, these form the main conclusion of the thesis. However, a final summary of our contributions, as well as the overriding conclusion in relation to our central thesis (above), together with an overview of future work, is presented in Chapter 10.

In the appendices we provide detailed overviews of the architecture of the platforms for which we develop our implementations. Appendix A describes the structure of GHC, its compiler and runtime system, and supports Chapter 5 through to Chapter 7. Appendix B describes the Jikes Research Virtual Machine and supports Chapter 9.

# Chapter 2

# Introduction to Garbage Collection

The first garbage collected language was McCarthy's LISt Processor programming system — LISP, developed in 1958 and documented in 1960 in [McC60]. The language was developed for the *Advice Taker*, a deductive system for Artificial Intelligence research which operated by 'manipulating [symbolic] expressions representing formalized declarative and imperative sentences'. McCarthy refers to a garbage collection cycle as a *reclamation cycle* and describes a tracing mark-sweep algorithm (although he does not refer to it by this name) in order to obtain free registers once the free list has been exhausted. He subsequently admits to referring to the reclamation process as 'garbage collection' and says:

> ..., but I guess I chickened out of using it in the paper — or else the Research Laboratory of Electronics grammar ladies wouldn't let me.

And so, garbage collection was born!

In addition to its contribution for the management of dynamic memory, LISP introduces the *public push-down list*, a recursion stack, for callee register saves across recursive function invocations. Prior languages, such as FORTRAN, used *activation frames* with *statically allocated* slots, whose size and location were determined at compile-time, for the storage of lexically scoped data and results. The ramification of this was that only one invocation of a subroutine could be executing or 'active' at any one time — a problem for LISP designers attempting to exploit the recursive nature of list processing operations. This recursion stack was subsequently generalised in Algol-58, to a program's (*local*) *stack*, where activation frames are pushed when a subroutine starts executing and also where a caller/callee subroutine stores register values that will be 'clobbered' by subsequent subroutine invocations.

McCarthy's mark-sweep algorithm belongs to the *tracing* class of garbage collectors which traverse the reachable objects of the system, marking them as 'live' and collecting those

objects that are unreachable or 'dead', and therefore deemed available for re-use. In 1960 Collins devised an alternative class of algorithm to address 'inefficiencies' within the mark-sweep collector based on the observation that its complexity is not limited to the number of live cells within the heap but to all cells in the heap — they are all visited in the sweep phase. This class of algorithm, *reference counting* collection [Col60], is the single alternative approach to tracing collection. It is so called because a count of referrers to an object is stored within the object and when the count becomes zero, the object can be safely collected and re-used. Although these two classes of garbage collectors are viewed as being fundamentally different approaches, they share much in common, and have in fact been shown to be duals of one another [BCR04] — tracing operates over the live objects, whilst reference counting operates on the dead. Furthermore, Bacon et al. show that for every operation performed by one class of collector, there is an 'anti-operation' performed by the other.

McCarthy states the following as his reason for the introduction of garbage collection:

> This [reclamation] process, because it is entirely automatic, is more convenient for the programmer than a system in which he has to keep track of and erase unwanted lists.

This reasoning is still one of the main motivations for the use of garbage collection in today's programming languages. The majority of these languages allow the run-time allocation of variable sized data structures as *dynamically allocated memory* whose lifetime is detached from the programs imperative lexical scope. Lexical scoped storage allocation provides a simple memory management mechanism which is predominantly transparent to the programmer. However, there are several shortcomings that motivate the use of dynamically allocated storage and ultimately garbage collection: activation frames are unable to share local variable bindings; incrementally constructed data structures such as lists and other recursive data types cannot be allocated, since their size is unknown when the activation frame's slots are created; the data contained within the activation frame cannot outlive its caller; and only an object whose size can be determined at compile-time can be returned as the result of a subroutine. The dynamic allocation of data structures in the heap resolve all of these shortcomings and facilitate many of the language features that are taken for granted in high-level and object-oriented languages such as Java. Jones motivates garbage collection as a necessity for many high-level, declarative and functional programming languages [Jon96]. Indeed, explicit deallocation would be exceptionally hard for a programmer to get right, in the presence of closures, sharing and delayed execution of suspensions, all of which have unpredictable execution orders in functional and constraint logic programming languages. In addition to this 'language requirement', there are several motivations related to software engineering practise. McCarthy's reason alludes to this, although there are stronger cases than just convenience. Explicit deallocation performed by the programmer is fraught with danger — forgetfulness leads to *memory leaks* and ultimately memory exhaustion; overeager deallocation leads to

*dangling pointers*, which if dereferenced may yield incorrect data or program termination from the access of an illegal location; and finally, *double deallocation*, where an already free or reallocated object is erroneously freed a second time, may result in undefined program behaviour or program termination. In short, (type) safety guarantees and program correctness can be enforced, or at least, more easily verified in a garbage collected environment. In addition, the programmer is released from the chore of problematic memory management for data whose lifetimes are not determined by lexical scope. This allows them to focus on the high-level programming tasks of application modelling, interface abstraction and modularity.

The primary goal of any garbage collector is to free as much garbage as it can at time of invocation, usually when the heap has been exhausted. It works to achieve this by identifying the dead cells as garbage cells and returning them to the free store. A correctness invariant must be maintained by the collector such that no live cells are identified as garbage. This process of liveness determination is merely a conservative approximation based on the reachability of an object from the *root set*, most generally, static data, the machine registers, and the program stack(s). This is of course a logical liveness criteria since any reachable object may be referenced at some time in the future. However, references to objects are often prolonged beyond the last use of the object, and to a garbage collector, as long as the object is transitively reachable via a (root set) reference chain, it is live. This inefficiency is common to both classes of tracing and reference counting garbage collectors and is an area of ongoing research.

The reference chains form a directed graph, the *memory graph*, between the roots and all live objects within the heap. As the user program executes, so nodes and edges of the graph are created and destroyed. Hence the graph is mutated. We refer to the user program as the *mutator* [Dij75], to distinguish it from the runtime system and in particular the garbage collector. The graph can of course contain cycles, consider the reference chain of a circular linked-list. A tracing garbage collector periodically traverses the memory graph using a breadth-first or a depth-first graph traversal algorithm, and can therefore be seen as a distinct entity from the mutator. In the most basic uniprocessor tracing algorithms, the mutator is simply stopped while the garbage collector traces out the memory graph, marking live data — a *stop-the-world* algorithm. The boundaries between a reference counting collector and mutator are less clear, for the execution of the mutator and collector are interleaved — the algorithm is inherently *incremental*. Unlike a tracing collector, a reference counting collector is able to respond immediately to graph mutations by the mutator, as cell references are destroyed.

Cooperation between mutator and runtime system is not limited to just run-time, but can be further enhanced at compile-time by allowing the runtime system, and in particular the garbage collector, access to type information. Such a garbage collector is said to be *type accurate* or *exact* as the type information can be used to distinguish references from *immediate* or *primitive* values. McCarthy's collector is one such example. A garbage collector that does not have access to such information, for example, in the case of a non-garbage collected language

that has been augmented with a collector, is called a *conservative* collector [BW88]. The collector is conservative because it must treat immediate values that could be interpreted as valid heap addresses *as* heap addresses. The net result is the occurrence of false negatives in the identification of live cells — some cells are kept alive because of immediate values appearing to be 'fake' references. This is not erroneous but it is to the detriment of the collector's efficiency. An example of a conservative collector is the Boehm-Demers-Weiser mark-sweep collector for C and C++ as a replacement for the explicit dynamic memory management primitives of `malloc`, `new`, `free` and `delete`. Henderson shows how type- and liveness-accurate information can be augmented to a C program, via a transformation, allowing the use of a conventional garbage collector thus eliminating the problems associated with conservative collectors [Hen02].

One final characteristic that is independent of the class of collector is whether or not it is a *moving* or *non-moving* collector. A moving collector is free to relocate allocated and free objects, enabling the *coalescing* of objects and reducing the *fragmentation* of the heap. A non-moving collector simply chains free cells through the heap, and allocated objects assume a fixed address. This often results in increased heap fragmentation and adds overhead to the allocation of variable sized structures in return for a cheaper deallocation operation. A mark-sweep collector is an example of a *non-moving* collector, because the sweep phase simply chains free cells onto the free list. Conservative collectors are most often non-moving collectors (consider the 'fake' pointer problem). The allocation / deallocation overheads and the issue of fragmentation that can be highly problematic in a non-moving environment are discussed by Wilson et al. in their excellent survey and critical review [WJNB95] and also in [JW97, Joh97, JW98]. In fact, these issues are arguably more pertinent to the design of allocators that operate outside of garbage collected environments [Lea97, BB99, BZM01, BZM02].

In the following sections we discuss the basic garbage collection techniques of which modern algorithms are variants. A more detailed and comprehensive survey can be found by combining [Coh81, Wil92, Jon96, PS95, AR98].

## 2.1 The Reference Counting Algorithm

For each object created in the heap by the mutator, a count is maintained of the number of objects that reference it (including itself). When a reference to the object is created, the count associated with the object is incremented, and when this reference is destroyed, for example by assignment to 'null' or another object, the count is decremented. If the reference count is equal to zero then the object is no longer reachable and can therefore be reclaimed. However, before the collector can reclaim the memory allocated to the cell, any cells which are referenced by this garbage cell must also have their reference counts decremented. As a result other cells may also become garbage. The collector therefore 'chases' references iteratively or recursively applying the algorithm to those cells reachable via the reference chain. The mechanism by which the collector 'traps' memory graph mutations on creation, destruction

or reassignment of a reference is referred to as a *write barrier*.

The main advantage of this scheme is that the memory management overheads are distributed throughout the computation of the user program, with the reference count handling and the reclamation of garbage cells interleaved with its execution. This is in contrast to stop-the-world or *stop-start* algorithms which result in the pause of the user program, while the collector runs. Incremental collectors are preferable in many applications, in particular real-time systems. Another advantage is that it is a relatively simple scheme to implement, usually be defining a macro to handle the reference counter when pointer assignment occurs, and/or having the compiler generate this book-keeping code automatically.

Unfortunately, there are more disadvantages to the scheme than there are benefits and so in production language environments, it is little used. The distribution of processing (i.e. the granularity of interleaved processing) is rather 'lumpy' with the cost of deleting the last reference to a portion of the memory graph being dependent upon its size. Not only is there a time overhead incurred through increased instruction count, but there is a space overhead associated with the reference count field for each object — it is usual to store the count within the object. As we shall see in subsequent chapters, incurring a space overhead for the support of a runtime system service such as garbage collection is undesirable. Generally, space overheads equate to time overheads, or worse still memory exhaustion, as the mutator has less memory available to it for the computation. In a tracing collector, a reduction in available memory results in an increase in the total number of garbage collections, which in turn prolongs mutator execution.

The efficiency of the scheme as a whole is very low with a high price being paid for its simplicity — a high processing cost in updating the counters that maintain the reference count invariant. Each time a pointer is updated, the reference counts in both the old and possibly the new target objects have to be adjusted (naïvely, iterating over a reference counted data structure forces reference count adjustment with no overall net change). This cost is even higher in a multi-threaded environment, where the reference count operations must be synchronised as multiple mutator threads are able to perform concurrent pointer updates. Finally, the most serious disadvantage in the scheme is in its inability to reclaim cyclic structures, which occur frequently in functional and object-oriented languages, and may result in serious space leaks. It is worth noting that cycles were not an issue for Collins' LISP reference counting algorithm [Col60], for, in his design of LISP, McCarthy disallowed cycles, even though his mark-sweep collector was capable of collecting them.

## 2.2 Mark - Sweep Garbage Collection

McCarthy's mark-sweep collector is a non-moving, (and therefore non-compacting and fragmenting), stop-the-world tracing algorithm, consisting of two phases. The first phase runs as the result of a failure to fulfil a heap allocation request. The mutator is paused and the collector runs tracing the memory graph of live cells, starting from the root set, and *marking*

as live each referenced cell. It terminates when all leaf cells of the memory graph have been marked. The cost of this marking phase is clearly proportional to the number of live cells in the heap, i.e. the amount of live data. With the marking phase complete, the second phase, the sweeping phase, executes. The collector starts at the beginning of the heap, traversing it in its entirety, visiting each cell to determine if it has previously been marked. If it has, the collector unmarks the cell and progresses onto the next. If it has not, the cell is *swept*, i.e. linked onto the appropriate free list for future allocation, and the collector progresses onto the next cell. The cost of the sweep phase is therefore proportional to the total number of cells in the heap, i.e. the size of the heap.

Mark-sweep algorithms exhibit several advantages over reference counting. Two are particularly significant. The first is the trivial collection of cyclic structures/references. Cyclic references are implicitly handled because the collector is a 'global state' observer of heap memory. If a cycle is reachable via the live memory graph traversal, all its constituent cells are marked live. If it is not, its cells will be swept. The second is that with the collector fully decoupled from the mutator, there is no overhead associated with pointer manipulations, i.e. no write barrier.

These benefits have lead to the implementation of (variants) of the scheme in many functional and object-oriented systems, and most notably in 'uncooperative' environments where a collector is required to be conservative and non-moving, for example, the Boehm-Demers-Weiser mark-sweep collector for C and C++.

In a mark-sweep algorithm, the collector runs less frequently and is invoked when the heap, is exhausted. As a result of this stop-start execution, the cost, that is the mutator pause time, is dependent on the size of the heap since the collector must fully traverse it during the sweep phase. Whilst it is true that the reference counting algorithm suffers delays when traversing the reference chains, it is highly unlikely, even in the worst case, that these chains contain all the cells, at least within a functional environment such as Haskell. Indeed, the research of Stoye et al. finds that the majority of cells in a functional program exhibit a reference count of one [SCN84]. The inherently incremental nature of reference counting algorithms, where the allocation space for a single cell can be recycled as soon as the last reference to it is destroyed, leading to comparably shorter pause times, is often cited as its raison d'être. However, a second-level tracing collector, such as a mark-sweep collector, is often employed to collect cycles. As a result, there is a tradeoff between incrementality and reduced pause times, and the amount of memory left unavailable due to uncollected cycles.

The major deficiency of the mark-sweep algorithm is in its tendency to fragment memory. Cells swept to the free list are 'littered' among allocated cells. If allocated objects are of an equal single fixed size, (and reside within size segregated heap spaces), then there is no issue of (external) fragmentation. If however, heap objects vary in size and are not segregated according to their size, i.e. reside in a single heap space, then portions of memory may be rendered unusable for considerable periods of time, because the available fragment is too small to fulfil successive allocation requests. The effect of fragmentation can be exacerbated

or reduced by the coalescing and splitting policies that are employed by the allocator and collector in their free list management. It should be noted, that while this is an inherent deficiency of the mark-sweep algorithm, a reference counting algorithm generally allocates from fixed size (segregated) free lists that support coalescing and splitting and as a result suffers similar issues of fragmentation.

### 2.2.1 The Mark Phase

McCarthy employs a bit stealing technique to mark a cell as reachable. Assume for simplicity that a cell consumes one word of heap space. The most significant bit, the sign bit, of the word is hijacked during the collection process. As the memory graph traversal proceeds, reachable cells are marked live by setting the bit, and the sweep phase subsequently clears it. This is of course problematic for cells representing signed numbers, and in particular negative numbers, and also whole-word data types. McCarthy resolves this by segregating a portion of the heap and reserving it for these whole-word data types and an associated bit table where each of these data type instances may be marked as active.

The naïve and most simple implementation of marking is a recursive algorithm:

**Function** `recursiveMark(`*cell*`)` Recursively mark cells reachable from *cell*

**Input**: *cell* Cell to recursively mark

```
1 if cell is marked then
2     return;
3 setMark(cell);
4 foreach reference in cell do
5     recursiveMark(reference);
```

This algorithm uses the procedure call stack to record and follow the branch points in the live memory graph visiting parent-child nodes top to bottom, left to right, in a depth-first tree traversal. When a leaf node is encountered, it is marked if necessary, and the call stack unwinds, with each `mark()` procedure returning and its associated activation frame popped until the next 'queued' branch point is 'found' and marking resumes. Having processed the rightmost leaf node (or marked internal node) of the memory graph, the stack unwinds completely, popping all `mark()` activations, and the mark phase terminates. The call stack must have enough space reserved to accommodate the largest reference chain from root node to deepest child — the maximum stack depth. It should be apparent that:

- Because activation frames consume call stack space, an arbitrarily large memory graph can overflow and exhaust the memory available for utilisation by the call stack.
- Tracing the memory graph from each root set reference, and the presence of cycles, can result in multiple, (and therefore unnecessary), subgraph traversals.

## 2.3 Mark - Compact Garbage Collection

The problem of fragmentation that the mark-sweep algorithm exhibits may mean that an allocation request fails, even though the total amount of free heap space is sufficient to accommodate the object, because the available free list blocks are too small to contain the object. A free list allocation algorithm generally involves an element of search in order to locate a suitably sized block. When allocating an object, the performance of the *fits* algorithm employed, whether it be first-, best-, next-, worst- or type / size segregated, can be critically impacted by the fragmentation of memory and the size blocks available. In addition, the efficiency of the mutator, with respect to its cache performance, can be severely reduced as the spatial locality of the objects in the heap is destroyed. In this case, objects allocated and dereferenced together in time are located sufficiently far apart within the heap such that they do not reside in the same cache line, level one or even level two cache.

Compaction of live data resolves these problems. Firstly memory allocation is simple, it is done by incrementing a heap pointer (usually known as *bump pointer allocation*), and allocated in linear increments allowing all objects to be allocated with the same cost regardless of their size. Secondly, compaction of live objects into a contiguous portion of the heap preserves and possibly increases the spatial locality of objects. Finally, the compaction of live data, and the heap, result in a reduction in *working set* size for a program, allowing the program to operate in smaller amounts of memory, i.e. with a smaller *footprint*.

Algorithms which compact the heap, eliminating the 'holes' resulting from fragmentation, and leaving both the heap and the free-space store contiguous as opposed to 'interleaved', are used. They are either *copying* or *in-place compacting* algorithms. In this section we concentrate on those that compact, and in the next, those that copy.

In-place compacting algorithms often use multiple passes to perform the compaction and work by calculating a *forwarding address* for the object under relocation. This forwarding address is stored in the object (for relocation in the next pass) or in the place of the object. The price paid for compaction and a contiguous heap is often high and hybrid collectors have therefore been developed which use heuristics to determine when compaction needs to be performed [San92].

These algorithms are often employed when there is only a limited address space, i.e. a memory constrained embedded environment [CKV+03], or the heap consists of a number of objects that have consistently long lifetimes, i.e. the residency is high. In such cases, the use of a copying algorithm is inefficient and undesirable due either to space limitations or repeated copying. The garbage collectors of declarative logic systems based on Warren's Abstract Machine (WAM) [War83, AK91] are predominantly *sliding* compacting collectors [ACHS88, BCRU83, PBW85, Sch90] for three reasons:

- The heap is allocated in a stack-like fashion which grows with forward execution and the allocation of *choicepoint* segments associated with non-deterministic execution. These segments are popped, in their entirety, on backtracking as the result of failure of a par-

ticular choice. This facilitates cheap, constant time deallocation of unbounded amounts of memory. As a result, segment ordering must be preserved.

- Some Prolog systems implement term comparison based on the terms' relative address ordering within a segment. As a result, term ordering must be preserved.
- The use of a copying collector in a WAM-based system can result in the double copying of a structure's internal cells. This is somewhat subtle but potentially results in a heap that is larger post-garbage collection [BL94].

Jones categorises compacting algorithms into three classifications [Jon96], based on the effect of compaction on cell ordering. *Arbitrary* compaction algorithms move cells making no explicit attempt to maintain their original order. These algorithms may be fast to run and simple to implement, but it is possible for orderings of cells to occur such that the resulting structures exhibit poor spatial locality through a reduction in cache hits and virtual memory performance. *Linearising* algorithms, where possible, move those cells which reference one another into contiguous heap locations (or at least as close to each other as possible) resulting in improved spatial locality. *Sliding* algorithms shift the live cells to one end of the heap, eliminating free space fragments (free space will be contiguous from where the live cells end), preserving the order of the cells and thus the spatial locality.

One of the earliest compacting algorithms was Edwards' *two-finger* algorithm [Edwwn] for fixed sized cells. Although it is possible to enhance Edwards' algorithm to handle variable sized objects by segregating the heap and compacting same sized objects into these regions, a more satisfactory one that also preserves object ordering is the sliding *Lisp 2* algorithm [Knu73]. A set of algorithms that do not require the additional storage overhead of the Lisp 2 algorithm, are suited to variable sized object compaction and slide objects to preserve their ordering are *table-based* compactors [HW67, Weg72, FN78]. Jonkers' compaction algorithm [Jon79] is a widely implemented compaction algorithm that is not only space efficient but also avoids the table management overheads of table-based compactors. It is a more general algorithm than Morris' [Mor78], upon which it builds. The algorithms consist of the standard tracing mark phase followed by two compaction phases which scan the heap in its entirety, require no extra space for the storage of relocation addresses, and are order preserving sliding compaction algorithms.

## 2.4 Copying Garbage Collection

The alternative to in-place compaction of live data, is compaction via copying. Like in-place compactors, allocation is cheap, with the use of a bump pointer, and the heap exhaustion check has low cost — a simple pointer comparison. In addition, because object compaction occurs implicitly during copying, and the allocation of the object copy uses bump pointer allocation, variable sized cells are trivially handled.

Copying collectors partition the heap, minimally into two, on which, for ease of explanation, we focus. The first partition, *to-space* is designated the heap, from which new objects are allocated. As the heap nears exhaustion and an allocation request fails, the stop-and-copy collector is invoked. Initiation of the collector results in a 'flip' of the two partitions, with to-space becoming *from-space*, and the second partition, previously reserved as from-space becoming to-space. The roots of the live memory graph are then copied, in a process known as *evacuation*, from from-space to to-space. As each cell is evacuated, it is overwritten with a forwarding address to the relocated cell in to-space. If a second attempt to evacuate the cell occurs, (the cell has multiple referrers), the to-space address found within the forwarding pointer is returned. It must therefore be possible to distinguish forwarding pointers, and this is most usually achieved using a reserved tag bit, which must be tested before the evacuation of each cell. Having evacuated and forwarded the cell, the naïve form of the algorithm, due to Fenichel and Yochelson [FY69], proceeds by recursively *scavenging* the cells, the process by which the references of this forwarded cell are themselves evacuated and copied to to-space. The recursive invocations of this evacuation routine return when all references of the cell undergoing evacuation have been forwarded and scavenged. This results from the scavenger encountering either a non-reference type at the leaf node of the live memory graph, or a forwarded cell (which has therefore already been, or is in the process of being, scavenged). It should be apparent that this process of implicitly marking the live memory graph with forwarding addresses parallels the explicit marking algorithm of Function `recursiveMark()`. In addition, it is combined with a depth first compaction phase that leaves all live data residing with referential ordering and spatial locality preserved, within to-space. Fenichel and Yochelson's algorithm is in fact a variant of Minsky's explicit marking LISP collector that utilises 'secondary' storage, i.e. a disk or drum, for to-space [Min63]. It is worth noting that Minsky's use of explicit marking eliminated the need for a marking stack. With evacuation and scavenging complete, from-space has now been *condemned* and can be thrown away, viz reinitialised if necessary, for use as to-space in the subsequent collection cycle. The allocation request that previously failed is now re-attempted, from the new to-space, if it again fails, the program terminates with an out-of-memory error, since the collection cycle has been unable to free sufficient memory. If it succeeds, the mutator resumes and the collection cycle is complete.

The pseudo-code for the allocator and collector routines are outlined in the following algorithms and functions. For illustrative purposes, it is assumed primitive and reference types occupy the same number of bytes.

- Algorithm 2 defines and initialises the semi-space collector's global variables.
- Function `allocate()` is the heap allocation routine and triggers the collector on heap exhaustion.
- Function `flip()` swaps the two semi-spaces, to-space and from-space, at the beginning of the collection cycle.

- Function `recursiveEvacuate()` recursively copies a from-space object to to-space with depth first evacuation of its reference fields.
- Function `recursiveSemiSpaceGC()` is the main routine of the garbage collector that invokes the flip and iterates across the roots initiating their evacuation.

**Algorithm 2**: Semi-space garbage collector global variable initialisation

```
1 Pointer memoryStart =<start of memory>;
2 Pointer memoryEnd =<end of memory>;
3 Pointer
  semiSpacePartition = memoryStart + ((memoryEnd − memoryStart)/2) + 1;
4 Pointer toSpace = memoryStart;
5 Pointer free = toSpace;
6 Pointer fromSpace = semiSpacePartition;
7 Pointer heapLimit = semiSpacePartition;
```

**Function** `allocate(`*size*`)` Dynamic object allocation

```
  Input: size Integer size of object to allocate
  Output: ptr Pointer to allocated object
1 Pointer ptr = free;
2 if free + size ≥ heapLimit then
3 |   recursiveSemiSpaceGC ();
4 |   if free + size ≥ heapLimit then
  |   |   /* Garbage collector failed to free enough memory to satisfy
  |   |      allocation                                                  */
5 |   |   abort (Out of memory);
6 free = free + size;
7 return ptr;
```

**Function** `flip` Flip to-space and from-space

```
1 if toSpace == memoryStart then
2 |   free = toSpace = semiSpacePartition;
3 |   fromSpace = memoryStart;
4 |   heapLimit = memoryEnd;
5 else
6 |   free = toSpace = memoryStart;
7 |   fromSpace = semiSpacePartition;
8 |   heapLimit = semiSpacePartition;
```

Figure 2.1 depicts the state of the heap as an allocation request fails, and a collection cycle is initiated. Objects $O_1$ and $O_2$ form the roots of the live memory graph while $O_8$ through

**Function** `recursiveEvacuate`(*reference*) Recursive evacuation of object pointed to by *reference*

```
   Input: reference Pointer to object to evacuate to to-space
   Output: toSpaceReference Pointer to to-space residing copy of object pointed to by
           reference
   /* If reference has already been evacuated and forwarded do nothing    */
 1 if reference ≥ toSpace and reference < heapLimit then
 2 |   toSpaceReference = reference;
 3 |   return toSpaceReference;
 4 Integer size = size of object pointed to by reference;
 5 Pointer fieldPointer = free;
 6 Pointer toSpaceReference = free;
 7 Word firstFieldCopy =dereference(reference);
 8 Boolean firstFieldIsReference = false;
 9 if dereference(reference) is a reference type then
10 |   firstFieldIsReference = true;
11 free = free + size;
   /* Set the forwarding address before recursively processing fields     */
   /* This ensures termination and prevents redundant copying             */
12 dereference(reference) = toSpaceReference;
   /* Evacuate the first field                                            */
13 if firstFieldIsReference then
14 |   dereference(fieldPointer) = recursiveEvacuate(firstFieldCopy);
15 else
16 |   dereference(fieldPointer) = firstFieldCopy;
17 fieldPointer = fieldPointer + 1;
   /* Depth-first evacuation of references                                */
18 for fieldCount = 1 to size − 1 do
19 |   field = reference[fieldCount];
20 |   if field is a reference type then
21 |   |   dereference(fieldPointer) = recursiveEvacuate(field);
22 |   else
23 |   |   dereference(fieldPointer) = field;
24 |   fieldPointer = fieldPointer + 1;
25 return toSpaceReference;
```

**Function** `recursiveSemiSpaceGC` Recursive semi-space garbage collector

```
1 flip ();
  /* Depth-first evacuation of the root set                               */
2 foreach root in roots do
3 |   if dereference(root) is a reference type then
4 |   |   dereference(root) = recursiveEvacuate(root);
```

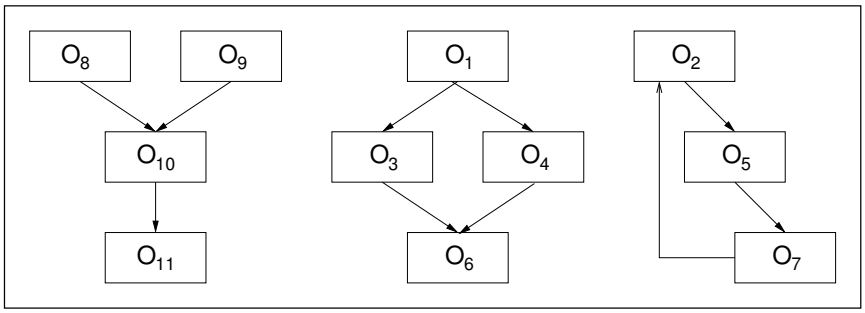



Figure 2.1: Exhausted heap prior to garbage collector initiation. Objects $O_1$ and $O_2$ form the root set.

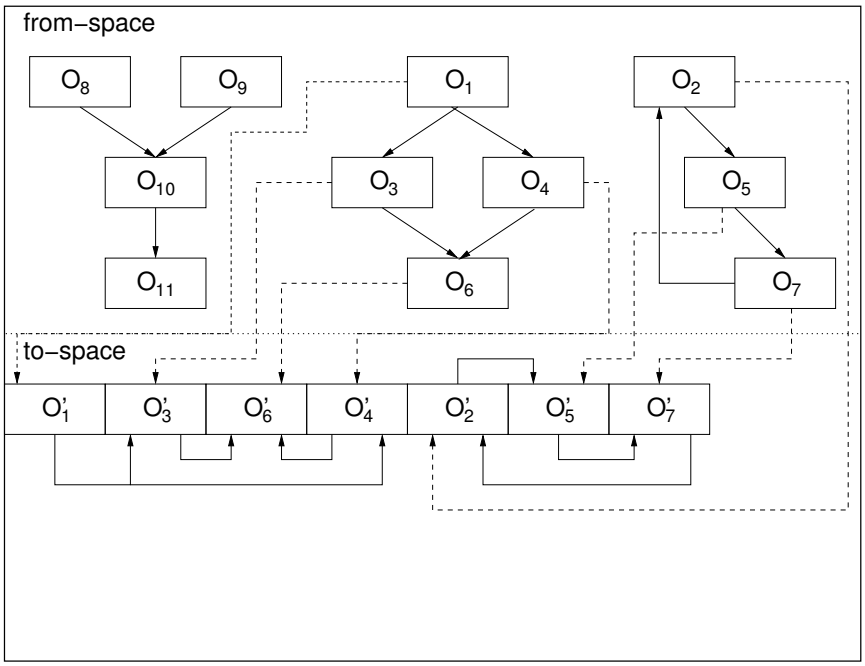



Figure 2.2: Depth-first evacuation of live objects to to-space.

$O_{11}$ are garbage. Figure 2.2 depicts the state of from-space and to-space on completion of depth-first evacuation and scavenging.

Fenichel and Yochelson's copying algorithm suffers from the same problem as the mark phase introduced in Section 2.2.1, consuming call stack space with each recursive invocation of Function `recursiveEvacuate()`, and the potential consequence of stack overflow. They recognised this, and suggested the use of the slower and more complex Deutsch-Schorr-Waite pointer reversal algorithm [SW67]. Fortunately however, Cheney constructed an efficient iterative algorithm [Che70] that does not incur this extra space overhead, based on the observation that it could be implicitly absorbed by the cells relocated to to-space. In order to achieve this, cells are now evacuated in a breadth-first manner, and the scavenger no longer scavenges the references of the from-space copy of the cell but the relocated to-space copy. Root evacuation proceeds as before, but the recursive evacuation routine is replaced by the Function `evacuate()`, that simply copies the cell to to-space and places the forwarding address. With all the roots evacuated, the scavenger is invoked, linearly scanning and scavenging the newly evacuated to-space cells. This scavenging process moves a 'scavenge' pointer towards the 'free' pointer at which new to-space cells are allocated. Again, the scavenger is evacuating only the immediate from-space references of the current object in this breadth-first fashion and their to-space copies are *queued* for scavenging. The stack has been replaced by an implicit to-space queue of objects that must be scavenged. When the 'scavenge' pointer catches the 'free' pointer, all to-space objects have been scavenged and therefore all live from-space objects evacuated. The modifications to Fenichel and Yochelson's algorithm required to yield Cheney's are reflected in the pseudo-code as follows:

- Function `evacuate()` replaces the Function `recursiveEvacuate()`, and simply copies a from-space object to to-space, no evacuation of its referents is performed.
- The Function `scavenge()` is responsible for processing the conceptual to-space scavenge queue evacuating referents in a breadth-first fashion.
- Function `iterativeSemiSpaceGC()` replaces the Function `recursiveSemiSpaceGC()` as the main routine of the garbage collector that invokes the flip and iterates across the roots initiating their evacuation.

Figure 2.3 depicts the state of from-space and to-space on completion of breadth-first evacuation of live objects to to-space and to-space scavenging.

The cost of a copying collector is the compelling motivation for its use and the primary reason for the widespread implementation of (hybrid) copying variants in the runtimes of many modern virtual machines, such as those for the Java and .NET platforms. Unlike the majority of mark-sweep and mark-compact collectors, a copying collector executes in a single phase operating on the live data, as opposed to the (multiple) entire heap scan(s). The cost is therefore proportional to the aggregated size of the live objects rather than the size of the entire heap. It should be apparent, from the pseudo-code above, that the implementation of

**Function** `evacuate`(*reference*) Evacuation of object pointed to by *reference*

**Input**: *reference* Pointer to object to evacuate to to-space
**Output**: *toSpaceReference* Pointer to to-space residing copy of object pointed to by *reference*

```
   /* If reference has already been evacuated and forwarded do nothing    */
 1 if reference ≥ toSpace and reference < heapLimit then
 2 |   toSpaceReference = reference;
 3 |   return toSpaceReference;
 4 Integer size = size of object pointed to by reference;
 5 Pointer fieldPointer = free;
 6 Pointer toSpaceReference = free;
 7 free = free + size;
   /* Simple field copy, no evacuation of references, actually implemented
      by bulk memory copying operation                                     */
 8 foreach field in reference do
 9 |   dereference(fieldPointer) = field;
10 |   fieldPointer = fieldPointer + 1;
   /* Set the forwarding address                                           */
11 dereference(reference) = toSpaceReference;
12 return toSpaceReference;
```

**Function** `scavenge` Iterative scavenging of to-space

```
 1 Pointer scavenger = toSpace;
 2 while scavenger < free do
   |   /* Breadth-first evacuation of references                          */
 3 |   foreach field in scavenger do
 4 |   |   if field is a reference type then
 5 |   |   |   dereference(field) = evacuate(field);
 6 |   |   scavenger = scavenger + 1;
```

**Function** `iterativeSemiSpaceGC` Iterative semi-space garbage collector

```
 1 flip ();
   /* Breadth-first evacuation of the root set                             */
 2 foreach root in roots do
 3 |   if dereference(root) is reference type then
 4 |   |   dereference(root) = evacuate(root);
 5 scavenge ();
```

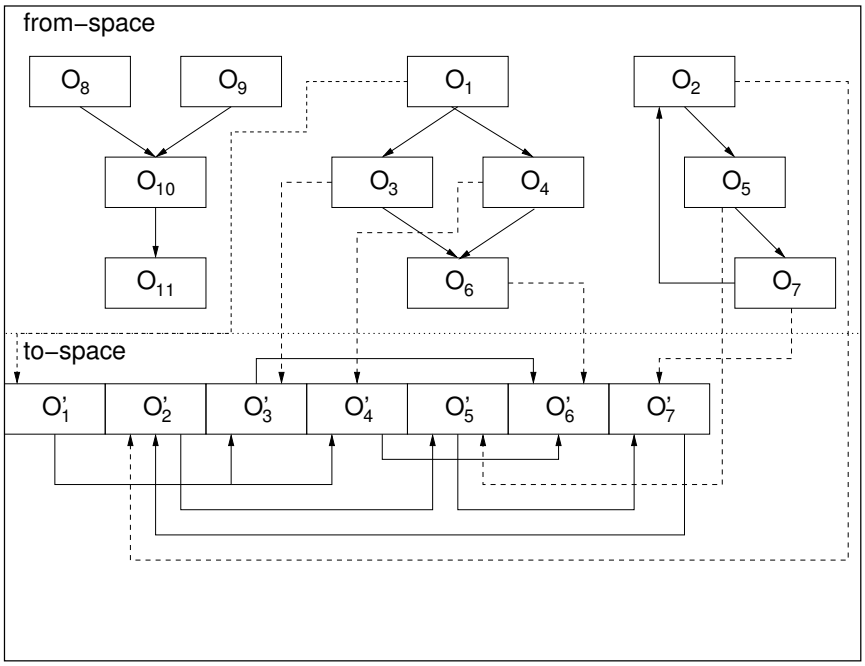



Figure 2.3: Breadth-first evacuation to, and scavenging of live objects, in to-space.

a stop-the-world semi-space copying collector is relatively simple when compared to the other forms of tracing collectors, that employ multiple phases to complete the collection cycle and restore the heap to a coherent state.

The main disadvantage of a copying scheme is that the size of the 'usable' heap is halved. However, on a virtual memory system, the from-space partition may be swapped to disk in-between collection cycles. There are however performance implications associated with this, for example, the costs of paging, especially at collector initiation when the flip occurs.

While Cheney's breadth-first copying algorithm is much more efficient than the Fenichel-Yochelson depth-first algorithm, the spatial locality benefits, in terms of cache performance are most likely reduced. As demonstrated in Figure 2.2, the depth-first ordering packs complete reference chains through to the leaf node contiguously, increasing the likelihood that the next dereference operation will be to cached data. However, the breadth-first, as depicted in Figure 2.3, operation packs child and then grandchild nodes contiguously reducing this likelihood as the distance between referrer and referent is increased.

If the lifetime of objects is long then the costs of copying the objects at each flip is expensive and can result in a long delay before the mutator can be resumed, i.e. an increased residency of live data sees a reduction in collector performance due to an increase in the amount of data copied. As a result, the majority of optimisations to copying algorithms focus on reducing the copying effort and employ heuristics based on object lifetimes, connectivity or other such demographics. For example, *large objects* — objects whose size exceeds a specified threshold,

may undergo only the scavenging process and not the evacuation process. The large object is effectively pinned in memory and the cost of copying is traded for the segregation of the large object space and the potential fragmentation that may result. In the following sections various collector architectures are discussed that exploit one or more of these object demographics.

### 2.4.1 Generational Garbage Collection

An object demographic that is particularly effective in reducing the amount of data copied and hence the duration of the stop-the-world pauses is that of object age. In order to utilise the age to optimise collection, the majority of objects must support one of two possible hypotheses. In the first, Ungar's *weak generational hypothesis* [Ung84], "most objects die young". If this is indeed true, then reclamation effort should focus on the collection of young objects, since they are the most likely to be garbage. Conversely, in the *strong generational hypothesis*, the older the objects are, the more likely they are to become garbage — as a result reclamation effort should concentrate on those objects that have lived the longest. Several researchers [DB76, FF81, Ung84, Moo84, App89, Zor89, Sew92, SP93, Wil94, Hay91, BZ93b] have confirmed the weak hypothesis, primarily in declarative and object oriented languages, while little evidence that the strong hypothesis holds exists.

*Generational* garbage collection exploits the weak generational hypothesis implementing policies that focus on more frequent collection of younger objects. For this reason, the technique is sometimes referred to as *ephemeral* [Moo84] (short-lived) garbage collection. As a copying variant algorithm, the heap is divided into a number of ordered partitions known as *generations*, where the *youngest* houses newly allocated objects. Live objects are *promoted* — copied into the next generation, at first from the youngest generation, and then subsequent generations, as they age. Thus each generation contains objects of a particular age range. What differentiates one generational garbage collector from another is usually the number of generations, the criteria used to determine the age of the objects, the *tenuring* or promotion policy, the frequency at which differing generations are collected and the intra-generation collection algorithm employed.

When a generation becomes full, the collector is invoked. Using the root objects in the older generations and pointers to younger generation objects, it collects objects no longer in use. A collection policy is employed that determines the frequency at which the generations are collected. Generally, the younger generations are collected more frequently than the older ones — with the focus on the most likely garbage objects. The collection algorithms used are implementation dependent and it can often be beneficial to use different algorithms to collect different generations, so that the dynamic properties of the specific generation being collected are exploited [San92]. The premise of the weak generational hypothesis, combined with the ability to collect generations independently of one another, results in mutator pause times, each of which, is less than that of a collector that pauses for a full collection and a lower *mark/cons ratio* — fewer objects are traced and copied and the quantity of data moved over

the entire program execution is reduced. The mark/cons ratio is the ratio of the total amount of data copied by the collector to the total amount of newly allocated data. It is a measure of work done by a copying collector, the implication being that the higher the ratio, the longer the algorithm will take, since it must do more copying per unit of memory allocated.

Lieberman and Hewitt [LH83], Ungar [Ung84], and Moon [Moo84] are all credited with significant contributions to generational algorithms. Both Lieberman and Hewitt [LH83] and Moon [Moo84] discuss generational and ephemeral collection within the context of Baker's incremental copying collector, which we describe Chapter 3. Ungar is credited with the first stop-the-world generational algorithm.

Consider a generational collector configured with two generations, and the memory available for heap allocation as contiguous. The heap is divided into two contiguous, segregated partitions, one for the young generation or *nursery* and one for the old generation. The mutator performs fresh allocations from the nursery. When the nursery is full, a *minor* collection occurs. This is a Cheney style copying collection except there is no 'flip', the live data is immediately promoted into the old generation. The nursery is now empty and available for re-use. Eventually, after subsequent minor collections, the old generation will become full and a *major* collection that collects the *entire* heap must be performed — the equivalent, in pause time, of a full stop-the-world semi-space collection. Ungar's collector used a mark-sweep algorithm to collect the old generation, but inplace compacting algorithms have also been widely implemented. The advantage of using a compaction algorithm in the old generation is twofold. Firstly, where old generation objects have long life times, coupled with increasing residency, repeated object copying is avoided. Secondly, there is no need for a to-space or *copy reserve*. Unfortunately a generational algorithm that employs such an immediate promotion policy does not perform particularly well. The young generation objects need a chance to age and therefore die off in order to avoid needlessly filling up the old generation. Ungar further splits the young generation into a to-space and a from-space and performs Cheney-style copying collection between the two until objects have survived sufficient minor collections and are promoted. When are the number of minor collections deemed 'sufficient'? Many criteria can be used in determining object age from the age of the object, in time, since its creation, to the number of bytes of heap space allocated or the address of the object itself. Care must however be taken to ensure that the age recording mechanism for each object does not needlessly waste space. If the object has a header then bits may be reserved that are set according to the number of collections for which the object has survived.

A technique that is widely employed to achieve both the efficient ageing of objects and efficient utilisation of memory is to split the young generation into *tenuring steps*, *buckets* or *ageing spaces*. Objects are allocated to the *eden* area and are then aged between each step, bucket or space at each minor collection before being promoted into the next generation. If only one ageing space is employed along with the eden (as originally described by Ungar) then for objects to survive multiple scavenges, an age recording mechanism for each object must also be employed. If multiple spaces are used, then the age of the object is implicitly

represented by its address. There are several tradeoffs that must be balanced:

- Multiple ageing spaces require careful arrangement so as to minimise the size of the copy reserve required for a minor collection.
- Age recording by anything other than an object's address carries additional space overhead.
- The fewer the ageing spaces and generations the less chance an object has to age and die off before promotion and the amount of *tenured garbage* increases. Tenured garbage arises from the promotion of live objects to the old generation that then soon die and hold objects live in the nursery at minor collections. This results in excessive copying which drives up the mark/cons ratio of the collector.
- The sizing of the generations also contributes significantly to the presence of tenured garbage. If the young generation (and ageing spaces) are too small, then frequent minor collections occur that result in rapid promotion of objects and increased probability of tenured garbage. However, if they are sized to big then minor collection pause times are unnecessarily increased.

**Pretenuring and Thread-Local Heaps**

The use of ageing spaces and adaptive tenuring techniques provide effective mechanisms for the performance tuning of generational collectors and have been studied in [UJ88, Sha88, Wil89, Zor89, UJ92, BZ93a]. *Pretenuring* is a technique that can significantly reduce the mark/cons ratio and therefore workload of a generational collector by avoiding the copying of objects altogether. Either static and/or dynamic run-time analyses are employed to determine whether an object, a type of object, or objects allocated at a particular callsite, should be newly allocated to the young generation nursery or directly into one of the older generations. The success of the technique must balance the reduction in copying with the cost of the analyses (especially if dynamic callsite profiling is used) and the potential creation of tenured garbage. Various pretenuring techniques are presented and evaluated in [Han90, BZ93b, SZ97, CHL98, Har00, AG00, BSH$^+$01, GM04, HSaC04, JBM04, RS05, MJR07]. Although many researchers present results showing reductions in collection times of as much as 30% for their benchmark suites, the effectiveness is application specific. It is for this reason that few production collectors implement pretenuring as standard. A related set of (static) analyses to those used for pretenuring are those used in determining whether objects can be allocated purely on the stack or within *thread-local* heaps. A thread-local heap is a heap in which only data 'visible', and therefore in use by, its associated thread is allocated. Locally allocated data may carry less overhead with respect to mutator-mutator and mutator-collector thread synchronisation. All other data is allocated in a shared heap. Escape analysis is most usually employed in order to determine whether the data 'escapes' the context of the thread or is

local to it. [GS98, CGS+99, GS00, Ste00, DKL+02, Kin04, JK05] report techniques for, and the effectiveness of, thread-local heap allocation.

### Appel's Generational Collector

A collector of notable design is Appel's fully copying generational collector [App89] that adopts a two generation configuration. It was devised to enable fast allocation, therefore utilising bump pointer allocation in all generations (as opposed to free list allocation and mark-sweep collection in the old generation). Two generations are used so as to maximise the size of the young generation, therefore reducing the likelihood that a young object is ever copied. For Standard ML of New Jersey, the environment in which the collector was originally implemented, this is a very effective strategy — the majority of data is immutable and modifications therefore require fresh allocations resulting in rapid turnover of young generation heap memory. The challenge then is how to effectively arrange the generations so as to avoid having to reserve half of all memory for a major collection. The nursery and old generation are laid out in virtual memory so that the old generation may grow to consume half of the available heap space while the nursery expands towards an inaccessible page of memory. When an allocation from this page is attempted a hardware interrupt is generated that when handled triggers the minor collection cycle. The old generation and nursery are separated by a copy reserve region that is always the same size as the nursery. The minor collection cycle immediately promotes all live data to the old generation, allocating out of the copy reserve, and the old generation therefore grows towards the nursery. A major collection is invoked when the old generation consumes just under half of the available heap space (to guarantee termination) and immediately follows a minor collection. The newly promoted region of the expanded old generation remains untouched - it is not copied, although it is rescanned for live objects. The remainder of the old generation that is live is copied to the nursery side of this newly promoted block of objects and allocated next to it. This now forms the contiguous old generation. It is however displaced from the start of the heap, so it must be block moved back again and all pointers adjusted by the displacement.

### Write-Barrier Synchronisation

Generational collection can be viewed as a 'halfway house' between single phase stop-the-world collectors and fully incremental and concurrent collectors. The goal of reduced mutator pause times is the same, both must handle the tradeoff between more frequent smaller collections and the overhead that results from increased 'context' switches between collector and mutator. Furthermore, both require synchronisation mechanisms between mutator and collector to provide a coordinated view of the live memory graph. Incremental and concurrent collectors are able to achieve further reduced pause times, but face additional overheads from more stringent synchronisation.

The ability to independently collect generations complicates generational collector imple-

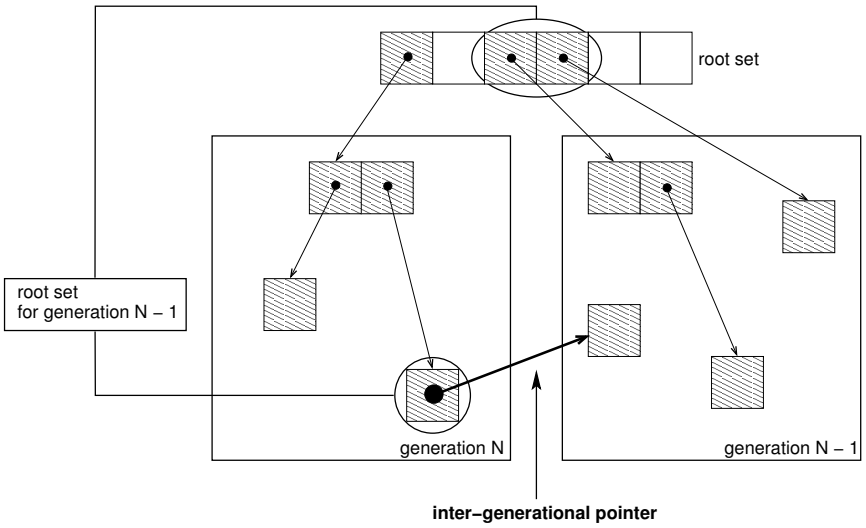



Figure 2.4: Two generation configuration with inter-generational pointer.

mentations and is arguably their main disadvantage (aside from to-space reservation for the generation(s) undergoing collection). Determining the root objects within a generation is more difficult than determining the roots of the entire heap. Not only must the registers and stack be scanned but *inter-generational* pointers must also be found and treated as roots — the mutator must synchronise and coordinate this information with the collector. Figure 2.4 shows two generations spanned by one such pointer. These pointers are created either by the promotion of an object with references into an older generation, or by storing in an object a reference to an object in another generation.

The easiest way to locate inter-generational pointers is to traverse the entire live memory graph from the root set at a minor collection, evacuating only those objects that are young generation objects and merely following references from older generation objects. However, not only is this inefficient, but the benefits of depth-first Cheney-style scavenging are lost and older generation objects must be marked as having been visited. Furthermore, traversing the entire memory graph degrades the performance benefits usually associated with sizing the nursery so that it fits in the cache — the cache is dirtied as old generation objects are dragged through it. Instead, the most widely adopted method of locating inter-generational pointers is to record them as they are created. Two overheads are associated with the introduction of inter-generational pointers, the first is that the collector must keep track of them when promoting objects and the second is the need for a *write-barrier* to trap pointer assignments by the mutator at time of storage. The use of a write-barrier for every pointer store would be extremely expensive, but the cost is substantially reduced as local variable stores need not be trapped since they already belong to the root set. In addition, it is common to store only older-to-younger [Ung84] generation pointers, not younger-to-older. When a generation

is then collected, all generations 'younger' than it, must also be collected. As a result, shorter pause times are traded for the overhead imposed on the mutator by the write-barrier.

Inter-generational pointers can be recorded as they are created by the mutator in several ways, each of which trades the amount of space required to record the existence of a pointer against the cost of the write-barrier and the work the collector must do when subsequently locating and following that pointer at collection time.

Lieberman and Hewitt's collector is the first collector to address this issue through the use of *entry tables*. Each generation has an associated entry table, except for the oldest. On creation of an old-to-young generation reference the address of the younger generation object is placed in the younger generation's entry table. The old generation reference points to the table entry, not to the young generation object itself. These younger generation references are therefore accessed by following an indirection through the entry table. Furthermore, for multiple referrers of a young generation object, there are multiple entries in the table unless the table is searched as each entry is made. For these reasons, no modern collectors adopt this approach, instead they track address of the older generation referring object. This approach of *remembered set* recording of inter-generational pointers was introduced by Ungar. With each generation a remembered set is associated, except for the youngest. The write-barrier traps pointer store operations, verifies the store is a reference to a younger generation object from an older generation object and places the address of the old generation object in the remembered set. A bit in the header of the old generation object is then set to indicate that the object has already been added to the remembered set. There is no need for the indirection associated with entry tables, nor are there duplicate entries. However, a bit must be available in the object header and (large) objects with multiple fields of which a few are the referrers must be scanned each time by the scavenger in order to find the actual old-to-young generation reference. A *slot remembering* write-barrier therefore records the address of the referring field (slot) itself instead of its parent object. Of course, there is now the issue of duplicate entries, unless there is a bit available in each inter-generational pointer.

The use of remembered sets and entry tables requires space to record the entries. Space may be reserved as a fixed size array or table outside of the heap. The problem then is when there is no available space for further entries in the table or array. A simple resolution is simply to invoke a major collection at the point at which the table becomes full. A more practical alternative is to allocate the remembered set in the heap, although care must be taken in placement of the remembered set so as to minimise reduction in 'useful' heap space. Alternatively, *sequential store buffers* (SSBs) can be used [HD90, HMS92]. The write-barrier is unconditional — it does not verify the store is an inter-generational reference, nor does it test for duplicates. Appel [App89] first employs this technique and lets the collector follow only the relevant references. The data structure of an SSB is a fixed size array of $2^i + k$ entries. It is operated as a circular hash table. The address of the old generation object or slot is hashed to obtain $i$ bits with which the array is indexed. If the index is empty, the entry can be made, similarly if it already contains an entry for the item no further action need be taken.

Otherwise the next $k$ slots are examined to find the next free slot. If it is still not possible to place the item then a circular search of the array is made. The array is kept relatively sparse by growing it whenever a circular search is required and 60% or more of the table's slots are full. If the SSB cannot be expanded and overflows then entries are made into the youngest generations SSB and then during collection are moved to the appropriate generation's SSB.

Because the write-barrier can be expensive, a technique that has often been used in its implementation is to exploit the hardware trap by the operating system of a write to a protected page of virtual memory. The pages of the heap are write-protected and the first write to a clean page results in the page's *dirty bit* being set. The page is then unprotected so that all subsequent writes are 'for free'. At collection time, those pages with dirty bits set are scanned for inter-generational pointers [Sha88, BDS91]. A refinement of this technique that reduces the effort in scanning an entire page is to split a page into multiple *cards*. A bitmap, the *card table*, has a bit associated within it for each card. The pages are never unprotected. On trapping the write the appropriate bit within the card table is now set. The cards are usually split so that a fast mapping function can be applied to determine the card table bit for the occurring write. The tradeoff in these two approaches is the cost of scanning an entire page versus the cost of the hardware trap handler invocation for all writes. This second approach is generalised in a more practical software implementation of the write-barrier, known as *card marking* [Sob88, WM89, Ḧ93]. The technique is no longer dependent on virtual memory page protection however the compiler must be modified to emit the barrier code for each store operation. The heap is divided as standard into generations but also smaller contiguous regions known as cards. The software write-barrier then sets the appropriate bit for the card in the generation's card-table.

### 2.4.2 Multiple-Partition Collectors

Throughout the remainder of the section we describe algorithms that are on the whole simply variations in the architecture of multi-generation collectors. They attempt to redress generational 'deficiencies' by collecting partitions in alternative orderings or according to alternative age heuristics or object demographics.

#### Large Object Space Collection

The majority of production copying collectors not only recognise the benefits on performance that can be achieved by segregating objects according to object age, but also the performance gained by segregation according to size. Most usually, heap objects that exceed a size threshold are allocated to a segregated, contiguous heap area known as a *large object space* (LOS), that employs an alternative collection algorithm to the rest of the copying collected heap. It should be clear, that the larger an object, the greater the cost in copying it. For generational collectors where large objects are allocated and collected as standard within the main heap, these copying costs can be exceptionally punitive when excessive copying occurs

as the result of being held live by remembered set references. Large object spaces are either collected using a mark-compact algorithm, or the objects are 'pinned' remaining unmovable and may be mark-sweep collected or reference counted. As a result they undergo only the scavenging process rather than both evacuation and scavenging in order to copy any standard heap objects they reference. There is very little literature devoted purely to large object space collection [HHMN98], instead LOS collection strategies are discussed in passing when describing complete collector implementations.

#### Leveled Garbage Collection

Generational collectors perform at their best when i) objects die young, and more specifically before they are promoted to an older generation; ii) when those objects that are promoted then live a long time; and iii) the majority of pointers are from young generation objects to older generation objects, so as to minimise the remembered set. Operationally, minor collections reclaim the majority of memory required for ongoing execution, and major collections are infrequent. If the assumptions of i) and ii) are violated then time is wasted copying objects during promotion. If iii) is violated, minor collection cycles are longer due to a larger root set and garbage may be promoted because of remembered set pointers from dead objects in the older generations.

*Leveled Garbage Collection* [Ton97] (LGC) is a collection heuristic that attempts to maintain a balance between freeing space in the nursery (the young generation or allocation area) and avoiding eager promotion so that objects may remain there to age and thus hopefully die for reclamation by a minor collection. There is however, much less emphasis on object age. The operation of a leveled garbage collector is most easily described for a two-level partitioning of a contiguous heap, but it generalises easily to more.

A contiguous heap of size $H$ is split into two contiguous areas, the *top level*, of size $S$, and the *bottom level*, of size $M = H - S$. The top level is the nursery or young generation, in which all new allocations occur. The bottom level not only plays the role of the older generation but also to-space. The parameter $F$ is a threshold setting that defines the amount of free space that must be available in the top level on completion of a collection cycle. On allocation failure, a minor collection, a hybrid of mark-sweep and Cheney's depth first copying algorithm, is initiated. Mark-sweep collection proceeds with a marking traversal of the top level from the root set. When the space consumed by the live objects marked so far totals, but not exceeds $S - F$, all subsequent objects marked as live are immediately evacuated to the bottom level to ensure that the free space reservation of $F$ will be satisfied. However, if the amount of allocated data in the bottom level, $m$, is such that there is not enough free space available to house the maximum amount of live data that could be promoted at a minor collection, $F$, then a major collection is initiated i.e. $m > M - F$. The major collection algorithm uses mark-compact to arrange the live objects contiguously in the bottom level. At the end of a major collection, heap expansion occurs in order to ensure that there is enough

free space for the next minor collection, i.e. $m \leq M - F$. Clearly, when $F = S$, the leveled collector behaves as a generational collector with an immediate, eager promotion policy.

In their evaluation of their leveled collector the authors of [Ton97] find that while the leveled collector performs at least as many minor collections as an eagerly promoting generational collector, it makes fewer major collections and furthermore that virtual memory performance is better. It is however worth noting that they shy away from evaluating a generational collector which tenures objects within a generation before promotion.

#### Older-First Collection

Stefanovic et al. propose a classification for the generalisation of age-based collectors and introduce *(Deferred) Older-first* garbage collection that delays the premature promotion of the very youngest objects [Ste99]. Generational collectors are therefore classified as *youngest-first* collectors. The rationale is that although generational collectors operate on the premise of the weak-generational hypothesis where the youngest objects have been observed as having the greatest mortality rates, high mortality does not necessarily equate to 'most likely garbage'. It is not the *mortality* function but the *survivor* function that indicates the probability that an object is live — at age 0, the value of the survivor function is always 1. Of course, those of us that work with generational collectors are acutely aware of this, and our production collectors tenure, thereby ageing, the object within a generation prior to promotion. Stefanovic et al. are suggesting however, that this is not enough. To achieve better performance, the focus should be *only* on those objects that are most likely to be garbage, not these *as well as* those that are least likely. With a standard generational collector this occurs because the younger generations than the one under consideration are collected along with it. Recall that this is done so as to reduce the remembered set sizes and eliminate the need for a write barrier during object allocation. It is based on previous observations that the majority of pointers are from younger to older objects because most writes occur during object initialisation. However, in their study of Smalltalk and Java benchmarks they find that only about 35% of pointers are in this direction and furthermore, neither direction is particularly dominant.

Stefanovic et al. view the heap as an age ordered (left: oldest, right:youngest), list and classify age-based collectors according to either the *true age* or *renewal age* of objects. True age based collectors timestamp objects at allocation. The relative (left-right) order of objects within the heap, with respect to the timestamp, never changes throughout program execution. For renewal age collectors, objects surviving collection have their timestamp updated as if they were newly allocated objects. They therefore move these objects to the young (right) end of the age ordered list.

A *true youngest-first* collector collects a subsequence of the rightmost true age ordered objects. A *true oldest-first* collector collects a subsequence from the leftmost ordered objects. A *deferred older-first* collector collects a subsequence of objects immediately to the right of the survivors of the previous collection. Collection therefore sweeps a collection window to

the right end of the ordered list. Once the right end is hit, the window is repositioned at the left end. A *deferred younger-first* collector operates in a similar way, however, it slides its collection window to the left end of the age ordered heap. These collectors can be further classified depending on whether they employ a fixed or variable size collection window. A *renewal youngest-first* collector is equivalent to a true youngest-first collector. However, a *renewal oldest-first* collector collects a leftmost subsequence of objects appending them to the right end of the renewal age ordered heap. Stefanovic et al. find that the renewal oldest-first collector effects a mixing of the true object ages such that excessive object retention (of garbage objects from region-crossing pointers recorded in the remembered set) occurs and object locality with respect to mutator access is destroyed.

The evaluations of older-first collection [Ste99, SHB+02], find that for Java, the throughput of the older-first collector is higher, with a lower mark/cons ratio, than for fixed-size and Appel variable-size nursery generational collectors. Furthermore, for the majority of benchmarks, the total execution time of the older-first collector is less and minimum mutator utilisation is higher.

There are two main drawbacks in the use of the older-first collector. Given that objects that do not die young tend to live longer, the older-first collector may perform redundant work through the excessive copying of old objects. Furthermore, because it does not collect the entire heap, it does not guarantee that all (unreachable) garbage will be reclaimed i.e. it is not *complete*.

#### Connectivity-Based Collection

Hirzel and Hind investigate the relationship between a Java object's connectivity (and the objects it references) and the lifetime and deathtime of the object [HHDH02]. They classify connectivity according to i) connectivity from stack locations, ii) connectivity from global variables, and iii) connectivity from heap allocated objects. Connectivity is analysed both directly and transitively. They find that there is strong correlation between connectivity and object lifetime and deathtime. In particular, objects reached from stack locations are usually shortlived; objects reachable from global variables tend to be immortal; and finally, that objects forming a pointer reference chain usually become garbage at the same time, regardless of direct or transitive referencing.

Following this investigation Hirzel et al. developed a new family of connectivity-based garbage collectors [HDH03]. These collectors exploit potential connectivity characteristics, arising from the above analyses, to segregate objects into disjoint heap partitions that can be more independently collected than in a generational context, and does so without requiring write barrier application. *Partition maps* are associated with each partitioning so as to be able to determine in which partition an object resides — often this may be implicit in the address of the object.

The partitioning must be both:

1. *conservative* — If it is possible for a pointer to exist between two heap objects, then the objects must reside within the same partition or the partitions in which the objects reside must be marked as having a dependency. A partition dependency graph with edges between the two dependent partitions is maintained. This graph is a directed acyclic graph.

2. *stable* — If two objects reside in the same partition, then they must continue to be located together for as long as they live.

Connectivity-based collectors choose a subset of the partitions to be collected when a minor collection is required. When a partition is selected, all the partitions that precede it in the partition dependency graph must also be collected. Partition selection occurs using two functions. The *estimator* function estimates the number of live and dead objects in a partition. The *chooser* function selects the set of partitions to collect based on these estimates working to collect sufficient garbage at minimal cost. An example estimator function may define a survivor function where, if an object in a partition is reachable from a global variable, then its survival rate is 90%, while if it is reachable from a stack location it is merely 10%. For heap objects an estimator function may determine object liveness according to an inverse exponential function of the average age — over time, more objects are likely to become garbage. Partitioning can be based on: allocation time; whether an allocation occurring at one program callsite subsequently references an allocation occurring at a second callsite or vice versa — both callsites now allocate to the same partition; and even the type of the objects referencing one another at these call sites. In this case, strongly connected components in the type graph are aggregated into a single dependency graph node in order to create the partitionings.

Hirzel et al. evaluate various connectivity-based collectors within a simulated environment (as opposed to complete JVM implementations). They find that many of the collectors outperform an Appel variable-size generational collector with respect to both pause times and memory footprint. The ability to collect partitions with greater independence than within the generational context provides a finer collection granularity that reduces the number of major collections required. Furthermore, the problem of excessive object retention, where objects in older generations prolong the liveness of an object in a younger generation, is not an issue for this family of collectors.

### Mark-Copy Collection

Sachindran et al. [SM03] present the *mark-copy* algorithm that extends Appel's standard generational collection algorithm in order to provide at least equivalent performance while reducing the memory footprint of the executing program.

Minor collections employ the standard generational copying, tenuring and promotion algorithms. However, the major collection algorithm is modified to use explicit marking in conjunction with copying. The heap layout of a two generation collector is modified only

in the old generation. The old generation is split into $n$ fixed size windows, where the size of each window is the smallest increment in the old generation that can be collected. The windows are numbered monotonically in the range $1..n$, with lower numbered windows being collected first. Unlike a major generational collection where half the available memory must be reserved for to-space copying, a mark-copy collector requires a space reservation of the size of a single old generation window. A write barrier is used to trap writes to objects in the nursery.

On initiation of a major collection, the entire live memory graph is traversed and live objects explicitly marked. During the marking process, the total amount of live data in each old generation window is calculated and a remembered set for each window constructed. These per-window sets record pointers from higher ordered windows into lower ordered windows. With the marking phase complete, the copy phase proceeds, however, it is split into a number of passes, with each pass copying a subset of old generation windows. Because the remembered sets are uni-directional the windows must be copied in order from $1..n$. Furthermore knowledge of the amount of live data in each window allows multiple windows to be collected together in a single pass even though the copy reserve is the size of a single window. On completion of each copying pass the collector unmaps those pages associated with the windows that have just been copied thereby limiting the amount of virtual memory in use by the collector.

The major benefit of mark-copy collection is that a major collection occurs only when the available free heap space is the size of a single old generation window while a standard generational collector invokes such a collection when the heap occupancy is at 50%. When operating in the same amount of heap space therefore, a mark-copy performs fewer major collections and therefore benefits from increased throughput. The disadvantages are clear, the mark-copy collector scans every object twice in order to evacuate an object, firstly in marking, and then during copying. Furthermore, the explicit marking of an object incurs a space overhead, requiring a mark bit be available within the object or be made available in a separate bitmap. Most significantly, an old generation of $n$ windows can require up to $n$ passes of the root set (including stack), although the authors claim it is in general much less.

#### The Beltway Framework

The Beltway garbage collection framework [BJMM02] is a toolkit, developed originally for the Jikes RVM Java virtual machine, that generalises the implementation of existing copying collectors and caters for the construction of hybridised copying algorithms. The Beltway framework is able to generalise copying algorithms by exploiting the fundamental premises upon which copying collectors are based but also by combining them all into a single framework:

- Most objects die young — The weak generational hypothesis
- Delay old object collection — Objects that do not die young tend to live longer suggesting that old generations should be collected less frequently.

- Youngest objects must be given time to die.
- Incrementality increases responsiveness — Performing a collection cycle in bounded sized increments increases the responsiveness of the collector.
- Increased locality through copying — Depth first copying can increase referential locality by placing objects that reference one another close together, thus reducing cache misses and even eliminating paging costs.

The conceptual architecture of Beltway collectors is simple, and is based on two key types of components. An *increment* is a contiguous region of memory that can be collected independently of other memory regions (and therefore increments). A *belt* is a first-in-first-out (FIFO) arrangement of increments which are collected in this strict FIFO order. Hence the name 'Beltway' — belts are analogous with conveyor belts. Belts are arranged in levels, with the first level belt most usually acting as the nursery and then each subsequent level above as an older generation. Belts fill from the left to the right and objects are promoted to higher level belts. During collection the belts 'roll' to the left as each increment is collected in FIFO order. For example, a simple semi-space collector consists of a single belt containing two increments, each of which is sized to half the available memory. Clearly the increments represent from-space and to-space and the effect of the belt rolling to the left at each collection results in the flip with from-space and to-space swapped. An Appel style generational collector is modelled by a Beltway configuration of two belts, the lower belt being the young generation nursery and the upper belt being the old generation. The lower belt contains a single increment, while the upper has two. The rightmost increment of the upper belt is the to-space increment or copy reserve. Allocation occurs to the lower belt and object promotion during a minor collection evacuates objects to the upper belt. When both the lower increment and the leftmost upper increment are full and all of the available memory, except for the copy reserve, consumed, a major collection is initiated. Semi-space collection of the old generation occurs with the from-space and to-space increments 'rolling' for the flip. Live objects in the lower increment are promoted to the upper belt to-space increment. By focusing collection on the lower belt the weak generational hypothesis is exploited and excessive copying of older objects is avoided. By adding further increments to the belts, tenuring increments can be added that allow objects to age before promotion. In turn, this gives youngest objects time to die. With collections occurring in increments, the responsiveness of the user program is increased. Finally, by allocating objects that reference one another together in the nursery and promoting or copying them to the same increments, referential locality is preserved.

It should be clear from the above examples that the Beltway architecture is flexible and can generalise the implementation of all copying collectors, indeed Beltway configurations for older-first and connectivity-based collectors are easily arranged.

Beltway is *not* an incremental collector in the sense that collection and mutator execution are interleaved. It gains increased responsiveness in a similar way to a generational collector,

where it is able to stop-and-copy at most a single increment before resuming the mutator. A truly incremental collector would allow the mutator to be resumed while the increment is being collected. Major collections will collect a subset of all of the belts, and the number collected in a stop-the-world fashion will determine the effect on the responsiveness of the collector. Like the older-first collector certain Beltway configurations and collection policies will result in incomplete configurations where not all the garbage can be reclaimed. Cyclic garbage that spans increments is eventually reclaimed if the belts/increments are considered together or the promotion policy guarantees that eventually the cyclic structure will span a single belt or reside completely within an increment. Blackburn et al. present a Beltway configuration that extends the Appel generational collector with the tenuring increments as above. This is not a novel configuration, indeed our GHC generational collector has such tenuring steps. The novelty is provided by collecting a single increment in an old generation belt, not the entire belt. This collector *Beltway.X.X*, where 'X' specifies the maximum increment size, suffers the problem of completeness outlined above. As a result, they introduce 'Beltway.X.X.100' which rectifies this problem by adding a further belt that can grow to consume all the available memory which is then collected in its entirety like a standard generational major collection.

Like all collectors that allow memory regions to be collected independently of one another (to a certain extent), Beltway uses a standard generational write barrier and maintains remembered sets for each increment and/or belt. Beltway uses *frames*, contiguous power-of-two aligned virtual memory regions that contain an increment. Like the older-first collector, an integer identifier is associated with each frame, so that frames can be collected in a designated order. The frame alignment allows cheap determination of an inter-frame pointer, while such a reference need only be recorded in the remembered set of the frame if the frame will be collected sooner than the frame from which the reference originates.

In order to provide flexibility in collection policy, Beltway provides a range of *collection triggers* that allow collection to be initiated at times other than heap exhaustion. For example, a *remset trigger* initiates collection when a remembered set exceeds a certain capacity — by avoiding lengthy remembered sets which are treated as roots to an increment collection, the collection pause for the increment can be limited and the responsiveness of the mutator maintained.

Like the mark-copy algorithm the ability to collect increments in a finer granularity than a generational collector reduces the memory footprint of the collector — the dynamic copy reserve can be smaller (sized to the largest increment), and greater throughput achieved. The disadvantages of completeness, and potential remembered set sizes are the same. In their evaluation of Beltway, against a standard Appel generational collector within Jikes RVM, Blackburn et al. are able to reduce execution time by up to 40% in similar sized heaps and by 5 to 10% in tighter heaps. The greatest benefit of the Beltway framework is in its enablement for greater copying collector design space exploration.

## 2.5 Region-Based Memory Management

In Section 2, the motivation for dynamic memory allocation and garbage collection over lexically scoped stack and static allocation was described. It should be apparent that garbage collection is no panacea for the problem of efficient deallocation and memory reclamation — we have briefly described some of the tradeoffs and disadvantages. In summary, garbage collection often adds significant overhead to the wall clock execution time of a program and complexity to the predictability of the program's run-time characteristics. In addition, reachability as a liveness criteria means that the collector may keep objects alive that are no longer needed. *Region-based* memory management [TT97, TBE+97, Tof98] provides a compromise between garbage collection and stack allocation — it is not as flexible as general purpose garbage collection but more so than fine grained allocation within procedure activation records that reside on a program stack.

This region-based paradigm partitions the heap into *stacks of regions* — segregated areas of memory often managed according to the standard stack discipline. The size of a region cannot always be determined at the time of its allocation (variable sized structures and recursive data types may ultimately reside there). As a result, only the regions whose size can be determined, can be allocated on and managed by a hardware stack. If a region is not stack allocated, or at least managed on an explicit stack with push (region creation) and pop (region deallocation) operations determined by lexical scope, then most likely, it will be managed by a garbage collector. Several criteria may be used to determine when to allocate a new region and which objects are allocated to a particular region. Regardless, when a region is deallocated, the objects it contains are simultaneously deallocated.

# Chapter 3

# Real-time Garbage Collection

Over the course of the following chapter we introduce real-time garbage collection, survey the state-of-the-art and examine in detail those algorithms that are fundamental to our work.

The primary goal of real-time garbage collectors is to reduce and strictly bound mutator pause times, ensuring sufficient mutator progress, such that real-time deadlines are met. Consider mutator execution to be a series of equal length time slices within each of which the mutator must execute a bounded amount of work. The real-time deadline is met if the mutator successfully completes the workload within the associated slice. In the presence of garbage collection, a time slice not only consists of a mutator workload, but also a collector workload. If pause times are bounded, then the time slice deadline must be adjusted to incorporate the maximum mutator pause time that occurs while the collector runs — in practice it is incredibly difficult to determine when the maximum pause due to collection will occur and so all pauses must be assumed to be this worst case.

For a stop-the-world collector the maximum pause is simply the time taken to perform the worst case collection cycle. In Chapter 2 the complexity of basic algorithms was discussed with the best performing ordered in the size of live data. The worst-case pause therefore occurs when the heap consists solely of live data. Such an entire heap collection for gigabyte sized heaps can take many seconds, which is unacceptable for interactive applications and real-time systems where sub-millisecond deadlines are not uncommon.

In order to reduce these long stop-the-world pauses, techniques are employed that interleave mutator execution with collector execution throughout the duration of a single collection cycle. These increments of collection work are then scheduled within the real-time time slices. In interleaving mutator and collector execution such that real-time deadlines are met, two key challenges must be addressed:

**Coherency** The mutator must manipulate a *coherent* view of the live memory graph and must therefore be synchronised with the collector to ensure correct operation. The collector 'hides' intermediate graph re-arrangement states that could result in erroneous mutator be-

haviour were the mutator to 'observe' them unchecked. Section 3.2 introduces the various mechanisms that enforce this synchronisation, each of which does so to varying degrees. The strength of synchronisation is a key determiner of a collector's performance characteristics — strong coupling between mutator and collector can result in serialisation bottlenecks, while weaker coupling allows finer grained 'parallelisation'.

**Scheduling** How is the amount of work that the collector must perform in a time slice determined? The problem is that the allocation rate of the mutator is a dynamic run-time property of the running application. Furthermore, it is usually dependent upon the inputs with which the application has been parameterised. As a result, without historical data, it cannot easily be predicted ahead of time. The collector increments must therefore be scheduled to ensure that the collection cycle has completed before memory is exhausted, so as not to trigger a new collection cycle. In the simplest case, if the collection cycle is not completed, the program must terminate with an out of memory error. A common solution to this problem is to adjust the collection rate, the collector workload at the next time slice, in reaction to changes in the allocation rate. It should be clear that if the collection rate exceeds the allocation rate then the cycle is guaranteed to terminate before memory is exhausted. However, during bursty allocation such an approach is in direct conflict with the mutator's requirement to meet its time slice deadline — either termination from memory exhaustion is risked, or the deadline may have to be missed. Section 3.3 revisits these issues in greater detail.

## 3.1 Real-time Terminology

### 3.1.1 Mutator Utilisation

If the rate of collection is adjusted with the rate of allocation then, for some scheduling policies, on occurrence of bursting allocation, the collection rate must be increased to compensate, resulting in an increase in the duration of collector pauses. The most likely consequence is the violation of real-time deadlines — *mutator utilisation* (or mutator progress), the proportion of the time slice for which the mutator runs, is low as individual pauses 'bunch' up as the result of increased allocation rate, effectively aggregating the pauses into one much longer pause. It is therefore insufficient simply to analyse pause time distributions when reasoning about the real-time capabilities of a collector. In addition, the *minimum mutator utilisation* (MMU) must be considered. Cheng and Blelloch introduce this measure of MMU for the study of their parallel soft real-time garbage collector for SML [CB01]. The MMU exposes the amount of time for which the mutator runs in a time window of specified size. The time window is sized, maximally, as the duration of the entire program execution and then reduced in increments to a size where the minimum utilisation is zero, i.e. the collector is on for the entire duration of the window size. Current research, ours included, strives to develop a collector that maximises

the MMU in the time window regions of between 1 and 10 milliseconds.

### 3.1.2 What is 'Real-time'?

We presented a simplified view of real-time event scheduling where the mutator must complete its workload within a fixed sized time slice. In reality, there is not necessarily a notion of such a 'time slice', instead, events occur which must be completed within a deadline relative to the time at which the event was initiated.

A real-time system falls into one of two categories:

- Hard real-time — Violation of real-time deadlines cannot be tolerated, exceeding deadlines results in catastrophic system failure.
- Soft real-time — Deadline violation is tolerable, the system can continue to operate 'correctly' as long as the workload is eventually completed. Until workload completion, the system may operate in a degraded state — most usually reduced performance or transactional throughput.

Existing literature on real-time garbage collection often misrepresents the term 'real-time', claiming that the collectors under study are real-time without consideration as to whether they are *hard real-time*, *soft real-time* or merely just *incremental*.

The categories are as follows:

- Incremental — The majority of the literature describes work-based incremental collectors where: Mutator and collector execution are interleaved and (average and) individual pause times may be bounded (by a small constant). However, there is little or no reasoning about the worst-case pause and minimum mutator utilisation, nor can guarantees be made about the extent to which real-time deadlines may be met. Real-time deadlines will most usually be violated through low mutator utilisation as individual pauses 'bunch' up as the result of increased allocation rate. Such a system is most usually employed in interactive environments to enhance responsiveness, the most significant benefit being that the user does not perceive a sloth-like system where they are 'frozen out' while a stop-the-world collection pause occurs.
- Soft real-time — The worst-case pause time is rigorously bounded; the minimum mutator utilisation is acknowledged to be so low that some (schedule of) deadlines will not be satisfied especially if memory constraints are imposed. A soft real-time collector may employ heuristics that trade violation of time deadlines with violation of memory bounds, i.e. the heap is grown so that the deadline need not be missed. The fundamental constraint that a real-time collector must satisfy is that: collector operations that block the execution of a mutator thread must consume no more than $x$ milliseconds within any $y$ millisecond execution window [Det04]. A soft real-time collector will meet this constraint with a high probability, but may occasionally violate it.

- Hard real-time — The worst-case pause time is rigorously bounded; the minimum mutator utilisation guarantees that, for whatever event schedule, all deadlines will be met and furthermore, the collector guarantees to operate within defined memory constraints (bounds). A hard real-time collector will guarantee to always meet the fundamental constraint described above.

### 3.1.3 Incremental, Parallel and Concurrent Classifications

In addition to the lack of clarity with which collectors are described as real-time, a lack of standardised terminology throughout the literature leads to further confusion as to how (each operating phase of) a 'real-time' collector is implemented. For simplicity, consider a 1:1 thread-processor model where a single thread is bound to a single processor and the processor executes that thread and that thread only. There may be several processors present. We adopt the following classifications:

- Incremental — A single thread (and by proxy, a single processor, regardless of the number of processors) performs mutator execution interleaved with increments of a single collection cycle.
- Concurrent — The mutator and collector execute concurrently in separate threads. On a dual processor system, the mutator runs in one thread on one processor, while the collector runs in another thread on the other.
- Parallel — The collector is multi-threaded and its threads run concurrently with one another. Note that this classification is concerned with the collector's concurrency with respect to its own threads, and not that of mutator-collector concurrency. If the mutator threads are paused while the collector threads run then it is a 'parallel stop-the-world collector'. Similarly if the collector threads are able to run concurrently with the mutator, then it is a 'parallel concurrent collector'. The collector may parallelise an individual collection phase (for example, the mark phase) between multiple processors, it may parallelise multiple phases for concurrent execution between processors (for example, the mark and sweep phases), or it may do both.

In particular, the literature confuses the 'concurrent' and 'parallel' classifications and they are used interchangeably. Furthermore, the notion of 'incrementality' has been implicitly included / excluded when describing parallel collectors.

## 3.2 Mutator - Collector Synchronisation Techniques

We have briefly mentioned the mutator-collector coherency problem that is inherent to interleaved mutator and collector execution. It should be apparent that the collector's architecture,

whether it be incremental, concurrent, parallel or a hybrid of all three will determine the granularity of mutator-collector and collector-collector synchronisation. A collector that strives to be classed as real-time must be capable of *active* synchronisation. In a *passive* environment the mutator is unable to perform work on behalf of the collector — it must block and yield to the collector. At the extreme, this is the synchronous mode of operation under which a stop-the-world collector runs. Active synchronisation enables the mutator to execute work on behalf of the collector and allows much finer grained asynchronous execution. This is simply another way of saying the collector must be incremental! However, this active / passive view of synchronisation hints at an abstraction that can be used to model all types of collector, not just incremental collectors.

### 3.2.1 The Tricolour Marking Abstraction

The *Tricolour marking* abstraction is commonly used to reason about correctness with respect to the mutator and collector view of the live memory graph (and therefore the synchronisation required to enforce a coherent view), the objects that are deemed to be live, and those that are most definitely dead. Although Dijkstra et al. devised tricolour marking for reasoning about their concurrent collector [DLM$^+$76], as we have explained above, it is suitable for reasoning about stop-the-world collectors too.

Nodes in the graph can be coloured in one of three ways, hence 'tricolour'. Consider a depth-first tracing collector. Preceding initiation of a collection cycle, all objects are viewed as potentially live and are coloured white. As the collector traces out the live memory graph it colours the nodes grey as it reaches them in the forward direction towards the leaves. The leaf nodes are coloured black. In the reverse direction, a grey node is coloured black when all its children have themselves been visited and therefore coloured black. The collection cycle terminates when all reachable nodes have been blackened. These black nodes are the live objects in the system, all other nodes are unreachable, remain coloured white, are dead and can be reclaimed. Breadth-first tracing differs slightly since cells are marked black in the single advancing traversal, as soon as all their immediate children have been pushed onto the mark stack, which implicitly colours them grey. Regardless of the traversal method, unscanned objects are segregated from fully scanned black ones by a 'fence' or 'fringe' of grey nodes (objects undergoing scanning) — depth-first, the fringe advances from top to bottom, left to right, while breadth-first, it advances left to right, top to bottom. Clearly, black nodes are no longer under consideration by the collector and we can reason about both progress and termination by examining the movement of the fringe towards the terminal leaf nodes of the graph.

### 3.2.2 The Tricolour Invariants

A system that ensures that no black nodes refer to white ones, except via intermediate grey ones, enforces the *strong tricolour invariant*.

The intermediate state in which a cell is implicitly coloured grey in a stop-the-world collector is not particularly interesting. However, when reasoning about incremental collectors and their associated invariants it is essential. In an incremental, concurrent or parallel collector hybrid it is possible for the mutator to create a reference from a black object to a white object. Recall that the black object is no longer under consideration by the collector. If the white object is not reachable on a path from a grey object, then this white object will never be blackened; instead it will be treated as garbage and collected at the end of the cycle. Unchecked, this violates the correctness of the user program. The collector must therefore be 'notified' of this graph mutation and recolour the black node grey or colour the white node grey. Notification occurs via a *barrier* synchronisation mechanism that allows the collector to maintain the coherent view of the live memory graph that we have previously discussed.

While recolouring the black node grey or colouring the white node grey allows the strong tricolour invariant to be maintained the synchronisation technique employed can be costly. An alternative invariant, the *weak tricolour invariant* allows an edge to exist directly from a black node to a white node as long as there is a path, (a set of edges), from a grey node to that white node that does not contain an edge to an intermediate black node. This weak invariant allows a potentially cheaper synchronisation mechanism to be employed, but is much more conservative in what it deems to be garbage. Under the strong invariant, if an edge leading to a white node is broken, resulting in the white node becoming unreachable, then the object it represents is garbage and can be reclaimed within the *current* collection cycle. However, under the weak invariant, the collector's view of live object reachability is much more conservative — *conceptually* a *snapshot* of the graph at the beginning of the collection cycle is taken. As a result, objects that die within the current collection cycle become *floating garbage* that cannot be reclaimed until the subsequent cycle. However, the weak invariant allows a much looser granularity of mutator coupling — the barrier synchronisation mechanism can be omitted when the edge from a black object is destroyed and a new one created as a reference is overwritten. The object has already been traced and so its edge was previously to a grey or black object. As a result, it could not be an edge of a path from a grey node to a white node that did not contain an intermediate black node. Furthermore, the overwriting reference must be either an existing reference within the graph or a newly allocated object. If it is an existing reference then the snapshot ensures that it will be traced via its original referrer. If it is a new allocation, then steps must be taken to ensure that the object is allocated black. For other algorithms, a newly allocated object can be coloured in any one of the three ways, and this is most usually determined by the barrier employed — this is further discussed in Section 3.5.

### 3.2.3 Barrier Synchronisation Mechanisms

There are broadly two types of mutator-collector barrier synchronisation techniques:

***Read barrier*** When the mutator attempts to dereference a pointer to a white object, on a pointer load, steps are taken to immediately colour the object grey before the mutator is allowed to manipulate the dereferenced object. As a result, the mutator sees only black or grey objects, and since it can never obtain a white reference, it can never write a white reference into a black object. The read barrier therefore enforces the strong tricolour invariant. It should be apparent that this is a fine grained barrier because it is executed on every pointer load, which on the whole, is a very frequent operation.

***Write barrier*** When the mutator attempts to perform the write of a pointer into an object, the write is trapped. There are two types of write barrier, the first of which enforces the strong tricolour invariant in an *incremental update* algorithm. When the barrier traps the pointer write operation, if the object is black, it is reverted to grey by recording its address so it can subsequently be rescanned by the collector. The edge is now between a grey and a white, grey, or black node, not between a black and a white node. A more aggressive version of the barrier may leave the node black and immediately colour the new referent grey. Recall that the mutator must not a) write a reference to a white object into a black object *while* b) destroying the original reference to the white object before the collector has 'seen' it. The second type of write barrier is employed in a *snapshot-at-beginning* algorithm that enforces the weak tricolour invariant by ensuring that b) can never happen. By *conceptually* taking a snapshot of the graph at the beginning of the collection cycle, the collector is subsequently able to visit this live white object. Of course, the topology of the graph need not be captured at the beginning of the collection cycle, instead the write barrier must simply record the reference that is being overwritten (as grey) so that the collector may subsequently trace and blacken it.

The choice of barrier mechanism is predominantly determined by the desired granularity of mutator-collector coupling; the resulting conservatism in object reachability; the dominating pointer operation — whether it be pointer loads or pointer stores; and most importantly the time and space costs of the barrier. It is worth noting that much of the literature ignores, or fails to analyse the space costs associated with the choice of barrier. In our work, we pay particular attention to these space costs. Intuitively, a barrier incurring a space overhead reduces the amount of memory available for heap allocation which ultimately results in more frequent collector invocation and a lower mutator utilisation. In Section 3.5 we discuss these issues in detail with reference to the algorithms surveyed.

### 3.2.4 Mutator Colour

Reasoning about the correctness of algorithms that employ mutator-collector barrier synchronisation techniques using the tricolour marking abstraction can be further aided, through extension, by considering the colour of the mutator itself [Pir98]. In analogy with object colour, Pirinen classifies a black mutator as a mutator whose 'state' has been processed by

the collector such that it need not be examined again. Similarly, a grey mutator is a mutator whose state requires (re)processing by the collector. The state of the mutator is essentially the root set that is presented to the collector at initiation of a collection cycle. The tricolour invariants apply to a coloured mutator through corresponding analogy — for example, a black mutator may hold references to both black and grey objects but not white objects, under the strong invariant.

Pirinen presents [Pir98] the following classification of black and grey mutator barrier techniques as complete — the minimal set that enforce the tricolour invariants across all possible algorithms that employ them. He reasons that all other techniques are variations derived from these techniques through combination, deferral, or elaboration, resulting in doing more than is necessary, although perhaps implemented more efficiently. It is instructive to refer back to, and relate this classification to, the fundamental real-time algorithms of Section 3.4.

Grey mutator techniques enforce the strong tricolour invariant using incremental update techniques all of which employ an *insertion* write barrier — a barrier that fires on the insertion of an object reference to prevent white pointers from being written into black objects. A major effect of mutator colour on the algorithm is its effect on termination. Because the mutator is grey, a read barrier is unnecessary, and the granularity of mutator-collector synchronisation is therefore looser and coarser. This looser synchronisation comes at a price — the black wavefront that is advancing from the root of the mutator's live memory graph to its leaves may retreat as black objects are reverted to grey ones. Algorithmically, the graph may be rescanned multiple times from the roots, using 'mini' collection cycles that are shorter (thus leading to termination), to ensure that all grey objects have been blackened.

The algorithms in which this trade-off in the loosening of mutator-collector synchronisation against a number of additional termination cycles is made, most usually facilitated through the use of a write barrier, is most evident in Section 3.4.2 describing Brooks' collector; in Section 3.4.3 detailing the replicating collectors of Nettles et al. and Cheng; in the description of the collectors of Pizlo et al. in Section 3.5.8; and in our own hybrid of Brooks' collector described in Section 7.5.

Pirinen's grey mutator technique classification follows:

- Steele's write barrier [Ste75] retreats the wavefront by reverting a black source object to grey on trapping the write of a white object reference into it. Steele's algorithm is described in Section 3.4.6.

- Dijkstra et al.'s write barrier [DLM$^+$76] advances the wavefront by colouring a white object grey as it becomes the referent of an object that has already been coloured black. This technique is described further, in the context of Dijkstra et al.'s algorithm, in Section 3.4.4.

- Boehm et al.'s write barrier [BDS91] is a coarser variant of Steele's and can either advance or retreat the wavefront as it colours the new referent grey regardless of its

actual colour. The barrier is implemented using the virtual memory manager's hardware page protection mechanism (see Section 2.4.1) that sets the dirty bit of a protected page as it is written to. The barrier therefore operates at the granularity of a page as opposed to an object. At termination, the dirty pages must be rescanned.

Black mutator techniques may enforce either the strong tricolour invariant using a read barrier that prevents the mutator from obtaining references to white objects; or the weak invariant using a snapshot-at-the-beginning algorithm that employs a *deletion* write barrier. The write barrier fires when an object reference is modified so as to prevent loss of reachability of an object that was live when the snapshot was taken. Recall that under the weak invariant, the mutator may still reference white objects. This is still the case for a black mutator, however, what defines it as 'black' is that its 'state', the original root set, has already been scanned and coloured black and need never be revisited. Algorithmically, termination is generally simpler than for a grey mutator and the shorter terminating collection cycles are, most usually, unnecessary. The compromise however, is the tighter coupling of the mutator and collector using the finer grained read barrier as its synchronisation technique.

Pirinen's black mutator technique classification follows:

- Baker's read barrier [Bak78] advances the wavefront by colouring the referent of a grey object. The algorithm and this technique are discussed in detail in Section 3.4.1.
- Appel et al.'s read barrier [ELA88] is a coarser variant of Baker's and, like Boehm et al., uses the virtual memory manager's page protection mechanism to trap read (as opposed to write) access to a grey protected page, colouring all its objects black once they have been scanned (scavenged). This technique is briefly described in Section 3.5.2.
- Abraham and Patel [AP87] and Yuasa [Yua90] independently devised the deletion write barrier that traps reference update to either a white or a grey object and colours the old referent, that is being replaced, grey. It should be clear that of all the techniques, this is the most imprecise because it retains the old referent that has become unreachable as a result of the update operation. This floating garbage cannot be collected until the subsequent collection cycle. Yuasa's algorithm is described in detail in Section 3.4.5.

## 3.3 Scheduling for Real-time

There are two scheduling challenges within a garbage collected real-time system. We have briefly introduced the first of these, where mutator execution can be viewed as an ordered schedule of events each of which must be completed within its associated deadline. The second, is that increments of a collection cycle must be scheduled in such a way that these deadlines are not violated, i.e. the mutator makes the necessary progress. There is further complication however — not only must the mutator make sufficient progress, but the collector

must also make sufficient progress to enable it to complete the collection cycle before memory is again exhausted and initiation of the next collection cycle required.

In truly concurrent and parallel collectors, where the collector executes asynchronously with respect to the mutator, a collection cycle is often initiated using heuristics that aim to prevent allocation of a new object from ever failing, as opposed to when an allocation has failed. This provides greater flexibility in the scheduling of the collector. This is not, however, the case for incremental algorithms: collector initiation most usually occurs at allocation failure and the collector attempts to perform only enough work so that the allocation request may be fulfilled and sufficient mutator *and* collector progress is made.

Throughout the remaining section we introduce three key approaches to collector scheduling.

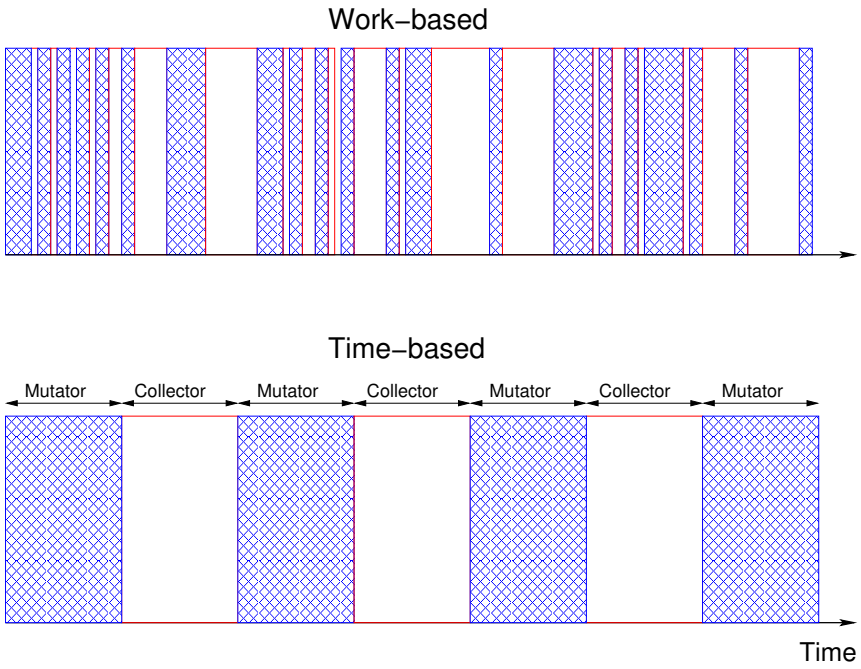



Figure 3.1: A work-based versus a time-based mutator/collector schedule

### 3.3.1 Work-based Scheduling

A *work-based* approach to solving this problem ties the collection rate to the allocation rate — for each unit of memory allocated, a unit of collection work, $k$ (the "mark/cons" or "allocate/mark" ratio), is performed. If the allocation rate increases so the collection rate increases to compensate. If it decreases then the collection rate decreases. $k$ is dynamically determined when collection is initiated in order to ensure that the live memory graph has been fully traversed before the available memory in the heap is exhausted and the next collection cycle is

required. By tying the allocation rate to the collection rate this is guaranteed. For a tracing collector, what is the tracing rate, $m$, that guarantees this? If there are $L$ live words in a heap of size $H$ at collector initiation, then the lower bound on $m$ is determined by:

$$m = \frac{H - L}{L} \tag{3.1}$$

where $L$ words are traced during the allocation of $H - L$ new words. In practice, the value of $L$ is unknown at the beginning of each cycle and so must be estimated by a function that uses the value of $L$ accurately computed at the end of the previous cycle.

In terms of the tricolour model, the assumption is that all new allocations are black allocations and are not traversed by the collector. It is worth noting, that if $L = H$ then the entire memory graph of $L$ words must be traced immediately in a non-incremental stop-the-world fashion. Furthermore, at the end of tracing, no memory has been made available for re-use by the collection cycle. The user program must terminate with an 'out of memory' error.

With a tracing rate of $m$, at least $k$ words of tracing must therefore be performed at each allocation, as determined by:

$$k \geq \frac{1}{m} = \frac{L}{H - L} \tag{3.2}$$

This technique of interleaving mutator and collector execution, by scheduling the collector with a workload of $k$ at each allocation was devised by Henry Baker [Bak78]. We revisit Baker's collector in Section 3.4.1. Siebert [Sie98] provides a detailed analysis for the amount of progress a copying collector must make for each unit of allocation based on the proportion of total memory in use as reachable objects by the user program. In addition, the upper bound on the maximum amount of allocated memory, as a function of the maximum reachable amount of memory is given. This allows an upper limit on the amount of free memory to be determined. The use of these analyses in combination allows collection time to be balanced against mutator memory requirements — the mutator may choose to expand the heap by allocating more memory rather than invoke the collector for too long when mutator progress requirements exceed a configured limit. Although this technique of effecting the collection rate based upon the allocation rate is both a simple and an elegant solution to incremental collector scheduling it is by no means the panacea. While $k$ ensures that the collection cycle is completed before the next one must be invoked *and* the amount of collector work at each allocation is bounded, there is still no guarantee that real-time schedules will be met. Unfortunately, the allocation rate of a program is seldom constant, and is often *bursty*. Furthermore, the time between two successive allocations is generally unpredictable and is often very short. If multiple allocations occur in quick succession, then the mutator may not make sufficient progress with respect to its real-time schedule — the collection pauses effectively aggregate into one much longer pause within the mutator time slice.

The inconsistent and unpredictable mutator-collector scheduling of a work-based collector

is depicted on Figure 3.1. Although such a collector is simpler to implement than its time-based counterpart, the disadvantages resulting from the disruptiveness of the collector to (minimum) mutator progress should be clear.

### 3.3.2 Priority-based Scheduling

Henriksson et al. [Hen94, MH95, Hen96b, Hen96a, Hen97, Hen98] present a body of research that does not completely eliminate these, problems but attempts to minimise them and provide greater flexibility in their management. Their focus is on hard real-time environments that consist of (typically) a few *high-priority processes* and rather more *low-priority processes*, all of which are fixed-priority scheduled. High-priority processes must start *on* time and complete *in* time. For example, a high-priority process may implement control software that coordinates the opening of one valve and the shutting of others. The *low-priority processes* are soft real-time with more relaxed deadlines — they are still time critical but the effect of missing their deadline can be tolerated. Processes that interact with a user or render video images to a screen are most usually soft real-time.

Henriksson's strategy is to prevent the garbage collector from ever interrupting high-priority processes. The processes are scheduled using real-time scheduling theory. In [SAr+04] Sha et al. present a comprehensive survey of scheduling algorithms from a historical perspective. The task of a scheduling algorithm is to determine the best order in which tasks should be executed in order to meet their deadlines. The algorithm is usually rule-based and searches for an optimal schedule using a feasibility test known as *schedulability analysis*. The analysis may be performed online or offline. An offline algorithm, although less capable of dealing with dynamically created tasks allows an optimal schedule to be sought. An online algorithm must return a viable schedule in a short period of time and may therefore use a non-optimal greedy algorithm. Schedulability analysis considers the ability of the tasks of a schedule to meet their deadlines as well as the worst-case latency bound, a bound on the worst-case system response time.

Henriksson's approach uses priority-driven scheduling where the priority level of the process determines the schedule. There are various means for the assignment of task priorities. Tasks may be *periodic*, where their inter-arrival times are constant, or *aperiodic* — tasks that i) may be bounded by a minimum inter-arrival time; ii) have an unbounded inter-arrival time but are bounded by an event density limit; iii) have unbounded, irregular inter-arrival times and no density limit. For real-time scheduling, aperiodic tasks are usually limited to those tasks whose worst-case inter-arrival time guarantees that the task's deadline is met, and are often referred to as *sporadic* tasks. The most usual way to reason about aperiodic tasks is to schedule a periodic task that is responsible for the batch processing of any pending aperiodic tasks. A *deadline-monotonic* scheduling algorithm assigns periodic tasks with the shortest deadline-interval, i.e. relative deadline, the highest priority. In a slight refinement, a *rate-monotonic* algorithm assigns the task with the shortest relative deadline the highest

priority, where the deadline is equal to the task's period. These are static algorithms where the priority of the task does not change. Finally, an *earliest deadline first* (EDF) algorithm, a dynamic algorithm where priority may change during task execution, assigns the priority based on the absolute deadline of the task — the earlier the absolute completion time of the task, the higher its priority.

Henriksson's collector is scheduled incrementally with the low-priority processes (when no high-priority processes are running) using Baker's work-based allocation scheduling technique. However instead of Baker's conditional read barrier, a Brooks-style unconditional read barrier is used in conjunction with a write barrier in order to maintain the tricolour invariant (this technique is discussed in detail in Section 3.4.2). The collector evacuates objects as normal, copying them immediately to to-space, however, the write barrier employs *lazy evacuation*: the mutator reserves space for the evacuee in to-space, but defers object copying to the collector task. The incremental work package of the collector is sized in proportion to the memory allocated by the high-priority processes, and to a lesser extent the low-priority processes. As a result, the collector may lag behind the rate at which it must work in order to guarantee termination before memory exhaustion. A modified generalised rate-monotonic scheduling analysis [LSR94] is therefore used to determine the amount of memory that must be reserved for high-priority process allocation to prevent memory exhaustion. The authors refer to this mutator-collector scheduling as *semi-concurrent scheduling* because the collector is partially incremental and partially concurrent. Hard real-time guarantees are therefore met for fixed-priority process schedules. However, because the high-priority processes may starve the collector of CPU time a large heap may be required in order to ensure there is sufficient free memory in which to operate. Furthermore, for each real-time system that employs this scheme, the collector must be tuned using the scheduling analysis. Finally, this scheme is not suited to dynamic schedules such as EDF scheduling. The scheme is a practical approach to the scheduling problem but there are inherent assumptions based on the operating environment such as the ratio of high- to low-priority processes, the deadline durations and the amount of available memory.

Kim et al. [KCKS99, KCS00, KCS01] schedule the garbage collector as a *sporadic server* [Spr90] task assuming multiple mutator tasks to be periodic and rate-monotonically scheduled. The sporadic server is a high-priority task that services aperiodic tasks. Collector scheduling is aperiodic because its inter-arrival times can be bursty, unbounded, and are determined by the memory utilisation of the periodic tasks. Kim treats them as sporadic because they have a deadline that must be met to prevent memory exhaustion. The sporadic server can execute an aperiodic task at any time, providing it has enough capacity to do so. If it does not, it must re-execute the remainder of the task when the capacity is replenished (or defer the entire execution if it is not preemptable). Replenishment is scheduled based on the sporadic server's priority level and the amount of capacity it has left, if any. The sporadic server approach to task scheduling is shown to reduce the worst-case memory requirement over background scheduling strategies whilst incurring lower overhead than slack

stealing algorithms. In contrast to the background scheduling of Henriksson, Kim et al. assign the sporadic server the highest priority. Given that the amount of memory that must be reserved for allocation during a collection cycle is determined by the worst-case collector cycle length, the sporadic server reduces the worst-case memory requirement at the expense of a reduced number of feasible schedules. This limits the flexibility in task scheduling and often the number of higher priority tasks that may be scheduled. In a similar body of work Assche et al. [MVA05] schedule the collector as an aperiodic task serviced by a *polling server* instead of a sporadic server. The polling server is scheduled as a periodic task that executes aperiodic tasks during each period with limited capacity. The capacity, i.e. time slots, of the periodic server may be used for lower priority task execution if there is no pending aperiodic task. Unlike the sporadic server any unused capacity in a period is lost and the replenishment criteria is therefore much simpler — it happens at the next cycle. The priority of the polling server need not be the lowest nor the highest. The approach has greater flexibility in scheduling than that of Kim et al. at the expense of a larger amount of reserved memory. However, because the priority is not the lowest, the worst-case reserve is lower than Henriksson et al.'s collector.

### 3.3.3 Time-based Scheduling

Robertz [Rob02] builds on Henriksson's semi-concurrent scheduling model to assist the rate-monotonic schedulability analysis and avoid low-priority process starvation as a result of increased collector workload resulting from a high frequency of allocation events by high-priority tasks — recall that the collector task is scheduled at a higher priority than the low-priority tasks. Each individual allocation is classed as either *critical* or *non-critical* and critical allocations of lower priority processes may now be scheduled before non-critical allocations of higher priority processes. Essentially, prioritised sub-task scheduling is now performed. To achieve this, the amount of critical allocations for each (high-priority) process period must be calculated *a priori* at application design time. While the flexibility that results from scheduling allocations is of obvious benefit, a more interesting consequence is that the collection cycle is of a fixed length. Using the amount of memory allocated by high-priority processes at each period, and the amount of memory reserved for high-priority process allocation, the length of a collection cycle is calculated and expressed in terms of high-priority process periods. The collector can then be statically scheduled as a periodic task along with the other fixed-priority processes. The collector is now *time-scheduled* and is no longer coupled to nor driven by the allocation rate.

Robertz and Henriksson [RH03] further develop time-scheduled collection and semi-concurrent scheduling to adaptively schedule the collector process for use in dynamically scheduled EDF environments using *Constant Bandwidth Servers* (CBS) [AB98]. The isolation property of a CBS, combined with its bandwidth allocation policy which reserves a fraction of the CPU for a task, gives the appearance that each task is running on a ded-

icated server, albeit with a slower processor. Furthermore, in recognition that calculating the collection cycle length at application design-time based on critical high-priority task allocations may not always be practical, they present an adaptive on-line collector scheduler. They estimate the allocation rate of the application and then calculate the collector cycle's deadline — the time at which remaining free memory will be exhausted. In order to provide resilience to deviations from the estimated allocation rate and variations in the amounts of floating garbage persisting within each cycle, they calculate the amount of memory that must be reserved using a worst-case analysis — when all the objects that die within a cycle float until reclamation at the end of the *next* cycle.

The key idea behind Robertz and Henriksson's work is that by determining the collector cycle time and therefore a collector task deadline, the collector can be scheduled using standard scheduling techniques without consideration to individual increments, or *quanta*, of collector work — the generic task scheduler handles this. Furthermore, by scheduling the collector according to the invariant that the fraction of collector work performed must be greater than or equal to the fraction of the garbage collection cycle time that has elapsed, the collector makes progress at a well-defined rate, regardless of when memory allocation occurs.

The alternative to time-scheduled collection is *time-based* scheduling [BCR03b] that interleaves mutator and collector work increments using fixed time quanta. The mutator is scheduled to run for a time quantum of $Q_T$ seconds before being interrupted by the collector which then runs for $C_T$ seconds before being reinterrupted by the mutator. Interleaved execution continues until the collection cycle terminates. If a perfect scheduler is assumed, where the context switches occur after *exactly* $Q_T$ and $C_T$ seconds, then the minimum mutator utilisation for a time interval $\Delta t$ can be determined by Equation 3.3 [BCR03b].

$$u_T(\Delta t) = \frac{Q_T \cdot \lfloor \frac{\Delta t}{Q_T + C_T} \rfloor + max(0, \Delta t - (Q_T + C_T) \cdot \lfloor \frac{\Delta t}{Q_T + C_T} \rfloor - C_T)}{\Delta t} \tag{3.3}$$

$Q_T \cdot \lfloor \frac{\Delta t}{Q_T + C_T} \rfloor$ specifies the contribution of whole mutator quanta within $\Delta t$, while the remainder of the numerator specifies the contribution of the remaining partial quantum, if there is one. It is worth noting that, as the size of the interval becomes large, this expression reduces to Equation 3.4:

$$\lim_{\Delta t \to \infty} u_T(\Delta t) = \frac{Q_T}{Q_T + C_T} \tag{3.4}$$

Figure 3.2 graphs the MMU assuming the perfect scheduling of 10 millisecond time quanta for both the mutator and collector work increments.

Recall that for a work-based collector, the MMU for an interval of $\Delta t$ is determined by the program's allocation rate and that bursty allocation drives the MMU low. The MMU is most usefully an observed property of a work-based collector for a *specific* application. Without knowing the worst-case allocation rate of the application the MMU cannot be reasoned about. Regardless, such reasoning leads to a derived MMU that is often so low as to force the collector

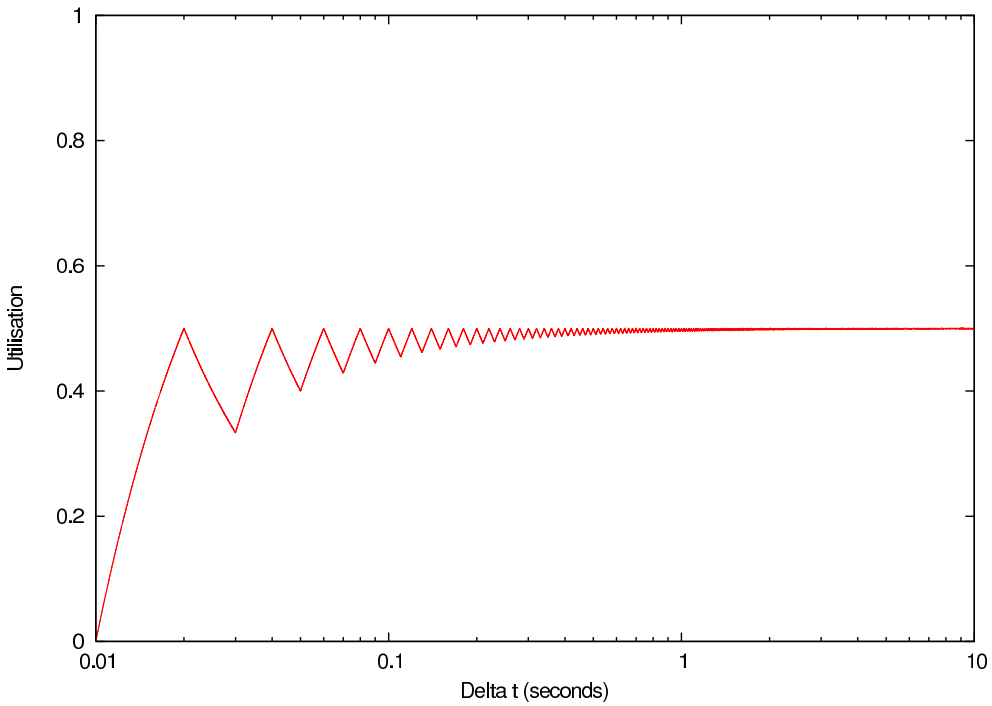



Figure 3.2: Time-based collector MMU with scheduling quanta of 10 milliseconds.

outside of 'real-time' classification. However, for a time-based collector, the MMU can be an active property, used to parameterise the collector. Unfortunately, the worst-case MMU, $Q_T$ and $C_T$ alone are insufficient parameters. Like Henriksson and Robertz's collectors, the decoupling of the collector from the allocator requires that memory be reserved for use when the collector lags the allocator. The size of the reserved memory is not only determined by the desired mutator utilisation characteristics, but also the collection algorithm employed. We discuss this issue further, along with the necessary runtime system parameters, in our consideration of Bacon et al.'s *Metronome*, presented in Section 3.5.7.

In defence of Baker's work-based collector, by adapting to an application's allocation rate it requires no complex analyses or application dependent parameterisations and very little tuning. As a result, for interactive and many soft real-time applications, the incrementality it provides can often be within acceptable bounds.

## 3.4 Fundamental Real-time Algorithms

Over the following section we describe the algorithms that have made fundamental contributions to, or have been key enablers of (soft) real-time, incremental, and concurrent garbage collection techniques.

### 3.4.1 Baker's Work-based Copying Collector

Baker's classic algorithm is an incremental variant of Cheney's semi-space copying algorithm that has previously been presented in Section 2.4. A read barrier is employed to trap mutator access to objects which have yet to be copied from the exhausted heap space to the new heap.

The memory available to the heap is divided into two partitions, *from-space* and *to-space*. from-space is the exhausted heap space where both live and dead objects reside before initiation of the garbage collector and from which the collector will copy the live objects. to-space is the new heap space where live objects are copied to, and from which new objects are allocated.

Like a stop-and-copy collector, the collector is initiated when the allocation of a new object fails due to insufficient free space of the current heap, to-space. The mutator is paused and the algorithm begins by performing a *flip* of the two partitions with from-space becoming the new to-space and to-space becoming the new from-space. Jones notes [Jon96] that this is not necessarily the best time to initiate the collector. Although the number of objects being copied is minimised by allowing the objects as much time as possible to die, it maximises the amount of heap allocated and, as a result, the number of page faults. If however the collector is run continuously, with the flip performed as soon as the previous collection cycle is complete, the probability of page faults occurring is minimised as live data is maximally compacted across fewer pages. Of course, care must be taken to prevent the overhead that would result were the two semi-spaces allowed to flip rapidly as a result of short collection cycles resulting from a low live data residency. Furthermore, there is a tradeoff between the overhead incurred from page faults and the increased execution time that would result from the continuous interleaving of mutator and collector.

Unlike Cheney's stop-and-copy collector, where the collector would now be run to completion, the whole root set is evacuated from from-space into to-space, but only $k$ objects are scavenged before the new object is allocated. Following the tricolour abstraction, these objects are now grey. Evacuation of an object entails copying the object to to-space, placing a reference to the copy on the scavenge queue and finally overwriting the from-space object with a forwarding pointer referencing the to-space object. Scavenging an object is performed by removing the object from the head of the scavenge queue and evacuating any (child) objects that it references. As we have previously noted, the scavenge queue is implicit in a Cheney configuration. The garbage collector now pauses and the mutator resumes so that progress can be made. The next time that a new object is allocated, the mutator is again paused, and the memory for the new object is allocated from to-space. Naïvely, the object is allocated as

grey and will undergo scavenging by the collector. However, with a little extra book-keeping, new objects can be allocated as black by arranging them at the opposite end of to-space to which evacuated objects are placed. This avoids having the collector unnecessarily scavenge new objects and also maintains any locality benefits that come with the breadth-first evacuation of child references. The collector now scavenges $k$ objects and the mutator is then resumed. In this way, garbage collection and mutator execution *appear* to be performed in parallel, with reduced mutator pause times, and garbage collection proceeding incrementally.

Because collection is performed incrementally it is possible for the mutator to resume with the heap in an inconsistent state, where objects which have yet to be evacuated from from-space (white objects) may be accessed by the mutator via references from scavenged (black) objects now residing in to-space. This is clear violation of the strong tricolour invariant. In order to prevent this, and provide the mutator with the illusion that collection is complete at the end of each collector increment (after $k$ pieces of work), a read barrier is installed to effect *all* pointer load operations. How does the read barrier maintain the invariant? Whenever the mutator attempts to load a field from an object to which it has a pointer, it must check whether the object has been scavenged and, if not, arrange to scavenge it first. The principal disadvantage of the scheme is the CPU overhead of these checks. Three main approaches have been used to implement the read-barrier.

**In software** The barrier is implemented as a conditional test on the address of the loadee to determine in which space it resides. Those found to be residing in from-space are immediately evacuated and a forwarding pointer installed. The final operation of the barrier code is to return the address of the grey to-space object. This involves inserting extra instructions to perform address range checks at each pointer-load in the program. It carries a typical overhead of 40–50% of the execution time of a program [Zor89].

**Using virtual-memory** This approach uses the machine's virtual memory protection mechanism to lock down grey copies of objects [AEL88]. Any attempt to access such an object causes a trap to the runtime system, the object is traced and blackened, the lock is removed and the mutator is resumed. The mechanism is operating system specific and so is not portable. The overhead is very sensitive to the trapping architecture of the system: an average range is around 13–63% [Zor89].

**In hardware** A small amount of extra hardware on a processor enables pointer-loads to be checked in parallel with normal execution [Joh88]. Again a trap to the runtime system occurs when the mutator trips over the barrier. Typical overheads with this approach are 9–11% [Zor89] although in all but a few customised hardware systems the option is simply not available.

The benefit of employing such a fine-grained barrier should be clear, garbage objects can always be reclaimed within a single collection cycle from the point at which they die — white objects (in from-space) that become garbage within the collection cycle before the scavenger

scavenges the grey objects that held them live at the *beginning* of the collection cycle can be reclaimed within *this* cycle. In the worst case, grey and black to-space objects (and in particular newly allocated objects) that die within this cycle cannot be reclaimed until the next cycle because of their to-space location. However, as previously discussed, the ability to eagerly reclaim these objects over a snapshot-at-the-beginning algorithm comes at the cost of tighter mutator coupling. We revisit this issue in our discussion of the Metronome (Section 3.5.7).

A read-barrier implementation offers a further potential advantage to the system [Cou88, Joh91]. With a read-barrier, objects are traced by two different mechanisms during a garbage collection. "Active" objects are traced by the mutator when it attempts to access them and trips over the barrier: "passive" (though of course live) objects are traced by the garbage collector itself when it is triggered by an allocation request. Thus active objects are naturally distinguished from passive objects and we can improve the dynamic locality of a program by grouping them together. This can reduce the paging costs of a program substantially, and can also improve its cache performance.

The diagrams below demonstrate the phases of a single Baker collection cycle. Notice how the algorithm handles cyclic references. The dashed lines represent the forwarding pointers, the dotted line separates the two semi-spaces, and the shaded boxes indicate those objects currently on the scavenge queue:

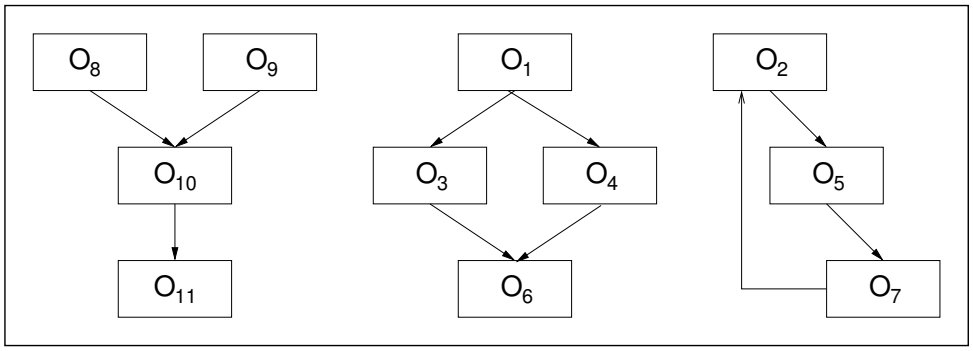



Figure 3.3: Heap state prior to initiation of Baker collection cycle: root set = $\{O_1, O_2.\}$

### Pseudo-code Algorithm

The following pseudo-code functions, describe the core functionality of Baker's algorithm. Function `mutatorPointerLoad()` outlines the read barrier code that effects pointer load operations. The barrier code is simply a call to a modified version of the Function `evacuate()` that was presented in Section 2.4. The Function `evacuate()` either copies the loadee object

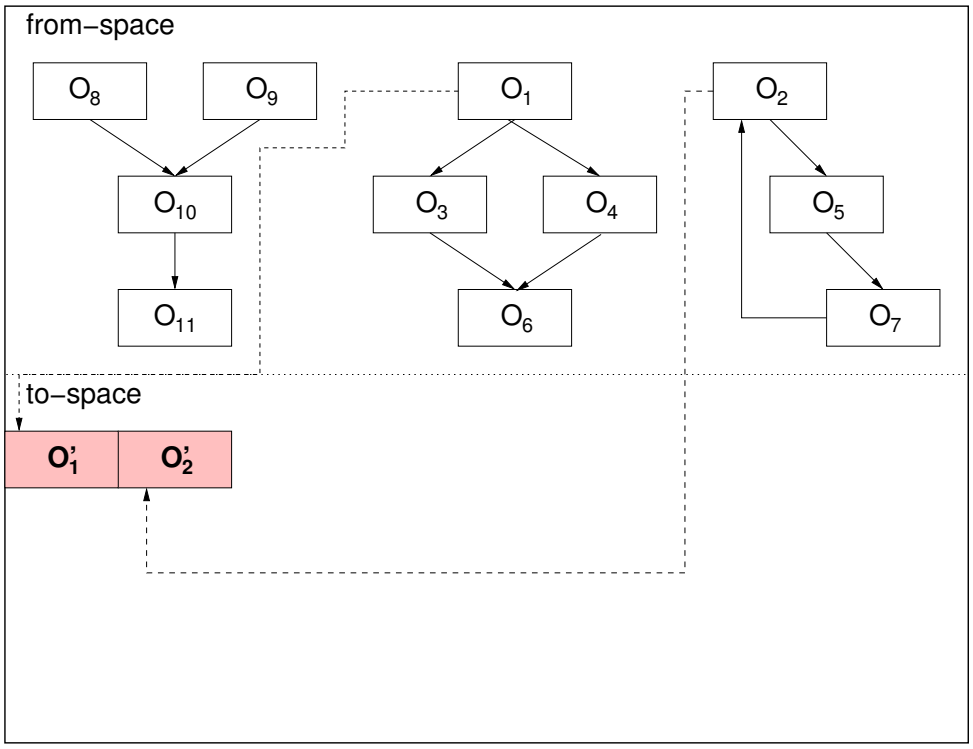



Figure 3.4: Semi-spaces after root evacuation.

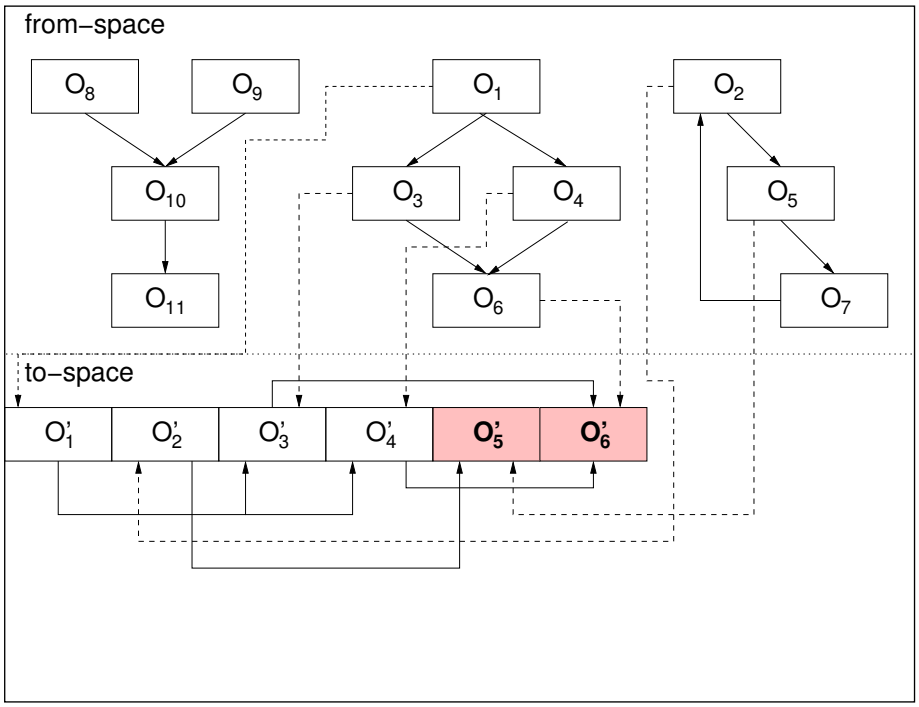



Figure 3.5: Semi-spaces after one call to Function `allocate()` and $k = 4$ scavenges.

to to-space and installs a forwarding pointer if the object resides in from-space and then returns this to-space address or simply returns the to-space address if it does not. Note that the `evacuate()` is used by both the mutator and the collector and while the *collector* may implement it as a standard function, the *mutator* barrier code is usually implemented inline using a macro for efficiency.

**Function** `mutatorPointerLoad(`$reference$`)` Baker's pointer load effecting read barrier

**Input**: $reference$ Pointer to loadee object
**Output**: $toSpaceReference$ Pointer to to-space residing copy of loadee object

```
1 Pointer toSpaceReference = evacuate (reference);
2 return toSpaceReference;
```

The Function `flip()` and the Function `evacuate()` are modified so that evacuated objects are copied to to-space locations whose addresses increase towards its end. The *scavengeLimit* pointer designates the next free address at which an object is to be copied to. The Function `allocate()` is modified such that newly allocated objects are allocated from addresses at the end of to-space and decrease towards the start. In this way newly allocated objects can be coloured black and need not undergo scavenging by the collector. Clearly, if the *scavengeLimit* and $free$ pointers collide, then memory is exhausted in spite of garbage col-

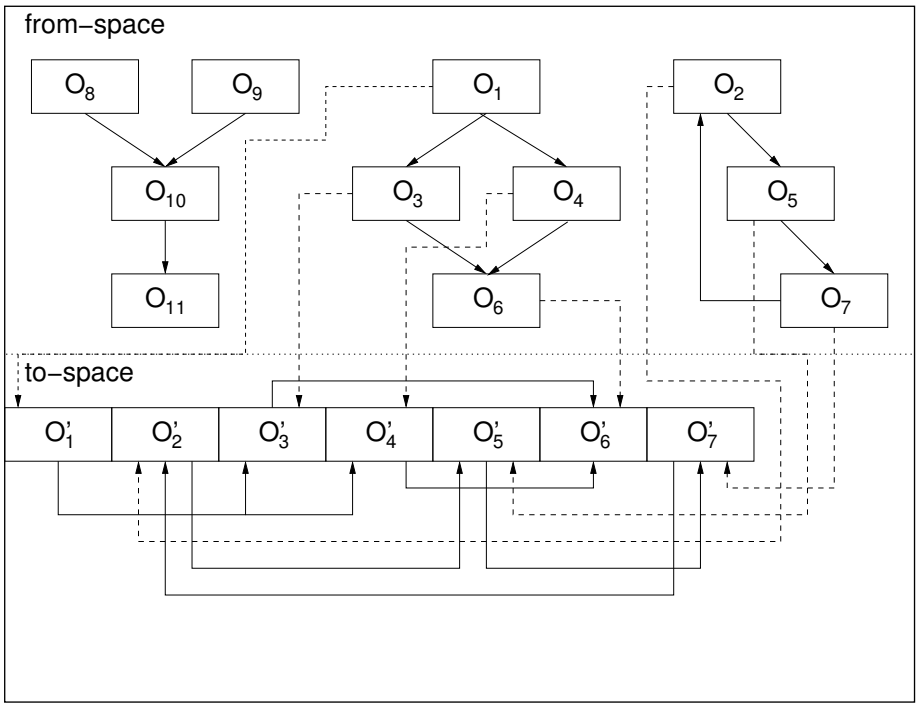



Figure 3.6: Semi-spaces after two calls to Function `allocate()`. Baker collection cycle completes before $2k = 8$ scavenges. The memory allocated to the from-space objects may now be freed.

lection.

**Function** `flip` Flip the Baker to-space and from-space

```
1 if toSpace == memoryStart then
2 |   scavenger = scavengeLimit = toSpace = semiSpacePartition;
3 |   fromSpace = memoryStart;
4 |   free = heapLimit = memoryEnd;
5 else
6 |   scavenger = scavengeLimit = toSpace = memoryStart;
7 |   fromSpace = semiSpacePartition;
8 |   free = heapLimit = semiSpacePartition;
```

**Function** `evacuate(`*reference*`)` Evacuation of object pointed to by *reference*.

```
  Input: reference Pointer to object to evacuate to to-space
  Output: toSpaceReference Pointer to to-space residing copy of object pointed to by
          reference
  /* If reference has already been evacuated and forwarded do nothing    */
1 if reference ≥ toSpace and reference < heapLimit then
2 |   toSpaceReference = reference;
3 |   return toSpaceReference;
4 Integer size = size of object pointed to by reference;
5 Pointer fieldPointer = scavengeLimit;
6 Pointer toSpaceReference = scavengeLimit;
7 scavengeLimit = scavengeLimit + size;
  /* Simple field copy, no evacuation of references, actually implemented
     by bulk memory copying operation                                     */
8 foreach field in reference do
9 |   dereference(fieldPointer) = field;
10|   fieldPointer = fieldPointer + 1;
  /* Set the forwarding address                                           */
11 dereference(reference) = toSpaceReference;
12 return toSpaceReference;
```

Function `allocate()` is further modified to perform $k$ pieces of work at each allocation if and only if the garbage collector is running. Naively, if the collector is running and an allocation request cannot be satisfied then the program is terminated with an out-of-memory error. A more realistic implementation will force the collection cycle to run to completion and then repeat the allocation request.

It is worth noting that there is no easy way for the collector to determine whether an object that it is about to scavenge within a $k$ work increment has already been scavenged by the mutator's read barrier. As a result all children of the encountered object must have `evacuate()` invoked on them. The Function `evacuate()` then determines whether the child object must actually be copied. These to-space checks incur an overhead that is somewhat

**Function** `allocate(`*size*`)` Dynamic object allocation

**Input**: *size* Integer size of object to allocate
**Output**: *ptr* Pointer to allocated object

```
1  free = free − size;
2  if free < scavengeLimit then
3      if garbageCollectorState == RUNNING then
           /* Failed to free enough memory to satisfy allocation despite
              in-progress collection                                    */
4          abort (Out of memory);
5      else
           /* Mutator is briefly paused while the incremental collector is
              initialised.                                              */
6          initialiseBakerSemiSpaceGC ();
7          free = free − size;
8  if garbageCollectorState == RUNNING then
       /* Perform k increments of work at each allocation              */
9      Boolean scavengeQueueEmpty = scavenge ();
10     if scavengeQueueEmpty == TRUE then
11         terminateBakerSemiSpaceGC ();
12 ptr = free;
13 return ptr;
```

**Function** `initialiseBakerSemiSpaceGC` Initialisation of a Baker semi-space collector

```
1  calculateK ();
2  flip ();
   /* Breadth-first evacuation of the root set                         */
3  foreach root in roots do
4      if dereference(root) is reference type then
5          dereference(root) = evacuate(root);
```

**Function** `scavenge` Iterative scavenging of to-space. $k$ increments of work are performed.

**Output**: *scavengeQueueEmpty* Boolean indicating that the scavenging of to-space is complete

```
1  Integer kIncrement = 0;
2  while scavenger < scavengeLimit do
       /* Have k increments of work been performed                    */
3      if kIncrement ≥ k then
4          scavengeQueueEmpty = FALSE;
5          return scavengeQueueEmpty;
       /* Breadth-first evacuation of references                      */
6      foreach field in scavenger do
7          if field is a reference type then
8              dereference(field) = evacuate(field);
9          scavenger = scavenger + 1;
10         kIncrement = kIncrement + 1;
11 scavengeQueueEmpty = TRUE;
12 return scavengeQueueEmpty;
```

unnecessary and we revisit this issue in the design of our own collector Section 7.4.3.

### Calculating $k$

We now complete the basic description of Baker's algorithm with an explanation of how $k$ is calculated and the Function `calculateK()` would be implemented.

If there are $L$ live words in the heap at the time of allocation failure, $k$ words are evacuated at each new allocation and an average of $A$ words are allocated for each new object, then $L$ words will have been evacuated after $\frac{L}{k}$ calls to the collector. Therefore, at the end of the collection cycle the number of live words in the (new) heap is given by Equation 3.5.

$$L(1 + \frac{A}{k}) \tag{3.5}$$

Therefore, in order to prevent starvation, the heap size, $H$, must be greater than the number of live words given by this equation. Obviously, these amounts are doubled for a two-space copying collector. If the heap size is fixed, then Equation 3.5 can be rearranged so that the value of $k$ can be dynamically determined for a particular collection cycle:

$$L(1 + \frac{A}{k}) < H \tag{3.6}$$

$$k > \frac{LA}{H - L} \tag{3.7}$$

An actual implementation of the collector may choose the units of $k$ to be the number of objects scavenged *or* evacuated within the incremental cycle as opposed to the number of

words evacuated/scavenged in the cycle. Clearly, by dividing the right hand side of Equation 3.7 by the average object size, $(A)$, this can be achieved.

We remark that an alternative to attempting to perform strictly $k$ fixed increments of work at each allocation, as determined at collector initialisation, is to calculate the amount of work to do based on the ratio of the size of the object being allocated to the amount of free space left. Although the work increments are now less rigorously bound and aggregated pauses occurring as a result of bursty allocation rates can now be much longer, tighter coupling to the allocation rate may prevent collection cycles having to be forced to completion as memory is exhausted while the collector is running — as its scavenging rate lags the allocation rate.

#### Limiting Incremental Bound Violations

The pseudo-code presents a generalisation of Baker's algorithm. Baker includes two more considerations in his algorithm. If the root set is large, for example by inclusion of the program stack, it will not be possible to maintain a small upper bound when the Function `allocate()` is invoked and a flip occurs (the entire root set is evacuated and $k$ objects scavenged). He therefore proposes that at each allocation request, $k'$ stack objects are also scavenged and at each flip $k'$ is recalculated so as to keep $\frac{k'}{k}$ equal to the ratio of stack locations to heap objects. The second consideration results from the realisation that the cost of evacuating an object is relative to its size. Baker therefore evacuates large objects lazily. Not only does a from-space object contain a forwarding pointer but the to-space object contains a pointer to the original from-space object. When a large object is evacuated, memory is allocated in to-space for it and the forwarding pointer and backward link pointers are set. The payload of the to-space object can now be copied incrementally throughout the duration of the collection cycle.

As a final remark, we comment on the space bounds of Baker's algorithm. For a stop-the-world semi-space collector processing $L$ live words, the space requirement is simply $2L$ — half of the available memory must be reserved for each semi-space. An incremental tracing collector allocates $L/k$ words while tracing $L$ words, and at the end of the cycle to-space contains $L(1 + 1/k)$ words. If only $L$ of these words are now live then $L$ words are copied to the new semi-space, while $L/k$ words are allocated. As a result, the maximum space requirement of Baker's collector is $2L(1 + 1/k)$. Furthermore, if $N$ arbitrary (large) sized objects are live, then an extra header word for each object (for the backward link pointer) is required. The final maximum space requirement is: $2(L(1 + 1/k) + N)$.

### 3.4.2 Brooks' Unconditional Read Barrier

Brooks [Bro84] modifies Baker's collector to reduce the tight synchronisation of the mutator-collector read barrier and the overhead that it incurs, at the expense of an extra word overhead for each heap allocated object. For the Lisp environment that is targeted, he suggests that this tradeoff is worth the reduced code density and execution times resulting from elimination of the conditional read barrier.

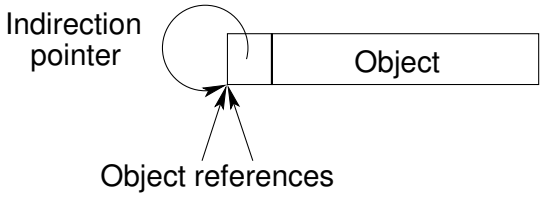



Figure 3.7: A Brooks-style indirection is prepended to all heap objects

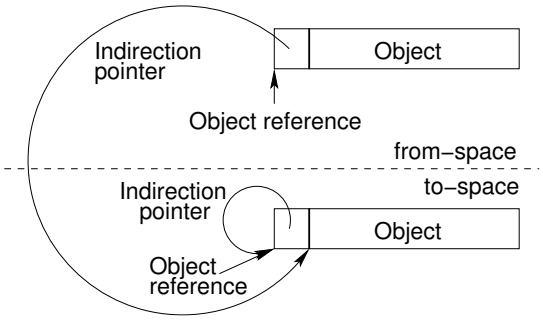



Figure 3.8: Arrangement of Brooks-style indirections for from-space and to-space objects

The extra word prepended to each heap object is for the storage of an *indirection pointer*. In this modified scheme, access to all heap objects is indirect via this pointer. When the collector is off, the pointer simply points to the object (see Figure 3.7). However, when the collector is on, and the object has been evacuated to to-space, the indirection pointer of the from-space object stores the forwarding pointer to its to-space copy (see Figure 3.8). In this way, if an object has been forwarded, the mutator always accesses the to-space copy. The conditional test and branch associated with Baker's barrier has been replaced by an unconditional pointer dereference and a dynamic space overhead. The code space and time costs (imposed on the mutator) are therefore reduced. Note that there is still a barrier on pointer load operations, it is however *unconditional*.

The elimination of Baker's read barrier results in violation of the strong tricolour invariant. In order to restore correctness to the algorithm, Brooks introduces an incremental update write barrier that maintains the to-space invariant. Whenever a reference is written to a heap object, the write barrier determines if the referent has been forwarded and if not, evacuates it. As a result, no grey to-space reference can be stored in a white from-space object.

To restore complete correctness, collector termination must be modified. If both to-space and the program stack have been fully scavenged then the only references to reachable white from-space objects reside in the machine registers. These objects must be evacuated and scavenged. This can be viewed as initiation of a shorter collection cycle. If it is processed in a stop-the-world fashion, then collection is complete when all register references have

been evacuated and scavenged along with any white objects to which they referred. Brooks notes that this takes a bounded amount of time since there are a fixed number of registers. However, for an environment with variable sized objects, the live reference chains from the register references may be long, resulting in unacceptable pauses and violation of real-time bounds. This shorter cycle can of course be performed incrementally, however, this may result in repeated scanning of the registers and invocation of multiple shorter collection cycles until all reachable white objects have been processed.

Brooks does not empirically evaluate his proposed scheme, but it outperforms Baker's provided that the work performed by the shorter collection cycles is negligible and the combined cost of the unconditional read barrier and that of the write barrier are less than that of a conditional barrier. Intuitively, for the latter condition, this is so in an environment where the frequency of writes to heap objects are less than reads. It is important to note that the indirection pointer incurs a dynamic space overhead that reduces the amount of heap available for use by the mutator. This can increase the number and frequency of the collection cycles, imposing a time penalty that can be greater than the overhead imposed by Baker's scheme for many configurations.

The final modification Brooks proposes to Baker's algorithm is the incremental scavenging of the stack at each allocation event. He argues that because the stack mutates most rapidly from the top down, the incremental scavenger should operate bottom up, concentrating effort on those frames that are likely to persist the longest. A *Stack-Swept-Marker* is used to record how much of the stack has been swept and relieves the stack push/pop operations from the burden and overhead of this book-keeping, at the expense of a much cheaper, single variable, book-keeping overhead in the collector.

### 3.4.3 Replicating Garbage Collection

In the context of copying garbage collectors, there are two general approaches to achieving incremental collection. The first, Baker's algorithm, presented above, uses the read barrier to prevent the mutator from accessing white objects in from-space that have yet to be copied. In the second, the *replicating* scheme of Nettles et al. [NOPH92, NO93], the reverse is true: the mutator is prevented from accessing the objects that have been copied until the collection cycle has completed. Over the duration of the collection cycle a replica of the memory graph is constructed in to-space, of which the mutator is oblivious. In order to synchronise the replica to-space copy with the mutator's memory graph, all mutations must be recorded in a *mutation log* as they occur. A write barrier is placed on all mutable operations that either a) eagerly applies the mutation to the replica, or b) records the mutation in a separate log table, whose entries are then applied *en masse* at the end of the collection cycle. Regardless of the strategy chosen, it is worth noting that the write barrier must also grey the updater object. In the case where mutations are applied via batched update, this amounts to re-greying a black object for re-processing. By relaxing the eagerness to which the replica graph is synchronised

with the mutator, the collector gains increased scheduling flexibility over Baker's algorithm. As a result, replicating algorithms are usually implemented as concurrent algorithms, with the collector running in a separate thread to the mutator.

Fundamental to the performance of the algorithm is the substitution of the "expensive" read barrier of Baker's algorithm with a rather "cheaper" incremental update write barrier, but at the expense of some additional work in maintaining the mutation log. While the write barrier *may* be cheaper or less frequent than the read barrier there are some disadvantages to this scheme. Since the mutator accesses the from-space object, its evacuation to to-space to build its replica must be non-destructive — in particular the forwarding address requires an additional word, of which the mutator is oblivious, be allocated for *every* heap object. If the write barrier applies mutations lazily via batched update from a log then additional space is required to store the log, which may be subject to its own book-keeping overheads, especially if the log itself must be garbage collected. Nettles et al. comment little on the log, namely because they exploit the existing infrastructure provided by the remembered set log of Appel's generational collector to which they augment their replicating implementation. As a result *all* writes are already, rather inefficiently, recorded and no generation test is required as part of the write barrier. Both these sources of dynamic space overheads reduce the amount of memory available to the mutator, which can increase the frequency and load of the garbage collector and ultimately lead to extended execution times. The most serious issue for a replicating collector is that of newly allocated objects and stack operations — these operations must maintain the from-space invariant and are therefore, in theory, subject to write barrier traps and to-space replication.

Nettles et al. describe how to avoid some of these disadvantages for their replicating collector that is implemented for Standard ML of New Jersey (SML/NJ). Others however, are simply side-stepped. Non-destructive forwarding of the from-space object is achieved by replacing the header word of an ML object with the forwarding pointer during object evacuation. This apparently destructive forwarding of the object is then masked by placing a read barrier on those operations that must access the object header. The read barrier simply determines whether the object has been forwarded and, if so, follows the forwarding pointer to get to the real header word. They indicate that the header is accessed infrequently for polymorphic equality operators and type-specific length operations, but do not present results for the frequency of these operations, nor the overhead incurred by the test. The program stack is allocated in the SML heap and stack frames are garbage collected, as opposed to the instant space reclamation of a conventional stack pop operation. As a result, the application of the write barrier to heap object mutations seamlessly handles stack operations without incurring further overhead. This leaves the issue of new object allocation. Clearly, if the from-space invariant is to be maintained, then newly allocated objects must be allocated in from-space. Nettles et al. suggest periodic retracing of live data from the roots to locate and evacuate these objects, but pay little attention to how such tracing should be scheduled, the effects on real-time bounds or the overhead incurred by redundant retracing. However,

in their actual implementation, which augments SML's generational collector, all data that is allocated during the collection cycle is conservatively treated as live and replication collection is only employed for major collections. Recall that the maximum space requirement for Baker's collector is $2(L(1+1/k)+N)$. If cells are *double allocated* in both from-space and to-space, then the maximum space requirements of a replicating collector is $2(L(1+1.5/k)+N)$. Furthermore, this assumes eager application of the mutations to replica copies by the write barrier, therefore eliminating the storage requirements of the write barrier log. This also assumes an ML-like environment where the non-destructive update for the forwarding pointer does not require an extra word in each object. If, however, an explicit mutation log is used, and an additional word is required for the forwarding pointer, then, if there are $O$ objects accounted for by the $L$ live words, then an additional $2O$ words are required, yielding: $2((L+2O)(1+1.5/k)+N)$.

It is these disadvantages in combination that make replicating collection prohibitively expensive as a general purpose uniprocessor algorithm and motivate its use within concurrent collectors, rather than the increased flexibility with which the collector can be scheduled. Furthermore, it is the very low data mutability rates within the SML runtime system that facilitates the "efficient, low-overhead" implementation of a replicating collector.

### 3.4.4 Dijkstra's On the Fly Mark-Sweep Garbage Collector

Section 3.2.1 introduced the tricolour marking abstraction and the associated invariants. Although the abstraction can be used to reason about the correctness and termination properties of any collector, Dijkstra et al. devised it in order to reason about various algorithms for concurrent mark-sweep collector implementation. The algorithms they present in [DLM+76] are all serial with respect to the two phases — the sweep phase must immediately follow the mark phase and cannot execute in parallel with it. The mutator is however able to operate concurrently with respect to both phases. The initial algorithm presented is an incremental-update algorithm that enforces the strong invariant during the marking phase. To do this, the mutator employs a write barrier that traps the write of a white, collector unvisited, object into that of a black, collector visited, object. As a result the barrier code colours the referent white object grey. An optimisation to the barrier, that reduces its overhead, has the barrier grey the referent regardless of the referrer's colour — it is redundant work if the referrer is itself white or grey. It is worth clarifying that 'greying' an object affects only white objects turning them grey and leaves grey and black objects unchanged.

A collection cycle is initiated by atomically marking the entire root set grey. To simplify the algorithm, all elements of the free list are also treated as reachable from 'special' root nodes. As a result, free list cells are initially white and will be traced during the marking phase with the write barrier ensuring that they are correctly coloured if subsequently allocated. Alternative schemes for the handling of the free list have been proposed, for example, in [KS77] Kung and Song propose the use of a fourth colour for free list cells that attempts to reduce

the length of the mark phase by avoiding the tracing of free list cells. The marking phase is then initiated and comprises of repeated linear scans of all memory cells. Whenever an object is visited whose colour is grey, its referents are all coloured grey and it is coloured black. This must be done atomically. The marking phase terminates when a full linear scan completes in which the collector has performed no marking actions and as a result no grey objects remain. It is worth noting that the algorithm can be implemented in many ways and that a more efficient implementation simply traces out the live data as opposed to repeatedly performing linear scans of all cells. If this is done, then the write barrier pushes those objects that it has coloured grey onto an explicit stack that the collector must process. When all the live data has been scanned and this stack is empty, the mark phase terminates. Of course this incurs the additional space overhead of the stack and care must be taken to handle its potential overflow.

The sweep phase, comprising of a single linear scan of all memory cells, is then initiated. If a white object is visited, it is garbage and is appended to the free list. If a black object is visited, it is live and its colour is reverted to white. At the end of the linear scan, the collection cycle is complete and the collector terminates. It should be apparent that there are no longer any black objects, although some objects may now be grey. A grey object results from the write barrier operating on a mutator store to a black object that had yet to be visited by the collector in the sweep phase. If collection cycle $n$ has just completed, any grey objects that now become garbage before collection cycle $n+1$ will 'float', only coming under consideration as garbage in cycle $n+2$. Furthermore, any black objects that became garbage during the mark phase will be under consideration in cycle $n+1$. As a result, the algorithm guarantees that no garbage will 'float' for more than two collection cycles.

### 3.4.5 Yuasa's Work-based Mark-Sweep Collector

Yuasa, like Brooks, devises an alternative algorithm to Baker's with the aim of eliminating the overhead incurred by the read barrier [Yua90]. However, the algorithm is presented in the context of an incremental mark-sweep collector, and unlike Dijkstra's collector, it is incremental, targeting a uniprocessor environment and employs a snapshot-at-the-beginning write barrier.

Collection is initiated when the number of cells on the free list falls below an acceptable limit, $M$. During the mark phase $k1$ cells are marked at each allocation, while during the sweep phase $k2$ cells are swept. The algorithm uses Baker's work-based scheduling technique to drive the collector. $k1$ and $k2$ are selected as small bounded constants. $M$ is selected so that the collection cycle completes before the exhaustion of the free list — repeated sweeping at each allocation, that violates $k2$ must be avoided. As a result, unlike the stop-the-world mark-sweep algorithm where the free list is empty on initiation of the sweep phase, the free list is left populated and untouched. Only those cells that are inaccessible and are not on the free list at the beginning of the mark phase can be reclaimed. This requires a bit within

the object header, or an additional field in each object, be reserved to indicate whether a cell is already on the free list. Furthermore, new cells are marked as live on allocation. As a result, cells that become garbage during the collection cycle cannot be reclaimed until the *next* collection cycle. Note this is more conservative than Baker's copying collector, where only to-space objects that die during the collection cycle cannot be reclaimed.

To enforce the invariant that all accessible cells at the beginning of the mark phase are eventually marked during the phase, a write barrier is employed. Unlike Brooks' write barrier, that operates on the referent to maintain the to-space invariant, Yuasa's operates on the target object of the write operation and records the reference being overwritten. The reference being overwritten was of course reachable at initiation of the mark phase, and it is for this reason the algorithm is a snapshot-at-the-beginning algorithm.

Recall that the root set comprises of the machine registers and the system stack. The machine registers are greyed *en masse* at collector initiation. However the program stack is by comparison much larger and must be processed incrementally so as not to violate real-time bounds. Yuasa proposes that the stack be processed with a similar snapshot mechanism. He introduces a *save stack*, to which the system stack is batch-copied. Stack pointers are then processed in increments bounded by constant $k3$, at the end of each marking increment.

### 3.4.6 Steele's Concurrent Compactifying Collector

In [Ste75] Steele presents a relatively complex algorithm for a concurrent mark-sweep collector that is in essence very similar to that of Dijkstra's collector. It too operates an incremental-update algorithm, employing a write barrier to maintain the strong invariant. However instead of colouring the referent grey, it is left white, and the referrer is reverted from black to grey. In fact, Steele doesn't explicitly colour objects; instead a mark bit is used in conjunction with an explicit stack upon which grey objects are pushed. The mark phase then terminates when the stack is empty. In the sweep phase, black, live, objects are distinguished from white, garbage, objects because the mark bit is set.

Steele separates the mark and sweep phase with two optional phases, *relocate* and *update*, that when used enable incremental in-place compaction of objects using a variation of Edwards' two-finger algorithm [Edwwn], that is synchronised to facilitate concurrent execution with the mutator. When an object is relocated, it is destructively updated with a forwarding address. As a result, the mutator employs a read barrier that determines whether an object has been forwarded based on a bit sequence in the object header and if so, follows the forwarding pointer. In the original stop-the-world two-finger algorithm free lists are unnecessary. However when operating concurrently, they are required so that the mutator may allocate new objects.

## 3.5 State-of-the-art "Real-time" Collectors

Despite a wealth of literature on incremental and concurrent collection algorithms only a small proportion have been fully implemented and fewer still provide any meaningful study of their real-time characteristics beyond the reporting of average and worst-case pause times. Furthermore, the majority avoid the mutator-collector scheduling issues entirely and simply optimise or improve existing algorithms, or focus on the bounds and guarantees of a specific collector phase.

In this section we summarise those collectors that currently lead the state-of-the-art in general-purpose concurrent and soft real-time garbage collection in each of the main categories. Only Bacon et al.'s Metronome and Detlef et al.'s Garbage-First collector attempt to achieve a user-specified mutator utilisation and therefore can be categorised as (soft) real-time. Henriksson's collector, and the analyses employed for its scheduling in real-time environments, have previously been described in Section 3.3.2. Recall that the collector is a semi-concurrent copying collector, which whilst scheduled so as not to preempt high-priority tasks, interrupts low-priority tasks with a work-based collector that may disrupt their desired MMU characteristics. The Real-Time Specification for Java adds support for both region-based allocation and time-based and time-scheduled collectors. The implementation approaches that the remaining algorithms and collectors adopt are however interesting and, in places, pertinent to our own.

Table 3.1 summarises the various characteristics of the collectors that we survey.

| Collector | Incremental | Concurrent | Parallel | (Soft) Real-time |
|---|---|---|---|---|
| **Reference Counting** | | | | |
| Recycler | ✓ | ✓ | | |
| **Hybrid Mark- Sweep/Compact** | | | | |
| Compressor | ✓ | ✓ | ✓ | |
| Metronome | ✓ | | | ✓ |
| STOPLESS | | ✓ | | Supports |
| CHICKEN | | ✓ | | Supports |
| CLOVER | | ✓ | | Supports |
| **Replicating** | | | | |
| Cheng | ✓ | ✓ | ✓ | |
| Sapphire | | ✓ | | |
| **Copying** | | | | |
| The Train | | | | |
| Henriksson | | Semi-concurrent | | High-priority tasks |
| Garbage-First | | ✓ | ✓ | ✓ |
| **Region-Based** | | | | |
| RTSJ | Supports | Supports | Supports | ✓ |

Table 3.1: Characteristics of state-of-the-art concurrent and (soft) real-time collectors

### 3.5.1 The Sliding View Recycler

Bacon et al. built on Lins' [Lin92, MWL90] cycle collecting reference counting algorithm to build the first *concurrent* cycle collector, the Recycler [BR01]. The synchronous version of the Recycler is a stop-the-world uniprocessor collector. The collector tracks those objects whose reference counts, when decremented, remain non-zero. The tracked objects are used as a candidate set of objects that may form part of a cyclic structure.

A tricolour marking scheme is used to detect actual cycles as follows. All objects are initially coloured black. In the first phase, the *mark* phase, the candidate set above is treated as the root set and all reachable objects are traced and marked grey. Grey objects are candidate members of a garbage cycle. As each object is visited, its reference count is decremented according to the number of *internal* references. On completion of this phase, each reachable cyclic structure contains at least one node with a non-zero reference count, while all nodes of an unreachable cyclic structure have a reference count of zero. In the next phase, the *scan* phase, the candidate set is traversed a second time, and all those objects that can be reached via an external reference (i.e. have a non-zero reference count), are coloured black. In addition, their children are also coloured black and the reference counts are reset to correctly identify all references from black objects. All other nodes are identified as belonging to an unreachable garbage cycle and are coloured white. In the third *phase*, the *collect* phase, the candidate set is traversed again and the white garbage objects are collected and deallocated.

This synchronous mode of operation is the basis for the first phase of the concurrent Recycler in which the mutator and collector run concurrently. During the collect phase, some of the objects that are coloured white may actually still be reachable. Thus they cannot be immediately reclaimed, but must be treated as a new candidate set of unreachable cyclic data. The second phase is scheduled to run *after* the next reference counting collection cycle and only those objects of the potential cycles, that are confirmed as belonging to an unreachable cycle are reclaimed.

The Recycler as presented, is not a *complete* algorithm — it is not necessarily the case that all unreachable objects will eventually be collected. Furthermore, the second concurrent phase requires additional traces of the candidate set. These repeated traversals impose a significant overhead. Both of these issues are the result of the collector operating concurrently without having a static snapshot of the heap with which to work.

Levanoni and Petrank present in [LP01] a concurrent reference counting garbage collector for Java based on a snapshot of the heap. The algorithm requires all mutator threads to be stopped simultaneously, to enable a (virtual) snapshot of the heap to be taken, whose consistency is then maintained in the presence of concurrent updates using a write barrier. Like Nettles and O'Toole's replicating collector, the write barrier copies objects prior to being dirtied by a write operation. The reference count for a dirtied object at the time of the snapshot is thus determined from the *local copy buffer*. A major contribution of the algorithm is that the synchronisation operations employed within a conventional write barrier

that protect (concurrent) pointer updates have been eliminated. The reference counting collector is hybridised with an "on-the-fly" mark-sweep collector, that runs infrequently, so as to guarantee that all unreachable objects, including cycles, are eventually collected. This hybrid algorithm is thus complete, and performs fewer object graph traversals by employing a write barrier. Furthermore, by updating reference counts based on the previous and current snapshots, fewer updates need to be performed. In order to reduce further the collector pause times, the snapshot is built based on a *sliding view* mechanism (which the mark-sweep collector also exploits). The mutator threads are no longer simultaneously stopped. The ability to stop the threads one at a time, yielding shorter pause times, comes at the expense of a more intrusive write barrier that additionally records all objects that are newly referenced. This set of objects augments the root set for the current collection cycle and as such, cannot be reclaimed until the next cycle. The heap snapshot used by the collector is said to be a "sliding view" because the heap objects are viewed at different time points.

In [PPB+05] the Recycler is augmented with the write barrier and sliding-view mechanism of the Levanoni-Petrank collector to yield a complete collector in which the second concurrent phase can be completely eliminated. A maximum pause time of 1.7 milliseconds, for the root scan of a single thread, across all benchmarks of both the SPEC JVM 98 and SPEC JBB 2000 benchmark suites is reported. Unfortunately the minimum mutator utilisation is not, and as a result, it is unclear how the aggregation of individual pauses might result in incremental bound violations when operating in a soft-real time environment.

### 3.5.2 The Compressor

Kermany and Petrank's Compressor [KP06] is a concurrent, incremental, parallel compacting garbage collector. It runs periodically as a "backup" compacting collector and is designed to eliminate heap fragmentation that results from ongoing mark-sweep collection of long running server processes. Its remit is to eliminate the long disruptive pauses that are usually associated with heap compaction. The novelty of the Compressor is that it operates in one pass, preserving the ordering of the compacted objects. This improves cache behaviour. While the Compressor achieves short pause times, it is not truly "real-time" because it does not implement a rigorously bounded MMU, nor is it time-scheduled or parameterised in an attempt to meet soft real-time deadlines or guarantee a minimum level of mutator progress within a specified time quantum.

Like many prior concurrent copying collectors and, in particular, that of Appel and Li [AEL88, AL91], the operating system's page protection mechanism is employed to trap accesses by the mutator to objects that have yet to be copied [AEL88, AL91, BDS91, OBYS04]. In essence this mechanism implements a fast read barrier using a hardware facility generally available on stock hardware; this is as close to a specialised hardware read barrier as one can get with a general-purpose machine. The barrier is coarse grained, acting at a granularity of a virtual memory page, which is typically 4KB — all objects on a page must be evacuated

atomically. This can lead to unpredictable pauses that may violate incremental bounds if objects on different unevacuated pages are accessed in quick succession. Operating at a page level granularity not only offers the benefit of a fast hardware read barrier but also allows the virtual memory subsystem to be exploited to present a *virtual* to-space. Instead of having to reserve a portion of the heap for to-space (in the worst case half of the available memory), only a few pages need be made available when compaction is initiated. Having fully evacuated a page, it can be immediately re-used i) as to-space for subsequent page compaction or ii) it can be returned to the operating system. Clearly, care must be taken using this technique to prevent the "thrashing" of the virtual memory subsystem as pages are written to and loaded from disk.

The Compressor requires a *markbit* vector as a prerequisite to initiating the compaction phase, and is most usually generated by a previous mark-sweep cycle. The markbit vector represents each heap word by a single bit (assuming objects are word aligned) and for each object that is to be marked as live, the bits associated with the first and last words of the object are set. Using the markbit vector an *offset* table is generated that is used to compute the address to which an object is to be relocated. It is worth noting that this table can be computed concurrently and without accessing the heap.

Multiple compaction threads run in parallel, each obtaining pages that are yet to be evacuated. The threads relocate the objects on each page to target addresses computed from the offset table. In concurrent operation, having computed the offset table, the virtual to-space pages are protected and the roots are updated with their target references using the table. When a mutator thread attempts concurrent access of an object on a protected page, i.e. that is yet to be populated with its live from-space page objects, an interrupt is generated. This results in the execution of a barrier trap handler by the mutator. The handler evacuates the live objects of the corresponding from-space page, relocating them to their to-space target addresses on this page. Having updated all object references on the to-space page with their target addresses, the mutator resumes.

Both the markbit and offset vectors impose a fixed-sized space overhead that can be computed statically and reduce the available memory for use by the heap accordingly. In the experiments of [KP06] the system is sized so that the markbit vector incurs a space overhead of 1/32 of the available heap size, whilst the offset vector incurs an overhead of 1/128. The ramification of these particular sizing decisions is that the barrier trap handler operates with a granularity of 8 pages, not 1. While the experiments report the "occurrence of meaningful mutator allocations after pauses of typically between 5 and 10 milliseconds" as an indication of mutator progress, no meaningful study of the real-time characteristics of MMU is presented.

### 3.5.3 Cheng's Parallel, Concurrent, Replicating Collector

In his PhD thesis [Che01], Cheng builds on the concurrent replicating collector of Nettles et al. (Section 3.4.3) to design and implement a parallel, concurrent, replicating collector for

shared memory multiprocessor architectures, with provable space and time bounds. Unsurprisingly, the target is the runtime system of an SML environment, TILT [TMC+96]. The core contributions of the collector are summarised in [CB01].

Cheng tackles those issues that Nettles et al. fail to address and which are required to implement a replicating collector with provable bounds. In particular:

- The compiler is modified to allocate a primary and replica copy of global variables that can then be interchanged subsequent to each flip of from-space and to-space — the mutator must then check a flag to determine which (primary) copy to use.

- The stack is arranged as a linked chain of fixed sized chunks known as *stacklets*; together they constitute the logical stack. Unlike Nettles et al.'s collector, stack writes are not subjected to the write barrier. Stacklets are processed concurrently, with collector threads working on those stacklets which are known not to include the currently active one. This arrangement incurs a book-keeping overhead on stack push / pop operations in order to determine when to allocate and deallocate stacklet chunks and to mark the currently active stacklet.

- The collector runs concurrently with the mutator and is multi-threaded. The collector threads are Baker work-based scheduled — each processor that allocates a unit of data must copy $k$ units of data. As a result it is possible for bursty allocations to result in violation of real-time bounds, albeit with reduced probability as the number of processors and collector threads is increased and idle processors are exploited as a result of work sharing. To limit further the likelihood of such violations, Cheng modifies the allocator to perform *two-level allocation* in which each processor performs lightweight allocation from a local pool. When the pool is exhausted it is replenished from the globally shared from-space heap. The collector threads need only be scheduled to perform their work at this replenishment point operating with $k$ in proportion to the size of the local pool.

- Unlike Nettles et al., who incrementally replicate only at a major collection, Cheng's collector is fully replicating, so as to perform within useful real-time bounds. As a result he is forced to allocate objects in both from-space and to-space (*double allocation*). Initially, he performs this cheaply enough by deferring allocation of the replica until a new processor local pool is allocated. The space cost of such a strategy is substantial. The effective survival rate increases from the liveness ratio, $r = L/H$, to $(r + r/k)/(1 + r/k)$, which results in significant increases in collection times [Che01]. To reduce this overhead, double allocation is not performed during the main collection cycle. However, like Brooks' scheme, an additional collection phase is required to replicate objects that were allocated during the main portion of the cycle. If this phase is performed incrementally, then additional phases must execute, until the amount of work that must be performed for each phase, is known to lie within the defined real-time bounds of the collector. A non-incremental termination cycle reduces the effective survival rate

to $(r + r^2/k + r^2/k^2)/(1 + r^2/k + r^2/k^2)$. Cheng reports that for typical values of $r$ and $k$, this results in an increase of, on average, 0.7% across the set of standard SML benchmarks.

In parallelising the collector, Cheng describes how to achieve scalability through load balancing. He employs a work sharing algorithm between multiple collector threads, where each thread moves grey objects from a globally accessible shared stack to processor local stacks for processing. Note that these stacks are explicit and depth-first traversal is employed, sacrificing the efficiency of Cheney's (breadth-first) to-space queue. However, using this arrangement, Cheng is able to minimise the contention on stack push / pop operations, and, more generally, relax the fine-grained write-write conflict synchronisation required to handle multiple writes to the same heap locations and race conditions from interleaved primary reads and replica updates. Cheng introduces the *rooms synchronisation* abstraction to enable this relaxation and enable concurrent reads and concurrent writes. The abstraction consists of a number of rooms, each of which may house any number of processors, but with the invariant that they cannot be simultaneously populated. In traditional terms, a room represents a critical section. A processor wishing to enter a room places itself on the room's waiting list. The last processor to leave a room opens the door to another room so that they are serviced fairly in a round-robin fashion.

For the shared stack, there are two rooms defined. When in the first room the processor may push grey objects onto the stack, while in the second, it may pop objects from the stack. As a result of the invariant, concurrent pushes and pops cannot occur. The replicating write barrier is comprised of three stages. The first records the reference being overwritten in the from-space object, recording it for subsequent batched processing. The latter phases copy and grey both this overwritten reference and the updating reference. The rooms abstraction is used to ensure that there are no concurrent read-write conflicts in the latter portion of the write barrier where the current value of the primary object is read and the replica is updated. These operations may only be performed when a processor is in the write barrier room.

Cheng's collector bounds each memory operation to $ck$ for where $c$ is a small constant, the time to collect one word, and $k$ is the number of words collected per word allocated. The maximum space requirement builds on that of Nettles et al.'s replicating collector, adding an an additional term, of $5PD$, to each semi-space footprint, where $P$ is the number of processors and $D$ is the maximum live memory graph reference chain "depth". The depth is calculated by summing the sizes of all objects resident on the reference chain: $2(L(1 + 1.5/k) + N + 5PD)$. Like the original replicating collector, this does not account for the space overhead of the mutation log (see Section 3.4.3) nor a header word required for non-destructive object forwarding.

### 3.5.4 Sapphire

Sapphire is a concurrent replicating collection algorithm designed with the specific remit of minimising the amount of time a mutator thread must be blocked for collector related activities. In particular, it avoids pausing all threads in a stop-the-world fashion at the flip phase where many algorithms scan and redirect the entire stack as part of the root set scavenging phase.

The Sapphire algorithm and a prototype implementation for Intel's Open Run-time Platform (ORP) are described in [HM01, HM03]. Sapphire is interesting because it is the first known application of replicating garbage collection to a language with high data mutation rates. Sapphire claims to improve the replicating collector of Nettles et al. in the following ways: i) the write barrier is used to propagate updates to the replica copy eagerly, as opposed to simply logging them; ii) the mutator is allowed to access both the original object and its replica copy; iii) heap and stack references are flipped incrementally. There are some underlying assumptions surrounding the heap layout and memory synchronisation characteristics of the runtime system in which the collector is implemented. These restrictions arguably limit its applicability. Indeed there are no known implementations other than the "proof-of-algorithm implementation" presented in [HM03].

Sapphire assumes the segregation of the heap into several distinct memory regions. The U region is an "uncollected" region that is not subject to reclamation within the current collection cycle — it is however scanned. The "collected" region, C is sub-divided into from-space O, and to-space N, regions. Each thread stack is represented by its own S region.

The algorithm allocates new objects in the U region and, as a result, they are subjected to the write barrier and its associated overheads. Adopting this strategy simplifies the algorithm to ensure termination guarantees — this is in contrast to the 'double allocation' and additional retracing phases of Cheng's collector.

The algorithm operates in two phases, the first comprises *Mark-and-Copy* while the second comprises the *Flip*. During Mark-and-Copy a root set marking traversal from each S thread stack region and the uncollected U region and through the from-space O region is performed. During the 'Copy' portion of the phase, the marked from-space objects of O are copied into the to-space region N. Mark-and-copy executes concurrently with the mutator threads which only access and modify from-space objects in the O region. Mark-and-Copy executes a concurrent marking algorithm followed by the copying of the marked objects. Unlike other replicating algorithms however, a live object, O, and its replica object, N, are *loosely* synchronised. Object updates made by a mutator thread between two synchronisation points are only propagated to the replica copy immediately before passing the second synchronisation point. As a result updates need not be propagated to the replica copy atomically. The algorithm exploits the memory synchronisation rules of the Java Virtual Machine Specification to achieve this. It is worth recalling that a key goal of replicating collection is to replace the "expensive" read barrier with a cheaper write barrier. However, the write barrier must be applied to *all* values

including non-pointer values. For a language like Java it is not at all clear that the write barrier (when combined with its effected frequency) is actually cheaper. Replicating collection does however allow weaker mutator-collector synchronisation that arguably, lends itself better to a concurrent implementation.

The algorithm enables threads to be stopped and paused for the stack marking phase independently of one another. As described, it scans and marks the thread stack in its entirety during which time the thread is paused. Furthermore, stack operations proceed unhindered and are not guarded by either a read barrier or write barrier. It is therefore possible for mutator threads to push references of unscanned objects onto the stack once it has already been scanned. As a result, stacks must be scanned repeatedly, although the authors propose some optimisations for the reduction of this overhead and speculate as to how a stack might be incrementally scanned.

The algorithm requires that it be possible to find a from-space copy of the object in O from its replica copy in N. Back pointers are not used, so as to avoid an extra pass over the N region in which they are removed. Instead, a hash table is used. The table can be discarded after collection, but no consideration is given as to how the memory for this table should be managed or the overheads that such management might incur. This is not a trivial issue given the number of objects and large heap sizes in use by today's applications.

The goal of the Flip phase is to eliminate pointers to the from-space region O. To achieve this, a write barrier is installed that tracks those objects containing pointers to objects resident in O. Next all pointers from the U region into O are flipped to point at their to-space N region replica. Finally, pointers in each thread stack S must be flipped to point at their to-space N region replica. Unflipped threads are allowed to access both the O and N region copies of the object. The flipping of each thread is performed both concurrently and independently of one another and thus affords flexibility as to how and when it is scheduled for each thread.

Unflipped threads may access both the O region object and its N region replica. As a result, operations, such as variable (reference) equality tests are subjected to what is essentially a read barrier. The authors remark that such operations are, however, rare.

The different phases of the algorithm require the application of various different write barrier mechanisms. The authors suggest ways to switch between each of the barrier implementations although it is interesting to note they don't indicate how this is achieved within their prototype.

### 3.5.5 The Train Algorithm

The Hudson and Moss Train algorithm [HM92] is an extension to the standard generational collection algorithm designed to reduce and eliminate the longer, disruptive pauses of a major collection. The algorithm divides the old(er) generation(s) — the "mature object space(s)" into equal sized *areas* which can then be collected at the granularity of a region as opposed to the entire generation. The collector is a pure copying collector, and as such compacts

live objects in clusters with their referents. A key contribution of the algorithm as presented in [HM92] is in guaranteeing that all garbage is eventually collected and, in particular cyclic garbage that spans multiple mature object space areas.

Hudson and Moss use a train metaphor to describe their algorithm. Each area is represented by a train *car* or *carriage*. Carriages are linked together to form a train. Each train groups related objects for management as a "unit". Like a generational collector where each generation has an associated remembered set to record inter-generation pointers, so each mature object space area has a remembered set that records inter-area pointers. In order to reduce the size of the remembered set, areas are scavenged in conjunction with all younger object spaces (generations) and the roots of the live memory graph residing in registers, stacks, and static data. As a result the remembered set only tracks references between mature object space areas. The remembered set for a train is therefore the union of the remembered sets of its carriages. In addition, the carriages of a train are processed in round-robin order — if the next carriage to be scavenged is assigned the next highest number in a monotonically increasing sequence, then only references from a higher numbered carriage to a lower numbered carriage needs be tracked by the remembered set of the lowered numbered carriage. This is similar to the collection ordering of a generational collector where a given generation and all generations younger than it are collected together.

The algorithm operates in collection increments of a train carriage, at which time the roots and younger generations are also scavenged. Before collecting an individual carriage, the collector determines whether there are any roots to its (parent) train and whether the train's remembered set is empty. If there are no roots and the remembered set is indeed empty then the entire train can be condemned and reclaimed. On promotion, younger generation objects, if referred to by a root, are promoted into any train. If references to an object undergoing promotion already exist within a mature object space area then it is promoted to a train containing one such reference. A carriage is evacuated and scavenged using Cheney's semi-space copying algorithm — a *to* carriage is selected to house the live objects evacuated from the *from* carriage. The 'from' carriage must belong to either a new train or some *other* existing train. Having performed the 'from' carriage evacuation of root referred objects, the carriage's remembered set is processed and those objects residing *outside* of the train of their 'from' car referents are evacuated to the train owning the 'to' carriage. All other objects that refer to the 'from' carriage reside in other carriages of the same train. These objects are evacuated to the last carriage of the *same* train. If there is no free space left in a train, a new carriage is simply linked onto the end of the train. The 'from' carriage can now be condemned and reclaimed.

The key point is that an object that is reachable from outside the train undergoing collection is relocated to one of the other trains, clustering live objects and collapsing linked garbage structures between fewer and fewer trains until eventually they reside in a single train where they are then collected. [HM92] reasons inductively about the correctness of the algorithm leading to guaranteed collection of garbage in $O(n^2)$ car collections where $n$ is the number of

cars. [SG95] presents a slight flaw in the algorithm stemming from a pathological arrangement and mutation of objects that starves the collector from processing any other trains of the system. Seligmann and Grarup extend the algorithm to handle this case and prove the algorithm formally correct. Furthermore, they implement and evaluate the algorithm within Mjolner runtime system for the BETA programming language. The results presented show an effective elimination of lengthy "disruptive" pauses on the order of a few seconds, replaced by "a series of non-disruptive ones, each just a few dozen milliseconds long". There is however no meaningful analysis of the real-time characteristics of the collector.

### 3.5.6 Garbage-First Garbage Collection

The Garbage-First collector of Detlefs et al. [DFHP04] is a concurrent parallel copying collector designed to meet soft real-time constraints *with a high probability*. The collector targets high throughput multiprocessor server applications which operate with large memory requirements. A more accurate categorisation of the collector is that it is partially concurrent and partially parallel — the global marking phase is concurrent with respect to mutator activity, and object evacuation is performed in parallel so as to increase throughput. Certain portions of the collector phases incorporate traditional 'stop-the-world' pauses where all threads must be stopped together. The authors claim that this simplifies the collector implementation.

The collector is parameterised by two user-specified values. The first places an upper bound on the size of the heap. The second describes a soft real-time constraint that defines the maximum time that can be devoted to stop-the-world collection during a specified time slice. Note that the collector is not incremental. Instead, it works to minimise the duration of the stop-the-world pauses. If the collector is unable to schedule the next stop-the-world pause within the current time slice, the heap is grown, subject to the maximum heap size, and the pause deferred to the next time slice. It is for these reasons that the collector is said 'to meet soft real-time constraints with a high probability'.

The Garbage-First collector partitions the heap into equal sized regions which can be collected independently of one another. As such, the remembered set for a region must record all pointers into it from the other regions. The remembered sets use the *card table* mechanism of [H93] — every 512 byte *card* in the heap maps to a card table entry of one byte. The remembered sets are hash tables of cards, and each thread has its own *remembered set log* that is a sequence of modified cards. When a mutator thread modifies a pointer a write barrier dirties its card table entry if necessary and then places the card into the thread's remembered set log. A dedicated thread is triggered and controlled by various thresholds and concurrently processes each thread's remembered set log — the corresponding card table entry is reset, and then the pointer fields of all objects whose modification might have dirtied the card are examined for references outside of the owning heap region. If such a reference is found, it is inserted into the remembered set for that heap region. In a private communication Detlefs acknowledges that much of the collector's complexity stems from the implementation

of the remembered sets and optimisations that allow parallel collector threads to manipulate the remembered sets unhindered.

To limit mutator thread contention during allocation, threads privately allocate to thread-local buffers within a heap region. The collector maintains a cost model for the collection overheads of each region, and uses this to determine the set of regions that should be collected within the next time slice.

The Garbage-First algorithm is a complete algorithm that employs a Yuasa-style snapshot-at-the-beginning concurrent marking algorithm. Unlike other algorithms (such as the Train) the concurrent marking phase operates so as to allow regions to be collected arbitrarily and independently of one another. Marking bitmaps are used in conjunction with an explicit mark stack to track those grey objects that have been marked but are yet to be recursively scanned. The root set marking phase requires a full stop-the-world pause of the mutators while those objects immediately reachable from the roots are marked. Additional data structures are initialised in preparation for the subsequent concurrent marking phase. Because mutator threads may update the live memory graph as the collector traces it a concurrent marking write barrier must record the old value of a field before it is overwritten with a new pointer value. This maintains the consistency of snapshot-at-the-beginning heap snapshot. This write barrier operates in a similar way to that of the remembered set write barrier — each mutator thread has its own local marking buffer, which when filled is enqueued in a global set of buffers on which the collector's concurrent marking thread operates.

During the concurrent marking phase the amount of live data in each heap region is (cheaply) determined. The algorithm uses this information along with current and expected pause time information for the immediate phases to determine the sets of regions and the per-time-slice sequence in which they should be collected. The algorithm prioritises those regions that contain the greatest amounts of garbage and least amounts of live data — a 'Garbage-First' strategy.

Detlefs et al. do not report the collector's minimum mutator utilisation characteristics arguing that such a metric is more applicable to collectors with hard real-time constraints. They argue that 'its inherently worst-case nature fails to give any insight into how often the application failed to meet its soft real-time goal and by how much'. Instead they approximate three statistics that attempt to quantify the 'how often' and 'by how much' an application violates its soft real-time goal using one millisecond quantised increments. The first presents the percentage of time slices for which stop-the-world pauses violated the specified pause time goal. The second presents the average amount by which the violating time slices exceed the pause limit expressed as a percentage of the target minimum mutator time in a time slice (i.e. the time slice minus the pause time limit). While the third presents the worst-case value for metric two, as opposed to the average case. A detailed evaluation is presented, but in summary, across a range of heap sizes up to 1GB and across the entire SPEC JBB 2000 benchmark suite, less than 5% of all time slices violate the target pause limit, while on average less than 2% do so.

### 3.5.7 The Metronome

Bacon, Cheng and Rajan's Metronome collector [BCR03b] is an incremental uniprocessor collector targeted at Java embedded real-time systems and is capable of achieving far higher minimum mutator utilisation during collection whilst requiring far less memory overhead than other real-time collectors. Furthermore, providing the application is accurately characterised in terms of maximum live memory usage and average allocation rate over a collection interval, time and space bounds can be guaranteed.

The collector is a *mostly non-copying* collector that hybridises an incremental Yuasa mark-sweep collector, employed as standard, with a Brooks variant of a Baker-style incremental copying collector, that is employed during periods of, or on detection of, high fragmentation. The mark-sweep collector is (obviously) a non-moving snapshot-at-the-beginning algorithm and allocates objects black.

The 'heap' is segmented into fixed sized pages, each of which is a power-of-2 segregated free list and is divided into blocks to service allocations of a particular size using a best-fit (smallest first) policy. The collector's architecture is defined based on the premise that fragmentation is rare and so the objects need not be moved. As a result, the primary collector is an incremental mark-sweep collector that maintains the weak tricolour invariant using Yuasa's write barrier. When fragmentation of a particular page is detected [BCR03a] an incremental Baker-Brooks style copying collector evacuates the live objects to another (mostly full) page with the unconditional read barrier maintaining a to-space invariant *for those objects that have been evacuated*. The collector is solely responsible for the evacuation of objects and forwarding pointer updates. This eliminates the mutator-collector coupling that can so adversely impact MMU as the mutator is forced to copy and scavenge objects when dereferencing pointers during the mutator's scheduled quantum.

The marking phase of the collector is augmented with a step that redirects from-space pointers encountered during traversal to their to-space copies. As a result, at the end of the marking phase, the fragmented pages from which live objects were evacuated at the last copying collection can now be reinitialised as completely free free list pages — all cells on the page are swept onto the free list.

In order to ensure that the collector does not exceed its scheduled quantum when performing atomic scavenging operations, large arrays are broken into small fixed sized chunks known as *arraylets*. Arraylets are arranged in a similar way to, and their design derived from, Cheng and Blelloch's stacklets (Section 3.5.3). An overhead is incurred on array access operations, although Bacon et al. don't quantify this. Somewhat surprisingly, the Metronome does not itself employ the use of stacklets. Instead, as with a snapshot-at-the-beginning collector, the thread stacks and global variables that form the root set must be atomically copied at collector initiation. It is therefore quite possible that the collector will exceed its scheduled quantum during root set evacuation. Bacon et al. cite the reason for this as the inability of the stacklet approach to scale with large numbers of threads and the problematic snap-

shot operations that are required when a thread returns from the topmost stacklet into a lower, unsnapshotted stacklet. A high frequency of stack push and pop operations is therefore problematic when reasoning about real-time bounds. They suggest the solution is to weaken the snapshot-at-the-beginning approach when handling the thread stacks — no stack reference may "escape" the stack without being recorded in the mutation log. As a result Yuasa's write barrier must be augmented with Dijkstra's, so that both old values *and* now new values written from stack variables into heap objects are recorded. It does not appear that the Metronome, as described and evaluated in the literature, implements this technique and certainly neither the cost of the Yuasa write barrier or this augmented write barrier are evaluated. In a private correspondence with the authors, they explain that this evaluation is unnecessary because it is uncontentious, having been performed for their runtime system (MMTk of Jikes RVM) by others, namely Blackburn et al. [BH04].

The collector and mutator can be scheduled using either Baker's work-based scheduling algorithm or the time-based algorithm that has previously been introduced. In order to guarantee that the collector achieves guaranteed provable real-time bounds, the application must be correctly characterised ahead of time. More specifically, both the maximum amount of live data in use and the peak allocation rate over a collection interval (for all collection intervals) must be specified. Furthermore, the collector must be parameterised by its tracing rate. With these characteristics specified, the performance of the system may then be tuned using three inter-related parameters: the system's total memory consumption, the minimum mutator utilisation, and the minimum time window for which the MMU is calculated. For the specified time resolution, either the MMU or the maximum memory consumption is user specified and the other dependent parameter is then calculated.

### 3.5.8 STOPLESS, CHICKEN, and CLOVER

In [PFPS07] Pizlo et al. present STOPLESS, a concurrent collection algorithm that targets real-time systems and supports lock-free parallel computation with fine-grained synchronisation. [PPS08] introduces two additional algorithms, CHICKEN and CLOVER, that trade the complexity of STOPLESS for its guarantees on lock-freedom and object relocation resulting in algorithms with lower overhead on average but degraded worst-case performance.

Operations are deemed to be "lock-free" or "non-blocking" if it can be guaranteed that any one of the (mutator) threads can make progress after the system (i.e. the threads collectively) has executed a finite number of steps. In practice, mutator threads never wait for a collector thread for more than a small and bounded number of steps. This ensures that while individual threads can potentially be starved, "globally" the mutator makes computational progress.

The three algorithms all share a common infrastructure that is implemented within the Bartok compiler and its runtime system for C#. The infrastructure as presented in [PFPS07, PPS08] avoids the complexity of real-time collector scheduling by devoting the collector threads to concurrent execution with the mutator threads on "spare" pro-

cessors. As such, the mutator threads are prevented from executing on these processors. In their most basic configuration the collector executes two threads, one responsible for mark-sweep reclamation, and the other for eliminating fragmentation through copying compaction collection. All the algorithms execute the same concurrent mark-sweep collector in the first thread and thus the algorithms differ solely in how they implement the compacting collection thread. The authors suggest the use of the formulae or mechanisms of [OBYG+02, BCR03b, Hen97, Rob02, RH03] in order to determine how many collectors must be executed concurrently in order to keep up with the allocation rate of the mutator threads.

The concurrent mark-sweep collector builds on the on-the-fly mark-sweep collector of Doligez-Leroy-Gonthier [DL93, DG94] and the subsequent enhancements of Domani et al. [DKP00, DKL+00] yielding an allocator and collector that are lock-free i) during allocation, ii) when manipulating the mark-stack, and iii) when load-balancing using a work-stealing mechanism. Furthermore, determination that the collector's termination condition has been satisfied is done without blocking any of the mutator threads. Fundamental to the operation of the concurrent marking phase is an incurred dynamic space overhead of one word for *every* heap object. This word is used to store the object's mark bit and also to link the object on to a thread-local linked list of grey objects that have yet to be scanned. It is worth noting that only when the compaction and mark-sweep phases execute serially with respect to one another can the extra word reserved for non-destructive forwarding pointer installation during copying compaction be re-used for this marking word. When the phases are allowed to execute concurrently the algorithms incur a dynamic space overhead of an additional two words per heap object.

CoCo is the concurrent compaction algorithm employed in the compaction thread of STOPLESS. Because the overhead of the three compaction algorithms, CoCo, CHICKEN and CLOVER, is relatively high and increases with the number of objects to be moved, they are only *partial* compaction algorithms, with the mark-sweep collector tasked with the completion of (root) pointer updates to relocated objects.

The prerequisite to the relocation of an object by CoCo is that it has been marked for movement (using a reserved header bit, which is most usually set during the sweep phase) and threaded onto a list for processing by CoCo. CoCo is then responsible for creating and populating the to-space object copy, ensuring a coordinated switch to a coherent to-space copy occurs whilst maintaining lock-freedom. To do this CoCo forwards the from-space object to a much larger temporary *wide* to-space object and employs both read and write barriers to ensure the correct copy is accessed and that it is coherent with respect to updates. Once a wide version of the object has been allocated in to-space, a forwarding pointer to it is installed in the from-space copy in one of the additional reserved words of the object. A read barrier now enforces access to the to-space temporary wide object copy. CoCo then evacuates each of the payload fields from the from-space copy to the wide object. As each field is copied a *status* field is associated with it in the wide object which the read barrier uses to determine

where the up-to-date version of the field resides. The wide object is larger than the original and so named, because each payload field carries an additional dynamic space overhead of one word (or two on a 64-bit architecture) for the status field. Just as the read barrier inspects the status word to ensure the most up-to-date copy is used, so a write barrier must inspect the word to determine which copy to keep up-to-date. The write barrier ensures that updates to the objects are not lost using the processor's compare-and-swap (CAS) instruction. Once population of the wide object has completed an original sized to-space copy of the object is allocated and each of its fields are populated. Again, read and write barriers, the status words of the wide object and CAS instructions are used to ensure that this narrow to-space copy of the object is eventually populated with the up-to-date fields and is then kept up-to-date. In essence, the status words, in combination with the object-level forwarding pointers, act as field-level forwarding pointers, redirecting, as necessary to one of the three object copies.

The authors of [PFPS07] claim a novel optimisation technique for the implementation of the fast and slow paths of their read and write barriers. The fast path handles the case where the barrier need take no action and amounts to checking a variable to determine the collector phase. They optimise the barriers, removing them for phases where they are unnecessary, by providing specialised copies of the code. We remark that this is similar to our barrier optimisation technique as originated in [CFS+00] (Chapter 5), and further developed in our method specialisation and virtualised execution environment for Java [CFNP07, CFNP08] (Chapter 9).

STOPLESS was designed with the specific remit of ensuring that the objects that have been marked for copying are actually copied whilst ensuring that mutator threads are never blocked when performing a heap operation. While the mutator threads have a progress guarantee, the CoCo collector thread does not — in the worst case, if writes occur repeatedly to the same field in the temporary wide to-space object, the copying thread can be delayed indefinitely by the repeated re-execution of the CAS operation.

CHICKEN and CLOVER take an alternative approach to STOPLESS. Instead of employing complex heavyweight barriers to ensure objects are copied, they rely on chance. CHICKEN exploits an optimistic assumption that mutator threads are unlikely to modify the object undergoing evacuation from from-space during the brief period that it is being copied. If the assumption is violated for that object, then its associated copying operation is aborted. Hence the mutator and collector play "chicken" with the object.

Unlike STOPLESS, CHICKEN copies objects in their entirety without the use of an intermediate wide object. Instead, like the Metronome, a Brooks-style unconditional read barrier and its associated write barrier are employed. The indirection-based read barrier requires an extra word in each object for the forwarding pointer. Having dereferenced the forwarding pointer, the write barrier allows writes to an object to proceed unhindered if the object has been copied (in its entirety) to to-space, or if it has not been marked for copying. If the write barrier finds that the object is in neither of these states, then the optimistic assumption above has been violated and the copy operation is aborted. The abort is implemented with a

CAS operation which can only fail if another thread has already aborted the copy, or if the copier has completed the copy. As a result, the write barrier is wait-free because it always completes in a constant number of steps. Similarly, the use of a CAS operation for the installation of the forwarding pointer when redirecting the from-space object to its to-space copy ensures that the copy operation is wait-free and aborts the copy if a mutator thread writes concurrently to the object. Unlike CoCo, CHICKEN does not guarantee that marked objects are actually copied and, as a result, there is a danger that objects whose copy operations have been aborted, will *never* be copied.

CLOVER, unlike CHICKEN, guarantees to relocate an object that has been marked for copying. However, lock-freedom is not guaranteed in the worst case and the mutator threads may be forced to block. Such blocking is designed to occur with "negligible" probability. CLOVER is, however, simpler to implement than STOPLESS and operates with lower time and space overheads. CLOVER exploits an observation from probability theory — that a value chosen at random from a range of $2^l$ values has a probability of $2^{-l}$ of being written by the mutator at each store operation. Thus, CLOVER proceeds by generating a random value and subsequently assumes that this value will never be written in normal execution by the mutator. As a result, the collector is free to use this value, in conjunction with a CAS operation, to mark a field as obsolete (in a similar way to CoCo's status word), once it has been evacuated to its to-space copy. To verify this assumption and guarantee correctness, a write barrier verifies that no mutator store operation ever writes this reserved value. If the write barrier traps an attempt at such a store, the mutator thread is blocked until the copying phase terminates.

All three copying compaction algorithms require a termination phase to be executed that "fixes up the heap" and requires a complete mark-sweep collector cycle to be executed to eliminate any remaining from-space references and flip them to their to-space copies. During concurrent execution of this phase a read barrier, similar in operation to the eager read barrier of the Metronome, must be employed in conjunction with a mutation logging write barrier to ensure that reads and writes occur to the to-space object copies. Furthermore, the collectors use a *soft handshake* mechanism to communicate each of the collector phases to the mutator threads, that stalls the collector threads until each of the mutator threads has acknowledged the phase change. Because collector threads are devoted to their own processors this avoids mutator-collector scheduling issues that would otherwise block the mutator threads. In addition to avoiding the issue of collector scheduling, the collectors do not implement incremental stack scavenging, nor are large arrays broken up into arraylets, all of which must be addressed to eliminate pauses which violate real-time bounds when employed in a production environment. It is for this reason that we address these issues in our production collector of Chapter 7.

[PFPS07, PPS08] present experimental results for the three collectors with a focus on throughput, responsiveness and execution times relative to the vanilla stop-the-world mark-sweep collector. No detailed analysis of barrier overheads or code bloat as a result of the

code specialisation for barrier optimisation are presented. STOPLESS imposes an execution overhead for the majority of benchmarks of between 0.4 and 1.6 over that of the stop-the-world mark-sweep collector, with a geometric mean of 1 across all benchmarks. The first additional reserved header word halves the heap space available, while the addition of the second word, as presented, appears to impose very little additional overhead on this. The temporary wide objects impose an additional overhead in the worst case of about 40% although on average it is closer to 10%. Because i) of the way in which the space overheads are calculated, ii) the collector is allowed to expand the heap, and iii) because the copier works at a page-level granularity, it is not clear exactly how these space overheads translate into an increase in execution time, number of collection cycles, or the true reduction in available memory for a fixed-sized heap. When benchmarked against the concurrent mark-sweep collector in [PPS08], both STOPLESS and CLOVER impose a relative overhead of about 1.22 on average, whilst CHICKEN imposes an overhead of around 1.17.

To demonstrate short pause times and mutator responsiveness the three collectors are executed within the context of a benchmark that simulates the frequency of high quality audio samples. The benchmark generates events at a rate of 108KHz with processing computation required to end before the next event fires. CHICKEN and CLOVER perform better than STOPLESS. CHICKEN is able to consistently handle a workload that copies an array of 256 reference values. STOPLESS and CLOVER are not, instead consistently managing a smaller task of 64 reference values. For this smaller task CHICKEN delays task completion by about 1 microsecond, CLOVER by around 3 microseconds and STOPLESS by 4 microseconds. The worst-case pause time was recorded as: 70 milliseconds for the stop-the-world mark-sweep collector, 5 milliseconds for STOPLESS, 3 milliseconds for CLOVER and 1 millisecond for CHICKEN.

### 3.5.9 The Real-Time Specification for Java

The Real-Time Specification for Java (RTSJ) [Pro00, Pro09] enhances the original language and runtime system specifications with a set of features and runtime system enhancements to resolve many of the problems encountered when using Java in real-time (and embedded) environments. More formally, RTSJ equips application developers with a set of APIs, semantic JVM enhancements and JVM to operating system interface modifications that enables a developer to reason about the temporal characteristics of applications. Particular emphasis is placed on thread scheduling and garbage collector predictability to avoid delays from collector pauses and jitters. RTSJ provides the following key enhancements:

- **Real-Time Tasks and Thread Scheduling** — RTSJ introduces periodic, aperiodic or sporadic task schedules each with optional deadlines. If a deadline is missed then a *deadline miss handler*, previously associated with the task, is fired to execute remedial actions or dynamically modify and update the schedule of outstanding tasks. These tasks are executed within the context of a new real-time thread type. The JVM-

OS interface is enhanced so that real-time threads can be scheduled with one of 28 different fixed priority levels. These real-time threads are bound to underlying kernel threads and typically rely on the underlying real-time operating system thread scheduler to enforce multiple priority schedules and the preemption of lower-priority threads by higher-priority ones. RTSJ incorporates *priority inheritance* into its scheduling policies so as to avoid *priority inversion* race-conditions that prevent a higher priority thread from running because it is blocked by a lower priority thread that has control of a particular resource. Real-time threads are essentially scheduled on run-until-blocked as opposed to time-slicing — the scheduler preempts a real-time thread when a higher priority real-time thread is ready to execute. Real-time threads are subclassed by a *no-heap real-time thread* type that targets hard real-time applications and, as such, is unable to allocate or manipulate values in the garbage collected heap. This restriction prevents such threads from being interrupted by the garbage collector and they are therefore not subject to collector delays or jitters. No-heap real-time threads allocate memory in segregated non-garbage collected memory regions.

- **Non-Garbage Collected Memory Regions** — RTSJ adds two further types of segregated memory region in addition to the garbage collected heap. Objects which are allocated into an *immortal* memory region persist throughout the lifetime of program execution and the region is only condemned and reclaimed when the program terminates. When a *scoped* memory region is allocated it persists for the duration of execution of the current block scope. When the scope is exited, the region is immediately condemned and reclaimed. For example a scoped region can be defined for the scope of a method, at the granularity of the scope of one particular branch of a conditional, or if necessary an arbitrary scoped block of just a few statements. Because these regions are not subject to garbage collection, there is no risk of mutator disruption due to GC delays or jitters.

- **Direct Physical Memory Access** — RTSJ has augmented the language specification with primitives for direct access to the physical memory of the host machine. As a result, hardware (DMA) device drivers may now be written purely in Java. Furthermore, subject to security restrictions, Java applications are able to communicate and synchronise with native applications running on the host via physical memory accesses.

- **Asynchronous Event Handling** — RTSJ allows asynchronous even handlers to be defined to respond to events that occur externally to the JVM without disrupting the "temporal integrity" of the hosted real-time application. Furthermore, the JVM safe-point mechanism, at which threads yield and transfer control, have been enhanced to cater for asynchronous interruption by another thread in a safe and controlled manner.

- **High-Resolution Timers** — The RTSJ JVM-OS interface has been enhanced to expose the high-resolution nanosecond-accurate timing infrastructure of the underlying real-time operating system.

Given the above, one might be forgiven for thinking that garbage collection has been deemed too disruptive to play a part in real-time Java environments. This is not so. In recognition of its importance to developing safe, scalable and maintainable software, where it may neither be practical nor possible to avoid dynamic heap allocation (consider shared data mutation, coordination, and communication between threads) both IBM's and Sun's RTSJ platforms have incorporated real-time garbage collectors. IBM have incorporated the Metronome into their real-time WebSphere application server while Sun have incorporated a variation of Henriksson's approach (Section 3.3.2). The Sun collector is a fully concurrent collector benefitting from no stop-the-world phases, that runs Henriksson's algorithm in one or more real-time threads, each of which run at a priority lower than all instances of the no-heap real-time threads. The collector is parameterised by a memory threshold and a maximum priority so that the JVM can raise the priority of the collector up to the maximum to ensure that sufficient collector progress is made whilst still allowing higher priority critical threads to preempt the collector in conjunction with expanding the heap to satisfy its allocation requests.

# Chapter 4

# The Non-stop Spineless Tagless G-machine

GHC, The Glasgow Haskell Compiler is an optimising compiler for the *de facto* lazy functional programming language Haskell, and inherent to its efficient compilation is a *pure* dynamic dispatch mechanism. Although GHC is now the fastest state-of-the-art production compiler and interactive environment for Haskell execution, its roots are buried firmly in lazy functional language research. The architecture of the compiler has been designed in a modular pipeline so as to better facilitate research via module replacement and allows us to more easily investigate language and runtime system optimisations that focus on the exploitation of dynamic dispatching.

Over the following sections we discuss the basic implementation techniques that span (lazy) functional programming language implementations, with particular focus on the abstract machines that underpin them.

In Section 4.6 we introduce the *Non-stop Spineless Tagless G-machine*, that augments the Spineless Tagless G-machine — GHC's abstract machine, with very basic modifications to add support for efficient incremental garbage collection of the machine's heap-allocated closures. The *While and Field Collector* that results has never been implemented, and is in no way complete, but is fundamental to our general approach and subsequent implementations.

## 4.1 Lazy Functional Language Implementation

The $\lambda$-calculus [Chu41], provides a 'universal' model of computation, which means that any computation that can be expressed as a Turing machine can also be expressed using $\lambda$-calculus. It is, at some level of abstraction, at the heart of every programming language, and allows the expression of (anonymous) functions (or function abstractions), function applications and recursion. In essence, it 'provides a formalism for expressing functions as *rules of correspondence*

between arguments and results' [Han94].

The evaluation of $\lambda$-terms is performed by the application of *reduction rules*. The current $\lambda$-expression being reduced is called the *redex* (short for reducible expression) and the evaluation of expressions is performed by the repeated reduction of its redexes until no more redexes exist. When a $\lambda$-expression cannot undergo further reduction, it is said to be in *normal form*. Abstractly then, a functional program can be thought of as consisting of a series of unreduced $\lambda$-expressions that are evaluated, to a normal form using reduction rules. A naïve scheme for the evaluation of a program simply starts at the top-level expression and continues to select, reduce, and overwrite each redex with the reduction result until there are no more redexes.

Strict (eager) and lazy functional language implementations differ in their choice of which redex to reduce next — the reduction order. *Applicative-order* reduction (AOR) reduces the leftmost innermost redex first. This results in the evaluation of the argument expression before attempting to substitute a variable into it. In contrast, *normal-order* reduction (NOR) reduces the leftmost outermost redex first. This delays the evaluation of any redexes within the argument expression until there is no alternative redex available, by substituting variables before attempting any reductions within the argument expression. Eager evaluation employs AOR to reduce expressions and corresponds to the *call-by-value* mechanism of imperative languages. If however, NOR is used, the calling mechanism corresponds to *call-by-name*. However, this strategy risks the multiple evaluation of expressions. To prevent this, the concept of *sharing* must be introduced that 'links' the copies of the argument expression, and results in all copies being updated with the value that results from the first evaluation of the expression. The addition of sharing to call-by-name semantics results in *call-by-need*, which in a functional language behaves equivalently, because it is not (naturally) possible for an argument expression to perform side-effects. The final addition of *lazy constructors*, where arguments to constructors are assumed to be unevaluated, results in lazy evaluation.

### 4.1.1 Graph Reduction

There are two models of reduction and expression evaluation, *environment-based* and *copy-based*. In an environment-based implementation, such as Landin's SECD machine [Lan64], an *environment* is associated with each instance of a $\lambda$-expression that maps the bound values to the free variables of its body. An alternative approach is that of *graph reduction* [Wad71], where an expression is represented as a directed graph. Each node represents a function call and its subtrees represent the arguments to that function. Subtrees are replaced by the expansion or value of the expression they represent. This is repeated until the tree has been reduced to a normal form value in which there are no more function calls. In practice reduction proceeds until the resulting expression is in a restricted normal form known as *weak head normal form* (WHNF). This avoids having to resolve name clashes that can occur during reduction when the argument expression contains free variables which occur bound in the body of the applied abstraction. As a result, evaluation of an expression stops before the

function body of an expression is entered, no free variables are therefore encountered.

Constants are represented at the leaves of the tree; function applications of the form $FA$, the application of $F$ to $A$, are represented using *apply-nodes*,'@', with $F$ on the left-hand leaf and $A$ on the right; currying is used to write multi-argument functions as a series of one argument function applications to yield the binary pair that is needed for the tree representation; finally, $\lambda$-abstractions of the form $\lambda x.M$ are represented using a $\lambda$-*node* '$\lambda$', with the bound variable $x$ on the left-hand leaf and the (subtree) representing the body $M$ on the right.

Consider an expression graph $G$. The *left spine* of G, corresponding to the chain of application nodes along with the root, is found by taking successive left branches from the root node. Evaluation of an expression within G, and hence reduction, begins by locating the next redex node in the graph. For NOR all that must be done is to traverse (*unwind*) the left spine of G until the node currently being traversed is no longer an apply-node.

Having located the redex for reduction, *graph transformations* are performed that are meaning preserving and which correspond to sub-expression reduction. The application of a $\lambda$-abstraction sub-graph to an argument sub-graph is a variant of the $\lambda$-abstraction sub-graph where each leaf node representing the bound variable has been replaced with the argument sub-graph. To provide sharing and hence cater for lazy-evaluation, each pointer to a bound variable leaf within the function graph is replaced with a pointer to the root node of the argument sub-graph, the redex node is then overwritten with the result (the root of the modified function graph). Unlike the environment-based mechanism of the SECD machine, graph reduction is copy-based — the application reduction rules result in the copying of the body of the applied $\lambda$-abstraction so that the substitution of bound variable nodes occur within the copy itself. The copy of the body requires new graph nodes for every node other than its root node, which is written into the redex node as required. Having performed the argument substitution, the redex is physically overwritten by the result of its reduction. This updating of the redex may result in other nodes in the redex becoming 'disconnected' from the root node. If these nodes are not shared then they are no longer required in the evaluation and the space that is allocated to them can be reclaimed by the garbage collector.

The model of graph reduction, as described, suffers an unfortunate inefficiency, namely that when function application is performed, the function body must be copied before the binding of free variables to their values can occur. One of the key factors governing the practical implementation of a functional programming language is the efficient implementation of function application. Most generally this is achieved by generating a fixed code sequence for each $\lambda$-abstraction. Recall that in the SECD machine such code sequences are associated with an environment which contains the values of the free variables for each application. The alternative approach, most usually adopted in the graph reduction model, is based on *combinator*s. A combinator is a higher order function whose body is defined solely by applications of primitive functions, and other combinators, each of which is a closed expression — has no free variables. The use of combinators eliminates the need to perform the copy by trans-

forming each $\lambda$-expression using variable abstraction into a fixed set of combinators [Tur79] or by *lambda lifting* into Hughes' *supercombinator*s [Hug82]. Lambda lifting is the process of abstracting the free variables from one $\lambda$-body at a time and replacing the $\lambda$-abstraction by the partial application of a new combinator which has one more argument than the number of free variables abstracted. The extra argument corresponds to the bound variable of the $\lambda$-abstraction.

The application of a supercombinator to its arguments is achieved by overwriting the application node within the expression graph with an instance of the supercombinator body where the formal parameters are replaced by pointers to the arguments. As a result, a supercombinator implementation naturally lends itself (via compilation) to native execution on stock hardware. Since they have no free variables, each supercombinator can be compiled to a fixed sequence of instructions that when executed, construct an instance of the supercombinator body. Furthermore, because by definition, any lambda abstractions occurring in the supercombinator body must also be devoid of free variables, they do not need to be copied when instantiating the body.

Although the *naïve* reduction model requires the copying of the function body, it has several advantages over the environment-based model, notably, that sharing is supported without the need for an environment structure and that NOR evaluation is easily expressed and relatively efficient to implement. These advantages promote graph reduction as a most suitable computational model for the implementation of lazy functional languages. It is for this reason that the *G-machine* [Joh84], a compiled lazy implementation for functional languages that results in an abstract machine, was designed, and a variant of which has been adopted by GHC.

## 4.2 The G-machine

The G-machine, like the SECD machine performs execution steps through state transitions and a state is represented by a 7-tuple ($O$,$C$,$S$,$V$,$G$,$E$,$D$) where:

- $O$ — is the output produced so far, consisting of a sequence of integers and booleans.
- $C$ — is the *G-code* (the language of the G-machine) sequence currently being executed. When mapped to the target machine it is the target code of the currently executing function and the program counter.
- $S$ is a stack of pointers into the graph being evaluated. On the target machine, this maps to a data area for the pointer stack and a stack pointer register, *ep*.
- $V$ — a stack of basic values, integers and booleans on which the arithmetic and logical operations are performed. On the target machine, it is mapped to the system stack and stack pointer, *sp*.

- $G$ — the graph, a mapping from node names to nodes. On the target machine, this maps to the heap and a register pointer, *hp*, that points to the next free location. It is this structure that is of the most significant interest where garbage collection is concerned, for it is storage area for structures generated during the evaluation of the program. When these structures are no longer used in evaluation, they become garbage cells and can be reclaimed by the garbage collector. New structures are added to the heap by storing them at the current free location indicated by *hp*. When the storage space in the heap is exhausted, the garbage collector is invoked to release the memory currently allocated to the garbage cells.

- $E$ — a global environment, mapping from function names to pairs consisting of the number of curried arguments and its code sequence. E corresponds to the code segment on a target machine.

- $D$ — is a dump used for recursive calls to the expression evaluation function. It is a stack of pairs consisting of a stack of node names, $S$, and a G-code sequence, $C$, before evaluation. On a target machine, this corresponds to the system stack and stack pointer, *sp*. Unlike in the abstract machine, only pointers into the $S$ stack and system stack are pushed, not entire stack dumps.

The G-machine implementation uses four compilation schemes [Jon87, FH88]:

- The $C$-scheme — which generates the code to *construct* graphs representing expressions.

- The $E$-scheme — which generates code to *evaluate* expressions represented by the graphs constructed from the code generated by the $C$-scheme, leaving a pointer to the result at the top of the evaluation stack, $S$.

- The $F$-scheme — which generates the *function bodies* of user-defined functions, reducing the graph constructed by the code from the $C$-scheme, to yield the value that results from the function application.

- The $B$-scheme — similar to the $C$-scheme, but compiles code to evaluate integer and boolean valued expressions leaving their value on the dump stack, $D$.

The execution of a compiled G-machine program is based on the computational model of the graph reducer described in the previous section. The $S$ stack is used to unwind and rewind the spine of a graph and together with $D$, to evaluate function applications.

## 4.3 The Spineless G-machine

Each reduction step within the G-machine is fine-grained with an update occurring at each step. The G-machine starts from the root of the expression undergoing evaluation, and

descends the left-branching chain of binary application nodes. It is this descent, the unwinding of the spine, that copies the arguments from the graph nodes stored in the heap onto the stack. As soon as the reduction step is complete the spine is rewound and written back to the heap, and the update is performed. In the next reduction step the spine is immediately unwound back onto the stack. This requires that the entire result be constructed in the heap so that the graph can be updated. The reduction site is likely to be the target of the next reduction as this result is further evaluated. It is therefore highly likely that part or all of this result structure will be immediately discarded, and must subsequently be garbage collected. As a result, allocation and collection of these "intermediate" structures in the heap is costly and can often, by employing compile-time analysis, be avoided.

The *Spineless G-machine* [BPJR88] is a G-machine variant which addresses exactly this problem by incorporating *spineless reduction* and *sharing analysis* so that the graph is updated only when there is a risk that the result of an expression will be recomputed if its root is not overwritten with the reduced value, i.e. *loss of sharing* would occur. Clearly, if the graph node which is to be updated is not shared, then there is no risk of loss of sharing occurring by omitting the update.

The association of graph updates with sharing rather than each reduction step is the defining feature of the Spineless G-machine. The G-machine's evaluation mechanism is modified so that it proceeds, so far as is possible, without performing any updates. Consider the following (example from [BPJR88]):

$fx_1...x_n = gD_1...D_m$

where $g$ is a user-defined function, not a primitive. At the point of evaluation of $f$, its arguments, $x_1...x_n$, are on the top of the stack, and it is initiated with the execution of the code for $f$, which:

- Performs the *argument satisfaction check* to verify whether there are sufficient arguments on the stack, i.e. that $g$ has at most arity $m$, or whether a partial application must be built.
- Constructs the closures for $D_1...D_m$ and pushes a pointer to each of them on to the stack.
- *Squeezes* out the $n$ arguments by sliding the top $m$ elements down $n$ places.
- Executes the code for $g$.

The evaluation mechanism only deviates from the above for the *rarer* case where there is a risk of loss of sharing occurring. In order to determine this, each "updateable" node within the graph must be detected. An updateable node is one which is both shared and is also reducible (i.e. is not already in WHNF). There are several ways in which an updateable node can be detected, each, with varying levels of precision and overhead, which must at least exactly capture the set of updateable nodes, but may in fact capture a superset. The resulting

set is that of the *possibly-updateable* nodes. The Spineless G-machine represents these nodes with *SAP* tags — each possibly-updateable node is tagged (bits in its object header reflect this) and the evaluation mechanism always updates such nodes with their WHNF value when evaluation completes. Non-updateable application nodes are tagged with the original *AP* node. The following strategies are examples of those that can be used to build the set of possibly-updateable nodes:

- On construction of a closure assume it is updateable.
- Perform sharing analysis, a non-local compile-time program analysis, that determines whether an argument can be shared. In the example, if none of the arguments to $g$ can be shared then the code for $f$ can construct non-updateable closures for $D_1...D_m$.
- On construction of a closure, assume it is non-updateable, and at run-time dynamically change the tag to possibly-updateable if it becomes shared.

In the example above, were $x_i$ to replace $g$ in the function position within $f$'s body, then $x_i$ is now shared, and failing to update it risks a loss of sharing, assuming it is indeed reducible. As a result, $x_i$ is possibly-updateable. The evaluation mechanism treats closures that are tagged as possibly-updateable slightly differently to non-updateable closures so that when their WHNF value has been computed, they can be updated with it. It should be apparent that the evaluation mechanism must detect when evaluation of $x_i$ is complete. This simply requires it to "notice" when one of the stacked arguments $D_1...D_m$ is required for evaluation to proceed — $x_i$ has then been reduced to a function applied to too few arguments and is in WHNF. To "notice" this, when $x_i$, is first presented for evaluation, an *update* frame, containing its address is pushed onto the *dump* stack before continuing with its evaluation. When $x_i$ has been reduced to WHNF, as a function, $g$, applied to too few arguments, the argument satisfaction check at the start of the code for $g$ detects that too few arguments are present on the stack and initiates the update operation, which:

- Builds a closure, $C$, binding the application of $g$ to the arguments on the stack.
- Pops the update frame, retrieving the address of the updateable closure $x_i$.
- Performs the update.
- Reattempts the evaluation of $C$, applying $g$ to the arguments on top of what is now a deeper stack.

In contrast to the G-machine, the Spineless G-machine never writes the spine of the expression being evaluated out into the heap unless it is reducible and not until it has been fully reduced to its WHNF value. It is for this reason, the lack of manipulation of an (explicit) spine, that the Spineless G-machine is so named.

## 4.4 The Spineless Tagless G-machine

The Spineless Tagless G-machine [PJS89, PJ92], further enhances the G-machine, building on the Spineless G-machine above. For the majority of abstract machine implementations, compilation rules are defined which generate a sequence of abstract machine instructions from the high-level source code, each of which has precisely defined operational semantics using state transitions — the G-machine compiler generates G-code. When considering a supercombinator implementation, prior to this point, earlier compilation phases will have removed any syntactic sugar, type-checked the program and most importantly, performed lambda lifting. Unfortunately many abstractions are lost during these earlier phases. In particular, the stack is used to store intermediate variables. When generating the abstract machine code, the code generator must simulate stack operations if it is to eliminate much of this unnecessary stack usage by storing the values in registers. The code generator of the Spineless G-machine takes exactly this approach. However, not only does this require the introduction of named variables for each intermediate value, complicating code generator implementation, but the subsequent handling of the abstract machine code is more cumbersome and it is harder to optimise. As a result, when designing the STG-machine, Peyton Jones et al., take a unique and rather more exotic approach that results in a flexible abstract machine that is able to benefit from retaining the abstractions at a higher-level. The abstract machine language is *itself* a small and concisely defined functional language, with the standard denotational semantics. It is not unusual for functional-language compilers to represent their intermediate languages in this way. However, in the STG-machine, in addition to these denotational semantics each language construct also has corresponding operational semantics for the same language which can be (and are!) expressed using a state transition system. For example, the following STG language constructs map directly to operational mechanisms within the abstract machine:

- Function application — Tail call
- 'Let' expression — Heap allocation
- 'Case' expression — Evaluation
- Constructor application — Return to continuation

Compilation of a Haskell program proceeds by translating the source-level program into its *Core* language form. The definition of the Core language is sufficient to express the full range of Haskell programs efficiently, and no more. Programs are represented in the Core language by a series of supercombinator definitions. 'Simple' variables of a supercombinator are better explained using an example:

```
square x = x * x
main     = square 2
```

Both `main` and `square` are supercombinators and $x$ is a simple variable.

When a supercombinator has no arguments, it is called a *constant applicative form* (CAF), of which, `square2` is an example:

```
square2 = square 2
```

The core language is an extended $\lambda$-calculus that contains the additional expressions: `let`, that allows local definitions within supercombinators, `letrec` that makes possible recursive definitions, and `case` that eliminates the complex implementation of pattern matching, whilst allowing Haskell to support it. In its Core language representation syntactic sugar has been removed; type checking performed; systematic overloading of functions and operations resolved; and simple single-level switched case expressions have replaced pattern matching. A number of optimisations and program analyses (including those of sharing and strictness analysis) are performed as a set of meaning-preserving program transformations.

The Core language representation is then translated into the *STG language* the abstract machine code which is then passed to either an *Abstract C* pretty-printer (standard C code encoding STG-machine data types) or to a native-code generator. The obvious benefit of pretty-printing C code is the portability that results since the majority of architectures support the standard GNU tool chain, and in particular, the `gcc` C compiler.

Not only is the STG-machine spineless, it is also *tagless*. Tagged implementations, such as the G-machine, require that objects are tagged so that heap objects built with different constructors can be distinguished from one another. The STG-machine eliminates tags through the uniform representation of its heap objects, as closures, by placing a code pointer in their first field. The object need not now be distinguished, because each object is handled by the code pointed to by this code pointer, so on evaluation, all that need be done is to jump to this code. *All* objects including data objects within the STG-machine are represented and evaluated in this way, the only real exception is in the representation and handling of unboxed data types. This is in contrast to the G-machine and its Spineless variant that manipulate trees of binary application nodes using interpreted evaluation code which switches on the type of the node undergoing evaluation, as determined by bits in its object header. Furthermore, additional bits may be tested to determine whether or not the closure is (un)evaluated.

In a higher-order language it must be possible to pass a function as a 'value' to another function. Similarly, in a lazy language with call-by-need semantics, values are passed to functions as suspensions which are simply data structures representing their unevaluated form. Furthermore, in a polymorphic language, a function value may be indistinguishable from a suspension using static analysis. Typically, compiled functional language implementations represent these two types of value using closures. Conventionally, closures represent unevaluated expressions. An elegant, natural, and fairly intuitive representation for a closure associates a block of static code, its *entry* code, which is shared among all instances, along with the value bindings for the free variables of the encapsulated expression. The STG-machine employs both a *self-updating* and a *push-enter* model of evaluation for each closure. The static code block generated for each closure not only handles its own evaluation, but is

also responsible for performing the update with its WHNF value if necessary. The code that "forces" evaluation of the closure simply *pushes* a continuation (a return address) onto the stack and *enters* the closure. Conceptually, entering a closure unpacks and pushes its arguments onto the argument stack and jumps to its entry code. When evaluation completes the value is returned. If the closure is a suspension, the entry code also pushes an update frame onto the stack (as previously described in Section 4.3) so that when evaluation completes, it is physically overwritten in-place with the resulting WHNF value[1]. If the closure is subsequently entered, then the static code for the closure simply returns its value. This ensures that laziness is preserved when an attempt is made to evaluate a shared closure multiple times. Less intuitively, data values are also represented as closures, the entry code for which simply returns the WHNF value. Because the STG-machine delegates responsibility for the yielding of a closure's value to the closure itself the interpretive overheads that result from testing the tags of an object in order to determine how it should be evaluated or processed by the garbage collector, are for the most part, completely eliminated.

Critical to Haskell's performance, and indeed that of an STG-machine implementation is the representation and handling of arithmetic data types as unboxed values. The language specification, its type system, and the Core and STG-languages all support the encoding of unboxed values, such as integers, 'as first class citizens', i.e. are built in ([PJL91]). This is in contrast to systems which box all such values in the heap and leave unboxing optimisations to be performed at the much later stage of code generation. Not only can boxing and unboxing optimisations be defined as a set of correctness-preserving transformations, but the results of strictness analysis can be applied within the same transformation framework. It is worth noting that unboxed types cannot be passed as arguments to polymorphic functions since the calling convention requires all arguments to be passed as closures laid out with the uniform representation.

The STG-machine, like the G-machine, is a supercombinator graph reduction implementation. Recall that in a pure supercombinator graph reducer, lambda lifting is used to abstract whole sub-expressions containing free variables, as opposed to just the free variables themselves — function abstractions are 'lifted' to the top level by associating the free variables as extra arguments. The result is the elimination of $\lambda$-abstractions in favour of supercombinators and results in a reduction in the number of function applications which are ultimately required. The code generator creates a new global function for each supercombinator. A lambda-lifted program consists only of suspensions (thunks) encapsulating the unevaluated arguments and supercombinators. The evaluation of a suspension unpacks its free variables onto the stack and executes the associated supercombinator function. However, the STG-machine does not require programs to be lambda-lifted. Instead, the free variables of $\lambda$-abstractions are identified, but the abstractions are left in place. Operationally, when entering a closure, its entry code, while accessing its argument values on the stack, does not unpack the values for its

[1]Note that some values are bigger than the closures that built them. In these cases the closure is replaced with a pointer to the value (an indirection).

free variables contained within onto the stack, but accesses them directly. This optimisation reduces the copying of values from the heap to the stack.

## 4.5 A Uniform Representation of Closures

The uniform representation of all objects as heap-allocated closures defines much of the elegance and simplicity of the STG-machine. The stock hardware implementation for each closure is simply a uniquely word-addressable contiguous region of heap allocated memory, the first word of which is a code pointer, the *info pointer*, which points to the closure's static *info table*. The words that follow are the closure's *payload* and consist of a sequence of *pointer words*, followed by a sequence of *non-pointer words*, that are the value bindings for the closure's free variables. Obviously, a pointer word points to a free variable binding, the value of which will be obtained on demand by evaluating the closure to which it points. Similarly, a non-pointer word is an *immediate value* of primitive type, a literal constant, and is the actual value for the associated free variable. For each `let` / `letrec` binding occurring in the program text, a single, statically-allocated info table is created. The dynamic instances of this binding are represented by heap-allocated closures whose info pointers all refer to this single static info table.

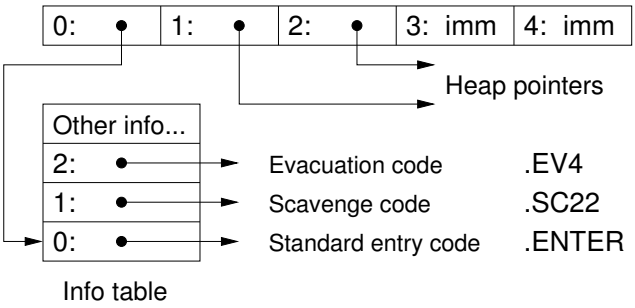



Figure 4.1: Example layout of a Spineless Tagless G-machine heap closure with two pointers and two immediate values.

Figure 4.1 depicts an example of an STG-machine heap allocated closure with a payload of two pointer words and two immediate values. When the evaluation of a closure is forced, the closure is "entered". On entry to a closure, its address is loaded into the dedicated abstract machine register, `Node`, and then its standard entry code, '.ENTER' on Figure 4.1, is executed by jumping to the label retrieved from the info table by indirecting via `Node`. The `Node` register acts as a conventional *environment pointer*, and the info table's various entry code blocks access the closure's payload by indexing it. In addition to containing the code pointer to the standard entry code, a closure's static info table also contains two further code pointers, both of which pertain to garbage collection.

### 4.5.1 Semi-Space Garbage Collection

The original design for the STG-machine utilised a contiguous memory region for the heap, sub-divided into two regions, and employed Cheney's iterative variant of Fenichel and Yochelson's stop-the-world semi-space copying collector, as described in Section 2.4. Recall that when a collection cycle is triggered, the semi-spaces are flipped and all live closures in from-space are evacuated to to-space, where they are then scavenged.

The implementation of the garbage collector adheres to the STG-machine's philosophy in avoiding the overhead of interpretive loops when processing closures. Most usually a garbage collector would inspect the tag of the closure to determine its object type and, using additional layout information, would determine the size of the copy to allocate on evacuation and the heap pointers that must be chased when subsequently scavenging it. However, by generating evacuation and scavenging code blocks that "know" the exact size and structure of the closure and are dedicated to the handling of these two operations for the closure itself, both interpretive loops and the need for additional layout information are eliminated. Not only is this a rather unusual implementation technique, but it is simple, elegant and seemingly rather more efficient. The true efficiency of the design is discussed in Section 4.6.

In addition to the standard entry code pointer, the static info table for the closure contains two additional code pointers, '.EV4' and '.SC22' on Figure 4.1, that point to these two code blocks. The evacuation code, .EV4, for the closure, copies it from from-space to to-space, overwrites the from-space copy with a forwarding-pointer to the to-space copy and returns the address of the to-space copy. Similarly, when scavenging a closure, the scavenging code, .SC22, invokes the evacuation code, .EV4', for each closure referenced, unless it has previously been evacuated, and replaces this original reference with its to-space reference resulting from the execution of .EV4.

The garbage collector then, is implemented, elegantly, as a rather simple function that is responsible for i) book-keeping and the gathering of allocation and collection statistics; ii) performing the semi-space flip; iii) jumping to the evacuation entry code pointer for each of the closures that it has identified as roots of the live memory graph within the heap; and iii) jumping to the scavenging code pointer for each closure that is encountered when linearly scanning to-space.

## 4.6 The Non-stop Spineless Tagless G-machine

While and Field augmented the design of the Spineless Tagless G-machine with an insightful variation to Baker's algorithm for an incremental garbage collector to create the *Non-Stop Spineless Tagless G-machine* [WF93]. The intent was to reduce the overhead imposed on the mutator by the read barrier and to allow an upper bound to be placed on the time taken to perform a store operation. Like the STG-machine, additional code pointers are generated for each static info table to support the garbage collector implementation. In particular,

by manipulating these code pointers during a collection cycle, each object can implement its own individual read barrier by tracing itself and the references to the values of its free variable bindings, if there is a danger of obsolete pointers being propagated to to-space. Figure 4.2 shows how the info table of our example heap closure is augmented with a further two additional code pointers, '.MUTSC' and '.SKSC22'. Inherent to the efficiency of the scheme, and key to its design, is the "straight-line" sequencing of the entry code fragments. In particular, the juxtaposition of .SC22 with .SKSC22 and .MUTSC with .ENTER.

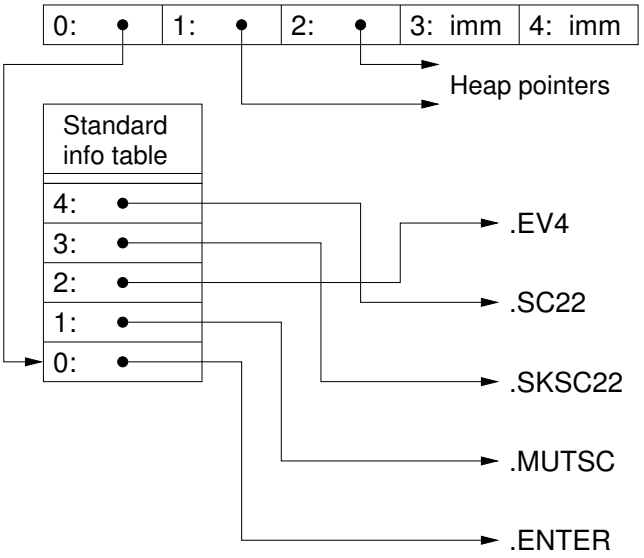



Figure 4.2: Example layout of a Non-stop Spineless Tagless G-machine heap closure with two pointers and two immediate values.

Over the remainder of the section the modified algorithm is described with reference to the description of Baker's algorithm in Section 3.4.1 and also its pseudo-code listed in Section 3.4.1. As before, we assume word-addressable stock hardware and a unit of storage (and thus the size of a [info] pointer) to be one word. The units of the offsets discussed are therefore word multiples. The modified algorithm begins, as standard, with the evacuation of the root set to to-space. Recall that each closure type has its own specific evacuation code, .EV4, so its evacuation is performed by loading the address of the evacuation code (the pointer at offset 2 of the info table) into the `Node` register and jumping to the code. Not only does the evacuation code copy the object into to-space and install the forwarding pointer, but it also *side-effects* the info pointer so that it no longer points at the standard entry code of the closure, .ENTER, but instead points to the *mutator scavenging code*, .MUTSC. At this point, one of two things can happen: either the closure will subsequently be entered by the mutator (in Baker's original algorithm, the read barrier check would be performed), or scavenged by the garbage collector as it carries out its $k$ pieces of work.

- If the mutator enters the closure at the newly side-effected info pointer pointing to .MUTSC then this self-scavenging code is executed, the info pointer is reset and the mutator drops through to the standard entry code at .ENTER. This self-scavenging code calls the evacuation code of the top level children it references and places them on the scavenge queue. Because all objects get placed on the scavenge queue, the garbage collector will eventually attempt to scavenge this object. This is undesirable because the object has already been scavenged. The garbage collector always executes scavenging code at the code pointed to by the offset of the info pointer + 1. The collector will therefore execute the code at *.SKSC22* (offset .ENTER + 1) which is the *skip-scavenge* code that simply checks to see if $k$ scavenges have been performed or garbage collection is complete and either continues scavenging the next scavenge queue object or terminates the collector. In this way scavenging of an object is prevented from occurring more than once.

- If, however, the garbage collector scavenges the closure before the mutator enters it, then the collector executes the self-scavenging code at *.SC22* (again offset 1 from the info pointer pointing to .ENTER). This code performs the same scavenging operations as the .MUTSC code but instead of dropping through to .ENTER, it restores the info pointer to .ENTER and drops through to the .SKSC22 code to check whether collection is complete or whether scavenging should continue. Any mutator entry will now occur in the normal fashion with entry to the closure at .ENTER.

### 4.6.1 Pseudo-code Algorithm

The following pseudo-code functions outline the core of While and Field's collector, and *loosely* demonstrate the side-effecting of the info pointer for the example closure in Figure 4.5.

Function `initialiseBakerSemiSpaceGC()`, Function `flip()`, and Function `allocate()`, all remain unchanged from Section 3.4.1, but are reproduced for completeness and to ease reference.

The absence of Function `mutatorPointerLoad()` should be noted — it is no longer necessary to place a global read barrier on *every* pointer load operation. Instead, the barrier is effected only when is necessary by side-effecting the info pointer. Recall that the mutator always enters the closure and executes the code pointed to by the info pointer. Similarly, the garbage collector always enters and executes the code pointed to at offset 1 from the info pointer. When the mutator modifies the info pointer to point at alternate entry code, so the garbage collector entry point must be altered to enter at offset 1 from this new mutator entry point. This is an abstract operation and is implicit in the adjustment of the info pointer, however it benefits our explanation to introduce the Function `sideEffectInfoPointer()` for this operation. We also introduce the abstract Function `mutatorEnterClosure()`, which demonstrates the possible execution flows that can occur when a closure is entered, via `Node`, for the purpose of evaluation. It should be apparent that the code to which the info pointer

points is the state encoding for each of these flows. In particular, the top-level conditional makes explicit the state encoding in which the read barrier is effected.

**Function** `sideEffectInfoPointer(`*closure, codePointer*`)` Update a closure's mutator and collector info table entry points.

**Input**: *closure* Pointer to closure
**Input**: *codePointer* Static info table code pointer

```
1 closure → infoTable.mutatorEntryPointer = codePointer;
2 closure → infoTable.gcEntryPointer = codePointer + 1;
```

**Function** `mutatorClosureEnter(`*closure*`)` Closure evaluation code demonstrating implicitly encoded read barrier

**Input**: *closure* Pointer to closure undergoing evaluation

```
/* Abstract test - this test is not performed, the mutatorEntryPointer
   of the info pointer has been adjusted to point to either .MUTSC or
   to .ENTER                                                          */
1 if closure → infoTable.mutatorEntryPointer == closure → infoTable.MUTSC
  then
    /* Adjust the closure's mutator and collector entry points       */
2   sideEffectInfoPointer(closure, closure → infoTable.ENTER);
    /* mutatorEntryPointer points to .ENTER, hence gcEntryPointer
       points to .SKSC22                                             */
    /* Breadth-first evacuation of references                        */
3   foreach field in closure do
4     if field is a reference type then
5       dereference(field) = evacuate(field);
    /* Drop through and execute standard evaluation code at .ENTER   */
  /* mutatorEntryPointer points to .ENTER                            */
  /* closure has already been evacuated and scavenged so just execute the
     standard evaluation code                                        */
6 execute(closure → infoTable.ENTER);
```

Function `evacuate()` is further modified to side-effect the closure's info pointer to .MUTSC so that if it is first entered by the mutator, it will be scavenged, before evaluation proceeds. similarly, Function `scavenge()` is modified to either scavenge and restore the info pointer to .ENTER, if the closure is unscavenged, or simply skip it if the mutator has previously scavenged it.

It is worth noting that there is no easy way for the collector to determine whether an object that it is about to scavenge within a $k$ work increment has already been scavenged by the mutator's read barrier. As a result all children of the encountered object must have `evacuate()` invoked on them. The Function `evacuate()` then determines whether the child object must actually be copied. These to-space checks incur an overhead that is somewhat unnecessary and we revisit this issue in the design of our own collector Section 7.4.3.

**Function** `initialiseBakerSemiSpaceGC` Initialisation of incremental semi-space collector

```
1 calculateK ();
2 flip ();
  /* Breadth-first evacuation of the root set                    */
3 foreach root in roots do
4     if dereference(root) is reference type then
5         dereference(root) = evacuate(root);
```

**Function** `flip` Flip to-space and from-space

```
1 if toSpace == memoryStart then
2     scavenger = scavengeLimit = toSpace = semiSpacePartition;
3     fromSpace = memoryStart;
4     free = heapLimit = memoryEnd;
5 else
6     scavenger = scavengeLimit = toSpace = memoryStart;
7     fromSpace = semiSpacePartition;
8     free = heapLimit = semiSpacePartition;
```

**Function** `allocate(`*size*`)` Heap allocation of closure

**Input**: *size* Integer size of closure to allocate
**Output**: *closure* Pointer to allocated closure

```
1  free = free − size;
2  if free < scavengeLimit then
3      if garbageCollectorState == RUNNING then
           /* Failed to free enough memory to satisfy allocation despite
              in-progress collection                              */
4          abort (Out of memory);
5      else
           /* Mutator is briefly paused while the incremental collector is
              initialised.                                        */
6          initialiseBakerSemiSpaceGC ();
7          free = free − size;
8  if garbageCollectorState == RUNNING then
       /* Perform k increments of work at each allocation         */
9      Boolean scavengeQueueEmpty = scavenge ();
10     if scavengeQueueEmpty == TRUE then
11         terminateBakerSemiSpaceGC ();
12 closure = free;
13 return closure;
```

**Function** `evacuate(`*closure*`)` Evacuation of heap object pointed to by *closure*.

```
   Input: closure Pointer to closure
   Output: toSpaceClosure Pointer to to-space residing copy of closure
   /* If closure has already been evacuated and forwarded do nothing      */
 1 if closure ≥ toSpace and closure < heapLimit then
 2 |   toSpaceClosure = closure;
 3 |   return toSpaceClosure;
 4 Integer size = size of heap object pointed to by closure;
 5 Pointer fieldPointer = scavengeLimit;
 6 Pointer toSpaceClosure = scavengeLimit;
 7 scavengeLimit = scavengeLimit + size;
   /* Simple field copy, no evacuation of references, actually implemented
      by bulk memory copying operation                                    */
 8 foreach field in closure do
 9 |   dereference(fieldPointer) = field;
10 |   fieldPointer = fieldPointer + 1;
   /* Set the forwarding address                                          */
11 dereference(closure) = toSpaceClosure;
   /* Adjust the closure's mutator and collector entry points             */
12 sideEffectInfoPointer(toSpaceClosure, toSpaceClosure → infoTable.MUTSC);
   /* Static info pointer points .MUTSC, hence gcEntryPointer points to
      .SC22                                                               */
13 return toSpaceClosure;
```

The collection algorithm, as Baker suggested, also incrementally scavenges the stack using a similar scheme to the above. The code for each closure is augmented with a subroutine call before each of its return points, in the same way as .MUTSC precedes .ENTER. The subroutines scavenge the stack frame of the closure, decrement the next return address on the stack and branch back to the 'normal' return code [WF93]. The (pointer-) arguments on the stack are therefore scavenged as the currently entered closure returns to its caller. The garbage collector also scavenges the stack 'bottom-up' at each call to Function `allocate()` to ensure that the whole stack is scavenged before the collection is complete. Like this stack scavenging technique, the collection scheme also makes provision for the scavenging of a separate update stack, on which update frames are pushed when the update of a thunk by its WHNF value is initiated.

The While and Field collector provides an elegant and efficient scheme for the elimination of (the overheads associated with) Baker's read barrier. However, it has never been implemented. Instead, a preliminary investigation, that simulated the collector, performed using version 0.10 of GHC, suggested that the scheme adds an overhead of around '10% to the execution time of a program' [WF93]. The simulation could not, however, be described as fully incremental because it scavenged both the update and return stacks at the flip. In addition, no consideration was given to the effect that the additional entry code fragments had on the size of the binaries, the effect of the increased fragmentation of the instruction

**Function** `scavenge` Incremental iterative scavenging of to-space. $k$ increments of work are performed.

**Output**: *scavengeQueueEmpty* Boolean indicating that the scavenging of to-space is complete

```
1  Integer kIncrement = 0;
2  while scavenger ¡ scavengeLimit do
       /* Have k increments of work been performed?                    */
3      if kIncrement ≥ k then
4          scavengeQueueEmpty = FALSE;
5          return scavengeQueueEmpty;
       /* Abstract test - this test is not performed, either gcEntryPointer
          has been adjusted to point to .SC22 or to .SKSC22            */
6      if scavenger → infoTable.gcEntryPointer == scavenger → infoTable.SC22
       then
           /* Adjust the closure's mutator and collector entry points  */
7          sideEffectInfoPointer(scavenger, scavenger → infoTable.ENTER);
           /* Static info pointer points .ENTER, hence gcEntryPointer
              points to .SKSC22                                        */
           /* Breadth-first evacuation of references                   */
8          foreach field in scavenger do
9              if field is a reference type then
10                 dereference(field) = evacuate(field);
11             scavenger = scavenger + 1;
12             kIncrement = kIncrement + 1;
13     else
           /* gcEntryPointer points to .SKSC22                         */
           /* The closure pointed to by scavenger has already been scavenged
              so just skip it                                          */
           /* The instructions of .SKSC22 encode the number of bytes to skip
              and thus implicitly, the size of the closure             */
14         scavenger = scavenger+ sizeof(dereference(scavenger));
15 scavengeQueueEmpty = TRUE; return scavengeQueueEmpty;
```

cache, the actual upper bound of a store operation, or the likely MMU and maximum pause time characteristics.

In practice, the implementation of the Non-stop STG-machine cannot be efficiently realised, for the design of modern stock hardware precludes it. Specifically, in GHC version 4.0, all code pointers were eliminated from the static info tables (the info pointer points at the entry code and directly follows the info table as opposed to pointing at it) for two key reasons:

- Performance is sacrificed as a result of instruction cache fragmentation when compared to the benefits gained through simplified collector implementation.
- The generated fragments of collection code were not only hard to optimise but also prevented the implementation of more global short-circuiting optimisations that result when the fragments can be combined and sequenced (in an interpretive loop).

The Non-stop STG-machine study concludes with a summary of the additional overheads that the scheme is *expected* to impose. A single overhead is incurred for 'standard' mutator execution in that, at each allocation, the mutator must test whether or not a collection cycle is in progress in order to determine whether the collector need be invoked. The scheme places three time overheads on garbage collection:

- A closure's info pointer must be adjusted during its evacuation and scavenging.
- A function call is necessary when scavenging a stack frame and the return address of each callee closure that is active during garbage collection must be decremented.
- A count variable must be maintained for the number of closures, stack pointers and update frames that have been scavenged so that when this count hits the scavenge limit, $k$, the mutator can be resumed.

Over the remainder of the dissertation, we explore the design space for incremental collectors that build on the key insight underpinning While and Field's collector — the exploitation of dynamic dispatch for a cheaper implementation of Baker's read barrier. Furthermore, we evaluate comprehensively the time and space overheads that each scheme imposes.

We start by proposing a modified While and Field collector, for incorporation into the current version of GHC, that is still based on the technique of self-scavenging closures, but provides a *theoretically* less efficient implementation which should, however, still reduce the costs that would be imposed by the read barrier.

# Chapter 5

# Non-stop Haskell

The major problem with the implementation of Baker's scheme, at least in software, is the very high cost associated with the read barrier, which is a check required at each pointer load to ensure that the referent object has been copied if it needs to be (Section 3.4.1).

In this chapter we present a portable software technique for incremental garbage collection in language implementations that already make use of dynamic dispatch.

We suggest a simple technique that exploits an underlying dynamic dispatch mechanism to implement the read barrier (Section 5.1). The idea is to "hijack" the method table of an object that must be scavenged before being used, thereby implementing a simple per-object read barrier. This means that there are *no* read barrier overheads associated with object references when either i) the garbage collector is off, or ii) the collector is on and the object has already been scavenged. In Section 5.1.5 we present variants of this generic self-scavenging object that are employed during the evacuation and the scavenging of large objects in order to ensure that the incremental scavenging bounds are not violated.

We extend the idea to treat stack activation records in a similar way, so that thread stacks can also be collected incrementally (Section 5.1.3).

In Section 5.1.2 we describe the *non-destructive* installation of the incremental copying collector's forwarding pointer, that obviates the need for *either* additional dynamic space overhead or the use of a read barrier for *all* objects.

To evaluate our technique we have developed a prototype implementation for the Glasgow Haskell Compiler (Section 5.2). Experiments with this implementation show that the average overhead on program execution time is less than 4% for the benchmarks tested and that sub-millisecond average pause times are achieved in all cases (Section 5.4).

Initially, we build on the semi-space bulk stop-and-copy collection scheme of GHC runtime system rather than a generational scheme. The scheme we describe can be implemented generationally (see Chapter 7), but here we focus on the more straightforward bulk copying scheme for evaluation purposes.

### 5.0.2 The Spineless Not-So-Tagless G-machine

As discussed in Section 4.6, the static info table layout of the STG-machine was re-designed by Microsoft's GHC team to eliminate the code pointers from the info table of a closure. The info pointer now points directly at the first instruction of the entry code, thus eliminating an extra level of indirection. This is immediately preceded, in static memory, by the info table itself. This new layout is depicted in Figure 5.1. Because the collector must take specific actions, based on the type of the closure, for the purposes of garbage collection each closure is also tagged with its type, via an additional field in its info table. With the closure evacuation and scavenging code no longer associated with the info table, a more standard collector implementation has been adopted, in which the scavenger operates via an interpreted loop. As a result, it is now necessary for each closure to be associated with its payload layout information, via a new field in its info table. A convention of specifying the number of pointer words, followed by the number of non-pointer words is adopted for the representation of the layout information so that generic collector code can interpret and manipulate all closures. In Figure 5.1 the second field of the info table contains the layout information of two pointer words followed by two immediate values.

The evaluation mechanism is unchanged and operates without requiring access to a closure's layout or type information and thus continues to execute without any interpreted overheads.

In the light of these modifications, the 'STG' acronym of the the STG-machine now stands more appropriately for the *Shared-Term Graph-machine*.

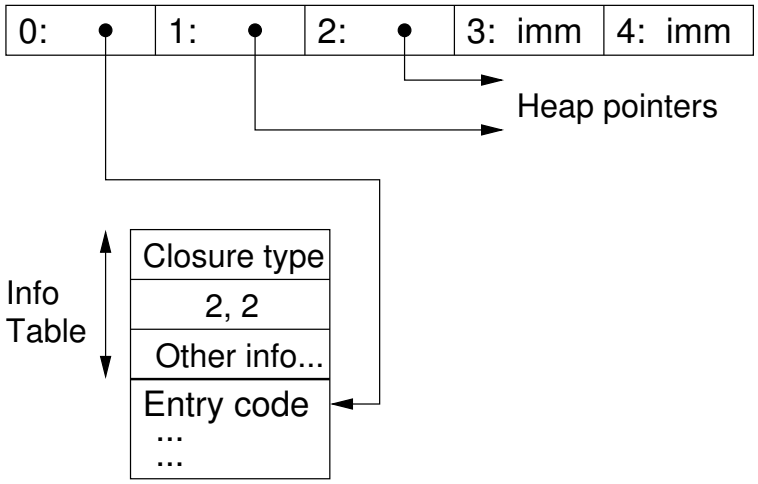



Figure 5.1: Example layout of a Shared-Term Graph-machine heap closure with two pointers and two immediate values.

## 5.1 A Prototype Incremental Semi-Space Collector

In this section we introduce the design for our prototype incremental semi-space collector. Like While and Field's collector, the main idea is the cheap implementation of the read barrier invariant where the mutator must see only to-space pointers.

Our system is only useful where the bulk of all object accesses use dynamic dispatch – that is, the fields of an object are private, and used only by the object's methods. This is so in GHC's implementation of lazy evaluation, and in some pure object-oriented systems. Certain static optimisations, in which the target of a dispatch can be computed statically, are no longer valid; we cannot quantify this loss in general, but in GHC it is small (Section 5.2).

The STG machine spends most of its time executing code generated from Haskell function definitions. This code executes in an immediate environment, or *activation*, consisting of (a) registers, (b) locations in the function's *stack frame*, and (c) the fields of the *currently-active closure* (CAC). This is quite conventional. The CAC is necessary to support first-class functions and thunks: it contains the environment captured when the function or thunk was allocated. (In an object-oriented environment the CAC is known as the *this* pointer.) As observed by While and Field, the STG machine captures a separate flat (i.e. non-nested) environment in each such closure, a design decision that turns out to make incremental garbage collection much easier.

Since the mutator only accesses the current activation, we can maintain the read barrier invariant thus:

> *Ensure that all the fields of the current activation have been scavenged; that is, they point into to-space.*

In operational terms, there are two major elements to implementing this requirement.

- Before entering the code for a closure, we must first scavenge the closure (Section 5.1.1).
- When a function returns to its caller, we must first scavenge the caller's stack frame (Section 5.1.3).

### 5.1.1 Entering a Closure

The first thing that happens to a closure when it is entered during a garbage collection cycle is that it is evacuated and placed on the queue of closures in to-space waiting to be scavenged. At some subsequent time, the collector scavenges the closure. If the closure is entered between being evacuated and being scavenged, there is a danger of from-space references being propagated into to-space. We avoid this by arranging for the closure to be scavenged if it is entered while it is on the collector queue.

When a closure is evacuated, its info pointer is adjusted to point at the entry code of the info table for our generic "self-scavenging" closure. The entry code simply invokes the

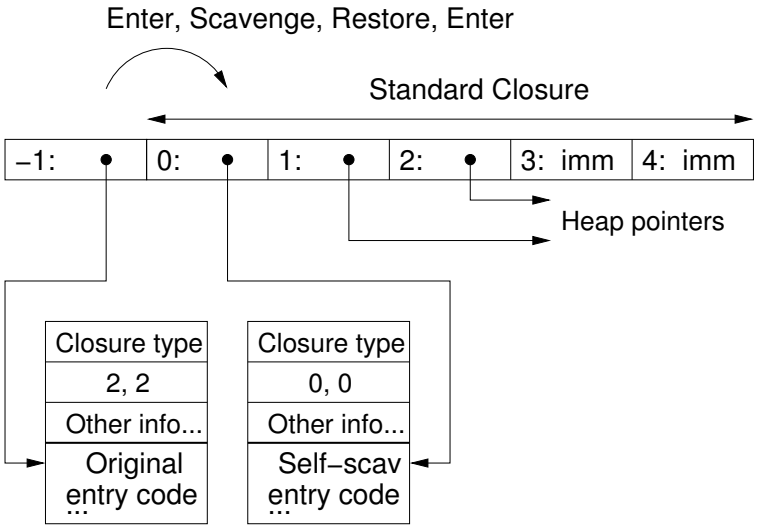



Figure 5.2: The layout of a Non-stop Haskell closure, with two pointers and two immediate values, post evacuation and pre scavenging.

interpreted single closure scavenge routine of the garbage collector before (re-)entering it. In this prototype implementation the original info pointer is remembered in an extra header word — Word −1 — in the evacuated copy of the closure itself, as illustrated in Figure 5.2.

Note that there are several ways of avoiding this info pointer copy. We could instead have used part of the original copy of the closure in from-space to store the info pointer, but this would require all objects to have a minimum size of three words, and in GHC, the minimum is two (the size of an indirection closure). An alternative is to make each info table contain both entry code and a copy of the self-scavenging code, both with associated layout information. The info-pointer of a closure can then be switched between the entry code and the self-scavenging code transparently from the point of view of the mutator. This is an attractive option but requires a substantial increase in the size of each static info table. From the point of view of a prototype, the info pointer copy also turns out to be useful should a closure be updated before being scavenged: the scavenger can use the original info pointer to determine the size of the original closure and hence how far to skip after the closure has been scavenged (see Section 5.1.4). As a result, only a single generic self-scavenging info table is needed, and as shown on Figure 5.2. It has uninitialised layout information that is never accessed. Various improvements to the current prototype, including the removal of the extra word, are discussed in Section 5.6.

After evacuation, there are two possibilities: the closure will either be entered by the mutator or scavenged by the garbage collector.

- If the closure is entered by the mutator first, the mutator executes Self-scav code. This code uses the layout info accessed via Word −1 to scavenge the closure, restores the

original info pointer to Word 0, and then enters the closure as it originally expected. The garbage collector will eventually reach the closure, but as Word 0 no longer points to Self-scav code, it "knows" that the closure has already been scavenged, so it does nothing.

- If the closure is reached by the garbage collector first, the collector scavenges the closure, again using the layout information accessed via Word −1. It also copies the original info pointer back to Word 0, so that if the closure is eventually entered, entry occurs in the normal way.

In effect, the two segments of scavenging code cooperate to guarantee that the closure is scavenged exactly once *before* it is entered.

Note that Word −1 plays no role outside the garbage collection: new objects are allocated in a different region of memory to from-space and do not require the extra word. The space overhead therefore depends on the proportion of closures that are alive at a garbage collection, known as the *residency* of the system. However, this overhead does not require the heap to be any larger. For the same heap size, it means that garbage collection is performed more often; this is automatically factored into the timings.

### 5.1.2 Forwarding a Closure

In incremental algorithms that employ a read barrier to maintain the strong tricolour invariant, an object can be destructively updated with its to-space forwarding address, once it has been relocated. Since the mutator only ever accesses and manipulates to-space references, the structure of the from-space object need not be preserved. A sub-operation of the read barrier determines whether an object has been forwarded based on a bit sequence in the object header and if so, follows the forwarding pointer.

Algorithms, that enforce the weak tricolour invariant by allowing access to the from-space object, or algorithms that work to eliminate the read barrier in certain scenarios, typically employ Brooks' unconditional read barrier which incurs a dynamic space overhead of the size of a pointer (Section 3.4.2).

In a dynamic dispatch environment, it is trivial to support the destructive forwarding of an object, thus avoiding further dynamic space overheads, without the need for a read barrier. In GHC, we create a "self-forwarding" closure, which is simply an indirection closure referring to the to-space copy of the object. Such a closure is depicted in Figure 5.3. When a self-forwarding closure is entered by the mutator's evaluator, the entry code of the closure ('Evacuated' on Figure 5.3) simply follows the reference and enters the to-space referent. We describe how this is achieved in other dynamic dispatch languages such as Java in Section 9.6.

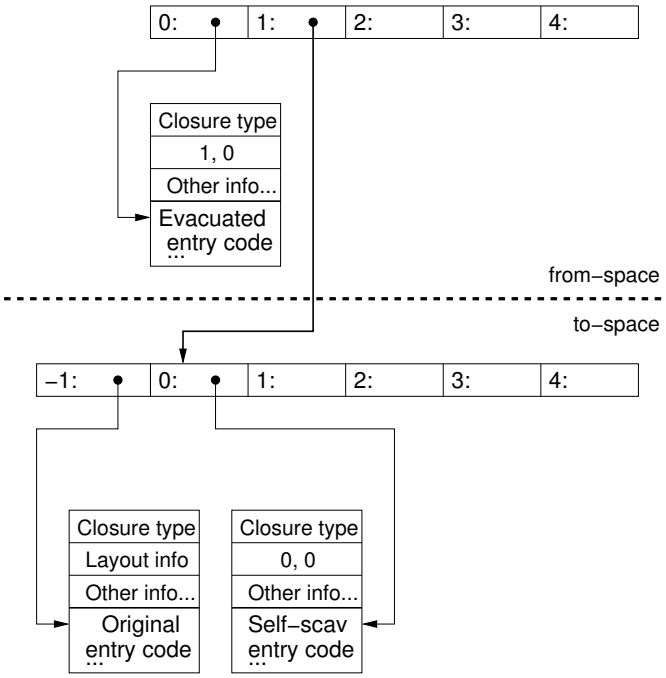



Figure 5.3: Arrangement of a self-forwarding closure and its self-scavenging referent immediately after evacuation.

### 5.1.3 Returning to a Closure

The second thing we must ensure is that the active stack frame is completely scavenged. One way to achieve this is to scavenge the whole stack at a GC flip, but this leads to an unbounded pause, especially since GHC supports concurrency, so there may be many stacks.

An alternative is to view the stack as a sequence of closures. The stack grows toward lower addresses, so each return address sits immediately below (in address terms) the stack frame to which it corresponds. In fact, GHC makes each return address look exactly like an info pointer, so stack frames really do have the same layout as heap objects.

Returning to the previous stack frame is very like entering a closure, and we could use the same technique to scavenge the stack frame incrementally as we do for closures. However, stack frames are contiguous, so the "Word –1" trick does not work so well. Instead, we adopt a compromise. Interspersed among the regular stack frames are *update frames*, whose role is to update a thunk with the value being returned. These update frames have a fixed return address. The compromise is this: we scavenge all the stack frames between one update frame and the one below, replacing the return address in the latter with a self-scavenging return address. This self-scavenging code scavenges the next group of frames, before jumping to the normal, fixed, update code. Since update frames can, in principle, be far apart, pause times can be long; but this case is rare. The basic idea is illustrated in Figure 5.4.

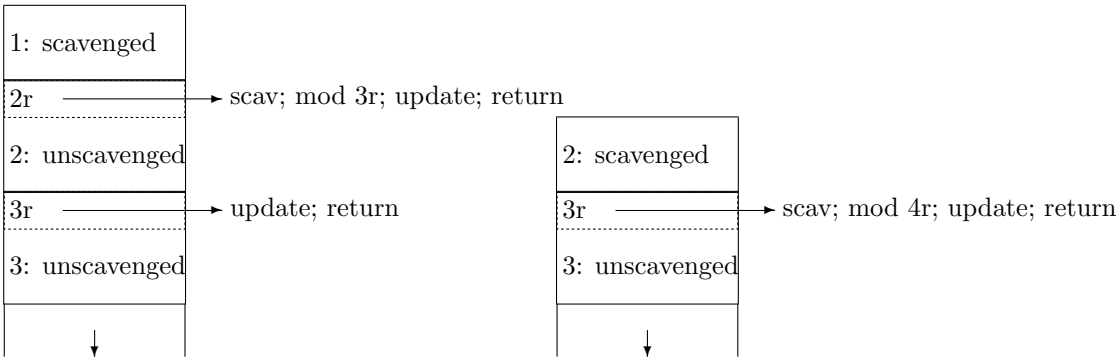



Figure 5.4: Before and after returning to the caller of group 2. Group 2 has been scavenged and the code-pointer in group 3 has been modified.

At the start of a collection cycle, we scavenge each stack down to the topmost update frame, whose return address we replace with a self-scavenging code pointer. After that, the stack is scavenged incrementally, either by the background collector, or when the stack retreats to an unscavenged update frame. The mutator and the collector maintain a pointer to the highest unscavenged stack frame so that each "knows" how far the scavenging has progressed. Thus the mutator and the collector co-operate in the scavenging of the stack, without the possibility of the collector encountering a frame that has already been scavenged by the mutator.

### 5.1.4 Updates

When garbage collection is performed incrementally, it is possible for a thunk to be updated between being evacuated and being reached by the linear scavenging process. This could cause a problem if the updated closure is smaller than the original: when the linear scan eventually encounters the closure and attempts to skip over it to the next closure it will not skip far enough and will interpret the dead space at the end of the previous closure as the start of the next. However, the collector can determine that the closure has already been scavenged because its info pointer no longer points at Self-scav code. The problem is therefore avoided by using the original layout information which has been saved at Word −1.

### 5.1.5 Evacuating and Scavenging a Large Closure

It should be apparent that in our scheme, as discussed thus far, the operations for both the evacuation and scavenging of individual closures, are uninterruptible and thus atomic — the closure must be evacuated in its entirety and similarly scavenged in its entirety. For large closures, this can lead to incremental bounds violations. Baker recognised this problem and addressed it in his algorithm through the lazy evacuation of large objects, as discussed in

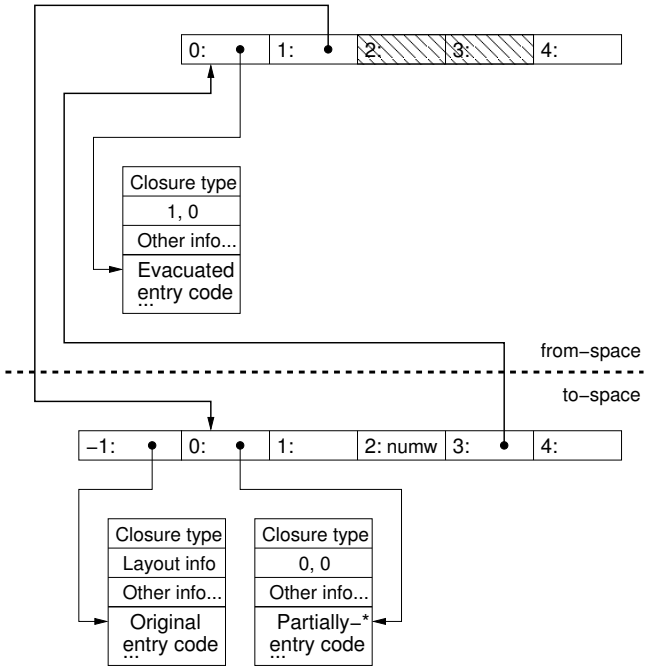



Figure 5.5: Exploiting dynamic dispatch for the bounded evacuation / scavenging of a large closure.

Section 3.4.1. In this section we present the mechanism for both lazy closure evacuation and scavenging that, in keeping with the fundamental design of our collector, exploits dynamic dispatch. The scheme requires a closure to have a payload of at least three words, but we remark that the large classification is typically used for closures with many more. For each of the two operations, we introduce a new closure type, both of which have the same to-space layout, but differ in their info table, type and entry code — Figure 5.5 shows this, and Partially-* is replaced by Partially-evacuated or Partially-scavenged for the appropriate operation.

When evacuation of a large closure is initiated, the first two words are evacuated as normal, with the info pointer word copied to Word –1, and the first payload word to its corresponding location in the to-space copy. The forwarding pointer is immediately installed into the from-space copy of the closure, and the info pointer of the to-space copy is adjusted to point to the Partially-evacuated entry code. The second payload word location (field '2' containing numw on the diagram) of the to-space copy is initialised with the integer value '3'. This field is used to record the number of payload words that have been processed as lazy evacuation or scavenging proceeds. Finally, a back-pointer to the from-space copy is installed into the third payload word location of the to-space closure copy.

At this point, if the object was evacuated by the mutator, the mutator scavenging of the CAC must continue. If however, it was evacuated by the collector in a scavenging increment,

then evacuation continues until the allocate/mark ratio limit is hit. If this is before the closure has been fully evacuated then numw is updated to reflect the number of words that have been evacuated, and the scavenge pointer remains pointing at the closure. At this point of evacuation, there are two possibilities: the closure will either be entered by the mutator or will continue to be evacuated by the garbage collector.

- If the closure is entered by the mutator first, the mutator executes Partially-evacuated code. This code uses the closure layout info accessed via Word −1, the back-pointer, and the numw field to *fully* evacuate *and fully* scavenge the closure. Both operations must be forced to completion in order to maintain the to-space invariant because the mutator is demanding the evaluation of the closure. The original entry code is then executed.

- If the closure is reached by the garbage collector first, the collector continues to evacuate the closure in allocate/mark ratio increments, resuming the mutator each time. When the closure has been fully evacuated the info pointer of the to-space copy is adjusted to point to the Partially-scavenged entry code and numw is again initialised to '3'. Note that the payload words of fields '2' and '3' of the from-space closure (shaded on Figure 5.5) remain unevacuated, and are scavenged and updated with the referents' forwarding addresses in the from-space copy. Scavenging now proceeds lazily in a similar way to that of lazy evacuation that has just been described. Again, if the mutator enters the closure and executes the Partially-scavenged entry code, the remainder of the closure must be scavenged and the original entry code entered. Otherwise, lazy scavenging is performed incrementally by the collector to completion, at which point payload words '2' and '3' are copied from the from-space closure into their correct places in the to-space copy, and the original info pointer from Word −1 is finally restored.

In this way, lazy scavenging and evacuation of large objects by the *collector* can be performed so as to maintain the *individual* incremental bound of the allocate/mark ratio limit. It is only if evacuation or scavenging of these larger objects is performed by the *mutator* that these bounds risk being violated. We discuss the ramifications of this, along with the aggregation of collector scavenging increments and quick successive barrier trips by the mutator, with respect to mutator progress and minimum mutator utilisation, in Section 5.5.2.

## 5.2 Implementation

We initially constructed a prototype implementation of the scheme on top of GHC v4.01 in order to determine its viability and guide subsequent design decisions for the production generational implementation of Chapter 7. Because the runtime system is fairly complex and the modifications required to support the incremental generational collector subsume those that support the incremental semi-space collector, we defer detailed description of the implementation to Chapter 7. Some points are worth noting, however:

- GHC v4.01 uses a two-level storage allocation policy (fully described in Appendix Section A.4.2). The central storage manager maintains a free list of 4KB-blocks of memory. When the mutator encounters an allocation request it obtains a block from the storage manager and chains it onto the heap. The mutator then allocates individual closures within that block until it is exhausted, at which time it obtains a new block. This continues up to some specified maximum number of blocks. The advantages of using a block allocated scheme are that areas of memory used by the system need not be contiguous; blocks that are freed during a collection become available for immediate re-use. Also, new objects are allocated to a separate area (the *nursery*) from those being evacuated, so the scavenger never encounters them.

- The block-allocation scheme makes it easy for the storage manager to vary the size of to-space at run-time. The manager tries to keep the residency (how much heap is reachable, sampled at the previous garbage collection) at about 33%. It gets more memory from the operating system to achieve this, up to a settable maximum. However, it has a default minimum heap size of 256KB, so for small live data sets the actual residency can be very small.

- Objects which are classified as "large" have their own "pinned" storage block(s) allocated solely to themselves. Evacuation of these blocks does not require them to be copied, and hence takes constant time. As a result where a large closure is allocated in such a way, the mechanism employing the Partially-evacuated entry code of Section 5.1.5 need not be applied. However, the object is still lazily scavenged, using a variation of the mechanism that employs the Partially-scavenged entry code, where numw is stored in an additional reserved word at the head of the closure, offset Word −2 — note that there is no danger of the scavenger trying to interpret this field as an info pointer because objects that are pinned in this way are handled differently. Those larger closures whose payload exceeds the allocate/mark ratio but are not allocated in their own pinned storage blocks, employ the mechanism described in Section 5.1.5.

- Some of GHC's object types are always evaluated at construction, and thus are never evaluated lazily; they are never updated or entered. These are called *unpointed objects*, and include primitive arrays and mutable variables. Although they are laid out according to the uniform representation of closures, they lack any useful entry code so we cannot use the self-scavenging scheme to collect them incrementally. They are therefore scavenged immediately after evacuation (*eager* scavenging). This can introduce longer pause times. However, the collector still scavenges a large object, such as a big array, which gets a storage block to itself, using the lazy scavenging technique described above. A Baker-style read barrier must then be implemented in all the access functions associated with the unpointed object to cooperate with the lazy scavenging of the object by the collector.

- The fact that *all* dispatching must be dynamic in order for the scheme to work means that some compiler optimisations have to be turned off. For example, when entering a dynamic function closure which has a known info table we can jump straight to the entry code rather than indirect via the info pointer in the closure. To exploit our incremental garbage collection scheme these short-cuts must be forbidden for otherwise a scavenging code pointer planted in the object during garbage collection might be missed. This constitutes an additional cost, although for GHC 4.01 the optimisation actually yields very small improvements (less than 2% on the benchmarks considered below).

The nature of the block allocator in GHC enables the incremental scheme to be supported at two alternative levels. In the traditional Baker scheme, a scavenging phase occurs at each new heap allocation. We can do the same here, although we remark that a single allocation in GHC typically delivers a 'chunk' of memory that can satisfy multiple lower-level allocations. We refer to this as EA (scavenge at Every Allocation). The alternative, which turns out to be beneficial for a number of reasons, is to scavenge at each 4KB block allocation. This increases the grain of each scavenge phase so that the impact of keeping the scavenger going during mutation is reduced. Also, we can tune the block size if necessary in order to reduce, or increase, the pause time. This alternative scheme is called EB (scavenge at Every Block allocation).

## 5.3 The Baker Collector

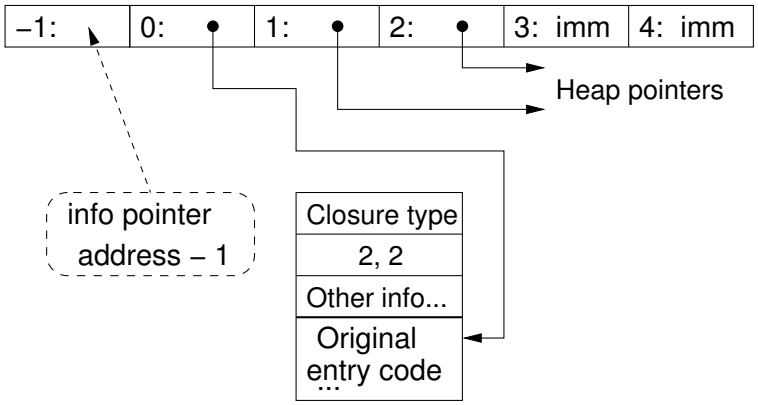



Figure 5.6: Closure layout used by the Baker collector.

A Baker collector can be implemented into GHC using most of the framework that the self-scavenging collector provides. This does not, however, provide the most optimal Baker implementation in terms of space overheads. Both our prototype self-scavenging collector

and the Baker collector implementations incur unnecessary space overheads that we work to remove in Chapter 7. However, because our current focus is solely on the overhead imposed on the mutator, dynamic space overheads that would affect the execution time due to the amount of garbage collection performed, can be largely ignored. Indeed, our initial experiments are devised with exactly this in mind.

The major differences between the two implementations are as follows:

- **Evacuating a closure** — When a closure is evacuated to to-space, the extra word that is prepended to the closure is retained. The original info pointer of the object is copied into this word as before. However, the address stored at this location has '1' byte subtracted from it (see Figure 5.6). The word that is the first 'valid' word of the closure, and hence contains the info pointer used by the evaluators, unlike the self-scavenging implementation, is not updated with a self-scavenging info pointer; it is left containing its original info pointer. The reason the info pointer copy word is retained is to resolve the update problem in which a closure is updated with one of a smaller size. As collector-scavenging of to-space progresses, this mechanism allows the collector to locate the start of the next valid closure in to-space (recall Section 5.1.4). Furthermore, it allows the read barrier to be optimised, see 'Scavenging a Closure' below.

- **The read barrier** — To implement a read barrier into GHC that is equivalent to trapping the 'pointer loads' as discussed in Section 3.4.1, the evaluator code that jumps to the entry code of a closure via the Node register is modified. It prevents the mutator from obtaining from-space references by trapping entry to live objects that still reside in from-space and evacuating them to to-space, *and* by trapping entry to unscavenged objects in to-space and scavenging them, in both cases before executing the standard entry code of the closure. The barrier is therefore effected on entry to all closures, regardless of i) the state of the collector; ii) the location of the closure; iii) whether or not the closure has previously been scavenged.

- **Scavenging a Closure** — Info pointer addresses are word-aligned. The subtraction of one byte from the address stored in the info pointer copy word results in an unaligned address stored at this location. When the the closure is scavenged, whether it be by mutator or collector, '1' is added to this address, restoring it to its original value. This optimisation enables both the read barrier and the scavenger to identify and thus skip those closures that have already been scavenged, essentially nullifying point iii) above.

## 5.4 Experiments

In order to evaluate the new scheme, and to compare it with stop-and-copy and Baker, we have tested it with six applications taken from the nofib test suite [Par93] and an additional

contrived benchmark called `contrived` designed to explore some of the extreme cases of heap usage.

We remark that none of the nofib benchmarks requires real-time response. The purpose of the experiments was to get a feel for both the average and extreme performance overheads incurred by the new scheme in relation to stop-and-copy for a representative cross section of applications. The benchmarks we selected were chosen arbitrarily with no preconception about the applications' expected behaviour using either scheme. The only criterion was that the application should be able to run to completion without invoking a garbage collection for a suitably chosen set of parameters and initial heap allocation. This was to enable the actual garbage collection and mean pause times to be determined accurately when the garbage collector is turned on.

We ran each application in two configurations: one with a heap large enough for the application to complete without causing a garbage collection (typically around 110MB), and one with a smaller initial heap (256KB). It is worth noting that the execution time of the application must be controlled such that the total allocation does not exceed the larger heap value (which would result in a garbage collection). The value is constrained to just below the amount of memory in the benchmark machine. With profiling turned off the difference between the two execution times tells us the total cost of garbage collection. For the new scheme the application was first executed with profiling turned on[1] in order to determine the total number of mutator pauses and the heap allocation rate in bytes/s. The timings taken from runs without profiling were used to determine the mean pause time. All experiments were run on a Pentium-II/350 with 128MB RAM running RedHat Linux 6.0.

We ran each application with four garbage collection algorithms: stop-and-copy (with all optimisations turned on), a traditional implementation of Baker with a software read-barrier, and the two versions of the new scheme, EA and EB. The Baker implementation was set to scavenge at each block allocation, as per EB. In each case the allocate/mark ratio was 2 for the runs of EA and 2048 for the runs of EB. Note that the smallest initial heap size is, by default, 256KB.

### 5.4.1 Overheads

In keeping with the basic Baker algorithm, the current implementation of the allocator tests whether there is a garbage collection in progress in order to decide whether it needs to do some scavenging. This involves testing a run-time flag called `gc_in_progress` each time an area of free space is requested by the mutator. The `gc_in_progress` test introduces an overhead on each allocation, the cost of which depends on the granularity of the garbage collector, i.e. how often it performs the test.

There are three time overheads on garbage collection itself: (i) the cost of adjusting the info pointer of each closure during evacuation and scavenging (ii) the cost of adjusting the code

[1] We found that profiling code significantly affected the execution times in the incremental scheme.

pointer in each stack frame that is active during the garbage collection, and (iii) the cost of counting closures as they are scavenged (this overhead can be eliminated if the allocate/mark ratio is hard-coded, e.g. to 1).

## 5.5 Results

The performance results reported are based upon the average of a number of identical executions of each application. This was typically between 5 and 10 runs, sufficient at least to keep the 90% confidence interval for the mean execution time within 1% of the mean.

### 5.5.1 Contrived Benchmarks

To begin with it is instructive to discuss the results for a contrived benchmark called `contrived` as we can control three of the key parameters which affect application behaviour. The code for `contrived` is shown in Figure 5.7. It contains a top-level function which first computes a list `x` of length `copy` and then recurses over a second list of length `run` whilst holding on to the first (by virtue of the final result being dependent on `x`). In each step of the recursion a tail-recursive function that performs no heap allocation is called `work` times. `copy` thus controls the amount of live heap data which has to be copied at each garbage collection; `work` controls the amount of work done between allocations and hence controls the allocation rate (in bytes/s) and `run` controls the length of the run and hence the number of garbage collections that need to be performed. The results reported here were undertaken with the EB scheme, i.e. the self-scavenging scheme with routine scavenging at each block allocation, although similar results are observed with EA.

By setting `work=copy=1` we obtain a program with a very high allocation rate (around 110MB/s) and a very small residency (around 0.1% of the available heap). This means that most of the overhead seen in the new scheme is incurred by the `gc_in_progress` test. We find that the overhead is around 21% in this case. To investigate this further, the benchmark was re-run with a 110MB heap which for `loop` $= 3 \times 10^6$ is sufficient to avoid a garbage collection. We were then able to safely turn off the `gc_in_progress` test in the new scheme. We found that this reduced the overhead to just over 2%. Indeed, a similar experiment in which the `gc_in_progress` test was redundantly added to the stop/copy scheme with the garbage collector turned on again brought the two execution times to within 2% of one another.

We next increased `copy` in order to increase the heap residency and so measure the effect of managing the info-pointer copies in to-space. The effect is quite subtle: as `copy` increases the allocation rate drops and the residency increases. Proportionally less time is now spent allocating fresh heap space, so the overall effect of the `gc_in_progress` test is reduced. If we set `copy=50,000` the residency remains at around the target value of 33%; the allocation rate drops to around 90MB/s and as a result the overhead reduces to 5.6%. Note that if `copy`

```
contrived
  = if length x == 0
    then [0]
    else if f [ 1..run ] == 0
         then x
         else []
  where
    run  = ...
    work = ...
    copy = ...
    x = take copy [ 1.. ]
    f []         = 1
    f ( x : xs ) = if g x work
                   then 1
                   else f xs
    g a b = if b == 0
            then False
            else g a ( b-1 )
```

Figure 5.7: The contrived benchmark code

is substantially increased the maximum heap size is approached resulting in the residency creeping towards 100%. At this point the extra to-space word has the effect of reducing the time between garbage collections, compared with stop-and-copy, and we find that the overhead swings the other way: as `copy` approaches the maximum value that can be handled by the memory manager the overheads increase by virtue of an increase in the number of garbage collections.

Finally, as `work` is increased, more work is performed between allocations and so the allocation rate is reduced. With `copy=50,000` as above, increasing `work` further reduces the overhead, as is expected. For `work=100` the allocation rate drops to around 16MB/s and the overhead falls to around 1.9%. Note that the residency is unaffected by the value of `work`.

These experiments provide some useful insights into the behaviour of the various collectors in extreme cases and also to the observed performance of some of the `nofib` benchmarks which follow. However, the interactions between the application and the block allocation scheme results in many subtle performance anomalies in both directions: the observations cannot be reliably used as a model of garbage collection performance but they can suggest the correct qualitative trend in many cases.

### 5.5.2 Experiments with nofib

The results obtained from our experiments with six of the `nofib` benchmarks are summarised in Tables 5.1 – 5.6. Recall that the parameter(s) in each case were chosen to allow the application to complete without garbage collection using a 110MB heap.

The runs of EB for these problem sizes show less than a 6% average overhead relative to

| | Stop-and-copy | Baker | Self-scav (EA) | Self-scav (EB) |
|---|---|---|---|---|
| Running time 110MB (s) | 2.03 | 3.20 | 2.31 | 2.14 |
| Running time 256KB (s) | 4.26 | 5.33 (+25%) | 4.59 (+8%) | 4.19 (-2%) |
| GC time (s) | 2.23 | 2.13 | 2.28 | 2.05 |
| Number of pauses | 19 | — | 1121409 | 5900 |
| Average pause time (ms) | 117 | — | 0.00203 | 0.347 |

Table 5.1: Results for pic 1500, a particle simulator.

| | Stop-and-copy | Baker | Self-scav (EA) | Self-scav (EB) |
|---|---|---|---|---|
| Running time 110MB (s) | 4.37 | 7.41 | 4.59 | 4.36 |
| Running time 256KB (s) | 6.97 | 12.75 (+83%) | 7.86 (+13%) | 7.06 (+1%) |
| GC time (s) | 2.60 | 5.34 | 3.27 | 2.70 |
| Number of pauses | 213 | — | 3182084 | 118556 |
| Average pause time (ms) | 12.2 | — | 0.00103 | 0.0228 |

Table 5.2: Results for anna ap_ListofList, a frontier-based strictness analyser.

| | Stop-and-copy | Baker | Self-scav (EA) | Self-scav (EB) |
|---|---|---|---|---|
| Running time 110MB (s) | 1.28 | 2.15 | 1.45 | 1.27 |
| Running time 256KB (s) | 3.89 | 5.51 (+42%) | 4.61 (+19%) | 4.16 (+7%) |
| GC time (s) | 2.61 | 3.36 | 3.16 | 2.89 |
| Number of pauses | 61 | — | 1896616 | 23425 |
| Average pause time (ms) | 42.8 | — | 0.00167 | 0.123 |

Table 5.3: Results for circ 8 125, a simple circuit simulator.

| | Stop-and-copy | Baker | Self-scav (EA) | Self-scav (EB) |
|---|---|---|---|---|
| Running time 110MB (s) | 1.65 | 2.59 | 1.66 | 1.64 |
| Running time 256KB (s) | 2.31 | 3.74 (+62%) | 3.08 (+33%) | 2.81 (+22%) |
| GC time (s) | 0.66 | 1.15 | 1.42 | 1.17 |
| Number of pauses | 353 | — | 1214107 | 420491 |
| Average pause time (ms) | 1.87 | — | 0.00117 | 0.00278 |

Table 5.4: Results for primes 1500. primes k returns the $k^{th}$ prime number using the Sieve of Eratosthenes.

| | Stop-and-copy | Baker | Self-scav (EA) | Self-scav (EB) |
|---|---|---|---|---|
| Running time 110MB (s) | 1.94 | 3.97 | 2.13 | 2.02 |
| Running time 256KB (s) | 2.09 | 4.17 (+100%) | 2.26 (+8%) | 2.17 (+4%) |
| GC time (s) | 0.15 | 0.20 | 0.13 | 0.15 |
| Number of pauses | 403 | — | 21106 | 403 |
| Average pause time (ms) | 0.372 | — | 0.00616 | 0.372 |

Table 5.5: Results for nqueens 11.

| | Stop-and-copy | Baker | Self-scav (EA) | Self-scav (EB) |
|---|---|---|---|---|
| Running time 110MB (s) | 2.94 | 4.98 | 3.10 | 3.03 |
| Running time 256KB (s) | 5.62 | 8.24 (+47%) | 6.12 (+9%) | 5.74 (+2%) |
| GC time (s) | 2.68 | 3.20 | 3.02 | 2.71 |
| Number of pauses | 139 | — | 1471723 | 37554 |
| Average pause time (ms) | 19.3 | — | 0.00205 | 0.0722 |

Table 5.6: Results for wave4main(2000), which predicts the tides in a rectangular estuary of the North Sea.

the standard stop-and-copy algorithm, but with substantially smaller pause times, typically much less than 1 millisecond. The mean pause times are, on average, approximately 360 times higher for stop-and-copy than for EB. Technically, we are interested in the maximum pause time (rather than the average), but our initial experiments show that this is too small to make accurate measurements with the profiling tools that were available. This is discussed in Section 5.5.2.

The runs for EA show significantly shorter pause times, of the order of 1 microsecond (approximately 18000 times shorter than stop-and-copy), but with larger overheads (an average of 15%), due to the cost of the `gc_in_progress` test at each allocation and the overheads of switching control between mutator and garbage collector.

The highest EB overhead (22%) is for primes 1500. The resident (live) data is quite small in this case and the nursery never exceeds the 256KB minimum allocation. The residency is around 24% and the allocation rate is approximately 40MB/s.

In an effort to explain the high overhead we ran `contrived` with parameters which matched quite closely the residency and allocation rate of `primes 1500` (`copy`=3000, `work`=25). With these parameters `contrived` showed a 21% overhead for EB when compared with stop-and-copy. This is very similar to that observed in `primes 1500`. As the problem size increases the amount of live data and the residency (as a %) both increase. Our earlier experiments with `contrived` suggest that the overhead should drop. Indeed, for `primes 6000` (see below) the live data averages around 220KB and the residency increases to around 30% with the measured overhead for EB reduced to just under 2%. By setting `work` and `copy` to approximate this behaviour the measured overhead for `contrived` also drops significantly (to around 5%). Qualitatively the relative performance moves in the right direction but we stress that making accurate quantitative predictions in general proves to be very hard.

**Remark:** In these benchmarks the stop-and-copy collector actually performs very well in terms of mean pause time; the longest mean pause time recorded in these experiments was just 117 milliseconds (`pic`). However, it is very easy to construct applications with substantially longer pause times. For example with `work=1, copy=2.5M` in `contrived` the mean pause time is around 1.6 seconds.

| Application | Param(s) | Stop-and-copy | Baker | Self-scav (EA) | Self-scav (EB) |
|---|---|---|---|---|---|
| `pic` | 8000 | 22.69 | 28.32 (+25%) | 26.87 (+18%) | 23.92 (+5.4%) |
| `anna` | preludeList | 152.06 | 468.3 (+308%) | 159.9 (+5.1%) | 153.7 (+1%) |
| `circ` | 8 500 | 15.26 | 22.13 (+33%) | 18.60 (+12%) | 16.59 (+8.7%) |
| `primes` | 6000 | 71.58 | 100.31 (+40%) | 79.82 (+12%) | 72.89 (+1.8%) |
| `nqueens` | 12 | 12.47 | 25.24 (+102%) | 13.97 (+12%) | 13.00 (+4.2%) |
| `wave4main` | 6000 | 17.58 | 25.33 (+44%) | 18.95 (+7.8%) | 17.66 (+0.4%) |
| | | **Arith. mean** | +92.0% | +11.2% | +3.6% |
| | | **Geom. mean** | +59.8% | +10.36% | +2.2% |

Table 5.7: Results for larger runs

### Longer Runs

To conclude, we executed each of the `nofib` applications with larger problem sizes than could be accommodated in the above experiments. These are summarised in Table 5.7 which also shows the arithmetic and geometric means of the overheads of each scheme compared to stop-and-copy. Compared with the earlier experiments we find that the overhead for two of the applications increases slightly with the problem size (`pic` and `circ`), three decrease (`anna`, `primes` and `wave4main`) and one stays approximately the same (`queens`).

We can understand some of these trends from experience with `contrived`. For example, an analysis of `circ` shows that with significantly larger problem sizes, the maximum heap size is being reached. This is analogous to the situation earlier when we tried to increase `copy` in `contrived` which lead to an increase in overhead. On the other hand, in `wave4main`, for example, the amount of live data increases more slowly and the allocation rate drops compared to the earlier problem size; this accounts for the slight drop in overhead. The results for `anna` are somewhat anomalous for Baker (308% overhead for `preludelist`). The allocation rate for this program is relatively low (an average of just under 3MB/s) with between 5 and 10MB allocated between each garbage collection. Existing heap objects are therefore being accessed at a very high rate relative to the allocation of new ones, which explains the very poor performance of the conventional read barrier implementation. In the new scheme, there is *no* read barrier overhead when the garbage collector is off. This conversely explains the very small overhead seen with both the EA and EB schemes. The other trends can broadly be explained in similar terms — `primes`, for example, was discussed earlier.

### Pause Time Distribution

Although we have been able to determine accurately the average pause time in each application, determining the pause time distribution by measuring each of the individual pauses is much harder because of the resolution of the clock available to profile the code. In fact, with sub-millisecond pause times, and the majority of pauses being on the order of tens of microseconds, direct access to the hardware register for the processor clock and cycle count are required. The graphs in this section were created from measurements captured by instrument-

ing our collector with cycle count profiling code that directly accesses the RDTSC processor register. This profiling code can only provide system-wide not process-local measurements, so operating system overheads and context switches pollute the measurements somewhat. The graphs are really only useful in confirming the results obtained above and in providing a rough indication of pause time durations and frequencies. Furthermore, they provide no insight into the minimum mutator progress of the runtime system, which in a work-based system such as our prototype, can severely degrade as a high frequency of short (microsecond) pauses aggregate to create much longer pauses that in reality violate the incremental bounds.

In Chapter 7 our benchmark results are collected on a faster, more sophisticated processor, using the profiling library, PAPI [TICL], that is able to collect accurate process-local measurements with the level of precision we require for a detailed study of our production collector's MMU characteristics.

## Pause Time Distribution for `anna (EA)`

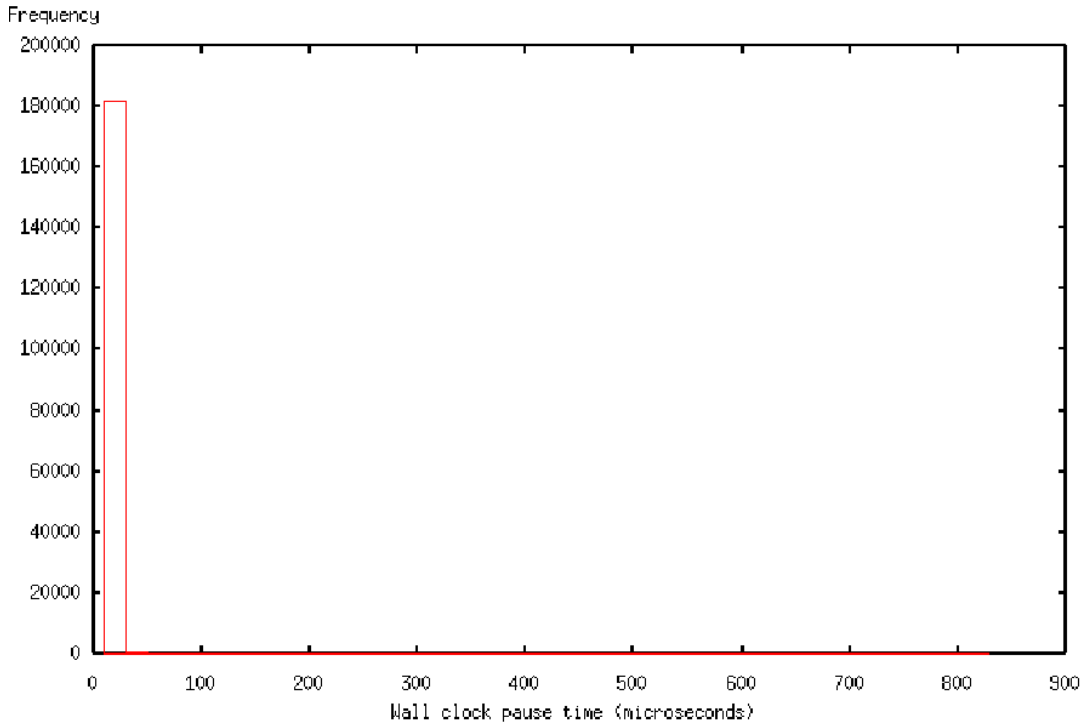



Figure 5.8: Pause time distribution for `anna (EA)`

| **Mean ($\mu$s)** | **Standard deviation ($\mu$s)** | **Sample variance ($\mu$s)** |
|---|---|---|
| 11 | 4 | 18 |

Table 5.8: Pause time distribution for `anna (EA)`

# Pause Time Distribution for `circ (EA)`

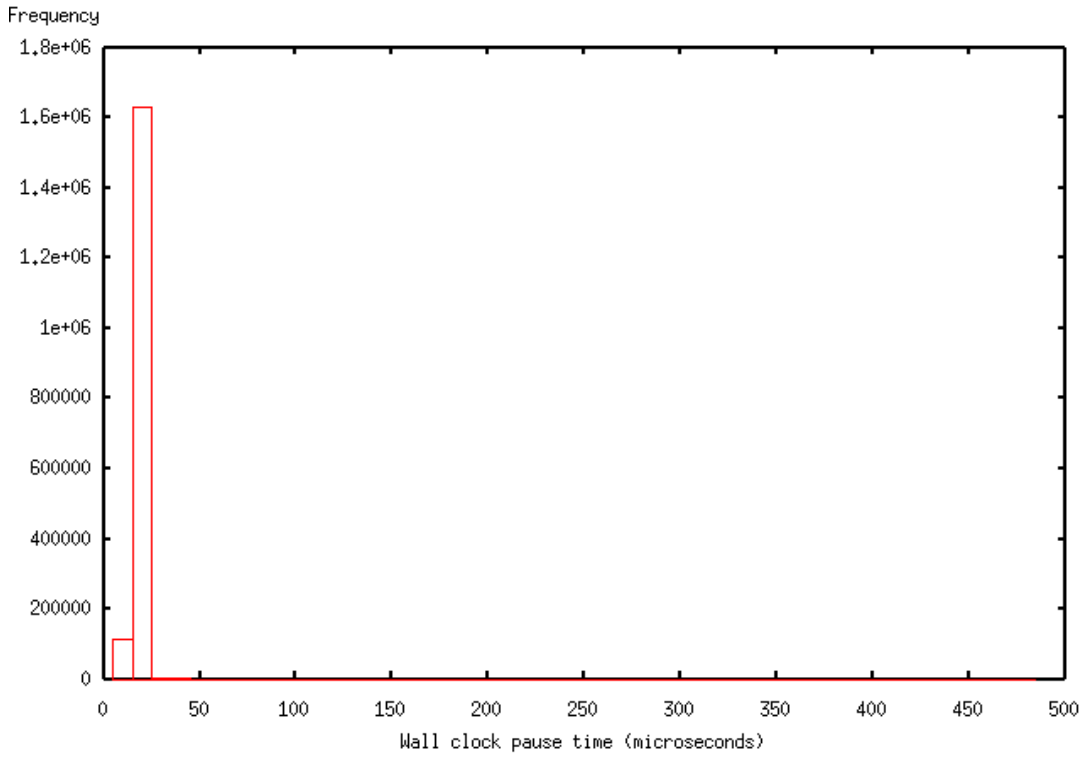



Figure 5.9: Pause time distribution for `circ (EA)`

| **Mean ($\mu$s)** | **Standard deviation ($\mu$s)** | **Sample variance ($\mu$s)** |
|---|---|---|
| 11 | 2 | 5 |

Table 5.9: Pause time distribution for `circ (EA)`

## Pause Time Distribution for `nqueens (EA)`

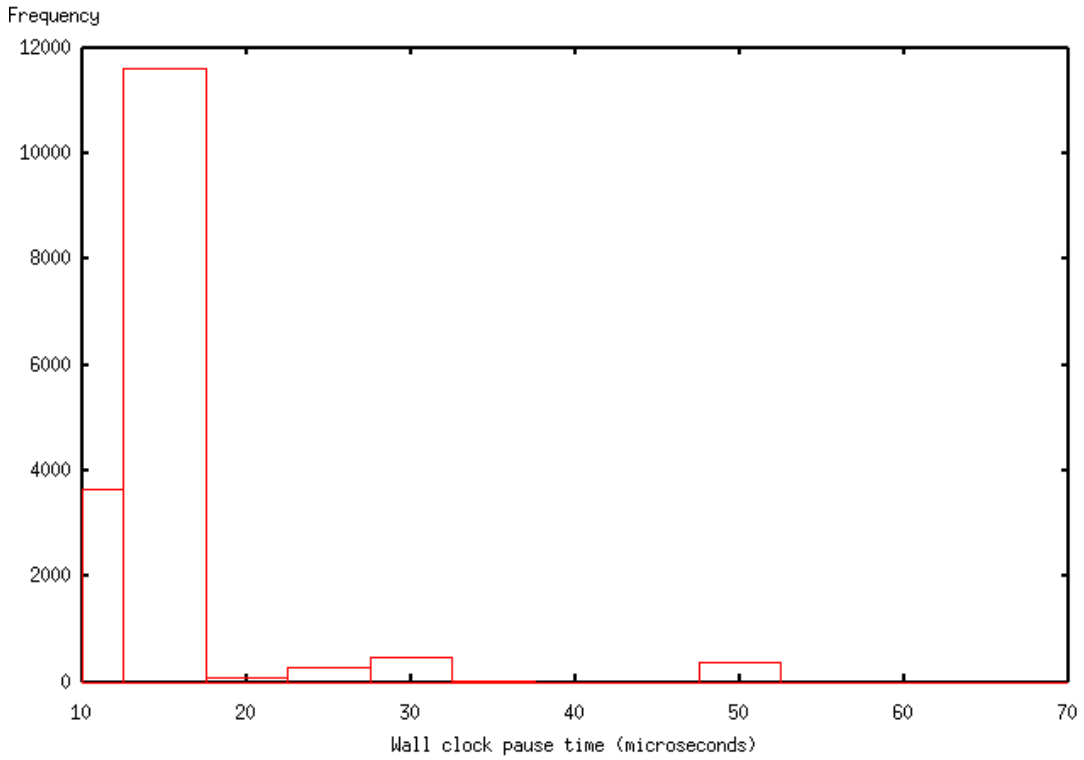



Figure 5.10: Pause time distribution for `nqueens (EA)`

| **Mean ($\mu$s)** | **Standard deviation ($\mu$s)** | **Sample variance ($\mu$s)** |
|---|---|---|
| 12 | 7 | 43 |

Table 5.10: Pause time distribution for `nqueens (EA)`

## Pause time distribution for `pic (EA)`

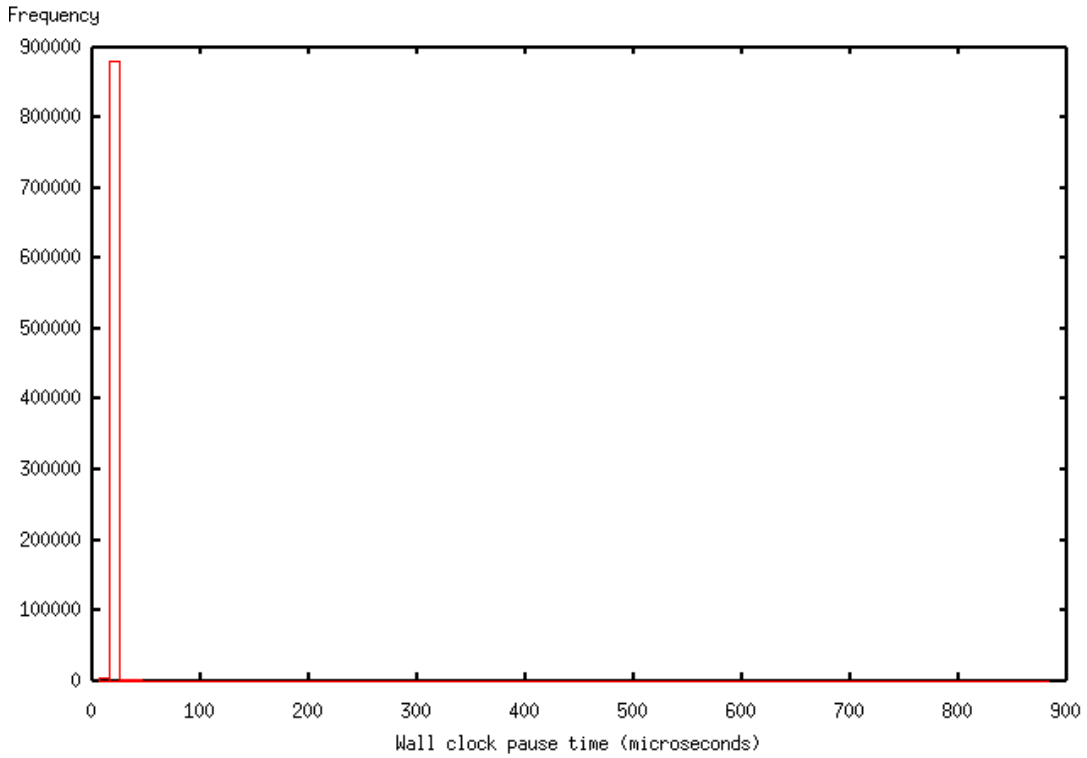



Figure 5.11: Pause time distribution for `pic (EA)`

| **Mean ($\mu$s)** | **Standard deviation ($\mu$s)** | **Sample variance ($\mu$s)** |
|---|---|---|
| 12 | 5 | 20 |

Table 5.11: Pause time distribution for `pic (EA)`

## Pause Time Distribution for `primes (EA)`

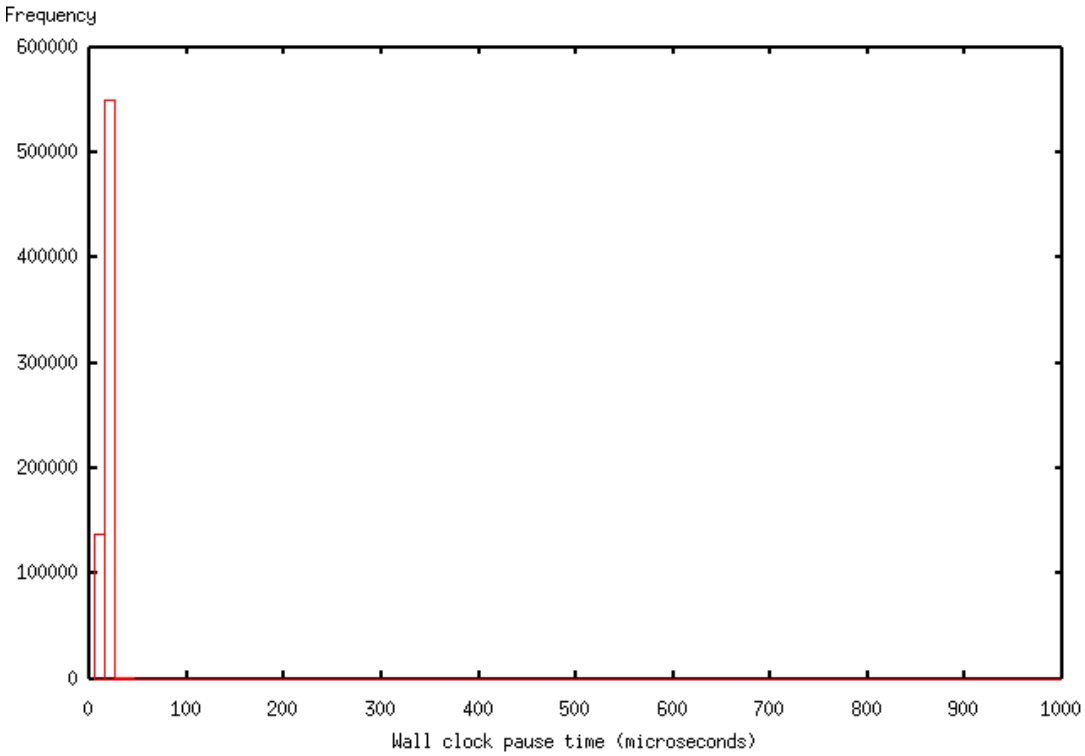



Figure 5.12: Pause time distribution for `primes (EA)`

| **Mean ($\mu$s)** | **Standard deviation ($\mu$s)** | **Sample variance ($\mu$s)** |
|---|---|---|
| 11 | 11 | 130 |

Table 5.12: Pause time distribution for `primes (EA)`

## Pause Time Distribution for `wave4main (EA)`

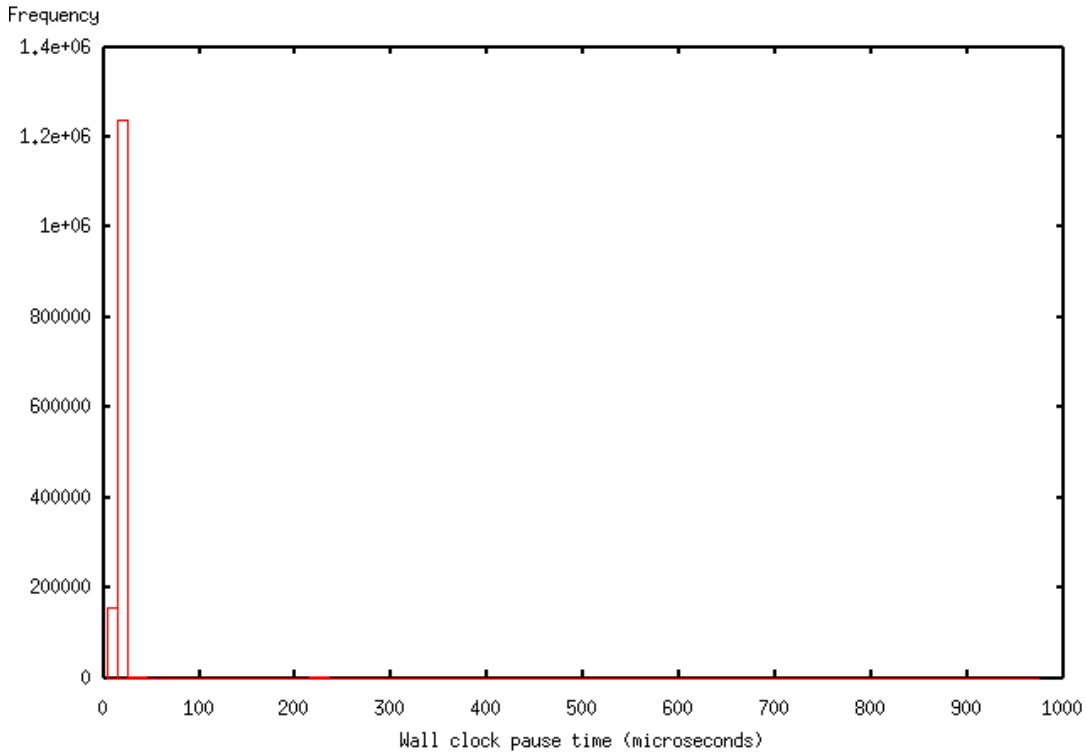



Figure 5.13: Pause time distribution for `wave4main (EA)`

| **Mean ($\mu$s)** | **Standard deviation ($\mu$s)** | **Sample variance ($\mu$s)** |
|---|---|---|
| 12 | 5 | 22 |

Table 5.13: Pause time distribution for `wave4main (EA)`

## Pause Time Distribution for `anna (EB)`

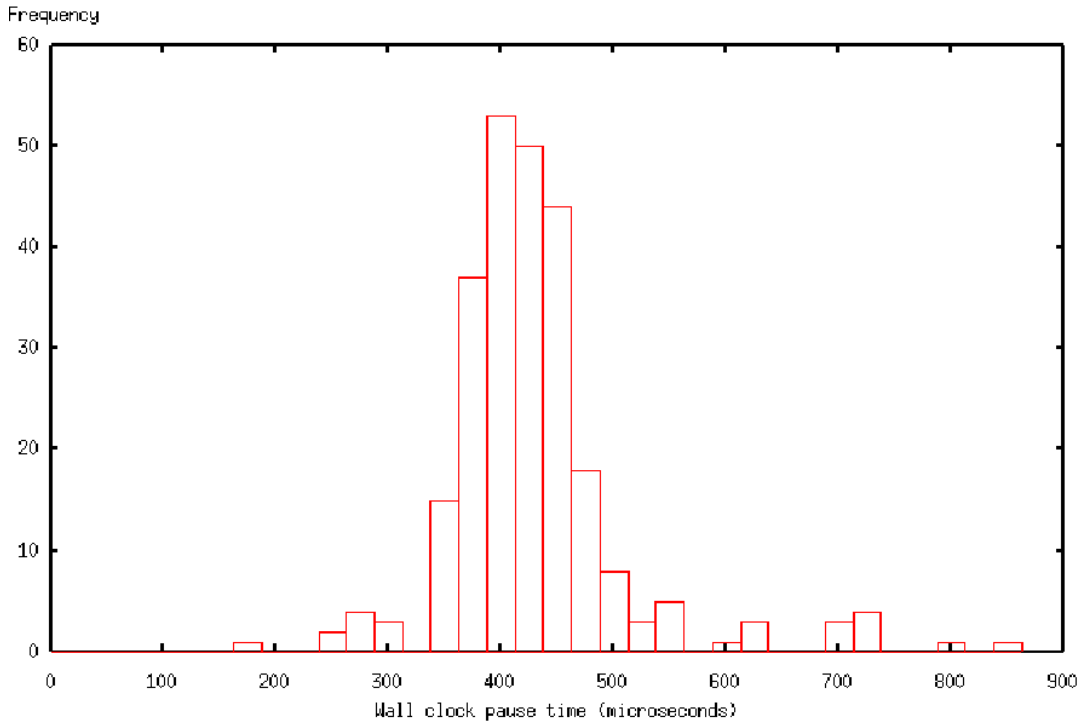



Figure 5.14: Pause time distribution for `anna (EB)`

| **Mean ($\mu$s)** | **Standard deviation ($\mu$s)** | **Sample variance ($\mu$s)** |
|---|---|---|
| 420 | 87 | 7511 |

Table 5.14: Pause time distribution for `anna (EB)`

## Pause Time Distribution for `circ (EB)`

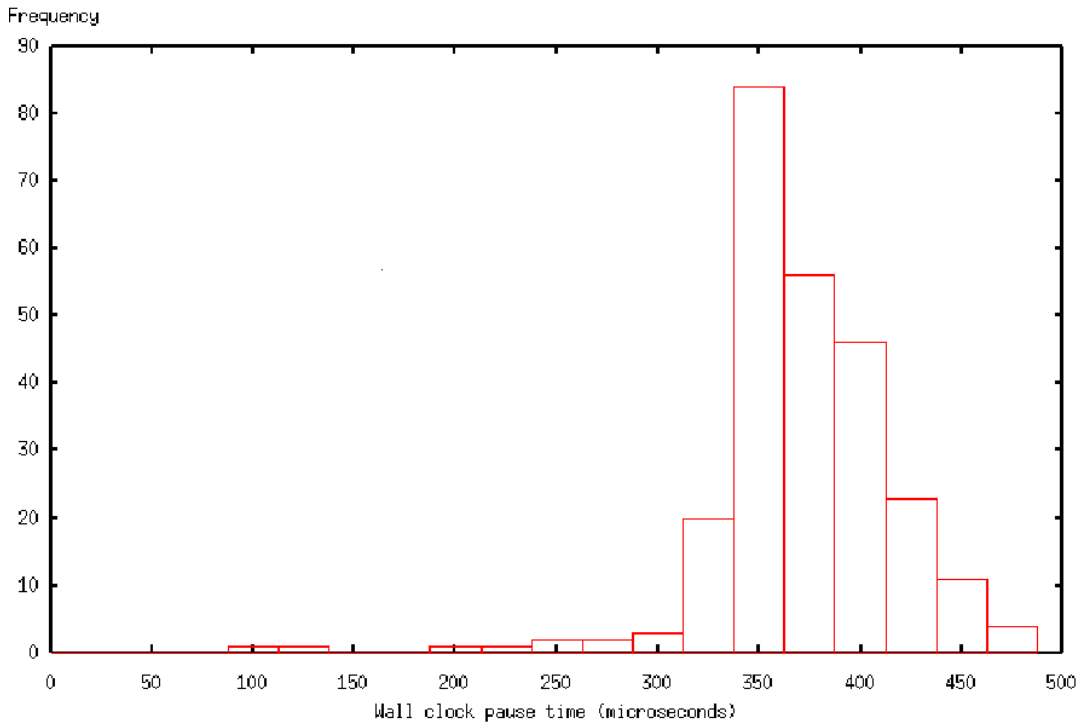



Figure 5.15: Pause time distribution for `circ (EB)`

| **Mean ($\mu$s)** | **Standard deviation ($\mu$s)** | **Sample variance ($\mu$s)** |
|---|---|---|
| 359 | 45 | 2069 |

Table 5.15: Pause time distribution for `circ (EB)`

## Pause Time Distribution for `pic (EB)`

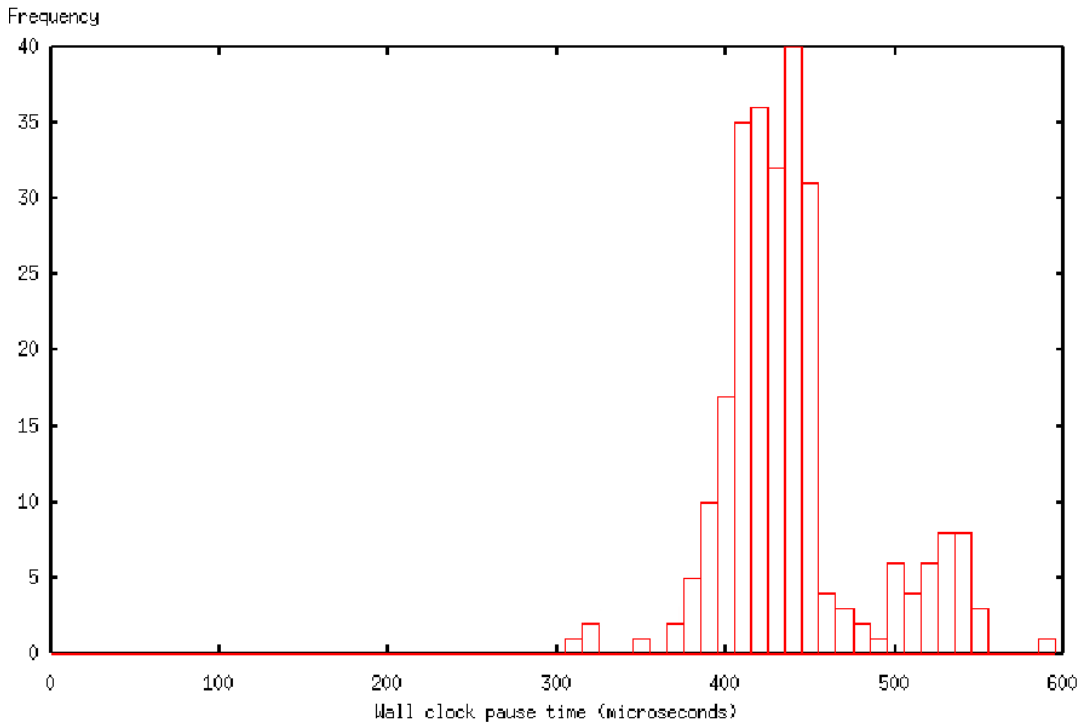



Figure 5.16: Pause time distribution for `pic (EB)`

| **Mean ($\mu$s)** | **Standard deviation ($\mu$s)** | **Sample variance ($\mu$s)** |
|---|---|---|
| 436 | 42 | 1765 |

Table 5.16: Pause time distribution for `pic (EB)`

## Pause Time Distribution for `wave4main (EB)`

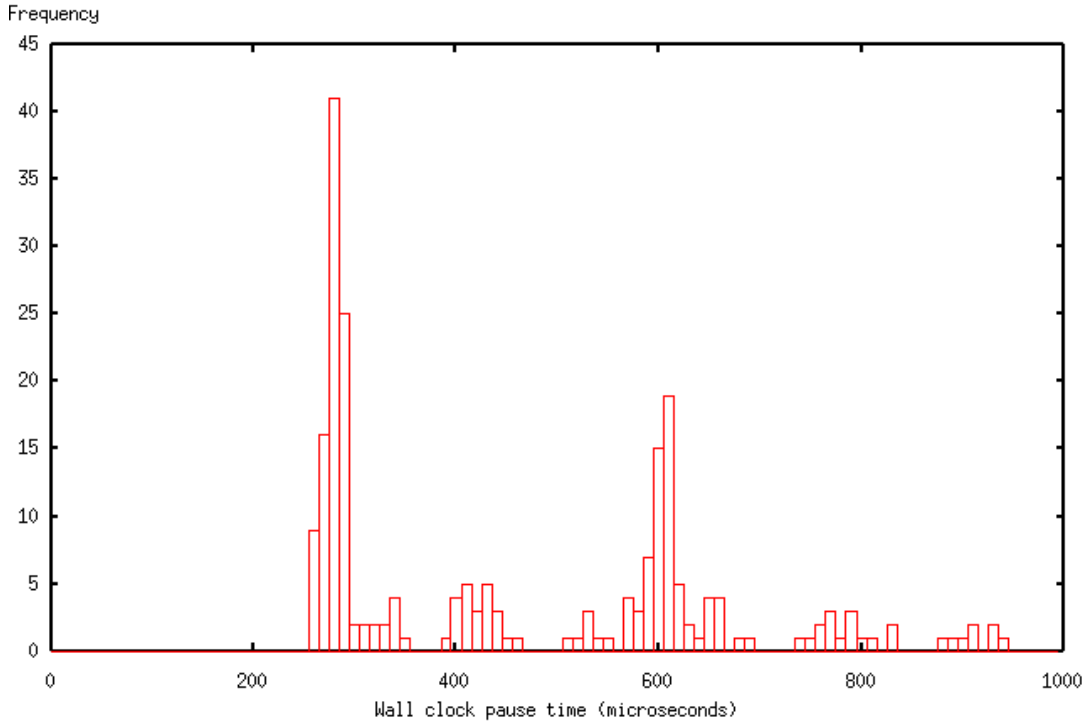



Figure 5.17: Pause time distribution for `wave4main (EB)`

| **Mean ($\mu$s)** | **Standard deviation ($\mu$s)** | **Sample variance ($\mu$s)** |
|---|---|---|
| 455 | 190 | 36411 |

Table 5.17: Pause time distribution for `wave4main (EB)`

## 5.6 Discussion

The results we have produced are extremely favourable and the experiments we have performed show that the software read barrier can be implemented in systems like the STG-machine at very low cost. The overheads and mean pause times we have reported are significantly lower than for other published schemes. However, we should make it clear that the basis of the scheme is indirection and it is the fact that indirection is already inherent in the STG-machine that the read barrier can be achieved so cheaply.

### 5.6.1 Pause Time Distribution

The two aspects of the current scheme for which there is no *guaranteed* upper bound on pause time is in the handling of unpointed objects and in stack scavenging.

For unpointed objects the evacuation cost is proportional to the size of the object, unless it is a large object; in this case it will reside in its own block and can be evacuated in constant time. The main cost is in the scavenging of unpointed objects: an unpointed object may contain references to other objects, including other unpointed objects. The pause time in such cases may be very hard, or even impossible, to bound. Our approach to this problem, using the techniques presented in Section 5.1.5, are an attempt at a practical compromise. We are able to avoid bounds violation of the allocate/mark ratio ($k$) increment when evacuating and collector-scavenging unpointed objects. However, as soon as the mutator attempts to access a partially evacuated or scavenged object, we are forced to eagerly evacuate (if necessary), and scavenge the remaining portion of the object's payload through to completion. An alternative approach is to implement a more intrusive explicit read barrier for each unpointed object primitive.

For reasons of simplicity incremental stack scavenging at present operates only between update frames. With some additional effort this can be adapted to work between arbitrary stack frames by hijacking *all* return addresses. This requires separate self-scavenging info tables to be available for all types of stack frame. Alternatively, we can build a generic self-scavenging info table provided we retain an additional copy of the return address in each stack frame. This is analogous to what happens in the current treatment of heap objects where we retain a copy of the old info table pointer in evacuated objects. These issues are explored in Section 7.1.4.

Our initial focus has been on minimising the individual pause times of our collector. However, as discussed in Section 5.5.2, we recognise that the work-based scheduling of the scavenger, where bursty allocation can lead to pause aggregation that violates the incremental bounds, can be problematic in achieving a minimum mutator utilisation that is useful for real-time applications. We address this issue with the implementation and evaluation of a lightweight time-based scheduler for our production collector in Section 7.2.3 and Section 7.3.

### 5.6.2 Space Overheads

In the current implementation of the scheme an additional copy of the original info pointer is maintained when evacuating an object. This simplifies some aspects of the implementation but it carries a (transient) space overhead on all evacuated objects which can affect garbage collector performance, both in maintaining the extra copy and invoking more frequent garbage collections when the heap approaches saturation. With some additional effort in compilation this additional word can be eliminated, by changing the layout of the existing info tables and adding extra fields to store the entry code and self-scavenging code addresses. A separate self-scavenging info table is also required for each closure type. This increases the total space required to store the info tables but the overhead is static (see Section 7.1.1).

With this modification in place, closure updates become more complicated, specifically if an object in to-space is updated with a value which is smaller than the original object. This can be overcome by adding a dummy info table pointer in the first word of the dead space at the end of the new object which appears to the scavenger to be an object of the same size as the dead space, but containing no pointer fields. Its effect is to cause the scavenger to skip automatically to the next evacuated object in the collector queue when it encounters it. The implementation of skip-scavenging slop objects are discussed in Section 7.1.5.

### 5.6.3 Generational Schemes

A number of incremental schemes have been developed based on generational storage structures in an attempt to avoid the read barrier. In a generational scheme, the heap is divided into two or more regions (generations), one holding freshly-allocated, but typically short-lived objects, and the remainder holding older objects copied from previous generations. Because the youngest generation typically contains a small amount of live data the cost of evacuating the data into an older region is usually small. These so-called *minor* collections consequently result in small average pause times.

The main challenge is in the incremental collection of objects in the older generations. As an example, the replicating collector of Nettles et al. [NOPH92, NO93] allows the mutator to access from-space whilst garbage collection is in progress. Correctness is maintained by patching the mutator roots at the end of a garbage collection and maintaining a mutation log of all changes to from-space objects from the mutator with the aid of a write barrier. The scheme yields a slowdown of less than 20% for the benchmarks tested and less than 10% when restricted to major collections. A similar approach is used in [HL93] also for ML which allows access to immutable objects in from-space.

In Section 7.5, we briefly discuss our own experiences with a prototype implementation of the replicating collector of Nettles et al. for GHC. In summary, we were unable to obtain anywhere near the performance reported by Nettles et al., instead incurring execution overheads well in excess of 370% and in the majority of cases, in excess of 650%, when executed on a uniprocessor platform.

Incremental schemes have also been developed for multi-threaded systems, such as [DL93] which is targeted at Concurrent Caml. Here immutable objects are allocated in private heaps and are copied to a shared heap during local garbage collection within the associated thread. The shared heap is collected by a separate thread operating Dijkstra's concurrent mark/sweep algorithm (Section 3.4.4). The scheme has the advantage that the threads can perform minor collections independently; however, mutable objects have to be allocated in the shared heap which incurs an additional expense. The average pause time for minor collections is favourable [DL93], but there is no guaranteed upper bound: the time is proportional to the amount of live data in the private heaps.

The scheme we have proposed here can be easily adapted to generational schemes and we examine such an implementation for Haskell in Chapter 7. Unlike the existing schemes, however, the fact that the read barrier comes at such low cost means that *all* collections, including minor collections, can, in principal, be performed incrementally. Alternatively, minor collections could be performed on the conventional 'stop-the-world' basis with the incremental scheme being used for the older generations.

## 5.7 Summary and Conclusions

We have described a scheme for incorporating incremental garbage collection into the STG-machine on stock hardware. The scheme is based on the manipulation of the code-pointers that are used to implement closure behaviour in the STG-machine. Each closure creates its own, 'personal' read-barrier by manipulating its code-pointer during execution and garbage collection. The technique derives its efficiency from the twin facts that these manipulations are very cheap and that they are performed on a per-closure basis, rather than a per-pointer-load basis, as in most implementations of the read-barrier. Experiments suggest that the new scheme carries a very low overhead, averaging less than 4% in execution time for the benchmarks tested with average pause time being sub-millisecond in all cases. This shows that in systems which implement dynamic dispatching exclusively, such as the STG-machine, a portable read barrier can be implemented extremely cheaply leading to very short mutator pauses with negligible overhead in execution time.

We have also found that the block-based memory allocator of the GHC runtime system provides an excellent level of granularity at which to perform incremental scavenging. By scavenging on each block allocation we can keep the scavenger going for longer, at the expense of an increase in pause time. This substantially reduces the overheads in the proposed scheme and provides an additional mechanism (the block size) by which to control the behaviour of the garbage collector.

# Chapter 6

# Bridging the Generation Gap

As discussed in Section 2.4.1, generational garbage collection is a well-established technique for reclaiming heap objects no longer required by a program. It reclaims short-lived objects quickly and efficiently, and promotes long-lived objects to regions of the heap which are subject to relatively infrequent collections. It is therefore able to manage large heap spaces with generally short pause times, these predominantly reflecting the time to perform minor collections.

A drawback of all but the simplest generation schemes is the need for a write barrier to capture references from older to younger generations. These must be added to the root set of the younger generation during a minor collection. Unfortunately, the write barrier must be applied to *all* objects, not just those in the older generation(s), as it is not possible to determine statically in which generation the object will reside. In some applications this can have a significant effect on the execution time of a program, particularly in lazy languages where updates to old objects are common.

In the previous chapter we described a prototype implementation of an incremental semi-space garbage collector that employs a dynamic dispatching read barrier to enforce the to-space invariant. In Haskell, data is largely immutable, and experiments with generational collector implementations within GHC, have demonstrated that the weak generational hypothesis holds, supported by high infant mortality rates [Sew92, SP93]. In this chapter we apply the techniques developed for the removal of the incremental read barrier to the remembered set write barrier of GHC's generational collector.

1. We would like there to be *no* write barrier when updating thunks in the young generation as these updates cannot generate inter-generational pointers.

2. When updating an object in the old generation we would like the thunk to be added to the remembered set *automatically* either when it is entered or when it is eventually updated.

Our ultimate goal then, is to produce a barrierless incremental generational garbage collector with low overhead and consistent utilisation.

The bottom line is that we want the update code to be different for the young and old generations. The trick is to hijack the info table pointer associated with a thunk at the point where the garbage collector promotes it so that it behaves differently depending on the generation in which it resides. One particular technique that we present (Section 6.2.2) achieves this by building specialisations of a closure's entry code. A similar idea was proposed independently Niklas Rojemo in [RÖ95], but is only briefly discussed and does not explore the possible design space or evaluate the efficiency of the scheme.

Rojemo describes a technique that avoids the write barrier in the Chalmer's LML/Haskell compiler. The idea is to hijack the info table pointer stored in each object so that it points to a different table for each generation. In particular, the object's entry code for the old generation(s) contains a preamble that automatically adds the object to the inter-generational pointer set when the object is entered for the first time. In order to do this, he proposed specialising the entry code of an object according to the generation in which it resides. The focus of the paper [RÖ95], however, is on the issue of space leaks, which can prove very significant in lazy languages. He observes that the proposed scheme can also be used to age objects efficiently within a generation, each age being associated with a separate info table for each object, but does not develop an implementation.

Our scheme, like Rojemo's, provides an elegant way of avoiding the tests usually associated with object updates, both when ageing the object and when updating it after it has been evaluated. However, unlike our scheme, Rojemo's suffers seriously from code bloat since each object type must have a separate info table for each generation, or worse for each tenure step. Furthermore, his scheme is much less efficient, prematurely adding objects to the remembered set and thus incurring additional time and space overheads. In this chapter we explore a range of techniques which variously i) trade dynamic and static space overheads, and ii) alter the time at which objects are added to the inter-generational pointer set. This culminates in the presentation of a new scheme for implementing generational garbage collection that avoids the write barrier whilst simultaneously avoiding the code bloat problems associated with thunk specialisation.

## 6.1 Generational Garbage Collection in Haskell

In order to set the scene for the subsequent description of our techniques it is useful to summarise the salient points of Section 2.4.1 and those most relevant from Section 4, with particular reference to their implementation within the Glasgow Haskell Compiler.

### 6.1.1 Single Mutation Thunk Updates

Recall from Section 4.4 that some closures represent unevaluated expressions, variously referred to as "thunks", "suspensions", or "futures"; these are the principal mechanism by which lazy evaluation is supported. When the value of a thunk is demanded, the mutator simply enters the closure by jumping to its entry-code without testing whether the thunk has already been evaluated. When the evaluation of the thunk is complete, the thunk is overwritten with an indirection to the closure's value. Note that when the heap representation of the resulting value is smaller than the original closure, the original closure can be overwritten directly with the value – called an *in-place* update – but this is rarely done. If the mutator tries to evaluate the thunk again, execution will now land in the code for the value (i.e. code that returns it), instead of code to compute it. These single-mutation closure updates occur with high frequency and most usually to young objects. These updates are by far the most common form of object mutation within Haskell, for the majority of data is immutable. In the rare case where such data must be modified, a copy is taken and the old copy is discarded and subsequently garbage collected.

There is an inherent delay between a closure being entered for the first time and the closure being updated with its value. An attempt to enter a closure that has already been entered but not updated results in an infinite loop. To trap this, objects are marked as *black holes* [Jon92] when they are entered[1]. The performance of the garbage collector can be substantially influenced by the presence of these black holes.

During program evaluation, the GHC stack contains, among other objects, update frames which have the same layout information as closures, with their own static info table. Update frames are pushed onto the stack by updateable function applications, which are applications with the full complement of arguments (*redexes*). The update frame contains a pointer to the application object (updatee) and to the next update frame in the stack. When the application has been evaluated the corresponding update frame will be at the top of the stack. This is popped and the application object that it references is updated with (most usually) an indirection to the value. Thunk updates thus ensure that redexes are computed at most once. When this is done, the evaluator returns to the stack object underneath the update frame by entering it. All thunks push an update frame as a result of executing their entry code — it is exactly this that our scheme exploits.

### 6.1.2 Heap Layout

As Figure 6.1 depicts, GHC's generational collector can be configured with any number of generations. With a single generation it operates as a simple semi-space collector. The most common configuration uses two generations, although the use of three generations is not uncommon. Each generation is further divided into a number of *tenure steps*, `S` (Figure 6.1).

[1] Optionally, black holing can also be performed lazily by the garbage collector by running down the stack and black-holing all thunks that are referenced from it.

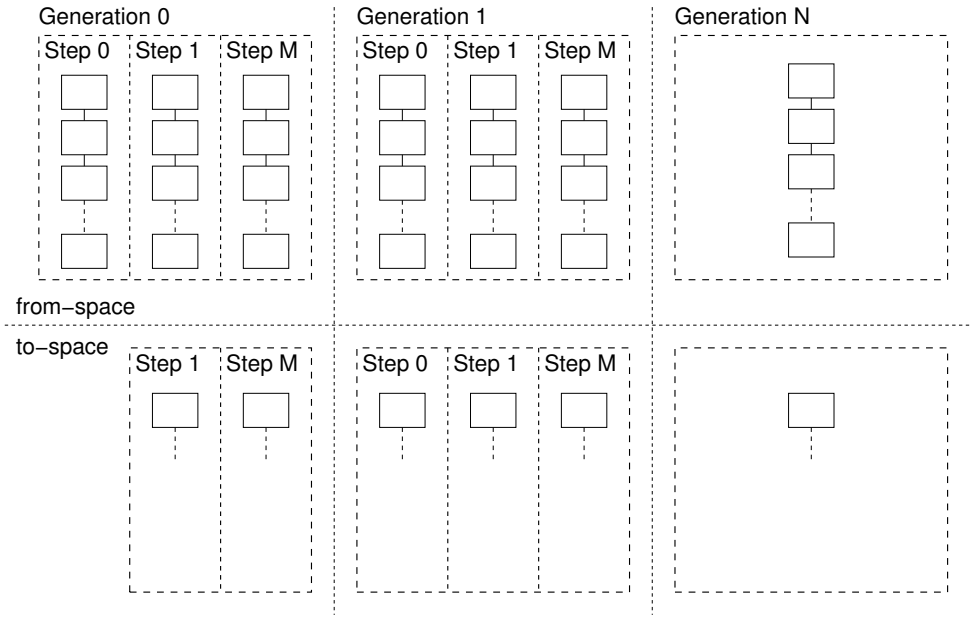



Figure 6.1: GHC's generational heap layout at the beginning of a major collection.

Step 0 of generation 0 is the *nursery* or *eden* allocation area from which new objects are allocated. The list structure within each step of each generation shown represents the block allocated heap area for that particular generation and step — recall that GHC utilises a layered storage system (Section 5.2), that chains 4KB-blocks together for the heap regions and then bump-pointer allocates data structures within each block.

As explained in Section 2.4.1 generational collection is motivated by the observation that most objects die young [Wil92], so that space can be recycled more efficiently by concentrating reclamation efforts on the space occupied by recently allocated objects. The most basic generational collector configuration employs two generations each with a single tenure step i.e. $N = 2$ and $S = 1$. The idea is to make all new allocations in the "young" generation or, to allow straightforward generalisation, "Generation 0". When this region fills up, the collector is invoked to reclaim the unused objects within it. The recovered objects are exclusively those that were created since the last collection and which subsequently died (young). Objects that are still live are copied to the other region — the "old" generation, or "Generation 1" — and we say that the object has been *promoted*. When the old generation itself fills up, the entire heap, i.e. the union of both generations, is garbage collected. Collections that work exclusively on the younger generation are called *minor* collections; those that involve both generations are called *major* collections.

### 6.1.3 Object Ageing

In this very simple scheme an object is deemed to be "old" if it is alive when the collector is invoked. This is not in general a good definition since the object may only just have been created when collection begins. A much better way of ageing the objects is to count how many collection cycles the object has survived. Only if the object age is beyond a specified limit (the *tenure age*, measured in collection cycles) is the object promoted to the old generation.

GHC ages objects by *tenuring* (intra-generationally promoting) objects into the next tenure step in the generation at both minor and major collection cycles. The default arrangement has been experimentally determined as a two step configuration for all generations but the last. GHC thus operates with two generations and two steps per generation, i.e. $N = 2$ and $S = 2$, as the default configuration. The generation number and step number that a 4KB heap block resides in is recorded in the header of the block itself — space need not be reserved within each object for this information.

### 6.1.4 The Immutable and Mutable Lists

With $N = 2$ and $S = 1$ the heap comprises two regions (an old and a new generation). An object $O$ in the old generation can refer to an object $Y$ in the young generation if $O$ is promoted but $Y$ is not, or if $O$ is overwritten with a reference to $Y$. The latter can happen in GHC when a promoted thunk is updated and replaced by (overwritten with) its value; typically this will be a reference to a newly-created object, i.e. one in the young generation. When collecting the younger generation it is important to keep track of these inter-generational references as they form part of the younger generation's root set.

To achieve this, the collector maintains a separate data structure, most commonly referred to as the *remembered set*, which is used to build the inter-generational *root set* of objects for a *minor collection*, i.e. the set of objects whose descendants collectively form the set of live objects in the young generation that are accessible from the old. The same idea generalises straightforwardly to more than two generations.

Note that this establishes an important invariant for generational garbage collection: *All references from the old generation to the young generation are accessible from the remembered set.*

In the context of GHC we refer to the remembered set as the *immutable list*, reflecting the fact that it comprises objects whose values (in the mathematical sense) cannot change – a consequence of Haskell's functional semantics. GHC employs an explicit immutable list in preference to the card marking approach discussed in Section 2.4.1. Whenever a generation is collected, the entire generation, that is all the steps within the generation, are collected together. As a result, there need only be a single immutable list associated with each generation, as opposed to one for each step.

GHC builds the immutable list by chaining together ("threading") its component objects using a reserved field within each object – the `link` field. This requires closures residing in

generations *greater* than 0, to occupy a minimum of three words: the indirection info pointer, the inter-generation reference to the WHNF value in the younger generation, and the link field.

The collector employs an *eager promotion* strategy, in which the evacuation code attempts to promote a value closure to the same generation as the object(s) that refer to it. Note that it is not always possible to do this. If an object has at least two referrers, each of which reside in different generations that are undergoing collection, the object can be evacuated only once in any given cycle and cannot be duplicated without loss of sharing. Furthermore, as a result of this "failed evacuation", if the object has been evacuated to the younger generation of the two referrers then the older generation referrer must ultimately be chained onto the immutable list of the younger generation.

It is important to understand that objects (specifically closures) in GHC can only be updated at most once, so will only be added to the immutable list at most once, as a side effect of tripping over the object's write barrier. However, GHC also supports *mutable* objects, such as arrays[2] which may be subject to an arbitrary number of destructive, i.e. non-functional, updates. To avoid adding the same mutable object to the immutable list each time it is updated, these objects are chained together on a separate *mutable list*. If a mutable object is relocated to a new generation it is automatically added to the mutable list for that generation. The traditional remembered set for a generation can thus be thought of as the union of its mutable and immutable lists.

### 6.1.5 The Write Barrier

Given the need for an immutable list, how do objects in the old generation get added to it? If an object contains a reference to a young generation object at the point when it is promoted it must be added to the immutable list immediately. However, an object may be overwritten during the normal course of execution and this may introduce a new reference from the old generation to the young. Therefore, each write operation must check the object that is being overwritten to see if it is in the old generation. If so, the object must be added to the immutable list. This additional test is called the *write barrier*. The main problem with the write barrier is that it must be applied to *all* objects since it cannot be statically determined in which generation the object will reside when a write takes effect. The write barrier can be a major source of inefficiency in some applications. It is worth noting that, in the majority of functional programming languages, the write barrier need only be applied to those write operations that are updates and not to those that initialise objects. Initialising writes create references that must necessarily be to older objects and thus need not be remembered.

The key point is that new references from the old to young generation (by 'new' we mean references that appear after an object is promoted) can come about only when a closure is updated with its value. Thus, the write barrier in a lazy functional language like Haskell

[2]Thread stacks are also modifiable in the same sense.

is implemented in software by planting test code around each update. If the object being updated is in the old generation, a reference to that object must be added to the immutable list. Function `updateWithIndirection()` outlines the pseudo-code for the write barrier and this enclosing update operation.

The cost of the write barrier depends on the way heap memory is organised. For a single monolithic heap, partitioned to form the two generations, the test is a simple comparison between the object's address and the partition boundary. In GHC's block-allocated memory system the object's address must first be mapped to a block identifier and the block header then examined to determine in which generation it sits. We have measured the fast-path cost of GHC's write barrier to be 15 instructions on a Pentium III, compared with just two instructions for a contiguous heap.

### 6.1.6 Completing the Picture

Figure 6.1 shows the from-space and to-space heap arrangement at the beginning of a major collection. By default, the collector operates solely as a fully generational copying collector, for all generations. If enabled, when the residency of the old generation reaches a specified limit, the older generation is collected using in-place compacting collection. Notice that Generation 0 step 0 is not allocated a set of blocks for to-space — live objects are tenured into Generation 0 step 1. Furthermore, the block allocated structure of the heap enables greater flexibility in both sizing and block allocation strategies. In particular, as depicted on the to-space portion of the diagram, the to-space block lists are allocated by chaining on empty blocks as and when they are needed. Generation 0 step 0 has a user-configurable fixed limit, defaulting to 256KB, which when exhausted results in the triggering of a minor collection. A major collection occurs simply when the old generation has doubled in size, or when a user-configured maximum heap size is reached. At the end of the collection cycle, the "flip" operation returns the current from-space blocks to the block allocator and marks the to-space blocks as now being the new from-space blocks. The nursery blocks are then re-used (modulo any necessary sizing adjustments). Note that the older generations grow on resize by a factor of twice the amount of live data, and also that they are never sized under 1 MB.

### 6.1.7 GHC's Generational Collector

We now briefly outline the generational collection algorithm using very basic C language syntax for the pseudo-code. We omit the specifics of "non-standard" object and block handling, such as those of large and "truly" mutable objects. These will be described more fully in Chapter 7.

### Generational Heap Structures

A `BlockDescriptor` structure is unique to each block managed by the storage manager and contains a block's book-keeping and accounting information. In particular:

- Pointer `start` — Start address of the block's memory.
- Pointer `free` — First free byte in the block's memory.
- BlockDescriptor Pointer `link` — Link field to next block in chain.
- Integer `generationNumber` — The generation number the block belongs to.
- Step Pointer `step` — The step the block belongs to.
- Word `blocks` — The number of blocks in the group if the block is the head of the group, 0 otherwise.
- Word `flags` — Various flag bits, most importantly, a bit indicating whether the block currently resides in to-space.

Block descriptors are not stored contiguously with their associated memory block. Instead they are aligned and padded to a fixed size, starting at a known location, so that the `getBlockDescriptor(Pointer memory)` macro can map a heap address to its block descriptor using pointer arithmetic and bit masking operations.

A `Generation` is represented by a structure containing:

- Integer `generationNumber` — The generation number.
- Step Pointer `steps` — The tenure steps into which the generation is divided.
- Integer `numberOfSteps` — The number of steps in the generation.
- Integer `maxNumberOfBlocks` — The maximum number of blocks in step 0. Exceeding this number requires the generation be collected at the next minor collection when the nursery is again exhausted.
- MutableClosure Pointer `mutableList` — The "truly" mutable objects in this generation (not used in Generation 0).
- MutableClosure Pointer `immutableList` — List of objects referencing younger generation objects.
- Fields used temporarily for the duration of a single garbage collection cycle:
  - MutableClosure Pointer `savedMutableList` – The "new" mutable object list.

A generation's tenure `Step` is represented by a structure containing:

- Integer `stepNumber` — The step number.
- BlockDescriptor Pointer `blocks` — The linked list of blocks in this step.
- Integer `numberOfBlocks` — The number of blocks in this step.
- Step Pointer `toSpace` — The destination step for live objects. This points to either the next tenure step in this generation, or step 0 of the next generation if this step is the final step in the generation. If this is the last step in the last generation then it is a self-reference to this step.
- Generation Pointer `generation` — The generation this step belongs to.
- Integer `generationNumber` — The generation number this step belongs to, cached for performance.
- Fields used temporarily for the duration of a single garbage collection cycle:
  - Pointer `currentHeapPointer` — The next free to-space location.
  - Pointer `currentHeapLimit` — The end of the current to-space block.
  - BlockDescriptor Pointer `currentHeapBlockDescriptor` — The block descriptor of the current to-space block.
  - BlockDescriptor Pointer `toSpaceBlocks` — The block descriptor of the first to-space block.
  - Integer `numberOfToSpaceBlocks` — The number of blocks in to-space.
  - BlockDescriptor Pointer `scavengeBlockDescriptor` — The block descriptor for the block currently being scavenged.
  - Pointer `scavenge` — The location of the scavenge pointer within the current block.

The generational collector uses the following global variables in which to capture state:

- Generation Pointer `generations` — The generations that make up the heap.
- Integer `numberOfGenerations` — The number of generations in the heap.
- Boolean `majorGC` — Whether the current collection cycle is a full heap collection of all generations.
- Integer `N` — The number of generations undergoing collection.
- Integer `evacuationGeneration` — The generation into which a referent should be evacuated. When scavenging an object and applying the eager promotion policy, this variable is set to the generation number in which the referrer resides.
- Boolean `failedToEvacuate` — A boolean indicating whether any referents of the object undergoing scavenging could not be evacuated into its generation. When set, the scavenging routine places the referrer onto the mutable list of its generation (Section 6.1.4).

**Pseudo-code Algorithm**

- Function `generationalGC()` is invoked as the result of failure of a heap overflow check that results from a failed allocation to the fixed (256KB, for example) nursery.
- Function `generationalGC()` first determines whether a major collection has been triggered. If not, it determines the number of generations, `N`, that are to be collected. A generation is deemed full, and thus is collected, if the number of blocks in step 0 of the generation have exceeded the maximum number of blocks for that step; the limit is determined at the end of the previous collection cycle based on either user specified parameters or a function of the amount of live data at the end of that cycle.
- Collection proper starts with the evacuation (see Function `generationalEvacuate()`) of those objects in the root set. The root set comprises of both the mutable and immutable lists *for all* generations, and all roots tracked by the mutator. Recall that each the thread stack is allocated as a large object with the uniform heap layout of a closure and is tracked on the mutable list.
- The to-space blocks for each step of each generation being collected are now repeatedly scavenged until no more objects can be evacuated. The same also applies to those heap blocks in the older generations that are not undergoing collection but into which objects are promoted. The method `scavenge(step *)` scavenges a step, resuming from the scavenge pointer positions (`scavenge` and `scavengeBlockDescriptor`) stored within the step. The generations and steps are scavenged in old-to-young generation order so as to efficiently support eager promotion (Section 6.1.4). Specifically, this ensures that objects are copied at most once, and minimises the length of the immutable list.
- When collecting `N` generations, evacuation (see Function `generationalEvacuate()`):
  - Leaves the object untouched and skips it if it is in a generation greater than the oldest generation being collected, i.e. is greater than `N`.
  - relocates the object in `step` $\rightarrow$ `toSpace`, i.e. the next highest step in its current generation or the first step in the next oldest generation if it is in the last step of its current generation.
  - For immutable objects, applies the eager promotion policy, by attempting to evacuate everything to which it refers into the same generation that the object is being tenured within, or promoted to. Note the objects are not relocated into the same step, just the same generation; they are still tenured within the generation and start in step 0. The global variable `evacuationGeneration` is used to determine the target generation of the object and those child objects to which it refers. If the children cannot be evacuated into its parent's generation (it may already have previously been evacuated), then the parent must be added to the immutable list, thus recording the inter-generation reference.

- The collection cycle terminates with the "flip" operation returning the current from-space blocks in the generations that have undergone collection back to the block allocator and marking the to-space blocks as now being the new from-space blocks of the heap. The nursery blocks are simply re-used (modulo any necessary sizing adjustments).

**Function** `initialiseGenerationalGC` Initialisation of GHC's generational collector

```
   /* Determine which generations are to be collected                    */
 1 N = 0;
 2 for g = 0; g < numberOfGenerations; g++ do
 3     if
       generations[g].steps[0].numberOfBlocks ≥ generations[g].maxNumberOfBlocks
       then
 4         N = g;
 5 majorGC = (N == numberOfGenerations − 1);
   /* For the generations being collected, discard the mutable lists and
      initialise to-space and its scavenging pointers.                   */
 6 for g = 0; g ≤ N; g++ do
 7     generations[g].mutableList = END_MUT_LIST;
 8     generations[g].immutableList = END_MUT_LIST;
 9     for s = 0; s < generations[g].numberOfSteps; s++ do
           /* If generational (not semi-space) collector, there is no
              to-space for Generation 0 Step 0                           */
10         if g == 0 ∧ s == 0 ∧ numberOfGenerations > 1 then
11             continue;
12         BlockDescriptor blockDescriptor = allocateBlock ();
13         step → toSpaceBlocks = blockDescriptor;
14         step → scavenge = blockDescriptor → start;
15         step → scavengeBlockDescriptor = blockDescriptor;
   /* Initialise the scavenging pointers in the older generations, objects
      that are promoted must still be scavenged.                         */
16 for g = N + 1; g < numberOfGenerations; g++ do
17     for s = 0; s < generations[g].numberOfSteps; s++ do
18         Step step = generations[g].steps[s];
19         step → scavenge = step → currentHeapPointer;
20         step → scavengeBlockDescriptor = step → currentHeapBlockDescriptor;
```

**Function** `generationalGC` The main function that drives GHC's generational collector

```
1  initialiseGenerationalGC ();
2  for g = numberOfGenerations − 1; g > N; g− − do
3  |  generations[g].savedMutableList = generations[g].mutableList;
4  |  generations[g].mutableList = END_MUT_LIST;
   /* Scavenge the immutable list roots in old-to-young generation order  */
5  for g = numberOfGenerations − 1; g > N; g− − do
6  |  scavengeImmutableList(generations[g].immutableList);
7  |  evacuationGeneration = g;
8  |  for step = generations[g].numberOfsteps − 1; step ≥ 0; step− − do
9  |  |  generationalScavenge(generations[g].steps[step]);
   /* Now scavenge the ''truly'' mutable object list roots               */
10 for g = numberOfGenerations − 1; g > N; g− − do
11 |  scavengeMutableList(generations[g].mutableList); evacuationGeneration = g;
12 |  for step = generations[g].numberOfSteps − 1; step ≥ 0; step− − do
13 |  |  generationalScavenge(generations[g].steps[step]);
   /* Attempt breadth-first evacuation of the root set tracked by the
      mutator                                                            */
14 evacuationGeneration = 0;
15 foreach root in roots do
16 |  dereference(root) = generationalEvacuate(root);
   /* Repeatedly scavenge generations in old-to-young generation order   */
17 for g = numberOfGenerations; − −g ≥ 0; do
18 |  for s = generations[g].numberOfSteps; − −s ≥ 0; do
   |  |  /* Generation 0 step 0 doesn't have a to-space when collecting
   |  |     generationally --- it does when operating as a semi-space
   |  |     collector.                                                  */
19 |  |  if g == 0 ∧ s == 0 ∧ numberOfGenerations > 1 then
20 |  |  |  continue;
21 |  |  Step step = generations[g].steps[s];
22 |  |  evacuationGeneration = g;
23 |  |  if step → currentHeapBlockDescriptor ≠ step →
   |  |  scavengeBlockDescriptor ∨ step → scavenge < step → currentHeapPointer
   |  |  then
24 |  |  |  generationalScavenge(step);
   |  |  |  /* Loop back to scavenging from the oldest generation       */
25 |  |  |  g = numberOfGenerations;
26 generationalFlip ();
27 resizeGenerations ();
28 resetNursery ();
```

**Function** `generationalFlip` GHC's generational "flip" of block-allocated from- and to-spaces

```
   /* Iterate through all steps of all generations and tidy up                    */
1  for g = 0; g < numberOfGenerations; g++ do
2      for s = 0; s < generations[g].numberOfSteps; s++ do
3          Step step = generations[g].steps[s];
4          if ¬(g == 0 ∧ s == 0 ∧ numberOfGenerations > 1) then
               /* Tidy up the end of the to-space chains                          */
5              step → currenHeapBlockDescriptor → free = step →
               currentHeapPointer;
           /* For those generations that were collected...                        */
6          if g ≤ N then
               /* Free the old blocks, and ''flip'' to-space into from-space
                  for all collected steps except the allocation area.             */
7              if ¬(g == 0 ∧ s == 0) then
8                  freeChain(step → blocks);
9                  step → blocks = step → toSpaceBlocks;
10                 step → numberOfBlocks = step → numberOfToSpaceBlocks;
11                 BlockDescriptor bd;
12                 for bd = step → blocks; bd ≠ NULL; bd = bd → link do
                       /* Clear the ''evacuated'' mark bit of the block
                          descriptor that designates the block as a to-space
                          block.                                                  */
13                     bd → flags &= ¬BF_EVACUATED;
14                 step → toSpaceBlocks = NULL;
15                 step → numberOfToSpaceBlocks = 0;
```

**Function** `generationalScavenge(`*step*`)` GHC's generational scavenging of a block allocated heap step pointed to by *step*

```
   Input: step Pointer to heap step to scavenge
   /* Retrieve step's inter- and intra-block scavenge pointers          */
 1 Pointer scavenge = step → scavenge;
 2 BlockDescriptor blockDescriptor = step → scavengeBlockDescriptor;
 3 failedToEvacuate = FALSE;
   /* Breadth-first scavenging loop of block allocated step             */
 4 while blockDescriptor ≠ step → currentHeapBlockDescriptor ∨ scavenge < step →
   currentHeapPointer do
       /* If the scavenge pointer is at the end of the current block, move on
          to the next one                                                */
 5     if blockDescriptor ≠ step → currentHeapBlockDescriptor ∧ scavenge ==
       blockDescriptor → free then
 6         blockDescriptor = blockDescriptor → link;
 7         scavenge = blockDescriptor → start;
 8         continue;
 9     Pointer closure = scavenge;
10     switch closure → infoTable.closureType do
11         case INDIRECTION
12             closure → indirectee = generationalEvacuate(closure → indirectee);
13             scavenge = closure → payload + closure →
                   infoTable.layout.numberOfPointers;
14         otherwise
15             Integer numberOfNonPointerWords = closure →
                   infoTable.layout.numberOfNonPointers;
16             Pointer end = closure → payload + closure →
                   infoTable.layout.numberOfPointers;
17             for scavenge = closure → payload; scavenge < end; scavenge++ do
18                 dereference(scavenge) =
                       generationalEvacuate(dereference(scavenge));
19             scavenge = scavenge + numberOfNonPointerWords;
       /* If there was a failed evacuation of any of closure's referents then
          it must be placed onto the immutable list, since it contains
          old-to-new generation references                               */
20     if failedToEvacuate == TRUE then
21         failedToEvacuate = FALSE;
22         addToMutableList(closure,
           generations[evacuationGeneration].immutableList);
   /* Update step's inter- and intra-block scavenge pointers            */
23 step → scavengeBlockDescriptor = blockDescriptor;
24 step → scavenge = closure;
```

**Function** `generationalEvacuate(`*closure*`)` GHC's generational evacuation of the heap object pointed to by *closure*

**Input**: *closure* Pointer to closure
**Output**: *toSpaceClosure* Pointer to to-space residing copy of closure

```
1  Boolean continueEvacuating = FALSE;
   /* Indirection short-circuiting and elimination loop                    */
2  repeat
3      continueEvacuating = FALSE;
4      BlockDescriptor blockDescriptor = getBlockDescriptor(closure);
       /* Is the closure in a generation older than the oldest being
          collected?  If so, do not relocate it.                           */
5      if blockDescriptor → generationNumber > N then
           /* If the closure is also in a generation younger than the target
              evacuationGeneration then its referrer must be added to the
              immutable list                                               */
6          if blockDescriptor → generationNumber < evacuationGeneration then
7              failedToEvacuate = TRUE;
8          toSpaceReference = closure;
9          return toSpaceReference;
       /* Determine the next step into which the closure is to be tenured or
          promoted                                                         */
10     Step step = blockDescriptor → step;
11     if step → generationNumber < evacuationGeneration then
12         step = generations[evacuationGeneration].steps[0];
13     switch closure → infoTable.closureType do
14         case EVACUATED
               /* The closure has already been evacuated, return the
                  forwarding address.  If the evacuee resides in a generation
                  younger than the requested evacuationGeneration, then its
                  referrer must be added to the immutable list.            */
15             toSpaceReference = closure → evacuee;
16             if getBlockDescriptor(toSpaceReference)→ generationNumber <
               evacuationGeneration then
17                 failedToEvacuate = TRUE;
18         case INDIRECTION
               /* Short-circuit and eliminate indirections by attempting to
                  evacuate the indirectee                                  */
19             closure = closure → indirectee;
20             continueEvacuating = TRUE;
21             continue;
22         otherwise
               /* Copy the closure to the to-space step using its size and
                  layout information in its info table.  Then forward the
                  closure by installing the forwarding pointer.            */
23             toSpaceReference =
               copyAndForward(closure, closure → infoTable.layout, step → toSpace);
24 until continueEvacuating == FALSE ;
25 return toSpaceReference;
```

**Function** `updateWithIndirection`($closure, value$) GHC's thunk update operation that overwrites the heap object *closure* with an indirection to its WHNF *value*.

**Input**: *closure* Pointer to heap closure of type "thunk"
**Input**: *value* Pointer to the WHNF value resulting from evaluation of *closure*

```
   /* The following line and subsequent conditional implement the write
      barrier                                                          */
1  BlockDescriptor blockDescriptor = getBlockDescriptor(closure);
2  if blockDescriptor → generationNumber == 0 then
       /* Set the reference to the WHNF value in the same (young) generation
          */
3      closure → indirectee = value;
       /* Set the info pointer to the static info table for a young
          generation indirection                                       */
4      setInfo(closure, IND_info);
5  else
       /* Set the reference to the WHNF value in the young generation  */
6      closure → indirectee = value;
       /* Chain closure onto the immutable list of its older generation */
7      closure → mutableLink = generations[blockDescriptor →
       generationNumber].immutableList;
8      generations[blockDescriptor → generationNumber].immutableList = closure;
       /* Set the info pointer to the static info table for an old
          generation indirection                                       */
9      setInfo(closure, IND_OLDGEN_info);
```

## 6.2 Dynamic Dispatching Write Barriers

In principle, generational collection requires all write operations to inspect the status of the object being written: if the object is in an old generation then it must be added to the immutable list for that generation. However, this does not apply to objects in the youngest generation, as such writes cannot generate inter-generational references. The idea is to exploit the info table pointer within an object so that the barrier is applied only to objects in an older generation. The barrier overhead is then removed completely from objects in the youngest generation. The trick is to hijack the info table pointer associated with the object when the garbage collector promotes it to an older generation. We can then control what happens when the promoted object is entered for the first time and/or updated. We consider firstly two schemes that work by introducing "special" indirection closures that effect the behaviour we require before redirecting entry to the thunk itself. These incur a *dynamic* space overhead that we then trade for equivalent schemes based on "thunk specialisation" (Section 6.2.2 and Section 6.2.3). As well as eliminating the dynamic space overhead, these schemes are computationally more efficient; they do, however, incur *static* space overheads on the code generated.

### 6.2.1 Special Indirections

Recall, that when a thunk is updated with its WHNF value, if the value closure is larger than the size of the original thunk closure, then it must be overwritten by an indirection closure to the value. Furthermore, when an indirection closure is entered it simply jumps to the entry code of its indirectee (i.e. enters it).

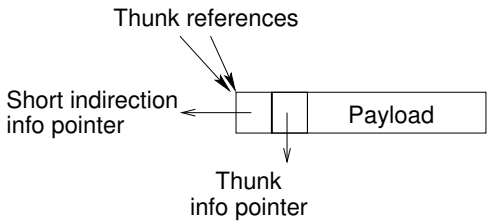



Figure 6.2: A thunk is indirected via a "short indirection" on promotion to an older generation

This is achieved by introducing several new "special" indirection closure types whose entry code (eventually) results in the threading of the closure onto the immutable list. We arrange for one of these special indirection closures to be "planted" in front of a thunk at the point where evacuation results in its promotion to an older generation. The forwarding pointer of the thunk's from-space copy is then installed and references this special indirection closure, as opposed to the normal case where it would directly reference the thunk. At the end of the collection cycle, all live references to the thunk now refer to it indirectly via this indirection closure. This is shown on Figure 6.2.

We optimise the space requirements of these special indirections. Normally, an indirection node in all but the youngest generation occupies three words: its info pointer, the indirectee reference, and the link field (in case it needs to be added to the immutable list). If special indirections also occupied three words we could potentially halve the occupancy of the old generation. However, we can reduce the space overhead to just one word by juxtaposing the special indirection to the thunk with which it is associated. This *short indirection* can never be updated, so its link field is superfluous. As it is now adjacent to its thunk its reference field is also superfluous. The entry code of the indirection always accesses, implicitly indirects to, and ultimately enters the closure adjacent to it (see Figure 6.2).

#### Indirected Add-on-entry

One way to preserve the generational invariant is to place the object on the immutable list as soon as it is entered. This is easily achieved using a special indirection that when entered immediately threads the closure onto the immutable list and then enters its indirectee thunk. We say that the closure 'goes on early' with the effect that the immutable list may contain references to closures whose updates have yet to happen. Such closures are technically in the

'black hole' state. The significance or otherwise of going on early depends on the amount of time the closure spends in this state.

How is the object added to the immutable list when it is entered? The usual trick is to use the space in the original thunk to store the immutable list entry. The thunk info pointer is overwritten with a reference to an "old generation indirection" info table and the second field is overwritten with a link to the immutable list. Rojemo's collector and the current GHC collector both do this. Other schemes employ a separate mutable *table*, for example [Sew92]. However, it does it when the object is updated, rather than when it is entered—we explore the latter option later on. Essentially, what we're doing here is performing a special type of *eager* black-holing, since adding the thunk to the immutable list has the side effect of marking it as having been entered (black-holing). In this scheme, the process of lazy black holing, that is performed by the collector when scavenging update frames, must be turned off for objects in the old generation. The experiments of [PJ92] report lazy black holing to be cheaper than black holing the thunks at the point where they are entered. This is not surprising since update frames that are created and popped before the next collection cycle will never have their updatee black-holed. The disadvantage is that for such thunks the very rare occurrence of an infinite loop will not immediately be detected and reported. The disabling of lazy black holing is part of the space/time trade-off associated with this scheme.

It turns out that adding the closure to the immutable list early using this particular technique is not such a good idea. Because the thunk reference must go on the immutable list early we need *two* extra words to hold the special indirection: one for the info pointer and one for the immutable list link field. Why can't we use the thunk info pointer, which the indirection "knows" how to find, to store the link field when the indirection is entered? This looks promising, and would reduce the overhead to one word. However, if a garbage collection happens during execution of the thunk entry code we have the problem that the thunk info pointer no longer points to thunk entry code. A garbage collection in GHC causes the object pointed to from the stack (the thunk in this case) to be *re-entered* when garbage collection has finished. Attempting to re-enter the thunk via its info pointer will therefore cause the program to crash.

#### Indirected Add-on-update

A seemingly preferable approach to the previous add-on-entry scheme arranges for a thunk to be added to the immutable list when it is updated, in keeping with conventional approaches such as [Sew92, SP93] and GHC's current generational collector.[3]

Again, when a thunk is promoted all references to it are directed at its special indirection, a *thunk indirection*, so that entering the thunk first involves entering the thunk indirection. The entry code for the thunk indirection pushes a *special update frame* on the stack, modifies its info pointer to a vanilla short indirection, whose entry code simply enters the thunk adjacent

[3]Note that after the update the thunk is actually a value or, more typically, an indirection to a value.

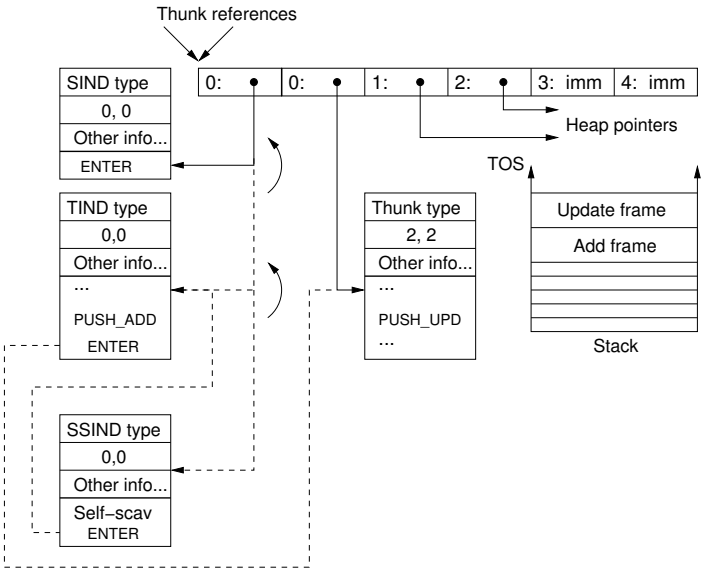



Figure 6.3: Indirection Nodes: Add-on-Update. A single-word object is juxtaposed with a copy of the original thunk. This flips from a special thunk indirection (TIND) to a vanilla short indirection (SIND) after being entered. It seamlessly integrates with Non-stop Haskell's self-scavenging object (SSIND).

to it, and then enters the thunk's standard entry code. The idea is to get the special update frame to add the closure it references to the immutable list. Because of its location in the stack this will happen *after* the thunk has been updated by its own standard update frame in the usual way.

Figure 6.3 shows the situation at the point where the thunk's entry code has just completed execution.

Once the thunk has been updated by the update frame on top of the stack, control passes to the entry code of the `Add` frame which sits beneath it. This adds the thunk (now an indirection) to the immutable list. This situation is shown in the top part of Figure 6.4 at the point where the thunk has just been updated (top) and after the `Add` update frame has extended the immutable list. Furthermore, the figure illustrates the important optimisation of the short indirection closures.

Importantly, note that in-place updates, which are rare anyway, must be turned off with this scheme so that thunks are *always* updated with indirections. Were an in-place update allowed to proceed in an older generation, then the operation of threading the closure onto the immutable list by the popping of the `Add` frame would result in the corruption of the value closure.

Referring back to Figure 6.3 notice that the extra field juxtaposed with the thunk assumes several different roles during its lifetime. Two of the roles have already been described, namely

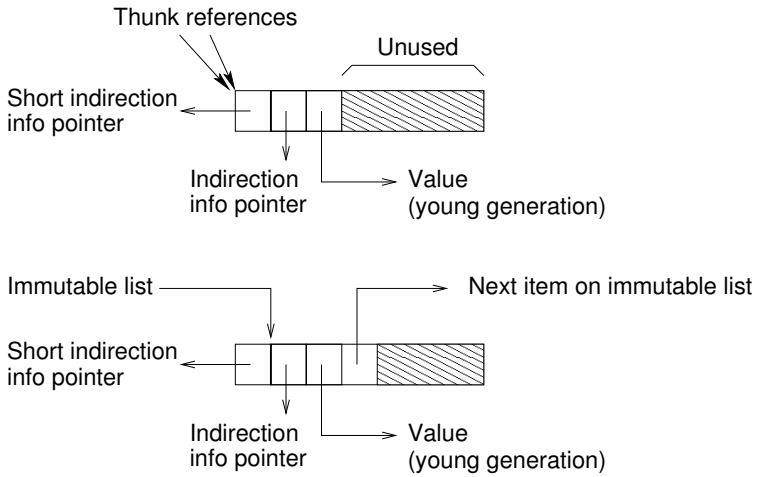



Figure 6.4: Adding an old generation closure to the immutable list via the `Add` frame.

that of the thunk indirection (with closure type `TIND`) and that of the short indirection (with closure type `SIND`). These are both shown in the figure; the dotted arrow shows the info pointer as it was before entering the thunk indirection, and the vertical arrows show the transition sequence between each info table. The third role employs a further special indirection closure, a *self-scavenging thunk indirection*, that realises our goal of a barrierless incremental generational collector. When the thunk is evacuated on promotion into an older generation, the self-scavenging indirection (with closure type `SSIND`) is installed as the special indirection instead of the thunk indirection. If the collector is the first to encounter this indirection closure and attempts to scavenge it before the mutator, then it simply modifies the info pointer from the `SSIND` info table to that of the `TIND` info table. Atomically with this operation, its associated thunk, that is adjacent to it, is also scavenged. Alternatively, if the mutator encounters the indirection closure first by entering it, the entry code of the self-scavenging indirection is executed. The entry code scavenges the adjacent thunk, and jumps directly to the entry code of the `TIND` that installs the vanilla short indirection closure and pushes the `Add` frame onto the stack. The self-scavenging thunk indirection therefore simultaneously enforces both the to-space and generational invariants, incurring no further dynamic space overhead on our Non-stop Haskell scheme. The special indirection word uses the space previously reserved for Non-stop Haskell's hidden Word −1 field. For all closure types other than these old generation thunks, the hidden Word −1 field remains in use in to-space for the copy of the original info table pointer that is hijacked by that of the self-scavenging closure.

It is important to note that a separate info table is required for each of the three indirection types, but that they themselves are then shared among all updateable closures.

### 6.2.2 Thunk Specialised Add-on-Entry

The extra to-space word overhead enables a simple technique for incremental and generational barrier optimisation and has assisted our exploration of the design space for various collector implementations. However, in a system where the minimum heap-allocated object size is 2 words, and on average between 3 and 4 words, this overhead may reduce the heap occupancy by between 33% and 50%. Over the remainder of the dissertation, we now concentrate on techniques that eliminate this dynamic space overhead. The techniques do this by trading it for the static space overhead that results from the 'specialisation' of a closure's static info table (i.e. a duplicated variant). We first explore this for the 'add-on-entry' scheme.

Recall that this scheme enforces the generational invariant by placing the thunk on the immutable list as soon as it is entered. Computationally the most efficient way to do this is to use a separate static info table for each thunk and for each generation. We say that the thunks are 'specialised' for each generation. The version for the old generation adds the object to the immutable list and then executes the standard entry code to evaluate the object. The young generation version just does the latter.

This is precisely the scheme described by Rojemo [R95] in his study of space leaks in the Chalmers LML/Haskell runtime system.

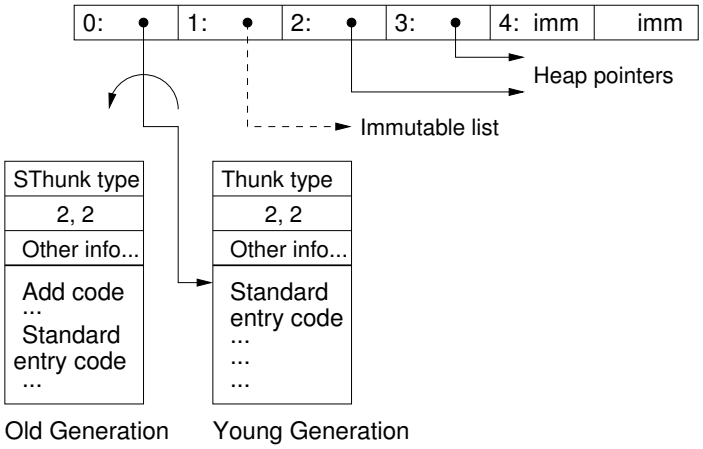



Figure 6.5: Specialised Thunks: Add-on-entry. When a thunk is promoted to the old generation, the info table pointer is flipped to point to entry code that adds the thunk to the immutable list when entered.

The scheme is illustrated in Figure 6.5. The info pointer shown is flipped when the thunk is promoted to the old generation (of a young and single old generation configuration). The entry code for the old generation first adds a reference to the thunk to the immutable list (`Add code`) and then executes the standard entry code for the thunk. Note that the scheme can be adapted trivially for multiple generations.

Again, placing the thunk onto the immutable list early suffers the same problem as the

Indirected Add-on-entry scheme discussed above, corrupting the thunk's payload if the thunk is ever re-entered. As a result old generation thunks must now incur an extra one word overhead that is reserved for use as the immutable list link field. This is an improvement on the two word overhead of the Indirected Add-on-entry scheme, but we can do much much better!

### 6.2.3 Thunk Specialised Add-on-Update

Using both thunk and update frame specialisation we can eliminate the dynamic space overhead incurred by the thunk, and, at the same time, ensure that the closure is threaded onto the immutable list at exactly the right time — when it is updated. The trick is to use a specialised update frame for thunks in the old generation, that "knows" which immutable list the closure must be threaded onto. Young generation update frames simply perform the update without performing the write barrier test. Similarly, old generation update frames simply thread the updatee onto the appropriate immutable list (without performing a generation test) at the time at which it is updated to an old generation indirection.

In order for the correct specialisation of the update frame to be pushed, thunks must also be specialised according to their generation. The thunk entry code will push a *special update frame* (`PUSH_SUPD`) when the thunk is in the old generation and a normal update frame (`PUSH_UPD`) when in the young generation. A special update frame overwrites the thunk with the old generation indirection as described above. This is illustrated in Figure 6.2.3.

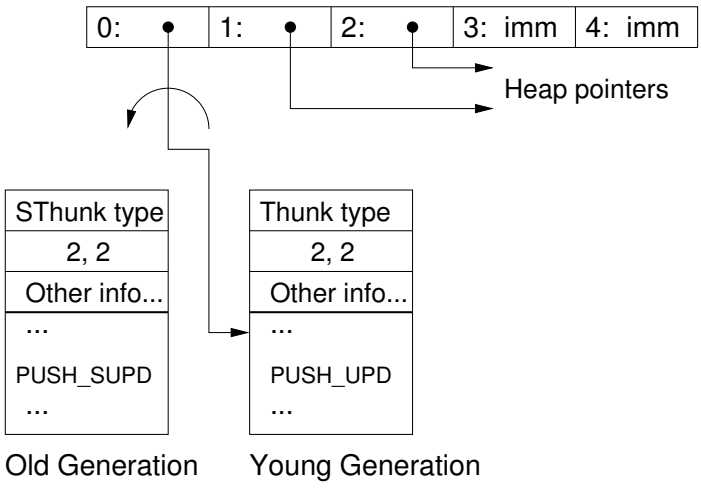



Figure 6.6: Specialised Thunks: Add-on-update. When a thunk is promoted to the old generation, the info table pointer is flipped to point to entry code that pushes a special update frame instead of a standard one.

## 6.3 Summary and Conclusions

We have presented four schemes that enable the implementation of a generational copying collector that completely eliminates the write barrier test on updates to young generation objects. While appearing simple to implement, the "add-on-entry" mechanisms highlight the complications and inefficiency of Rojemo's scheme. The remaining two place the referring old generation closure onto the immutable list at both the computationally "correct" and most efficient point of execution — when the write that creates the inter-generation pointer occurs.

The first "add-on-update" scheme employs various "special" indirections, the most important of which, ensures that the old generation closure is threaded onto the immutable list. The scheme incurs two dynamic space overheads — one word for each old generation thunk in the heap, and an extra stack frame in addition to each old generation thunk update frame, that threads the closure onto the immutable list. It is worth noting that the push operation that stack allocates the frame and the corresponding pop operation are exceptionally fast and the overhead they incur is far less than that incurred by the combination of the extra heap word and the additional immutable list management that results when old generation thunk closures are threaded on early. Furthermore, this mechanism is easily and efficiently extended to integrate with Non-stop Haskell to produce a barrierless incremental generational copying collector.

The "specialised add-on-update" scheme specialises every thunk for every generation in which it can reside. This specialised thunk entry code pushes an update frame that is also specialised for and shared by all thunks that may reside in that generation. Encoded within the update frame is the address of the immutable list for the generation onto which the updated thunk should be threaded. This scheme incurs no dynamic space overhead, either within the heap or the thread stack. However, the static code "bloat" — the overhead incurred by specialisation of each thunk's static info table, when required to support `N` generations, can be severe. This bloat can however, be limited by providing a single "old generation" specialisation for each thunk, which must perform the generation test to determine the immutable list target. The user is then free to trade the size of their application binary against its performance when the system is first compiled. It is worth noting that this scheme can be seamlessly integrated with our Non-stop Haskell collector to enable a barrierless incremental generational variant.

Over the course of the following chapter we describe the implementation of our production barrierless incremental generational collector that eliminates the overhead that the extra word incurs. We do this, again, through closure specialisation. The combined implementation of specialised self-scavenging thunk closures with their "specialised add-on-update" counterparts. This, our preferred scheme is evaluated in Section 8.6.

# Chapter 7

# Real-time Haskell

In Chapter 5 we described a low-cost mechanism for supporting the read barrier in GHC based on augmenting copied objects with an extra word which retains a reference to the original object entry code after the entry code pointer has been hijacked. This extra word comes at a price: in addition to the overhead of managing the extra word it also influences the way memory is used. For example, in a fixed-size heap a space overhead reduces the amount of memory available for new object allocation which reduces the average time between garbage collections and hence increases the average number of collections.

In this chapter we develop an alternative approach that avoids the one word overhead of our previous scheme. We do this by building, at compile time, *specialised* entry code for each object type that comes in two flavours: one that executes normal object entry code and one that first scavenges the object and then executes its entry code. This eliminates the need for the extra word as we can simply flip from one to the other when the object is copied during garbage collection. The cost is an increase in the size of the static program code (code bloat).

A key objective of this chapter is to detail how this code specialisation can be made to work in practice (see Section 7.1.1) both for single and multiple generation collectors. Although the idea is simple in principle it interacts in various subtle ways with GHC. We also incorporate a time-based scheduler, as an alternative to the previous work-based strategies, in order to develop a true (soft) real-time collector.

The following section describes the design of our incremental generational collectors and time-based extensions for real-time scheduling. The remaining sections detail their implementation paying particular attention to each component that must be modified, specifically, the compiler itself, the evaluation model, and the runtime system, its scheduler, storage manager and garbage collector interface. For reference, the (unmodified) architecture of each of these components is documented in Appendix A.

## 7.1 Design

Over the following section we describe the design of our incremental work-based and time-based collectors that employ closure specialisation within GHC.

Note that while the allocation layers of the Storage Manager need no significant modification to support incremental garbage collection we rewrote much of the Block Allocator to fix a major flaw in its coalescing algorithm, implemented by the `freeGroup()` API — the freeing of each block had a worst-case complexity of $O(N^2)$, where $N$ is the length of the free list, which we have reimplemented in constant time, $O(1)$[1].

### 7.1.1 Object Specialisation

We now show how the dynamic space overhead can be eliminated using object specialisation. This gives us the opportunity to evaluate the effect that dynamic space overheads have on program execution time in general (see Section 8.2).

The idea is simple: instead of employing generic code to perform actions like self-scavenging and using additional space in the heap to remember the object's original entry code, we instead modify the compiler so that for each closure type, in addition to generating its usual info table and entry code, we build one or more "specialised" versions that modify the object's usual behaviour when entered.

For each closure type we generate one additional variant. This inlines generic "self-scavenging" code with a duplicate copy of the standard entry code (but see Section 7.1.2 below). Info tables are extended with an extra field to contain a reference to their partner, i.e. the standard info table contains a reference to the self-scavenging info table and vice versa. The scheme is identical to the original, but now instead of copying the standard info pointer to Word –1 on evacuation, we simply replace it with its partner's (i.e. self-scavenging) info pointer. This obviates the need for Word –1. On scavenging, we do the reverse, thus restoring the standard info pointer. Notice that the static data that sits above the object's entry code must carry a pointer to the object's partner in order to facilitate this flipping of code pointers.

This new scheme is pictured in Figure 7.1. This scheme has replaced the run-time space overhead with a slight increase in compilation time and code space; both info table and entry code are stored in the text segment.

Note that the scheme described in Section 5.1.5 for the handling of large closures uses specialisation to eliminate the additional word overhead exactly as described above. This results in the generation of two info table and entry code specialisations, one for each "partially-evacuated" and "partially-scavenged" variant.

[1] The reworked allocator now features in all releases of the GHC runtime system subsequent to Version 6.4.

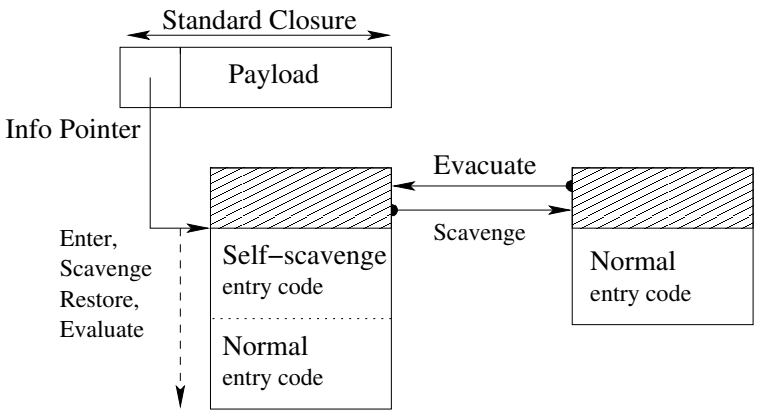



Figure 7.1: Object Specialisation

### 7.1.2 Entry Code Duplication vs Inlining

In Figure 7.1 the entry code associated with each info table partner is shown with an identical *inlined* copy of the closure's original entry code. Of course, we could replace one of these by a direct jump to the other, thus reducing the space overhead at the expense of additional instruction execution (JUMP). Although we do not expect much difference in execution time we evaluate both variants of this mechanism in Section 8.

### 7.1.3 Implications of Eval/Apply

In GHC 6.02 the *eval/apply* evaluation model has been adopted because, in terms of both performance (on average, 2–3% better), and the ease of its implementation and maintainability it has been found to be the superior evaluation model [MJ04].

In eval/apply closures representing function applications operate in a slightly different way to thunks. Instead of blindly entering the function's entry code an explicit "apply" function first evaluates the function (recall that Haskell is higher-order) and then inspects the result to determine the function's arity. At this point it determines whether an exact call to the function can be made or whether a partial application must be constructed.

This mechanism is different to previous implementations of GHC, and has important implications for our incremental garbage collector. A detailed description of the differences between the push/enter and the eval/apply evaluation models can be found in Appendix Section A.2.

Because "apply" is a generic operation, it must explicitly inspect the type tag of the closure and its arity. From the point of view of the garbage collector it is at this point that we must ensure the closure has been scavenged, in order to maintain the to-space invariant. The key difference between GHC's previous push/enter model, and eval/apply is that in the latter functions are *not* actually "self"-scavenging in the above sense since the scavenging is

performed by the apply function based on the object's type tag. For sure, we could make them so and simply arrange for the apply function to enter self-scavenging code, but there is nothing to be gained.

### 7.1.4 Incremental Stack Scavenging

Baker's original scheme focuses primarily on pointer dereferences for heap objects. The stack, however, is also subject to the to-space invariant, both from perspectives of correctness and incrementality — the pointers it contains form part of the root set. The simplest mechanism that enforces the former is to scavenge the entire stack at the start of the collection cycle, but this leads to an unbounded pause. The alternative, is to place a read barrier on stack pop operations, which adds further significant overhead. In Chapter 5 we devised a cheap means of scavenging the stack incrementally, this time hijacking the return addresses in update frames which are interspersed among the regular frames on the stack. These update frames have a fixed return address to common update code in the runtime system. We scavenge all the stack frames between one update frame and the one below, replacing the return address in the latter with a self-scavenging return address. This self-scavenging code scavenges the next group of frames, before jumping to the normal, generic, update code.

Since update frames could, in principle, be far apart, pause times could be long, although this case is rare in practice.

One of the consequences of the eval/apply model is that the stack now consists solely of stack frames. The key point is that the `layout` field of each stack frame's info table provides the meta information that enables the scavenger to sequentially traverse the stack. In the push/enter model the stack is also littered with pending argument sections (arguments pushed in preparation for a function call). Using eval/apply it is therefore possible to "walk" the stack *cheaply* one frame at a time, something that is not possible in the push/enter scheme.

In our implementation of Non-stop Haskell we could not scavenge the stack one frame at a time; the only stack object whose return address we could safely hijack was the next update frame on the stack. Although updates in GHC are quite common the amount of work required to scavenge between adjacent update frames (a "unit" of stack scavenging work) was impossible to bound. In the eval/apply scheme the maximum amount of work is determined by the maximum allowable stack frame, which is fixed in GHC at 1KB. This provides a hard bound on the time taken to perform one "unit" of stack scavenging work.

Given that the stack really is just like a collection of closures do we need to specialise the entry code for these stack frames in order to make the stack scavenging incremental? No, because the stack is accessed *linearly* from the top down[2]. To hijack the return address of the frame below we just need a single extra word (e.g. a register) to remember the original info pointer of the frame that is the latest to be hijacked. We cannot do the same in the heap as

[2]Unfortunately, however, we are forced to specialise unboxed tuple return addresses, as described in Section 7.4.3

the heap objects are subject to random access.

### 7.1.5 Slop Objects

When a closure is updated it is (always) replaced by an indirection to the closure's value. Because the indirection may be smaller than the original closure, the scavenger has to be able to cope with the remaining "slop", i.e. the now unused memory within the original closure. In short, the scavenger must know how to skip this dead space in order to arrive at the next valid object in the collector queue.

Note, that for the stop-the-world collector this issue of slop does not exist — the scavenger does not scavenge from-space where slop exists. Furthermore, the collection cycle is instantaneous with respect to mutator execution, and so it is not possible for an evacuated object to be updated before it is processed by the scavenger.

If we use the extra word, the scavenger can always determine the size of the original closure since its info table will be accessible from the object's original info pointer at Word –1. With object specialisation, this information is lost: when the scavenger has processed the updated version of the object it has no idea whether there is any dead space following the object, let alone how much. We solve this problem by creating a special "dummy" object, a *slop object*, which looks to the scavenger like a uniform closure whose size is that of the dead space created by the update and whose payload contains no pointer fields. When the scavenger encounters a slop object the effect is to skip immediately to the next object in the collector queue. This is illustrated in Figure 7.2.

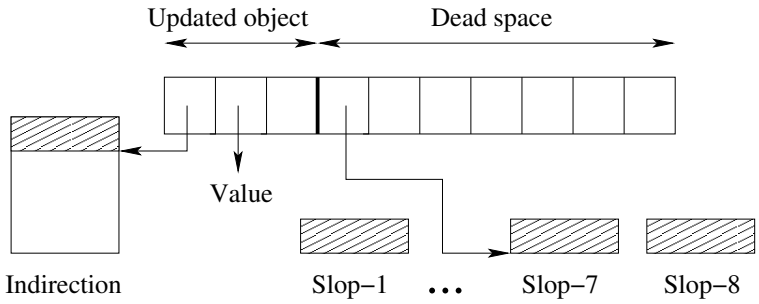



Figure 7.2: Slopping in to-space.

We do not want to build slop objects on the fly, so to make "slopping" fast for the vast majority of cases we pre-define eight special slop objects, one for each slop size from 1 to 8. A generic slop object is constructed dynamically for all other cases whose payload contains the slop size. This is more expensive but it very rarely required (we did not observe it at all in the benchmarks we used for evaluation). Note that slop objects will never be entered, and only their layout information is ever used, so the entry code for each is null. Figure 7.2 shows the situation after the object's update code has overwritten the original object with an

indirection and the remaining dead space with a pointer to one of the eight predefined slop objects — the seventh in this case as the dead space comprises seven words.

Recall that there is an inherent delay between a closure being entered for the first time and the closure being updated with its value. In a single-thread model, an attempt to enter a closure that has already been entered but not yet updated results in an infinite loop. To trap this, objects are marked as *black holes* when they are entered. Note that whilst initially appearing attractive, it is not possible for the thunk's update frame to perform the slopping by arranging for the deployment of a combined indirection and slop object — if the black hole is tenured before the the update occurs, the slop is eliminated, but when the indirection is subsequently deployed, its slop-filling payload will erroneously corrupt the heap.

In the case of the scheme depicted in Figure 7.1, where the entry code is inlined, we can perform a further optimisation. Recall that slop objects are only required in to-space, since this is the only place the scavenger visits. We can avoid slopping altogether in from-space by suitably specialising the entry code for objects after they have been evacuated. The appropriate slop filling code is generated and appended to the end of the duplicated (and inlined) entry code for self-scavenging thunks which are at risk of being entered and updated before the scavenger visits them. However, because slopping in our implementation is very fast — a single word assignment in almost all cases — the performance benefits of such specialisation turns out to be negligible (see Section 8).

All the self-scavenging collectors are able to determine that a closure has already been scavenged – they are implicitly marked based on their type. For collectors that allow mutations in to-space during the scavenging process, i.e. those that impose a to-space invariant or those with a relaxed from-space invariant, there is no way to distinguish whether a to-space closure has already been scavenged. Instead, they must test each reference of a to-space closure's payload and determine whether or not it must be evacuated. The Baker collector, and our Brooks collector and replicating collectors (see Section 7.5), all suffer this inefficiency. Note that the Baker implementation in Non-stop Haskell used the extra word.

### 7.1.6 Fast Entry

The crucial property on which our scheme depends is that every object is entered before it is used. However, GHC sometimes short-circuits this behaviour.

Consider this function (taken from [CFS+04]):

$$\texttt{f x} = \texttt{let } \{\texttt{g y} = \texttt{x} + \texttt{y}\} \texttt{ in } (\texttt{g 3}, \texttt{ g 4})$$

The function `g` will be represented by a dynamically-allocated function closure, capturing its free variable `x`. At the call sites, GHC "knows" statically what code will be executed, so instead of entering the closure for `g` via the slow entry point, dereferenced from the `Node` register, it simply loads a pointer to `g` into a register, and the argument (3 or 4) into another, and *jumps directly* to `g`'s fast entry point.

For our incremental collectors this optimisation for known function evaluation fails because we must be sure to scavenge `g`'s closure before using it. The evaluator cannot therefore just blindly jump to the function's fast entry point in case the self-scavenging code is pending execution. In short, closure entry must always proceed with an indirect jump to the entry code address obtained from dereferencing the `Node` register. The simple solution therefore, which we adopt, is to turn off the optimisation. It turns out (as we show in Section 8) that this has a performance cost of less than 2%.

Notice that this only applies for dynamically-allocated function closures. For example, the function `f` also has a closure, but it is allocated statically, and captures no free variables. Hence it does not need to be scavenged, so calls to `f` can still be optimised into direct jumps.

### 7.1.7 Scheduling for Real-time

GHC's block-allocated storage manager, as described in Appendix Section A.4.2, enables us to explore two levels of granularity at which to perform work-based incremental scavenging during garbage collection. The first invokes the scavenger at each *object* allocation — the object allocation code is inlined with additional code which first tests whether the collector is on and then, if it is, invokes the scavenger before allocating space for the object. The second plants the same code in the runtime system at the point where a new memory *block* (with a default size of 4 KB) is allocated. By scavenging at each block allocation the mutator does more work between pauses at the expense of increased pause times (the scavenger also has to do more work to keep up).

In Section 3.3 we discussed the issues surrounding real-time collector scheduling. In particular, we recognised that while work-based collection is adequate for interactive environments, the effects of unpredictable allocation rates make it less so for (soft) real-time systems. A more appropriate strategy is that of time-based scheduling introduced in Section 3.3.3. We incorporate such a strategy into our implementation by leveraging the timer-based functionality that implements the sequencing of thread time slices within GHC's lightweight scheduler (see Section A.4.1) — the context switch period of a thread is configured with the desired mutator time quantum. When a thread returns to the scheduler, the scheduler tests whether the mutator quantum has actually expired and whether the collector is enabled and if so, runs the collector for its time quantum.

In Chapter 8 we evaluate both these work-based and time-based schemes. The remaining sections of this chapter describe the implementation details of our incremental work-based and time-based collectors and the modifications that must be made to the compiler, and the runtime system's scheduler and garbage collector.

## 7.2 The Compiler

The majority of modifications made to the compiler, which is itself written in Haskell, are concerned with:

### 7.2.1 Compiler pipeline

Two additional command-line options are added to the compiler pipeline. The first, `fill-heap-slop`, ensures that compiler generates the slopping code for each updateable thunk. The second, `incremental-gc`, forces the compiler to generate the specialised self-scavenging variant of each closure. Certain features of GHC require certain parts of the compiler, as well as its libraries, to have been built in an associated *way*. Each "way" affects the compiler pipeline differently, and so specifies which options the feature requires, and with which other options it can and cannot generate code and execute alongside. Shared libraries are built for each "way" that encapsulate its specific code and are located by the compiler for subsequent linking with user-compiled code by a naming convention that encodes the "way" name. Various "ways" exist, such as those for debugging, profiling, the multi-threaded runtime system (RTS), the parallel environment, etc. We add an additional "way" that ensures the necessary parts of the compiler, the runtime system, the prelude and standard libraries are all consistently built with both the above options enabled thus generating the required support code for the incremental collector to be enabled.

### 7.2.2 Eval/Apply Code Generator

Within GHC's implementation of the eval/apply evaluator the return addresses for the most common call continuation code sequences are pre-generated (as `stgApply` functions) and built into the runtime system, each of which represents a call sequence for $1..n$ arguments. If more then $n$ arguments are required, a sequence of call continuations are pushed. The generic apply functions must be modified to handle each closure type's self-scavenging specialisation. To achieve this, the eval/apply code generation module is modified to generate C code that recognises the closure types of the specialisations and dispatches immediately to the `MutatorScavenge()` API that scavenges the closure and falls through to the generic apply code for the vanilla closure type. Note that this is not the same as inlining an explicit read barrier since no additional overhead is incurred due to additional runtime tests — there is still only one tag test applied. Listing 7.1 shows an example fragment of the `stgApply` function generated for generic application of a closure to a single pointer argument. The code enclosed by the conditional compilation tag `INCR_GC` is the additionally generated code that handles unscavenged closures in to-space, residing in their self-scavenging form. The implementation of `MutatorScavenge()` is described in detail in Section 7.4.4.

```
F_ stg_ap_p_ret( void )
{
    StgInfoTable *info;
    nat arity;

  again:
    info = get_itbl(R1.cl);
    switch (info->type) {
        #ifdef INCR_GC
        case FUN_SELF_SCAV:
        case FUN_STATIC_SELF_SCAV:
            CALLER_SAVE_R1
            STGCALL1(MutatorScavenge,R1.p);
            CALLER_RESTORE_R1
        #endif
        case FUN:
        case FUN_STATIC:
            arity = itbl_to_fun_itbl(info)->arity;
            ASSERT(arity > 0);
            if (arity == 1) {
                Sp += 1;
                JMP_(GET_ENTRY(R1.cl));
            } else {
                BUILD_PAP(1,1,(W_)&stg_ap_p_info);
            }

        #ifdef INCR_GC
        case PAP_SELF_SCAV:
            CALLER_SAVE_R1
            STGCALL1(MutatorScavenge,R1.p);
            CALLER_RESTORE_R1
        #endif
        case PAP:
            arity = ((StgPAP *)R1.p)->arity;
            ASSERT(arity > 0);
            if (arity == 1) {
                Sp += 1;
                R2.w = (W_)&stg_ap_p_info;
                JMP_(stg_PAP_entry);
            } else {
                NEW_PAP(1,1,(W_)&stg_ap_p_info);
            }

        #ifdef INCR_GC
        case AP_SELF_SCAV:
        case AP_STACK_SELF_SCAV:
        case BLACKHOLE_BQ_SELF_SCAV:
        case THUNK_SELF_SCAV:
        case THUNK_STATIC_SELF_SCAV:
        case THUNK_SELECTOR_SELF_SCAV:
            CALLER_SAVE_R1
            STGCALL1(MutatorScavenge,R1.p);
            CALLER_RESTORE_R1
        #endif
        case AP:
        case AP_STACK:
        case CAF_BLACKHOLE:
        case BLACKHOLE:
        case BLACKHOLE_BQ:
        case SE_BLACKHOLE:
        case SE_CAF_BLACKHOLE:
        case THUNK:
        case THUNK_STATIC:
        case THUNK_SELECTOR:
            Sp[0] = (W_)&stg_ap_p_info;
            JMP_(GET_ENTRY(R1.cl));

        case IND:
        case IND_OLDGEN:
        case IND_STATIC:
            R1.cl = ((StgInd *)R1.p)->indirectee;
            goto again;
        }
}
```

Listing 7.1: Self-scavenging `stgApply` function fragment for application of a single pointer argument.

### 7.2.3 Closure Entry

All places within the compiler that generate entry code for *dynamic closures* that jump directly to the fast-entry point must be identified and disabled so as to force entry via the slow-entry point that loads the `Node` register and indirectly jumps to the dereferenced info pointer address.

Haskell threads cannot be actively preempted; instead they yield to the scheduler, at its request, through the periodic checking of a global preemption flag. The machine code evaluator only checks the preemption flag on failure of a closure's *heap check* or *stack check*, the point at which the thread must return to the scheduler to allow memory management activities to occur. Failure of a heap check indicates that the current nursery block (4KB in size) is full and the bump pointer must be moved to the next block in the chain or that the nursery is full and that the garbage collector must be triggered. This 'preemption' strategy is simple and limits the code size because the check occurs from within the generic shared nursery extension code, but the downside is that if a thread performs little allocation then it may exceed its allotted time-slice. For the time-based collectors the mutator must be made more responsive to preemption requests so as to avoid the mutator overrunning its time quantum.

The first option available is to modify the machine code evaluator to check the preemption flag whenever a closure is entered. The heap check macro would need to be modified to perform a fast in-band test of the preemption flag on the fast path branch where the heap check succeeds. The result is that only those closures that perform a heap check, i.e. perform heap allocation, now *always* check the flag. In general very few closures omit the heap check and so trading potential preemption accuracy to reduce its overhead and avoid the test being performed twice is worthwhile.

The alternative option is to arrange for the next heap check that occurs to fail when the preemption flag is set — the heap check's slow path already checks the preemption flag and yields to the scheduler if necessary. This is significantly cheaper than the previous option since there is no overhead on the mutator when the collector is off or when the mutator quantum has yet to expire. The details of this approach are discussed in Section 7.3.

### 7.2.4 Self-scavenging Info Table Generation

The following structures define the static info tables, and the uniform layout of *all* closures:

```
typedef struct _StgInfoTable {
    struct _StgInfoTable *selfscav_infoptr;
    StgClosureInfo       layout;            /* Closure layout info (one word) */
    StgHalfWord          type;              /* Closure type */
    StgHalfWord          srt_bitmap;        /* Number of entries in SRT */
    StgCode              code[0];           /* Start of entry code */
```

```
} StgInfoTable;

typedef struct {
    const struct _StgInfoTable* info;
    ...
    /* Optional header structures based on the compiler way */
    ...
} StgHeader;

typedef struct StgClosure_ {
    StgHeader   header;
    struct StgClosure_ *payload[0]; /* Start of payload  array*/
} StgClosure;
```

The `StgClosure` structure defines the layout of a closure and incorporates the standard (fixed size) header and a pointer to the start of the object's payload. The standard closure header is defined by the structure `StgHeader` and contains a pointer to the info table of the object and optional structures that are defined based on the current compiler "way" (see Section 7.2.1). The `StgInfoTable` structure describes the portion of an info table that is common to all closure types.

Listing 7.2 outlines a simplified version of the info table generation macro for a generic apply function. Generating the self-scavenging variants of each info table is fairly straightforward. For info tables generated by the runtime system, these code generation macros simply generate each table twice, sharing much of the contents, but with the labels of the `info` table, `entry` code, and closure `type` fields of the self-scavenging variant receiving an identical value to which the `SELF_SCAV` identifier is augmented. The collector's scavenger distinguishes scavenged from unscavenged closures based on the `type` field of the closure's info table. The addresses of the `info` table labels are then written into the `selfscav_info` pointer fields of the partnering info tables. Each info table can now reference and even access its partner via the pointer value of its `selfscav_info` field. The entry code for the two info tables is then generated at the address represented by the `entry` code label. Where info tables are generated in Haskell by the compiler, the code generator incorporates the equivalent modifications.

When the mutator enters a self-scavenging closure it executes the `stg_SELF_SCAV_##entry` code, saving register `R1` to prevent it from corruption as it calls out to runtime system code, invokes the collector's `MutatorScavenge()` routine, with `R1`, the address of the closure, passed as an argument, then restores `R1` before reentering the closure jumping to its standard entry code or falling through to the inlined entry code in the case where it is duplicated and inlined. Note that when generating the code to re-enter the closure we can be more optimal than generating a runtime dereference of, and indirect jump via, the `Node` register, since the address of the partner's entry code, to which we must jump, is known at compile-time. The

```
INFO_TABLE_FUN_GEN(info,                                    /* Info table label */
                   entry,                                   /* Entry code label */
                   ptrs, nptrs,                             /* Closure layout info */
                   srt_, srt_off_, srt_bitmap_,             /* SRT info */
                   fun_type_, arity_, bitmap_, slow_apply_, /* Function info */
                   type_,                                   /* Closure type */
                   info_class, entry_class,                 /* C storage classes */
                   prof_descr, prof_type)                   /* Profiling info */
    entry_class(stg_SELF_SCAV_##entry);
    entry_class(entry);
    ED_RO_ StgFunInfoTable info;
    info_class const StgFunInfoTable stg_SELF_SCAV_##info = {
        i : {
            layout : { payload : {ptrs,nptrs} },
            srt_bitmap : srt_bitmap_,
            type : type_##_SELF_SCAV,
            selfscav_infoptr : (StgInfoTable *)(&(info))
            INIT_ENTRY(stg_SELF_SCAV_##entry)
        },
        srt : (StgSRT *)((StgClosure **)srt_+srt_off_),
        arity : arity_,
        fun_type : fun_type_,
        bitmap : (W_)bitmap_,
        slow_apply : slow_apply_
    };
    StgFunPtr stg_SELF_SCAV_##entry(void) {
        FB_
            CALLER_SAVE_R1
            STGCALL1(MutatorScavenge,R1.p);
            CALLER_RESTORE_R1
            JMP_(entry);
        FE_
    };
    info_class const StgFunInfoTable info = {
        i : {
            layout : { payload : {ptrs,nptrs} },
            srt_bitmap : srt_bitmap_,
            type : type_,
            selfscav_infoptr : (StgInfoTable *)(&(stg_SELF_SCAV_##info))
            INIT_ENTRY(entry)
        },
        srt : (StgSRT *)((StgClosure **)srt_+srt_off_),
        arity : arity_,
        fun_type : fun_type_,
        bitmap : (W_)bitmap_,
        slow_apply : slow_apply_
    }
```

Listing 7.2: The macro for code generation of a self-scavenging generic apply function.

scavenging operations of the `MutatorScavenge` routine are explained in Section 7.4.4.

Similar modifications are required for the 'barrierless' collector implementations that, in the addition to the above, incorporate the "specialised add-on-update" scheme of Section 6.2.3. Figure 7.3 shows a thunk and its four specialised variants. The example shows the info pointer set as it would be after promotion, before the thunk has been updated and after it has been scavenged during garbage collection. The diagram would be the same for a promoted thunk when the garbage collector is off. To complete the picture, self-scavenging variants of both the original entry code and thunk barrier entry code are needed as before: a total of four specialised thunk variants.

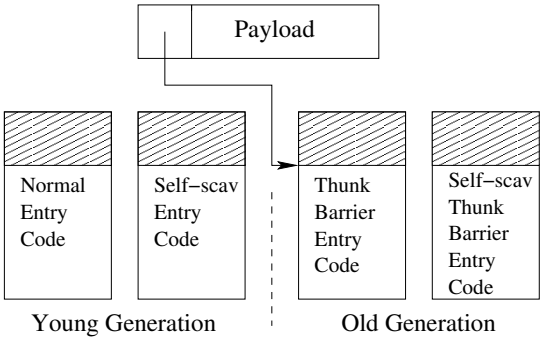



Figure 7.3: An evacuated closure in the specialised self-scavenging thunk barrier scheme.

### 7.2.5 Slopping

As described in Section 7.1.5 the scavenger "recognises" slop objects by type. Therefore, slop objects need only deploy the `StgHeader` word which incorporates their associated info table. `StgSlopObjectSpecific` define the eight pre-built slop objects, while `StgSlopObjectGeneric` defines the slop object that fills nine words of slop or more, the amount of which is encode by its `n_slop_words` field:

```
typedef struct {
    StgHeader   header;
} StgSlopObjectSpecific;

typedef struct {
    StgHeader   header;
    StgWord     n_slop_words;
} StgSlopObjectGeneric;
```

Those modules of the code generator responsible for generating i) closure update, and ii) eager black-holing code, are modified to incorporate the generation of slop-filling code that deploys the appropriate slop object when evaluation of the thunk is initiated, and if enabled, in conjunction with its eager black-holing. The main complexity that arises in the implementation is in ensuring that this generated code: i) does not clobber the thunk's payload until it is safe to do so; ii) is sequential and atomic; and iii) cannot be subjected to the statement reordering optimisations of subsequent compilation phases. Care must also be taken to ensure that the allocated registers do not conflict, and that they don't invalidate subsequent register allocation optimisations.

## 7.3 The Scheduler

The primary way in which a collection cycle is initiated is when a running Haskell thread returns to the scheduler on failure of a heap check (the `Hp` register meets the `HpLim` register) where the blocks available for chaining onto the nursery have been exhausted. The collector is invoked by the `GarbageCollect` API (see Section A.4.3).

The code that handles the heap overflow condition in the scheduler is modified to check the `gc_in_progress` flag to determine if the collector is already running. If it is, then the heap overflow has occurred during an incremental collection cycle. The heap size is calculated using a dynamic sizing policy based on the amount of live data at the end of the previous collection cycle (see Section 6.1.6). Since this is an approximation of the live data for the current cycle, it is often too conservative. Furthermore, bursty allocation within a work-based environment results in the mutator outpacing the collector. The naïve handling of this sequence of events forces the current cycle to run to completion, performing non-incremental collection, and then immediately initiating a new incremental collection cycle. This approach risks violation of the real-time bounds.

GHC's block-allocated heap presents an alternative solution. Instead of forcing a cycle to completion when the nursery blocks are exhausted the `allocBlock()` API is invoked and an attempt is made to chain an additional block onto the nursery. Only when memory is truly exhausted must a collection cycle be forced to completion. This enables some resilience to the problems that bursty allocation rates present to the work-based collectors but also greatly benefits the time-based collectors, enabling us to continue to schedule mutator and collector quanta at the same rate, albeit at the expense of a slight increase in the memory footprint.

For simplicity and efficiency we tie the quanta of the time-scheduled collectors to the scheduler's thread context switch period. This avoids the overheads of managing a timer stack and multiple outstanding signals. Furthermore, a simple reentrant (`SIGALRM`) signal handler can be defined. This handler already sets the preemption flag when it expires. However, as discussed in Section 7.2.3, the preemption flag is checked lazily on a heap overflow and so mutator quanta may overrun. To avoid this, we modify the context switch handler so that in addition to setting the preemption flag, the value of the `HpLim` register is driven below that of the `Hp` register forcing the heap check to fail. The scheduler must then detect the forced failure of the heap check and restore `HpLim` to its correct value so as not to initiate a new collection cycle, force the current one to completion, or needlessly chain on additional heap blocks. Note that we are careful not to corrupt the value of the `Hp` register since the synchronisation required, and the potential race conditions resulting from attempting to cache its old value before driving it to `HpLim` are complex and would carry a punitive overhead. Instead, it is trivial to determine the correct value of `HpLim` using `HpLim`'s block descriptor and the (constant) block size.

The remaining modifications within the scheduler are applied to the code that handles infrequent, unusual or abnormal thread operations, such as relocating the thread stack of a

TSO (heap allocated *Thread State Object*) on stack overflow, exception handling, and the detection and recovery from thread deadlock. All of these situations involve primitives that manipulate, but do not necessarily return to, stack frames. As a result, the code must handle the presence of hijacked (self-scavenging) stack frames that have yet to be scavenged by the incremental stack scavenger. Each TSO caches the incremental stack scavenge pointer so it is trivial to determine which frames, if any must be scavenged based on the closure type of the frame, or its position in the stack relative to the scavenger.

## 7.4 The Garbage Collector

### 7.4.1 Incremental Parameterisation

The following variables back new runtime system options that parameterise the incremental collectors:

- `int incrementalGC` — Specify which generations are subject to incremental collection. By default all generations are collected incrementally.
- `int minorEvacuationLimit` — overrides the dynamic calculation of Baker's $k$ value as calculated by Equation 3.7, setting `evacuation_limit` to a user-defined value for the number of words to be evacuated in each increment of a *minor* collection cycle.
- `int majorEvacuationLimit` — overrides the dynamic calculation of Baker's $k$ value as calculated by Equation 3.7, setting `evacuation_limit` to a user-defined value for the number of words to be evacuated in each increment of a *major* collection cycle.
- `int overflowChainBlocks` — Specifies the number of blocks to chain onto the nursery in order to avoid the forced-completion of the current collection cycle as a result of heap check failure (see Section 7.3). The default value is '1', a value of '-1' disables block-chaining. Note that each block-chaining operation must `closeNursery()`, fixup abstract machine registers, and then re-`openNursery()`. This option therefore enables the reduction in frequency, and therefore cost, of this operation at the expense of a larger copy reserve.
- `rtsBool timeBasedGC` — Specifies whether the incremental scavenger uses work-based or time-based scheduling. Time-based scheduling is the default and when enabled all attempts to override `evacuation_limit` are ignored.

If time-based scheduling is enabled the mutator and collector quanta default to the thread context switch period of 10 milliseconds. This can be overridden using the existing option that sets `ctxtSwitchTime`.

### 7.4.2 Garbage Collector Invocation

Figure 7.4 outlines the *basic* execution flow of the incremental collectors. The subsequent four sections detail the key phases (see rounded boxes of Figure 7.4).

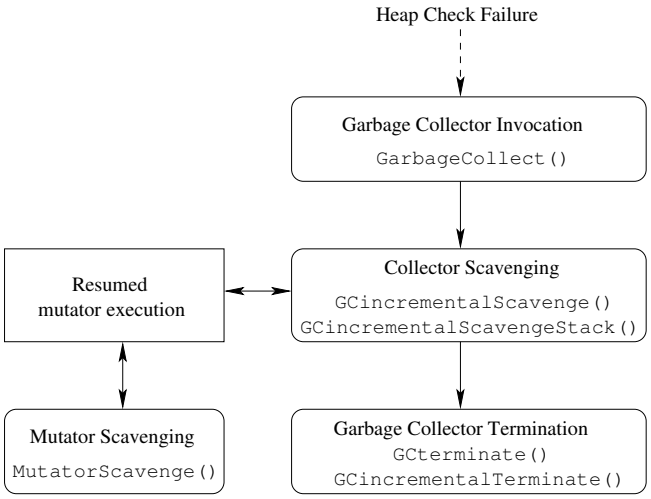



Figure 7.4: Basic flow of execution for the incremental collectors

When the scheduler triggers the collector by invoking the `GarbageCollect()` API, previously described by the Function `generationalGC()` of Section 6.1.7, the collector sets the `gc_in_progress` flag to indicate that an incremental collection cycle is in progress. The generations which are to be collected are determined, and their to-space blocks and scavenge pointers initialised, as outlined in Function `initialiseGenerationalGC()`. Recall that a block is marked as belonging to to-space by setting the `BF_EVACUATED` flag of its block descriptor.

Lines 5 to 16 of Function `generationalGC()` evacuated the roots in their entirety. This approach risks violation of the incremental and real-time bounds. The majority of the root set contains the thread stacks of the TSOs (Section A.4.1), the (im)mutable lists of each generation and the CAF (see Appendix Section A.4.3) and static closure lists. In order to limit the work done in processing the roots, and avoiding immediate evacuation, we arrange for the stacks to be scavenged incrementally, and those *pointed* members of these lists (i.e. those closures that can be entered) to be hijacked by *self-evacuating-and-scavenging* specialisations. In order for these lists to be initialised and subsequently populated with new objects that should not be subject to the scavenging routines within the current cycle, the handles to the (head of) these lists, as well as the large object lists belonging to each generation, are reassigned to a reference marking them as the list from the previous collection cycle.

Incremental stack scavenging (see Section 7.1.4) is initiated by invoking `evacuate()` on each TSO. TSO's are most usually large objects and so are not copied by `evacuate()` but only

scavenged, as described in Section 7.4.3. `evacuate()` invokes `scavengeTSO(StgTSO *tso)` which hijacks the top stack frame, switching it to the self-scavenging frame. Each TSO object maintains its own scavenge pointer, `scav_sp`, as well as the original info table pointer prior to hijacking, `hijacked_scavenge_frame`.

Lines 6 and 11 of Function `generationalGC()` are replaced with `prepareImmutableList()` and `prepareMutableList` that fully traverse each of the (im)mutable lists and hijack the info pointers of those pointed members, replacing them with their self-evacuating-and-scavenging closure specialisations that, on entry, `evacuate()` the closure to to-space, and then immediately invoke `MutatorScavenge()` to scavenge the forwarded closure, before finally invoking the closure's standard entry code. The same is done for the CAF and static object lists. The types of pointed closures that may reside on these lists are (static) indirections, CAFs (static thunks), and top-level (static) functions and constructors. Specialising indirections is trivial since they are few in number and their static info tables and entry code are defined within the runtime system. However for CAFs and top-level functions and constructors we are forced to modify the compiler to generate their specialisations in the same way as the self-scavenging specialisations are generated for dynamic closures (see Section 7.1.1 and Section 7.2.4).

Lines 17 to 25 are replaced by `GCincrementalScavenge()` which performs a single unit of incremental collector-scavenging before the mutator is resumed and the allocation of the object that previously resulted failure of the heap check is now reattempted (see Section 7.4.3).

Lines 26 to 28, which perform collector termination, are moved to two new APIs, `GCterminate()`, and `GCincrementalTerminate()` (see Section 7.4.5), which are invoked when the incremental cycle is complete and the collector is ready to terminate.

### Evacuating an Object

The `evacuate()` API does not just handle the evacuation of objects in the root set but is *eventually* invoked for every live object in the system. Evacuation invokes `copy()` on those closures that are to be copied to to-space (not static or large objects), and installs the forwarding pointer. The most significant modification to `copy()` is the hijacking of a closure's info pointer, by assigning it to the dereferenced value of its `selfscav_infoptr` (the info pointer of its self-scavenging specialisation). Listing 7.3 demonstrates this. Unpointed objects that are never entered, and so have no associated entry code, cannot operate using our self-scavenging technique, and so use the `copyUnpointed()` method that is identical to the old `copy()` routine. For those unpointed objects which are not predominantly manipulated by primitives in which an explicit read barrier can be embedded (e.g. arrays), `evacuate()` must now also immediately scavenge the object.

```
StgClosure * copy(StgClosure *src, nat size, step *stp)
{
  P_ to, from, dest;
  /* Find out where we're going, using the handy "to" pointer in
   * the step of the source object.  If it turns out we need to
   * evacuate to an older generation, adjust it here.
   */
  if (stp->gen_no < evac_gen) {
#ifdef NO_EAGER_PROMOTION
    failed_to_evac = rtsTrue;
#else
    stp = &generations[evac_gen].steps[0];
#endif
  }

  /* chain a new block onto the to-space for the destination step if
   * necessary.
   */
  if (stp->hp + size >= stp->hpLim) {
    gc_alloc_block(stp);
  }

  evacuated_count += size;

  to = stp->hp;
  SET_INFO((StgClosure *)to, get_itbl((StgClosure *)src)->selfscav_infoptr);
  to++;
  from = (P_)src + 1;
  size--;

  for( ; size>0; --size) {
    *to++ = *from++;
  }

  dest = stp->hp;
  stp->hp = to;

  /* Install the forwarding pointer */
  src->header.info = &stg_EVACUATED_info;
  ((StgEvacuated *)src)->evacuee = dest;

  /* Keep free-space pointer up to date */
  stp->hp_bd->free = to;

  return (StgClosure *)dest;
}
```

Listing 7.3: The copy function that hijacks the info pointer installing the self-scavenging specialisation.

### 7.4.3 Collector Scavenging

As the mutator executes, if the collector is enabled, it periodically performs a unit of incremental scavenging using the `GCincrementalScavenge()` API.

#### Work-based Scavenging

Work-based collectors incrementally scavenge either: i) at every bump pointer heap allocation that increments `Hp` — this is the scavenging policy of Section 5.2 referred to as 'EA'; or ii) as `Hp` hits the end of the current nursery block and is set to point to the start of the next block in the chain, assuming unused blocks exist — this is the scavenging policy of Section 5.2 referred to as 'EB'. Clearly, invoking the runtime system scavenger code "at every allocation" on the

bump pointer allocation fast-path incurs a significant overhead, as documented in Chapter 8. Invoking the scavenger "at every block allocation" within the heap check failure code which is on the slow allocation path incurs negligible overhead. In either case, the additional code must test the `gc_in_progress` flag to determine if `GCincrementalScavenge()` should be invoked. The scavenger is also invoked when storage space is requested from the Out-of-band Allocator (see Section A.4.2). We avoid testing the `gc_in_progress` flag at each invocation of the Out-of-band allocator through the use of a function pointer. If `gc_in_progress`, then this pointer points to a function that invokes `GCincrementalScavenge()` and performs the allocation. If it is not enabled, then the pointer points to the standard routine.

A unit of incremental scavenging involves the scavenging of all allocation areas, including those external to the heap, within the system. As a result `GCincrementalScavenge()` and its constituent scavenging routines are structured to loop for as long as: i) there exist some objects still yet to be scavenged; ii) the global `evacuated_count` is less than the `evacuation_limit`. These two (unsigned) variables record the number of words that: a) have been evacuated in the current work increment, and b) the total number of words to evacuate in that increment. The `evacuation_limit` is determined by Equation 3.7 using the amount of live data in the previous cycle. Note that this is not just the amount of live data in the heap, but also in the other areas that are incrementally scavenged, such as the thread stacks and the large, pinned, and out-of-band allocation areas.

#### Time-based Scavenging

Time-based collectors perform a unit of incremental scavenging on the subsequent (faked) heap check failure following the expiry of the thread context switch timer, as described in Section 7.3. The structure of `GCincrementalScavenge()` is identical to that of the work-based collectors. On expiry of the mutator quantum, when it is the scavenger's turn to run, we set the `evacuation_limit` to the largest representable 32-bit integer — this is equivalent to setting $k$ to 4GB, a huge value that is sure to violate the incremental bounds were that many words to actually be evacuated. On expiry of the collector quantum, when the mutator is to be resumed, we force the scavenger to terminate by setting the `evacuation_limit` to 0. Not only is the termination test efficient but the code can be shared between both the work-based and time-based collectors.

#### The Scavenge Queue

The benefit of the block allocated storage manager is that to-space is always separate from the heap and Out-of-band Allocator storage areas. Furthermore, there is no explicit scavenge queue, since all the objects in to-space must eventually be scavenged, and there is no danger of the collector scavenging those objects that are allocated during resumed mutator execution.

It is because the live objects of the system do not end up in a contiguous to-space chunk, as happens in Baker's algorithm, that incremental scavenging is somewhat more complicated.

`GCincrementalScavenge()` consists of a loop over four separate routines, each of which is described below, where each routine may result in the evacuation of objects whose scavenging must be handled by one of the other routines. The collection cycle is therefore only complete when a pass of the loop is made in which *no scavenges are performed* and hence no evacuations occur which would result in further objects to be scavenged.

On initiation of a collection cycle, once the root set has been evacuated, the `scan` pointers of each step of each generation being collected are initialised to point at the object at the start of the first to-space block. As collector-scavenging (`GCincrementalScavenge()`) operations are invoked, objects in to-space are scavenged with each `scan` pointer advancing by the size of the scavenged closure. Because each `scan` pointer makes a single, complete pass of its to-space region, all live objects currently evacuated to to-space will have been scavenged when all `scan` pointers have reached the end of their last to-space blocks. This corresponds to scavenging an object at the head of the scavenge queue and then removing the object from the scavenge queue so it will not be subsequently scavenged.

Note that the scavenger may encounter an object that has previously been scavenged by the mutator. Recall from Section 3.4.1 that in a standard incremental collector that scavenges forwarded to-space objects, there is no easy way for the collector to determine whether an object that it is about to scavenge within a work increment has already been scavenged by the mutator's read barrier. As a result all referents of the encountered object must be tested to determine whether they must be evacuated. This is not the case for our collectors — when the scavenger encounters an object that is not in its self-scavenging form, it "knows", based on its type, that the object has already been scavenged and can be skipped.

#### Incremental Stack Scavenging

The stack is scavenged incrementally as described in Section 7.1.4 on invocation of `GCincrementalScavenge()` by either `scavenge()` or `scavenge_large()` depending on the size of the TSO.

The most complex aspect of the implementation of the self-scavenging stack frame is the hijacking of unboxed tuple return addresses — the stack pointer points at the last argument on top of the return address, not at the return address itself. We therefore specialise the return addresses for unboxed tuples, in a similar way to standard self-scavenging closures, extending their info tables to contain the amount by which the stack pointer will be offset from the frame at the time of the return.

#### The Scavenging Loop

Figure 7.5 shows the structure and control flow of the loop body of `GCincrementalScavenge()` that invokes the four scavenging routines, that between them handle the different allocation areas in the system:

- to-space, using the `scavenge()` routine.

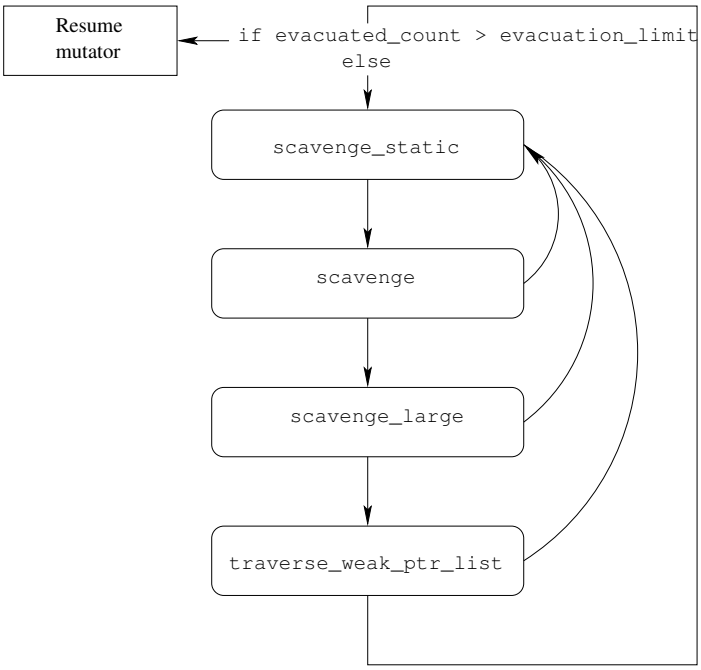



Figure 7.5: Control flow of `GCincrementalScavenge()`

- Statically allocated (fixed address) objects, using the `scavenge_static()` routine.
- The large objects list using the `scavenge_large()` routine. Large objects reside in their own storage area and are never evacuated from it, but are always scavenged.
- The weak pointer list, using the `traverse_weak_ptr_list()` routine.

**Scavenging to-space**

The scavenging of to-space is performed by the `scavenge()` function. It consists of loops for the iteration over: i) the to-space chains of each generation's steps, as previously described by Lines 17 to 25 of the Function `generationalGC()` of Section 6.1.7; and ii) the body of Function `generationalScavenge()` that scavenges the individual objects within a single to-space chain using the interpreted code that inspects the `type` field of each closure's info table. In essence this is the code of Listing 7.4 operating within a loop that traverses each block within the to-space chain and terminates either when the end has been reached or `evacuation_limit` - `evacuated_count` objects have been scavenged.

The scavenger automatically skips objects that have been previously scavenged and any associated slop objects of updated thunks.

### Handling Large Objects

Recall that GHC does not allocate large objects in the heap. Because each large object takes up one or more blocks, they are created by making a direct request to the Block Allocator for the number of blocks required. The block descriptor that is returned is then chained onto a doubly linked list, `large_objects`, that is local to each step of a generation, and which contains references to the block descriptors of every live large object in that generation.

As described in Section 7.4.2, each `large_objects` list is reassigned to its `old_large_objects` reference.

When a large object is evacuated, its block descriptor reference on `old_large_objects` is moved to the `new_large_objects` list where it awaits scavenging. Once scavenged, they are then moved to the `scavenged_large_objects` list. In this way large objects are pinned within their blocks and undergo scavenging but not copying. At the end of the collection cycle, any large objects on `old_large_objects` are no longer live and their blocks can be returned to the Block Allocator's free list. The `scavenged_large_objects` list is then chained on to its step's `large_objects` list.

Like standard objects, `MutatorScavengeLarge()` handles the scavenging of a single large object, while `scavenge_large()` is modified to perform a unit of scavenge work (the remaining `evacuated_limit - evacuation_count` objects).

### Handling Weak Pointers

`traverse_weak_ptr_list()` is called possibly many times during garbage collection in the main loop of `GCincrementalScavenge()`. Its order in this loop is essential and it is critical that the following invariant is maintained:

> `traverse_weak_ptr_list()` is called when the heap is in an idempotent state. That means that there are no pending evacuate/scavenge operations. This invariant helps the weak pointer code decide which weak pointers are dead: if there are no new live weak pointers, then the currently unreachable ones are dead.

The routine returns a flag indicating whether or not it called `evacuate()` on any live pointers.

The collectors traverse the list incrementally until the `evacuation_limit` is hit. The mutator will then be resumed. Because weak pointers are only removed from this list if they are live it is both undesirable and inefficient to keep scavenging the list from the top each time the routine executes. The position that the scavenger reaches is therefore maintained across calls to this routine so that traversal may be continued from where it was last left off.

On completion of the collection cycle, any elements still on the list are dead and may have their finalisers run. Unfortunately, it is possible for the mutator to complete its computation before the current collection cycle completes. The result is that the program may exit without the finalisers of the dead weak pointers being run. This can have a disastrous consequences,

for example the output weak pointer that is associated with `stdout` needs to have its finaliser run in order to dump the result of any computations performed to the screen. To ensure that the finalisers of such weak pointers are executed, if a collection cycle is in progress when the last/main thread finishes executing, it is forced to run to completion before the program may exit.

### Scavenging Static Objects

The only type of static objects that point to the heap are CAFs [3] (static indirections). The remaining static objects point only to other static objects either via references in their payload for static constructors or via the *static reference table* (SRT) for static functions and thunks.

In our original Non-Stop Haskell collector, we were unable to perform self-scavenging of statics (including CAFs) using the generic self-scavenging object, since there was nowhere to "remember" the hijacked info pointer. The result of this, was that the `scavenge_static()` routine had to be invoked just before mutator execution was resumed (i.e. in `MutatorScavenge()` and `GCincrementalScavenge()`) and had to process the list in its entirety as opposed to incrementally. Fortunately, using object specialisation, `scavenge_static()` can operate incrementally, terminating when the `evacuation_limit` is hit. Static objects are marked as live and "evacuated" by placing them on the `static_object` list in their self-scavenging form. When the (self-scavenging) static object is subsequently scavenged, by either the mutator or collector, `scavenge_static()` is invoked and the object is removed from this list and placed on the `scavenged_static_objects` list. If the object has an SRT, its SRT references are evacuated, its payload references are then evacuated.

When handling CAFs, the problem is in identifying all live static indirections before the mutator can obtain a reference to them. The static object list is itself built during the collection cycle, and it is only at the end of the cycle that the locations of the live static objects are known. This is a real problem. In order to solve this, a list of all the live CAFs must be maintained (`caf_list`). The function `newCaf()` does this. At the start of a collection cycle, the `caf_list` is reassigned to the `old_caf_list`, and all the CAFs currently on `old_caf_list` are treated as roots. However, instead of immediately evacuating their indirectees, the indirections are replaced by their self-evacuating-and-scavenging specialisations. When a static indirection is `evacuated`, it will be placed onto the `static_objects` list and eventually after `scavenge_static` has been invoked, it will be placed on the `scavenged_static_objects` list. At the end of collection, it is known which CAFs are live — they are on the `scavenged_static_objects` list and their `STATIC_LINK` fields are non-`nil`. Any CAFs on `old_caf_list` can now be removed and black-holed — since their heap values have been collected.

[3] Appendix Section A.4.3 describes the issues surrounding the garbage collection of CAFs.

### 7.4.4 Mutator Scavenging

Whilst executing, the mutator may enter a self-scavenging object. This object must be scavenged before the mutator is allowed to handle any object references in the payload. The entry code that is executed when an object of this type is entered has already been described in Section 7.2.3. The result of executing the entry code is a call to the collector's `void MutatorScavenge(StgPtr p)` API. A simplified version of `MutatorScavenge()`, that omits some of the self-scavenging specialisations is outlined in Listing 7.4. The closure is scavenged by: i) reverting its info table pointer from its hijacked self-scavenging variant to its standard variant (Line 8); ii) evacuating its payload referents and flipping their info table pointers to their self-scavenging variants; iii) adding the closure to the inter-generation remembered set, the generations mutable list, if one of its referents could not be evacuated into the referrer's generation (Lines 60 to Lines 63). Note that mutator scavenging scavenges only a single object, it does not continue to evacuate the outstanding `evacuation_limit - evacuated_count` words.

### 7.4.5 Garbage Collector Termination

Eventually, in a call to `GCincrementalScavenge()`, all the live data in the system will have been scavenged and the collection cycle is terminated. The completion of an incremental cycle is performed by `GCterminate()`. If necessary, the `GCincrementalTerminate()` API forces an incremental cycle to completion, in a stop-the-world fashion. Essentially it inlines the body of `GCincrementalScavenge()`, omitting the outer loop that terminates a unit of incremental scavenging, and that of `GCterminate()`. `GCterminate()` performs lines 26 to 28 of Function `generationalGC()` of Section 6.1.7, in which: the blocks previously allocated to the old from-space are now freed; the size of the nursery is determined in preparation for the next collection cycle and the older generations resized; dead CAFs are reverted (see Section 7.4.3); and various structures are tidied up. The latter include: clearing the block descriptor `BF_EVACUATED` flag bit for all newly allocated blocks; the chaining of scavenged allocation areas external to the heap (e.g.`scavenged_static_objects`, `scavenged_large_objects`, etc) onto those currently in use (e.g.`static_objects`, `large_objects`, etc); the switching of the Out-of-band Allocator's allocation function pointer to point to the function that does not invoke `GCincrementalScavenge()` on a new object allocation; and finally the collection statistics are bumped.

## 7.5 Replicating Garbage Collection

Whilst exploring the design space for incremental collector implementations within Haskell we developed prototypes for both a replicating collector and a variant of Brooks' collector. This section briefly describes the challenges in implementing such collectors for GHC.

```
void MutatorScavenge( StgPtr p )
{
  /* Obtain closures info table and block descriptor */
  StgInfoTable *info = get_itbl((StgClosure *)p);
  bdescr *bd = Bdescr((P_)p);

  /* Reset the info-pointer: indicates scavenged */
  ((StgClosure *)p)->header.info = info->selfscav_infoptr;

  /* Set evac_gen, the generation the closure undergoing
   * scavenging is in.
   */
  evac_gen = bd->gen_no;

  /* Scavenge the closure */

  switch (info->type) {
    case FUN_SELF_SCAV:
      scavenge_fun_srt(info);
      goto generic_object;

    case THUNK_SELF_SCAV:
      scavenge_thunk_srt(info);
      /* fall through */

    generic_object:
    case CONSTR_SELF_SCAV: {
      StgPtr end, q;

      end = (P_)((StgClosure *)p)->payload + info->layout.payload.ptrs;
      for (q = (P_)((StgClosure *)p)->payload; q < end; q++) {
        (StgClosure *)*q = evacuate((StgClosure *)*q);
      }
      break;
    }

    case THUNK_SELECTOR_SELF_SCAV: {
      StgSelector *s = (StgSelector *)p;
      s->selectee = evacuate(s->selectee);
      break;
    }

    case AP_STACK_SELF_SCAV: {
      StgAP_STACK *ap = (StgAP_STACK *)p;
      ap->fun = evacuate(ap->fun);
      scavenge_stack((StgPtr)ap->payload, (StgPtr)ap->payload + ap->size);
      break;
    }

    case PAP_SELF_SCAV:
    case AP_SELF_SCAV:
      (void)scavenge_PAP((StgPAP *)p);
    break;
  }

  /* If a payload referent couldn't be evacuated into this
   * generation, then record the referrer on the mutable list.
   */

  if (failed_to_evac) {
    failed_to_evac = rtsFalse;
    mkMutCons((StgClosure *)p, &generations[evac_gen]);
  }
}
```

Listing 7.4: `MutatorScavenge()` that scavenges a self-scavenging specialisation on closure entry.

Central to this algorithm is the ability to non-destructively forward the from-space object. Nettles et al. achieve this by replacing the header word of an ML object with the forwarding pointer when it is forwarded. A read barrier is then placed on those operations that must access the object header that determines whether the object has been forwarded and if so follows the pointer to get to the original header word. They document that the header is only accessed for polymorphic equality operator and type specific length operations but do not present results for the frequency of these operations or the overhead incurred by the test. In GHC, this header (the info pointer) is central to the evaluation model and is accessed frequently, so the overhead would be much more significant. As a result, an extra word must be prepended to *all* closures.

We thread mutation log entries through the heap in the same way that the (im)mutable lists are threaded — the mutation log is therefore heap allocated and there is no danger of its overflow. Because of this threading, a closure can only exist on one of either the (im)mutable list or the mutation log and so when adding a from-space closure to the mutation log we perform the generational test to determine if the to-space replica should be placed on the mutable list. We handle updates, which are the most frequent form of mutation slightly differently. Since updates undergo a single mutation (evaluation to WHNF), instead of placing the closure in the mutation log, we *eagerly* apply the barrier and immediately propagate the update to the replica. The generational test is applied to the replica and if necessary it is immediately placed on the immutable list. By sharing the field through which mutable closures are threaded onto the remembered set list, we save an additional word overhead that would usually be allocated for the mutation log entry. In the rare case where a mutable closure does not have such a field for placement onto the remembered set list, instead of increasing the size of the closure, we write into the to-space replica object a *mutation closure* which has an info pointer indicating it as such and a payload of two pointers. The first pointer references the mutated from-space closure and the second is the link field for the mutation log.

Nettles and O'Toole say nothing about the strategy for handling allocation of new objects, which is potentially a significant overhead. In order to maintain the from-space invariant new objects must be *doubly allocated*, that is, allocated in both from-space and to-space. Mutations to these from-space objects must then be logged. Cheng [Che01, CB01] reduces the cost of double allocation using a deferred scheme. Objects are allocated without replication and, when the scavenger completes scavenging to-space, a shorter collection cycle is initiated in which the newly allocated objects are copied. This strategy requires a new root set computation and the memory graph to be traversed in order to evacuate the live data from the nursery and to update the referring references. If this mechanism is employed incrementally then, pathologically, the collection cycle may never terminate with the scavenger always playing catch-up.

Before discussing how we allocate new objects, we turn our attention to large objects and stack mutations. The SML stack is heap allocated — when stack frames are popped they become garbage within the heap and are garbage collected. In GHC this is not so — an

explicit stack exists that *resides* in the heap-allocated TSO. Push and pop stack operations are therefore cheaper and the collector is invoked less frequently. We do, however, have the added complication of handling frequent stack mutations and stack relocation should the thread stack overflow. Finally, there is the issue of large objects (of which a TSO can be one) that do not undergo copying. Instead, they are scavenged in-place (synonymous with *pinned* memory blocks). Maintaining a strict from-space invariant now requires these objects to be copied. This adds further overhead in an incremental collector where large objects that undergo copying should do so within bounded time — this is usually achieved by splitting the large objects into smaller objects and placing an overhead on mutator access and manipulation operations.

In an attempt to resolve these issues of non-destructive forwarding, double allocation, stack mutation, and large object allocation, we relax the from-space invariant. Newly allocated objects, large objects and thread stacks are now allowed to obtain references to to-space pointers. The write-barrier is now more expensive, eagerly applying updates, and collector termination is more costly and complex. Once the mutation log has been processed and the flip occurs, the root set must be recomputed and the live data traversed to flip from-space references to their to-space replicas. Because only certain types of closure can potentially contain a from-space reference, this phase can be optimised so that the scavenger skips closures that cannot. GHC forwarding pointers (in the non-replicating collectors) are uniform closures, with an info table and entry code. The entry code is an empty code block (it terminates the mutator if entered and reports the error) and the payload is a single pointer to the to-space copy. If, however, we modify the entry code to act like that of an indirection, such that it enters the entry code of its referee we have created a dynamic-dispatching forwarding pointer similar to a Brooks-style indirection. The extra word reserved for the forwarding pointer (see above) can now be eliminated. This resulting collector implementation is a hybrid of Brooks' collector with its relaxed to-space invariant and a replicating collector. If we modify the replicating write barrier so that mutations are no longer logged, but instead to do the work of Baker's read barrier (i.e. evacuate any from-space references before writing them into to-space closures), then the collector is a pure Brooks implementation. At termination, as the flip is performed, from-space references in machine registers, the nursery, and the thread stacks are flipped to their to-space copies in a similar way to the replicating collector implementation, although a complete traversal of all live data is unnecessary. The only issue that must be resolved is to modify primitives that manipulate closures, without entering them (for example the generic apply routines) to recognise a forwarding closure and to follow the forwarding pointer to the closure on which it will actually operate.

Preliminary experiments with prototypes of these two collectors demonstrate overheads in excess of 650% over the baseline generational stop-the-world collector with the overheads of the termination cycles dominating. The conclusion is that, for Haskell, these collectors are both complex and prohibitively expensive and as such we consider them no further.

# Chapter 8

# Evaluation

To evaluate the various schemes of Real-time Haskell we have implemented each of them in version 6.02 of GHC. We selected 36 benchmark applications from the `nofib` suite [Par93] – see below for an explanation of the ones selected – and conducted a series of experiments using both the complete set and, for more detailed analysis, a selection of nine of the applications that exhibit a range of behaviours. Benchmarks for the incremental collectors were conducted on a 400MHz Celeron with 1GB RAM, a 128K level-2 cache, a 16KB instruction cache and a 16KB, 4-way set-associative level-1 data cache with 32-byte lines. The write barrier experiments were run on a 500MHz Pentium III processor with a 512K level-2 cache and identical level-1 cache, associativity and line size specifications. The systems ran SUSE Linux 9.0 with 2.6.4 kernels in single-user mode. We deliberately chose slow machines in order to increase the execution times and reduce their variance. All results reported are geometric averages taken over five runs and are expressed as overheads with respect to reference data. At the bottom of each table we show the minimum, maximum and mean overhead taken over all 36 benchmarks.

In evaluating the various schemes we focus particular attention on the following questions:

1. What is the overhead on the mutator execution time (Section 8.1) and the increase in compiled binary sizes, the "code bloat", (Section 8.1.1) of each of the various schemes?

2. How do the various collectors perform in a single-generation heap? In particular, how does the dynamic one-word overhead on each copied object in our original Non-stop Haskell scheme affect the behaviour of the mutator and garbage collector (Section 8.2)?

3. In moving to a generational scheme how much can be gained by the various optimisations that accrue from specialisation (Section 8.4)? Supplementary to this, is it worth trying to eliminate the write barrier (Section 8.6) using the techniques developed in Chapter 6?

4. How does incremental collection change the distribution of the pause times in a generational collector (Section 8.5)? In answering this question we also wish to consider

minimum mutator utilisation which is a measure of the amount of useful work the mutator performs in a given "window" of time.

## 8.1 Baseline Overheads

We begin by analysing the overheads imposed on the mutator by the various schemes. To do this each of the benchmarks in the nofib suite was executed and a subset of 36 selected that, appropriately parametrised, satisfied the following two conditions: i) They terminated without garbage collection using a 1GB fixed-size heap on the reference platform. ii) They executed for a sufficiently long time (at least 5 seconds, the majority much longer) to enable reasonably consistent and accurate timings to be made. In ensuring that the benchmarks have sufficient memory available to execute without the need for a single collection cycle, the cost that the incremental infrastructure of the collector places on the mutator (the cost of "going incremental") is determined.

The results are shown in Table 8.1 for GHC's single-generation stop-and-copy collector (`REF`), the same with fast entry points turned off (`REF*`) (see Section 7.1.6), the traditional Baker algorithm with an explicit read barrier (`BAK`), our previous 'Non-stop Haskell' collector (`NSH`), the new variant of `NSH` with object specialisation and shared entry code (`SPS`) and the same but with inlined entry code (`SPI`). For these last four, we report results where incremental scavenging takes place on every mutator object allocation (`-A`) and every new memory block allocation (`-B`) – see Section 7.1.7. In the last two columns the results for the time-based collectors (see Section 7.1.7 and Section 7.3) are reported. These collectors are `SPS` variants that employ specialisation with shared entry code and operate with mutator and collector scavenging quanta of 1 millisecond in the first collector and 10 milliseconds in the second. We list individual results for the nine benchmarks selected, in addition to averages taken over the whole benchmark suite. We report average overheads, but not average execution time. A positive overhead represents a slow-down.

When comparing each of the collector variants (`BAK`, `NSH`, `SPS`, `SPI`, etc), the `-A` and `-B` results for each variant, and the `-A`/`-B`'s between variants, expected trends emerge and the individual benchmarks broadly exhibit similar characteristics, with the same speed-ups and slow-downs. Furthermore, they follow the trend that emerges when taken over all 36 benchmarks. `symalg` is however an exception to this and is particularly unusual: it computes $\sqrt{(3)}$ to 65000 significant figures and is almost totally dominated by the construction of new data representing the result, none of which is "revisited" by the program. The execution time is practically identical over all schemes. The variations in the table are largely attributable to measurement noise.

The two variants of Baker's read barrier scheme incur an additional overhead of at least 35% on average over all the other schemes.

For the Baker variants, the overheads come from:

| Application | REF (s) | REF* (%) | BAK-A (%) | BAK-B (%) | NSH-A (%) | NSH-B (%) | SPS-A (%) | SPS-B (%) | SPI-A (%) | SPI-B (%) | SPS-1ms (%) | SPS-10ms (%) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| anna | 119.26 | –0.16 | +82.18 | +82.62 | –0.08 | +2.04 | +0.59 | +1.92 | +0.08 | –0.04 | +1.29 | +0.66 |
| circsim | 16.29 | +1.04 | +38.31 | +37.51 | +4.85 | +1.47 | +6.38 | +3.19 | +6.71 | +2.87 | +7.24 | +5.56 |
| lcss | 19.40 | –1.60 | +31.13 | +31.34 | –0.62 | –3.40 | +1.34 | –1.29 | –0.46 | –0.82 | +4.91 | +3.10 |
| pic | 25.33 | +0.99 | +33.64 | +32.81 | +6.36 | +7.03 | +7.11 | +4.70 | +6.84 | +4.96 | +7.11 | +5.24 |
| scs | 23.17 | +3.88 | +32.50 | +28.96 | +6.34 | +4.96 | +5.83 | +3.88 | +6.37 | +3.27 | +6.84 | +5.49 |
| symalg | 21.93 | –0.15 | –0.24 | –0.03 | +0.18 | –0.27 | +0.41 | +0.21 | +0.00 | –0.15 | +1.83 | +0.09 |
| wang | 23.85 | –0.92 | +31.28 | +29.31 | +0.67 | +0.34 | –0.13 | +1.89 | –1.46 | –1.35 | –1.21 | +1.60 |
| wave4main | 51.95 | +1.12 | +58.79 | +56.13 | +2.69 | +2.00 | +6.16 | +2.93 | +4.47 | +2.16 | +5.58 | +3.34 |
| x2n1 | 22.56 | +0.09 | +55.41 | +53.86 | +4.21 | –0.18 | +5.76 | +1.60 | +5.27 | +1.73 | –0.73 | +4.47 |
| Min 36 | | –1.60 | –0.24 | –0.03 | –0.62 | –3.40 | –0.96 | –1.29 | –0.97 | –1.13 | –1.21 | –3.07 |
| Max 36 | | +20.45 | +84.64 | +82.63 | +26.10 | +21.91 | +31.08 | +25.17 | +31.27 | +25.03 | +32.04 | +27.85 |
| ALL 36 | | +1.99 | +39.20 | +37.22 | +4.61 | +2.28 | +6.54 | +3.84 | +6.73 | +3.78 | +6.45 | +5.31 |

Table 8.1: Mutator overheads reported as a percentage overhead on GHC 6.02 execution times (REF) with no garbage collection

- The need to test whether the collector is on (`gc_in_progress`) at each allocation and at each object reference (Section 5.2, Section 5.4.1, and Section 7.4.3).
- The test to determine whether an object has been evacuated and/or scavenged at each object reference (Section 5.3).
- The fact that fast entry code must be turned off (Section 7.1.6)

For the remaining collectors the overhead comes from turning off fast-entry points and from the `gc_in_progress` test at either each allocation, for the work-based collectors, or when returning to the thread scheduler, for the time-based collectors. For all the work-based collectors, it should be clear that this first overhead will be greater if incremental scavenging occurs at each object allocation, rather than at each allocation of a new memory block (4KB in the current implementation). This is confirmed on comparison of the results in the `-A` and `-B` columns for each variant, with the "every allocation" test incurring an additional 2–3% over the "block allocation" test when taken over all 36 benchmarks. Similarly, for the time-based collectors, the overhead increases as the quanta are shortened as the currently executing thread is forced to return to the scheduler more often. While the work-based and time-based collectors cannot really be compared it is worth noting that the overheads of the 1ms time-based variant are closer to those of the `-A` counterparts while those of the 10ms variant are closer to its `-B` counterparts. The time-based results are however more consistent with the 1ms variant incurring an overhead in the range of 5–7% and the 10ms variant between 3–5%.

Taken over all benchmarks, turning off fast entry points costs on average just under 2% in execution time. The `gc_in_progress` test in `NSH` introduces a negligible additional overhead (less than 0.3%) when performed at each block allocation; when performed at each object allocation the overhead is more significant (around 2.6%). With no garbage collection the only difference between the code executed by the `NSH`, `SPS` and `SPI` variants is in respect of slop filling (Section 7.1.5). However, this introduces a small overhead of between 1.5–2%, as the vast majority of useful slop objects covering object sizes 1–8 have been predefined (Figure 7.2).

It is worth noting that although the code executed with the collector off is the same for both `SPS` and `SPI`, the measured times vary very slightly. Experiments with Valgrind [Sew] confirm there is no significant difference in the cache behaviour and we have confirmed the variation is simply due to measurement noise.

Notice that Table 8.1 reports the geometric mean taken over five runs for each benchmark, but not their variances. The variances of the underlying run data are so small that it is not particularly instructive to report them. The variation is attributable to measurement noise and the accuracy of the timer used to record benchmark execution times. However, to support this, we consider for a single benchmark, the variance, standard deviation, and the two-tail heteroscedastic Student's t-test, to refute the null hypothesis that the means of the populations are identical. We choose a benchmark with the least favourable result

when comparing the Baker scheme against a variant employing specialisation. We therefore select the benchmark `pic` and compare `BAK-A` against `SPS-A`. For the execution time for `REF` in seconds, the arithmetic mean is 25.334; the geometric mean is 25.334; the variance is 0.00078; and the standard deviation is 0.0279. For the execution time for `BAK-A` in seconds, the arithmetic mean is 33.856; the geometric mean is 33.856; the variance is 0.00058; and the standard deviation is 0.0241. The overhead on execution time of `REF`, as reported in Table 8.1, is therefore 33.64%. For the execution time for `SPS-A` in seconds, the arithmetic mean is 27.136; the geometric mean is 27.136; the variance is 0.00043; and the standard deviation is 0.0207. The overhead on execution time of `REF`, as reported in Table 8.1, is therefore 7.11%.

The probability of the two-tail heteroscedastic Student's t-test in support of the null hypothesis that the means of the `BAK-A` and `SPS-A` populations are identical is $1.006 \times 10^{-18}\%$ i.e. 0% — a rejection of the null hypothesis. This result applies to all benchmarks and the pairwise comparison of all variants.

### 8.1.1 Code Bloat

The compiled binary sizes are listed in Table 8.2. In our implementation of Baker's algorithm extra code is inlined around all object accesses to implement the read barrier. Code is also planted around each object allocation in the case of `BAK-A` that invokes the incremental scavenger during garbage collection. The same overhead is paid in `NSH-A`, `SPS-A` and `SPI-A`. For `BAK-B` (likewise the other `-B` variants) this code is attached to the block allocator in the runtime system so the additional code overheads are negligible.

For Non-stop Haskell the code bloat comes primarily from additional static code in the runtime system, including the code to perform incremental stack scavenging (1.8MB vs 280KB approximately). For larger binaries the overhead is proportionally smaller. For `NSH-A` the additional overhead comes from the additional code around each object allocation.

For the specialised code versions the overheads are primarily due to the additional entry code variants associated with each object type. Once again, the code bloat in the `-A` versions is higher than that in the `-B` versions because of the additional object allocation code, as we would expect. The time-based `SPS` collectors lack this additional object allocation code. They inline the `gc_in_progress` test, the code for the scheduling of mutator and collector quanta and the invocation of the collector scavenging code, within the runtime system's scheduler and its yield points. The variation in binary sizes between the 1ms and 10ms collectors is insignificant and arises due to the inlining of the different constants used to determine the mutator and collector quanta.

| Application | REF (KB) | REF* (%) | BAK-A (%) | BAK-B (%) | NSH-A (%) | NSH-B (%) | SPS-A (%) | SPS-B (%) | SPI-A (%) | SPI-B (%) | SPS-1ms (%) | SPS-10ms (%) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| anna | 918 | –0.02 | +14.36 | +8.52 | +8.60 | +2.77 | +29.00 | +23.04 | +48.04 | +40.54 | +34.68 | +34.68 |
| circsim | 286 | +0.00 | +12.95 | +8.46 | +13.56 | +9.09 | +30.56 | +26.06 | +44.79 | +39.61 | +34.39 | +34.40 |
| lcss | 239 | +0.03 | +12.72 | +8.46 | +15.16 | +10.92 | +30.18 | +25.92 | +42.37 | +37.58 | +33.60 | +33.61 |
| pic | 361 | +0.01 | +13.18 | +8.30 | +12.16 | +7.27 | +28.02 | +23.10 | +40.42 | +34.97 | +31.85 | +31.85 |
| scs | 492 | +0.02 | +14.22 | +8.90 | +10.54 | +5.25 | +27.82 | +22.45 | +41.13 | +35.01 | +32.35 | +32.35 |
| symalg | 423 | –0.03 | +12.99 | +7.92 | +11.10 | +6.04 | +27.94 | +22.82 | +40.22 | +34.43 | +31.92 | +31.93 |
| wang | 272 | +0.00 | +12.98 | +8.74 | +13.97 | +9.74 | +27.72 | +23.49 | +37.85 | +33.23 | +30.74 | +30.74 |
| wave4main | 174 | +0.04 | +13.11 | +9.23 | +16.84 | +13.01 | +29.28 | +25.45 | +40.21 | +36.27 | +31.57 | +31.58 |
| x2n1 | 324 | +0.00 | +12.74 | +8.21 | +12.76 | +8.24 | +28.30 | +23.77 | +39.63 | +34.54 | +33.81 | +33.82 |
| Min 36 | | –0.03 | +11.82 | +7.24 | +8.60 | +2.77 | +27.14 | +22.08 | +35.32 | +30.88 | +28.74 | +28.75 |
| Max 36 | | +0.14 | +14.36 | +9.28 | +19.16 | +15.60 | +30.56 | +26.79 | +48.04 | +40.92 | +35.41 | +35.42 |
| ALL 36 | | +0.02 | +12.99 | +8.55 | +14.21 | +9.76 | +29.14 | +24.68 | +41.54 | +36.49 | +32.60 | +32.61 |

Table 8.2: Code bloat reported as a percentage overhead on the stop-and-copy collector

| Application | NSH-1 | NSH | NSH-2 |
|---|---|---|---|
| anna | 129 | 85 (–34.11%) | 191 (+48.06%) |
| circsim | 69 | 57 (–17.39%) | 88 (+27.54%) |
| lcss | 450 | 207 (–54.00%) | 350 (–22.22%) |
| pic | 68 | 65 (–4.41%) | 72 (+5.88%) |
| scs | 561 | 560 (–0.18%) | 648 (+15.51%) |
| symalg | 3764 | 3764 (+0.00%) | 3764 (+0.00%) |
| wang | 19 | 17 (–10.53%) | 25 (+31.58%) |
| wave4main | 298 | 295 (–1.01%) | 306 (+2.68%) |
| x2n1 | 226 | 187 (–17.26%) | 286 (+26.55%) |

Table 8.3: Number of garbage collections for NSH, NSH-1 and NSH-2

## 8.2 Dynamic Space Overhead

Having two implementations of the barrierless Baker scheme, with and without a one-word space overhead on copied objects, gives us the opportunity to investigate the effects of dynamic space overheads. Recall that newly-allocated objects do not carry the overhead.

We work with our original Non-stop Haskell (NSH) scheme with the dynamic space overhead on copied (to-space) objects. However, we do not want the incremental nature of the NSH collector to interfere with the experiment – some extremely subtle effects can be observed when switching from stop-and-copy to incremental as we shall see shortly. We therefore remove the return instruction from the incremental scavenger so that the garbage collector actually completes in one go, essentially making the collector non-incremental. This enables us to focus on the effect of the space overhead independently of the issue of incrementality.

One further point must be understood before we proceed. All of GHC's collectors use a dynamic *sizing policy* which allocates free space for newly-allocated objects (the *nursery*) in accordance with the amount of live data at the end of the previous collector cycle. In other words, GHC does *not* employ a fixed-size heap. The default is to allocate twice as much nursery space as there is live data, with a view to keeping the residency at around 33% on average. However, the *minimum* nursery size is 256KB so the average residency may be smaller for some applications.

If each copied object carries a space overhead we create the illusion that the residency is larger than it really is or, rather, otherwise would be. Thus, the collector will allocate *more* nursery space than it would otherwise. This increases the mean time between garbage collections and hence reduces their average total number. This effect is countered somewhat by the 256KB minimum nursery size as we explain below.

Note that the increased memory usage per object can either be thought of as increasing the GC overhead per unit memory, or increasing the memory demand per unit run-time.

To explore the effect that the overhead has in practice, we therefore built an *adjusted* variant of NSH (NSH-1) that subtracts the total space overhead from the total amount of live

data *before* resizing the nursery. For both collectors, (new) allocations to the nursery do not incur the one word overhead and so this adjustment sizes the nursery to the same as that of the stop-and-copy collector. The number of garbage collections performed by this variant should now be the same as that of a baseline stop-and-copy collector – we were able to verify this as we had made the `NSH` variant non-incremental for the experiment.

What about a fixed-size heap, i.e. where memory is constrained? This would essentially *reduce* the nursery size compared with stop-and-copy. We could rework the GHC runtime system to do this. However we instead simulate the effect by building a second variant of `NSH` that subtracts *twice* the total space overhead from the total amount of live data before sizing the nursery. Whereas the previous reduction for `NSH-1` enabled both collectors to run in equivalent space, and house the same number of objects, this second reduction now correctly penalises the `NSH-2` collector for the one word overhead. It does this by appropriately reducing the amount of heap space available as would be the case when running in a fixed-size heap.

The results are summarised in Table 8.3. The average object size for our benchmarks is around 3.3 words so the space overhead is around 30% on all copied objects. This suggests that `NSH` (`NSH`-2) should reduce (increase) the number of garbage collections by around 30% when compared to NHS-1 (and the baseline stop-and-copy collector). However, the 256KB minimum nursery size reduces the effect somewhat. If there is less than 128KB of live data all three variants will allocate the same size nursery (256KB). We would therefore expect the average reduction/increase in garbage collections to be rather less than 30%, indeed the observed averages are -13.57% and +19.19%. For applications whose working set is less than 128KB all three variants will behave identically; we see this on a number of the smaller benchmarks.

In summary, the results clearly show that a dynamic overhead of one word imposes an unacceptable reduction in performance and available heap space particularly when the majority of heap objects are small at around 3 to 4 words in size.

## 8.3 Incremental Collector Performance

We now consider raw performance of the various incremental collectors in comparison with the baseline stop-and-copy collector, for a single generation. Note in particular that the `NSH` implementation is the unadjusted version - we do not correct for the dynamic space overhead when sizing the nursery.

For the two Baker variants, `BAK-A` and `BAK-B`, we report overheads for collectors *without* incremental stack scavengers. The stacks are scavenged in their entirety at the GC flip (see Section 7.1.4) – the addition of incremental stack scavengers would further degrade the performance of the collectors that already incur the highest overhead. All other collectors employ incremental stack scavengers. The results are presented in Table 8.4.

It is clear from the table that something quite unusual is going on. Averaging over all 36 benchmarks some incremental `-B` and time-based schemes are actually running *faster*

| Application | REF (s) | REF* (%) | BAK-A (%) | BAK-B (%) | NSH-A (%) | NSH-B (%) | SPS-A (%) | SPS-B (%) | SPI-A (%) | SPI-B (%) | SPS-1ms (%) | SPS-10ms (%) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| anna | 151.83 | +0.69 | +249.26 | +176.60 | +2.28 | +0.49 | +3.86 | +0.79 | +2.50 | +0.39 | +1.16 | +0.45 |
| circsim | 44.81 | –0.58 | +66.37 | +56.04 | +12.23 | +3.70 | +20.31 | +11.19 | +20.55 | +11.09 | –1.79 | –6.67 |
| lcss | 49.28 | –0.13 | +54.97 | +39.24 | +4.53 | –5.90 | +8.71 | –6.15 | +7.66 | –6.26 | –17.74 | –23.48 |
| pic | 4600.13 | –1.13 | –93.53 | –95.14 | –93.53 | –95.83 | –94.02 | –96.19 | –93.96 | –96.24 | –97.00 | –97.05 |
| scs | 41.01 | +1.35 | +107.76 | +101.31 | +86.30 | +76.34 | +68.40 | +59.34 | +67.30 | +59.20 | +63.29 | +56.00 |
| symalg | 74.18 | +0.45 | –64.89 | –65.02 | –65.06 | –65.29 | –65.22 | –65.37 | –65.25 | –65.34 | –19.62 | –20.51 |
| wang | 59.98 | +0.62 | +136.12 | +46.35 | +16.47 | +9.02 | +18.65 | +8.20 | +17.07 | +8.24 | –3.07 | –14.58 |
| wave4main | 457.91 | +0.31 | +107.72 | –47.70 | +68.39 | –76.20 | +48.72 | –77.52 | +47.13 | –77.55 | –43.69 | –68.84 |
| x2n1 | 46.23 | –0.81 | +48.9 | +32.29 | –0.82 | –10.94 | +22.06 | –3.22 | +24.20 | –4.09 | +2.53 | +1.51 |
| Min 36 | | –3.69 | –93.53 | –95.14 | –93.53 | –95.83 | –94.02 | –96.19 | –93.96 | –96.24 | –97.00 | –97.05 |
| Max 36 | | +11.01 | +249.268 | +204.94 | +92.22 | +116.73 | +139.77 | +116.73 | +137.83 | +115.28 | +63.29 | +56.00 |
| ALL 36 | | +1.49 | +47.22 | +28.89 | –1.12 | –12.78 | +4.73 | –9.88 | +2.48 | –10.22 | –8.16 | –12.90 |

Table 8.4: Single-generation incremental collector performance as an overhead on baseline stop-and-copy (REF)

than stop-and-copy. This can be observed for the individual benchmarks `lcss`, `pic`, `symalg`, `wave4main`, and `x2n1`. Closer inspection of the figures shows that some applications (`pic` and `symalg`) exhibit substantial speed-*ups* (20-97%) across almost all incremental variants. There are also significant outliers in the other direction (`scs`, for example). Curiously, `wave4main` exhibits extreme speed-*ups* across all `-B` variants and extreme slow-*downs* across all `-A` variants!

Because the effects are seen across the board it suggests that it is an artefact of the incremental nature of the garbage collector, rather than the idiosyncrasies of any particular implementation. Examination of the Valgrind cache traces reveals no obvious trend that might explain the differences.

These extreme behaviours are inherent aspects of our incremental collector's dynamic heap sizing policy, and in particular the (re)sizing of the nursery. Unlike a stop-and-copy collector, collection no longer happens instantaneously with respect to mutator execution. At the end of stop-and-copy collection the nursery is resized using a heuristic based on the function of the amount of live data, as determined, in this *current* cycle (Section 6.1.6 and Section 8.2). The semi-space collector simply uses a constant scaling factor, defaulting to 2. The incremental collectors must, of course, resize the nursery at the beginning of the collection cycle and as a result cannot use heuristics based on the amount of live data for the current cycle, since it has not yet been determined. Instead the heuristics are based on the amount of live data, as determined, in the *previous* collection cycle. Extreme speed-ups are observed for benchmarks where the amount of live data during the previous collection cycle was significantly greater than that of the current cycle, or where a large proportion of the working set dies before the next collection cycle starts. In these cases, the nursery is sized larger than for its stop-and-copy counterpart. Not only does this allow the mutator to execute for longer, thus delaying the next garbage collection cycle, but it can also reduce the number of collection cycles and the number of bytes copied. This can have a significant effect on execution time.

In [CFS+04] we attributed the extreme slow-downs to be an artefact of incremental collection cycles that are *forced to completion*. If the sizing policy does not size the nursery large enough for the collection of from-space to complete before the nursery is again exhausted, then the current cycle is forced to completion and a new cycle is started. This can result in the incremental collector doing significantly more work than the stop-and-copy collector.

In a work-based collector, one way to remedy this is to adjust the amount of work done at each allocation on the basis of forced completions in previous collector cycles. However, as GHC has a block-based allocator, the preferred solution is to allocate one or more extra blocks to the nursery as needed. Only when the total available memory is truly (close to being) exhausted and no more blocks are available must the collection cycle be forced to completion. This is exactly what is experienced by our previously observed outlier, `scs`. In such a case, it is more than likely that the program will terminate with an out of memory error and, as such, enforcing real-time bounds is rather moot. In essence, the physical resources available, in conjunction with the real-time parameterisation of the collector, are insufficient to meet

the user-specified real-time constraints. This block-chaining modification is straightforward and has been applied to all the incremental collector implementations (Section 7.3).

Intriguingly `wave4main` exhibits both of these traits. The stop-and-copy collector copies 7.3 gigabytes of data during 290 collection cycles, while the `NSH-A` collector copies a further 90%, and the remaining `-A` collectors a further 45%, over the stop-and-copy collector. In contrast, the `-B` and time-based collectors all copy around 80% less data than the stop-and-copy collector.

The incremental collection cycles of `wave4main` are long and span 98% of the total execution time. The working set is large, and remains so for the duration of program execution, occupying on average well over 60% of the available heap. In the work-based collectors, triggered by object allocation, bursty allocation results in frequent invocation of the scavenger and free blocks are consumed through the *standard* chaining of blocks onto to-space. Fewer blocks are therefore available for nursery expansion via block-chaining, and cycles must be forced to completion — of 452 cycles 219 are forced to completion. This increases the number of collection cycles and the amount of data that must be copied. In the work-based collectors, triggered by each block allocation, or, to a lesser extent, at the collector quantum of the time-based collectors, the scavenger is invoked much less frequently than at every allocation and as a result the rate at which blocks are chained on to to-space is much slower, making more blocks available for nursery expansion, if required. As a result, fewer cycles are now forced to completion. This, in combination with the dynamic resizing policy of the nursery, significantly reduces the amount of data copied during garbage collection. The work-based `-B` collector performs 86 collection cycles, none of which are forced to completion. The 10 millisecond time-based collector performs 123 collection cycles of which only 16 are forced to completion.

Note that `NSH` performs well (compare `x2n1`'s `NSH` results with those of the other incremental collectors) because the sizing policy incorporates the extra word in the live data calculations. In practice, the collector would never be chosen over an `SPS` or `SPI` collector, instead, the sizing factor, a configurable parameter to the collector, would be adjusted to provide equivalent performance.

## 8.4 Incremental Generational Collectors

We now investigate the raw performance of the various collectors with two generations and two steps per generation (GHC's current default configuration). This is of interest as it indicates the bottom line on performance that we would expect to see in a "production" environment. The results are shown in Table 8.5; the performance of each variant is expressed as an overhead on the execution time observed using GHC's baseline generational collector. Once again the results suggest that there is no advantage at all to inlining entry code in each variant. Indeed the mutator overhead that accompanies the code bloat often leads to worse performance when compared to `SPS` that shares a single copy of each closures' original entry code.

An interesting observation is that for the work-based collectors, the extreme behaviour we observed in the single-generation collectors is much diminished. Recall that the semi-space collector resizes the nursery by a constant factor dependent on the amount of live data, and by default just doubles it. The generational collector, however, applies such scaling factors to the older generations, but fixes the size of the nursery to 256KB (modulo our block chaining to prevent the forced-completion of cycles) so that the collection cycles are relatively small. For almost all the benchmarks (`lcss` is the exception) this all but eliminates the pathological speed-*ups*. Furthermore, the long-lived data is promoted to the older generation more quickly where garbage collection is less frequent; the extreme effects we saw earlier are altogether much less apparent. This applies to a much lesser extent to the time-based collectors where the speed-*ups* remain, although they are less significant. It is important to recognise that the chaining of extra blocks onto the nursery to prevent the forced-completion of cycles can also be a contributing factor in the speed-ups of the incremental collectors over their stop-the-world counterparts. The addition of the extra blocks provide additional headroom for the application to complete in fewer collection cycles, thus reducing the amount of collector work. Of course, the converse is also true and the extra blocks may result in an eventual increase in the collector workload, thus contributing to the slow-down. This results from fewer blocks being available for chaining at the next collection cycle and if an insufficient number are available, the collection cycle must be forced to completion.

| Application | REF (s) | REF* (%) | BAK-A (%) | BAK-B (%) | NSH-A (%) | NSH-B (%) | SPS-A (%) | SPS-B (%) | SPI-A (%) | SPI-B (%) | SPS-1ms (%) | SPS-10ms (%) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| anna | 138.55 | +1.13 | +100.31 | +89.09 | +1.84 | +0.94 | +1.10 | –0.61 | –0.09 | –0.74 | +1.78 | +1.17 |
| circsim | 40.32 | +0.12 | +37.64 | +34.87 | +22.34 | +13.46 | +13.07 | +6.14 | +12.29 | +6.55 | +6.94 | –6.80 |
| lcss | 54.79 | +3.31 | +46.58 | +35.91 | +6.04 | –10.49 | +0.73 | –18.04 | +0.63 | –18.30 | –27.35 | –40.72 |
| pic | 103.47 | +1.49 | +9.62 | +6.86 | +5.52 | +13.00 | +0.97 | +4.64 | +0.53 | +4.37 | –12.98 | –17.80 |
| scs | 29.35 | +0.15 | +46.21 | +44.08 | +10.57 | +6.38 | +8.27 | +4.18 | +7.11 | +3.47 | +10.51 | +4.23 |
| symalg | 26.36 | +0.17 | +33.86 | –2.47 | +9.11 | –1.13 | +7.83 | –1.56 | +7.96 | –1.42 | +9.04 | +6.86 |
| wang | 96.18 | -1.00 | +29.17 | +23.65 | +22.14 | +15.85 | +8.34 | +2.16 | +7.38 | +2.62 | –40.09 | –32.51 |
| wave4main | 178.25 | +0.66 | +17.07 | +34.39 | +4.52 | +7.89 | +3.25 | +3.86 | +2.18 | +4.20 | –13.39 | –18.09 |
| x2n1 | 222.39 | –1.66 | +3.10 | +6.86 | +3.32 | +1.43 | +2.25 | +1.47 | +1.92 | +1.41 | +0.62 | –4.22 |
| Min 36 | | –2.38 | +0.00 | –2.47 | –0.54 | –10.49 | –6.86 | –18.04 | –7.28 | –18.30 | –51.19 | –55.21 |
| Max 36 | | +9.91 | +100.31 | +89.09 | +33.41 | +24.22 | +29.95 | +19.99 | +29.64 | +19.87 | +36.01 | +28.56 |
| ALL 36 | | +1.57 | +31.30 | +29.54 | +11.40 | +7.57 | +7.70 | +4.74 | +7.25 | +5.08 | –5.23 | –5.58 |

Table 8.5: The Bottom Line: Incremental generational collector performance as an overhead on execution time compared to GHC's current generational collector (REF)

## 8.5 Pause Times and Mutator Progress

Chapter 5 demonstrated the mean pause times for both per-object and per-block (`-A` and `-B`) work-based scavenging schemes to be very favourable over a range of benchmarks. However, while the individual pause granularity is important, it is the collector's MMU characteristics that give insight into its resilience to incremental bounds violation in the presence of pause aggregation. In this section we focus on the pause time distribution and mutator progress for a small subset of the benchmarks. To gather these we used PAPI [TICL] to obtain access to the hardware performance counters and produce a trace of the *wall clock* time for each pause. It is important to understand the measurements are wall-clock timings; in some rare cases the pause times include context switches. Also, some very small pauses are subject to measurement granularity. We are forced to use wall clock time, as the time quantum PAPI is able to multiplex for per-process "user" time is generally larger than the incremental collector pause times.

In this investigation, we focus on GHC's baseline stop-and-copy collector, and our incremental work-based and time-based `SPS` variants that employ specialisation and shared entry code. Initially, pause time distributions and (minimum) mutator utilisation plots were produced for the entire nofib benchmark suite. Broadly consistent trends emerge across the suite and so we have chosen to present the most collector-intensive benchmarks and those that exhibit behavioural extremes. Therefore, of our subset of nine benchmarks, we show here results for both single-generation (semi-space), and the "two step, two generation" runs of `anna`, `circsim`, `lcss`, `pic`, `wang`, `wave4main`, and `x2n1`. The plots are arranged horizontally in rows for comparisons and are distinguished using the legends '1 Gen' and '2 Gen' respectively.

We start by analysing the pause time distributions of each collector and examining the average minimum, mean, and maximum pause times. The longest pauses of between 9 and 13 seconds are incurred during execution of the stop-the-world semi-space collector for the benchmarks `pic` and `wang`. The mean of the longest pauses for this collector is around 4.8 seconds, the mean smallest pause around 5.2 milliseconds, and of the mean pause durations themselves around 0.56 seconds. In comparison, the longest individual pauses for the baseline generational collector again result for `pic` and `wang` with durations of 8.3 and 10.3 seconds respectively to perform major collections. The mean of the longest pauses for this collector is around 3.9 seconds, the mean smallest pause around 1.9 milliseconds, and of the mean pause durations themselves around 13.84 milliseconds. The generational collector not only reduces the total execution times, and the shortest and longest pause durations, but also reduces the mean pause time by an order of magnitude. These results are consistent with those reported extensively in the literature (Section 2.4.1) and enforce the effectiveness of generational garbage collection for functional programming languages where the weak generational hypothesis prevails.

Unsurprisingly, the `SPS-A` collector exhibits the best individual pause duration characteristics with a sub-microsecond mean for the smallest pause, a couple of microseconds for

the mean pause, and a mean maximum pause of between 11 and 12 microseconds for both the baseline semi-space and generational collectors. Following, is the `SPS-B` collector with a 91 microsecond mean minimum pause duration, a 137 microsecond mean pause, and a mean maximum pause of between 12 and 13 microseconds for both the semi-space and generational baseline collectors. The longest pause for both `SPS` variants was consistently recorded across the benchmark suite at just under 14 milliseconds.

Recall that PAPI is used to measure the pause times, i.e. the times spent in collector scavenging phases. For the time-based collectors, the minimum observed pause times are usually substantially shorter than the time quantum, as they record the time taken to perform the final (partial) scavenging phase at the end of a collection cycle. The maximum observed pause times are often rather longer than the time quantum. These maxima are typically recorded: i) at the start of a collection cycle (where the root set is flipped and general collector initialisation is performed); ii) at the end of a cycle (where finalizers, e.g. for weak pointers, are executed); iii) where a cycle is forced to completion as a result of physical memory exhaustion (for example, `scs` and `wave4main`).

Note also that, because the scheduler is non-preemptive, the time spent in both mutator and scavenging phases can vary from the stated quantum in some cases (see Section 7.2.3 and Section 7.3), although this does not generally affect the minimum or maximum pause times. If the mutator overruns its time quantum the next collector quantum can be shortened to compensate. The block chaining mechanism enables this flexibility by providing additional headroom if required. As a result, while individual quanta are subject to disruption, over the combined period of a single mutator and collector quantum, the ability to meet soft real-time constraints can be restored. This is particularly beneficial if the eventron model [SAB+06] of periodic real-time task scheduling is used, where the sum of collector and mutator quanta is configured to be the period between events, the eventron frequency.

The `SPS-1ms` time-based collector exhibits similar characteristics to the `SPS-B` collector, although the mean minimum pauses are slightly lower at 62 microseconds and 34 microseconds for the semi-space and generational collectors respectively. Similarly the mean longest pause for both is around 12 milliseconds, while the mean average pause time is greater at between 1.2 and 1.6 milliseconds, which is of course as expected for a collector operating with a time quantum of 1 millisecond (the deviation from precisely 1ms is explained above). Again, the longest pauses are recorded at just under 14 milliseconds.

Finally, the `SPS-10ms` time-based collector exhibits similar mean minimum pauses with a duration of 81 microseconds and 74 microseconds for the semi-space and generational collectors respectively. The mean maximum pauses are larger than expected at around 20 milliseconds for both while the average of the means is 10.2 and 9.2 milliseconds respectively. However, visual inspection of the graphs for both the `-1ms` and `-10ms` time-based collectors show a significant number of pauses clustered around the 1 millisecond and 10 millisecond buckets. The clustering results from the inability to precisely terminate the scavenger at the exact moment when the current time quantum expires. It is this late termination, in combi-

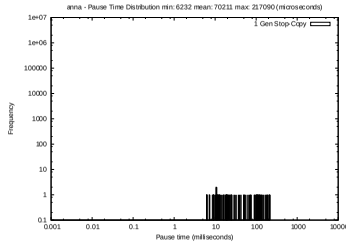
nation with the imprecise yielding by the mutator as a result of a non-preemtable scheduler that results in a soft real-time classification for our time-based collectors.

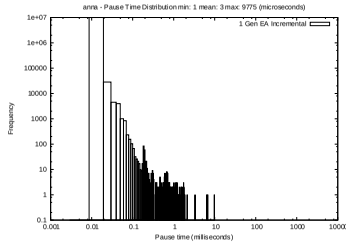
The overall trend when comparing the semi-space and generational collectors is that the mean pause times for the generational collector are very slightly longer, resulting from the additional complexity of generation management, but overall execution times are significantly shorter. While the pause time characteristics of the `SPS-A` collector appear to be exceptional, they come at the expense of significant overheads on both mutator execution times and extended collector cycle times. Furthermore, considering only the individual pause characteristics is deceptive as there can be disastrous effects on (minimum) mutator utilisation in the presence of pause aggregation resulting from bursty allocation and the forced-completion of cycles due to memory exhaustion.

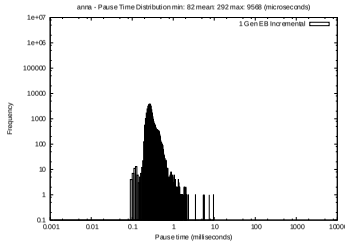
We have included the mutator utilisation plots beneath those of the pause distributions. They are really only useful for visual inspection and in confirming the characteristics of the bursty work-based collectors which rapidly fluctuate between as little as 0% and as much as 100% versus the more consistent utilisation of the time-based collectors. This is most clearly seen on the plots for `circsim`, `pic`, and `wang`. Where plots are missing the data collection process terminated abnormally, most usually due to the amount of data being generated. The results are, however, clear. The most important trend to note is that for the `-1ms` and `-10ms` time-based collectors, for the majority of the benchmarks' execution, the mutator utilisation is between 40% and 60% and is centred at around 50%. This is exactly as expected when running with equal mutator and collector time quanta.

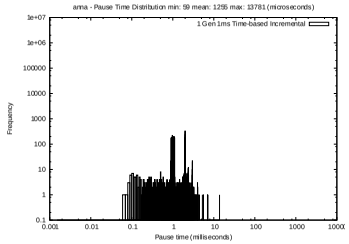
The minimum mutator utilisation (MMU) graphs are separated into plots for both the semi-space and generational collectors, each of which overlays the baseline stop-the-world collectors and each of the incremental work-based and time-based variants. These MMU graphs show the lowest average mutator utilisation seen over the entire program run, for any fixed length interval of time. On comparison of the two sets of plots, those of the semi-space collectors versus the generational collectors, clear trends emerge — the plots for `pic` are a good example. The minimum mutator utilisation of the semi-space collectors is almost always significantly better than their multi-generation counterparts. Furthermore, utilisation is greatest in `SPS-A` and least for the baseline collector. However, for both these trends this is as we would expect because short pause times lead to increased execution times. Both of these trends are, unsurprisingly, directly attributable to how the collector is scheduled: i) collector scavenging is often bursty; ii) the cycle times that result from incrementalising both minor and major collections are extended; iii) major collection cycles are occasionally triggered immediately after a minor collection. In combination, these result in incremental work-based MMUs that are less than 20% and often significantly below 10% within the 10 millisecond to 10 second window. Unlike the work-based collectors the time-based collectors universally show a consistent utilisation of 50% at around the 10 to 20 millisecond window. This is the major result of our work. Interestingly the MMU characteristics of the `SPS-1ms` and the `SPS-10ms` collectors are very similar — much closer than one would expect for the two.

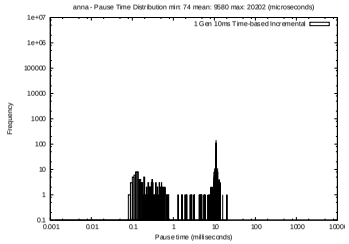
This demonstrates that smallest quantum that our time-based collectors are currently able to consistently achieve lies somewhere between 5 and 10 milliseconds. Tightening this range and targeting a quantum of exactly 1 millisecond, by enhancing the scheduling heuristics and reducing the granularity at which yield points are emitted, is the focus for the main body of future work.

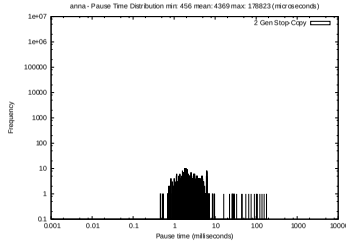

anna - Pause Time Distribution min: 6232 mean: 70211 max: 217090 (microseconds)
1 Gen Stop-Copy
Frequency
Pause time (milliseconds)

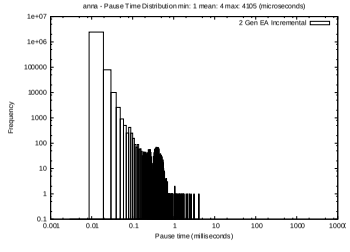

anna - Pause Time Distribution min: 1 mean: 3 max: 9775 (microseconds)
1 Gen EA Incremental
Frequency
Pause time (milliseconds)

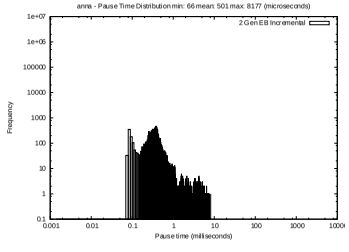

anna - Pause Time Distribution min: 82 mean: 292 max: 9568 (microseconds)
1 Gen EB Incremental
Frequency
Pause time (milliseconds)

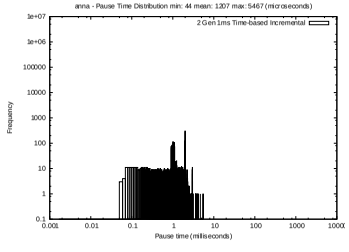

anna - Pause Time Distribution min: 59 mean: 1295 max: 13781 (microseconds)
1 Gen 1ms Time-based Incremental
Frequency
Pause time (milliseconds)

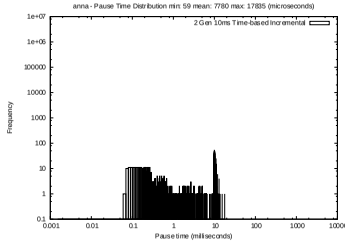

anna - Pause Time Distribution min: 74 mean: 9580 max: 20202 (microseconds)
1 Gen 10ms Time-based Incremental
Frequency
Pause time (milliseconds)

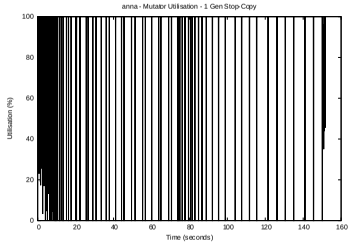

anna - Pause Time Distribution min: 456 mean: 4369 max: 178823 (microseconds)
2 Gen Stop-Copy
Frequency
Pause time (milliseconds)

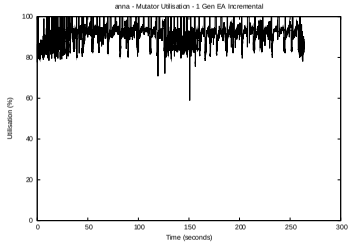

anna - Pause Time Distribution min: 1 mean: 4 max: 4305 (microseconds)
2 Gen EA Incremental
Frequency
Pause time (milliseconds)

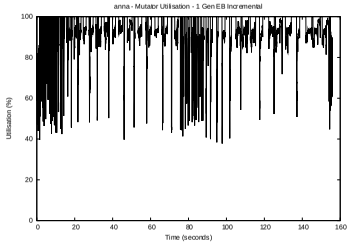

anna - Pause Time Distribution min: 66 mean: 501 max: 8177 (microseconds)
2 Gen EB Incremental
Frequency
Pause time (milliseconds)

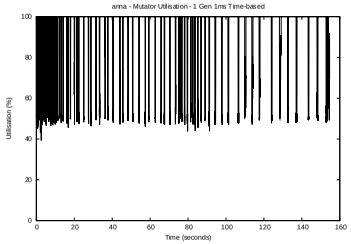

anna - Pause Time Distribution min: 44 mean: 1207 max: 5467 (microseconds)
2 Gen 1ms Time-based Incremental
Frequency
Pause time (milliseconds)

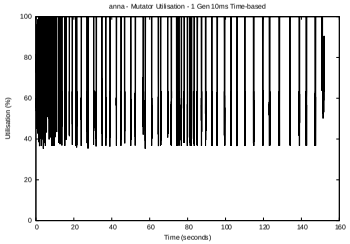

anna - Pause Time Distribution min: 59 mean: 7780 max: 17835 (microseconds)
2 Gen 10ms Time-based Incremental
Frequency
Pause time (milliseconds)

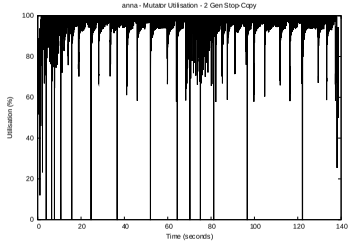

anna - Mutator Utilisation - 1 Gen Stop-Copy
Utilisation (%)
Time (seconds)

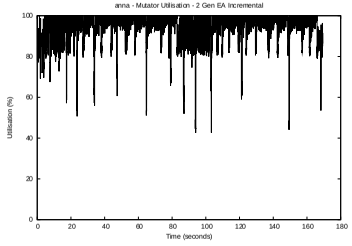

anna - Mutator Utilisation - 1 Gen EA Incremental
Utilisation (%)
Time (seconds)

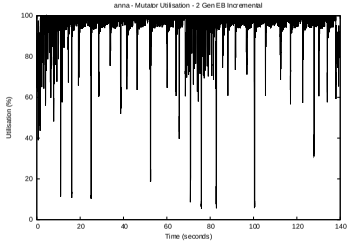

anna - Mutator Utilisation - 1 Gen EB Incremental
Utilisation (%)
Time (seconds)

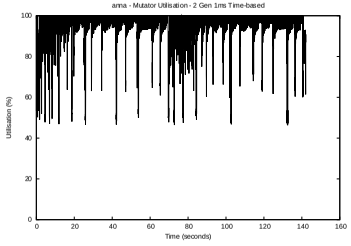

anna - Mutator Utilisation - 1 Gen 1ms Time-based
Utilisation (%)
Time (seconds)

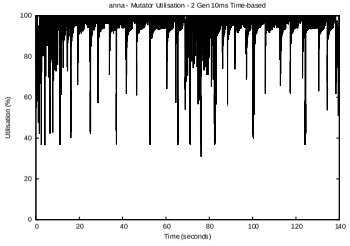

anna - Mutator Utilisation - 1 Gen 10ms Time-based
Utilisation (%)
Time (seconds)

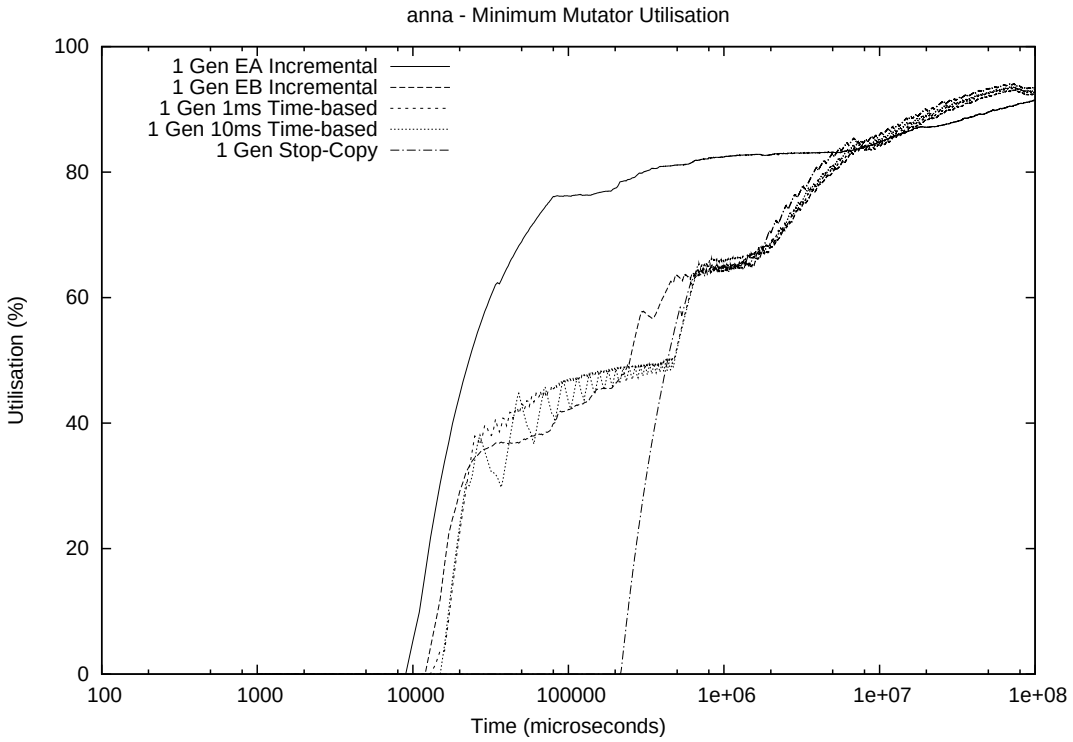

anna - Mutator Utilisation - 2 Gen Stop-Copy
Utilisation (%)
Time (seconds)

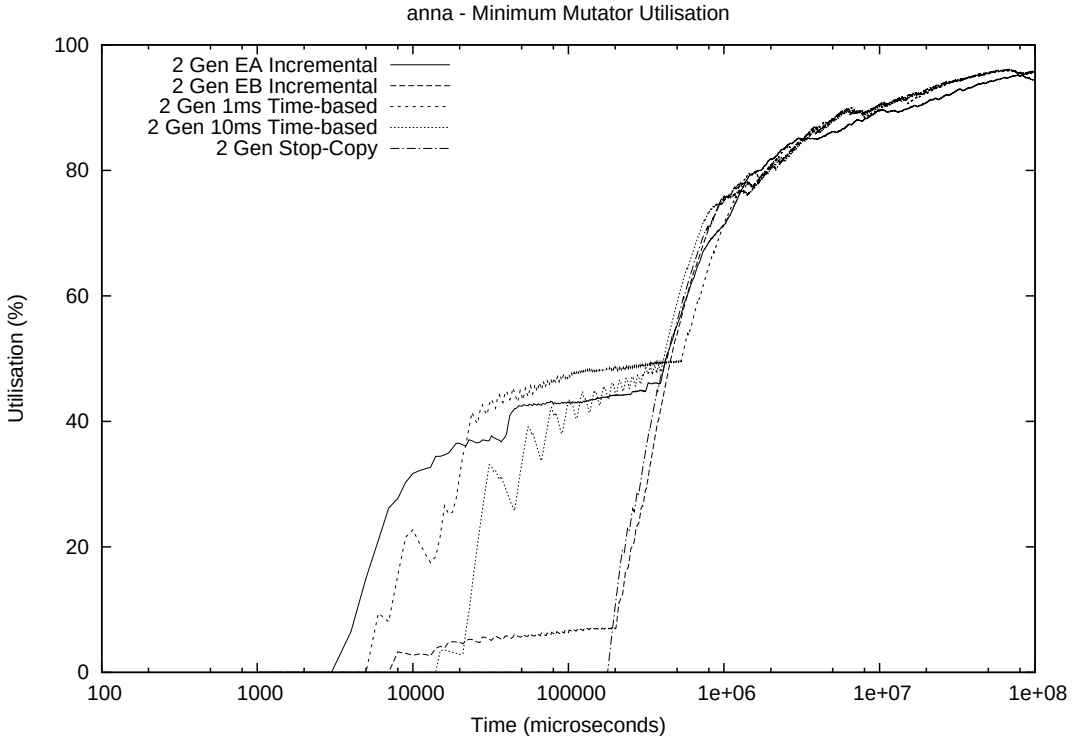

anna - Mutator Utilisation - 2 Gen EA Incremental
Utilisation (%)
Time (seconds)

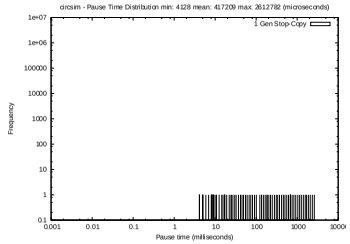

anna - Mutator Utilisation - 2 Gen EB Incremental
Utilisation (%)
Time (seconds)

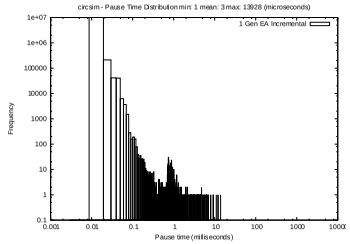

anna - Mutator Utilisation - 2 Gen 1ms Time-based
Utilisation (%)
Time (seconds)

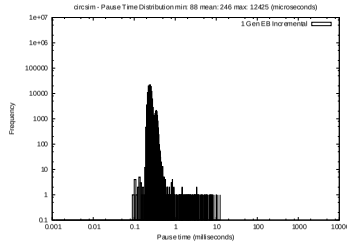

anna - Mutator Utilisation - 2 Gen 10ms Time-based
Utilisation (%)
Time (seconds)

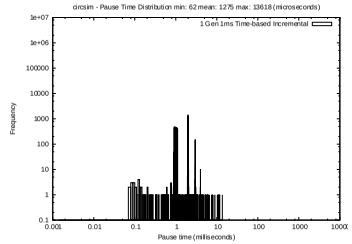

anna - Minimum Mutator Utilisation
1 Gen EA Incremental
1 Gen EB Incremental
1 Gen 1ms Time-based
1 Gen 10ms Time-based
1 Gen Stop-Copy
Utilisation (%)
Time (microseconds)
0
20
40
60
80
100
100
1000
10000
100000
1e+06
1e+07
1e+08

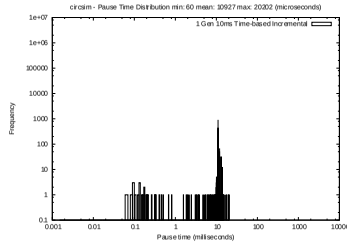

anna - Minimum Mutator Utilisation
2 Gen EA Incremental
2 Gen EB Incremental
2 Gen 1ms Time-based
2 Gen 10ms Time-based
2 Gen Stop-Copy
Utilisation (%)
Time (microseconds)
0
20
40
60
80
100
100
1000
10000
100000
1e+06
1e+07
1e+08

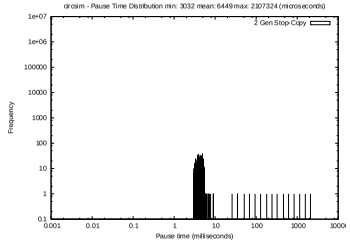

circsim - Pause Time Distribution min: 4128 mean: 417209 max: 2612782 (microseconds)
1 Gen Stop-Copy
Frequency
Pause time (milliseconds)

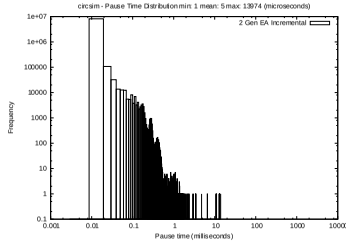

circsim - Pause Time Distribution min: 1 mean: 3 max: 13928 (microseconds)
1 Gen EA Incremental
Frequency
Pause time (milliseconds)

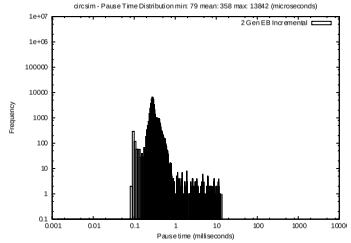

circsim - Pause Time Distribution min: 88 mean: 246 max: 12425 (microseconds)
1 Gen EB Incremental
Frequency
Pause time (milliseconds)

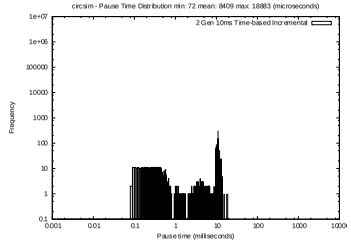

circsim - Pause Time Distribution min: 62 mean: 1275 max: 13618 (microseconds)
1 Gen 1ms Time-based Incremental
Frequency
Pause time (milliseconds)

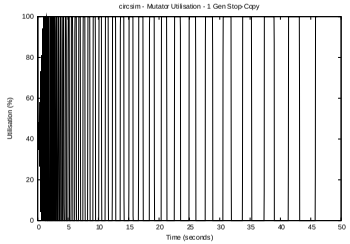

circsim - Pause Time Distribution min: 60 mean: 10927 max: 20202 (microseconds)
1 Gen 10ms Time-based Incremental
Frequency
Pause time (milliseconds)

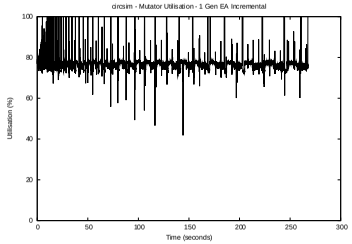

circsim - Pause Time Distribution min: 3032 mean: 6449 max: 2107324 (microseconds)
2 Gen Stop-Copy
Frequency
Pause time (milliseconds)

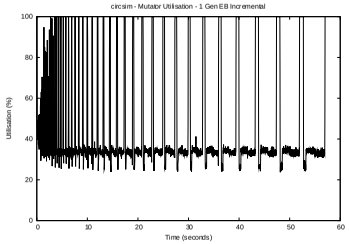

circsim - Pause Time Distribution min: 1 mean: 5 max: 13974 (microseconds)
2 Gen EA Incremental
Frequency
Pause time (milliseconds)

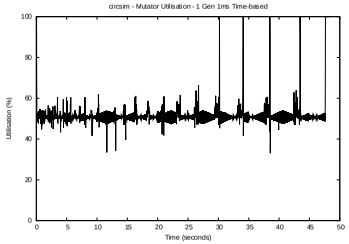

circsim - Pause Time Distribution min: 79 mean: 358 max: 13842 (microseconds)
2 Gen EB Incremental
Frequency
Pause time (milliseconds)

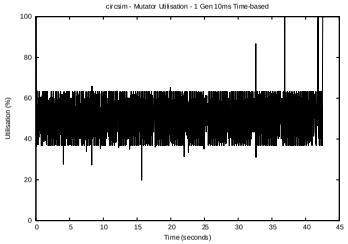

circsim - Pause Time Distribution min: 72 mean: 8409 max: 18883 (microseconds)
2 Gen 10ms Time-based Incremental
Frequency
Pause time (milliseconds)

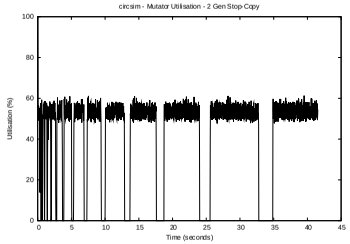

circsim - Mutator Utilisation - 1 Gen Stop-Copy
Utilisation (%)
Time (seconds)

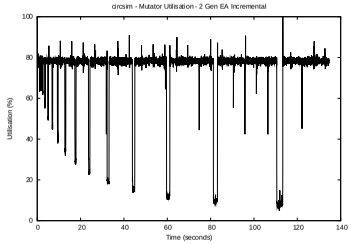

circsim - Mutator Utilisation - 1 Gen EA Incremental
Utilisation (%)
Time (seconds)

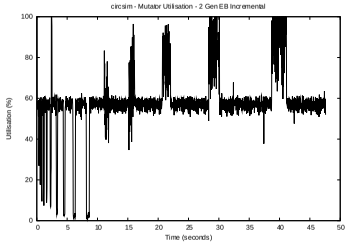

circsim - Mutator Utilisation - 1 Gen EB Incremental
Utilisation (%)
Time (seconds)

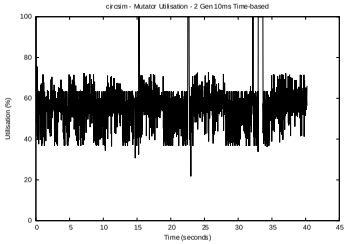

circsim - Mutator Utilisation - 1 Gen 1ms Time-based
Utilisation (%)
Time (seconds)

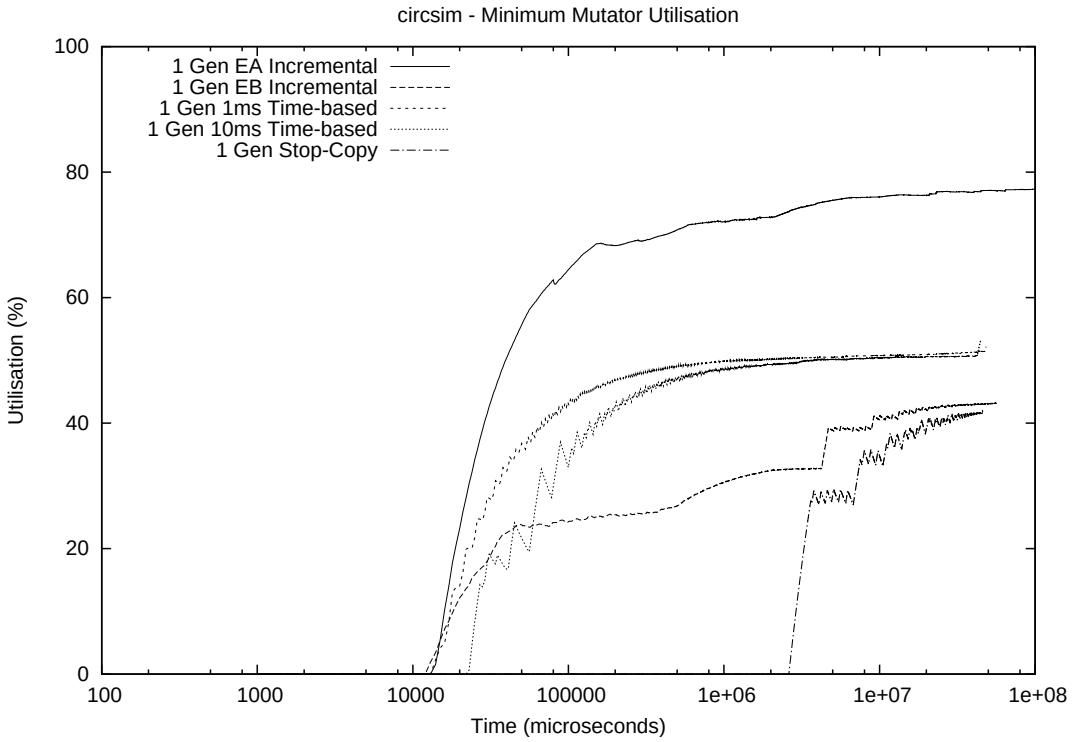

circsim - Mutator Utilisation - 1 Gen 10ms Time-based
Utilisation (%)
Time (seconds)

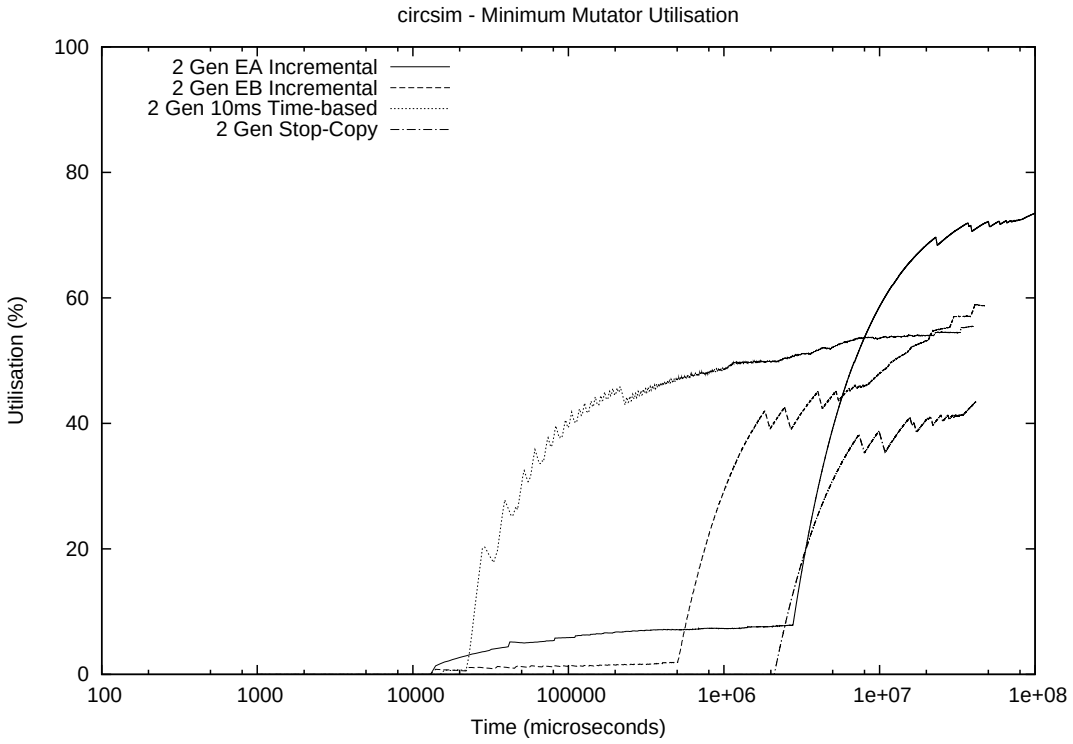

circsim - Mutator Utilisation - 2 Gen Stop-Copy
Utilisation (%)
Time (seconds)

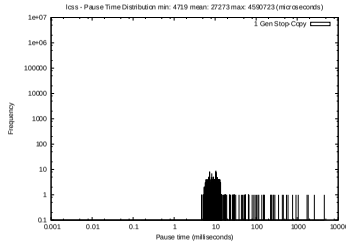

circsim - Mutator Utilisation - 2 Gen EA Incremental
Utilisation (%)
Time (seconds)

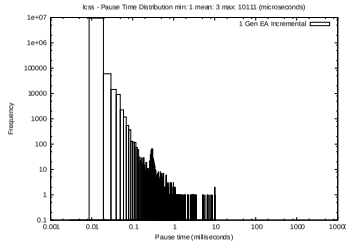

circsim - Mutator Utilisation - 2 Gen EB Incremental
Utilisation (%)
Time (seconds)

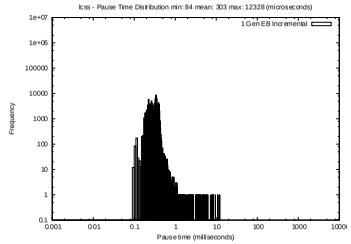

circsim - Mutator Utilisation - 2 Gen 10ms Time-based
Utilisation (%)
Time (seconds)

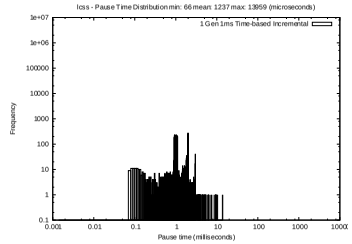

circsim - Minimum Mutator Utilisation
1 Gen EA Incremental
1 Gen EB Incremental
1 Gen 1ms Time-based
1 Gen 10ms Time-based
1 Gen Stop-Copy
Utilisation (%)
Time (microseconds)
0
20
40
60
80
100
100
1000
10000
100000
1e+06
1e+07
1e+08

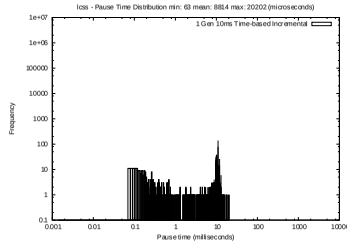

circsim - Minimum Mutator Utilisation
2 Gen EA Incremental
2 Gen EB Incremental
2 Gen 10ms Time-based
2 Gen Stop-Copy
Utilisation (%)
Time (microseconds)
0
20
40
60
80
100
100
1000
10000
100000
1e+06
1e+07
1e+08

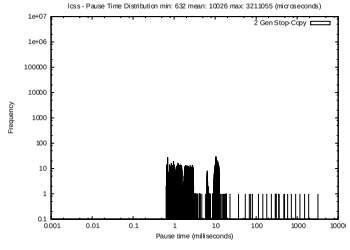

lcss - Pause Time Distribution min: 4719 mean: 27273 max: 4590723 (microseconds)
1 Gen Stop-Copy
Frequency
Pause time (milliseconds)

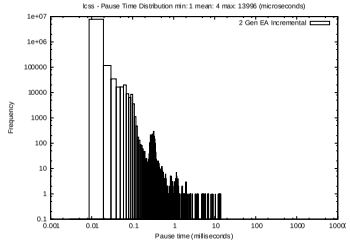

lcss - Pause Time Distribution min: 1 mean: 3 max: 10111 (microseconds)
1 Gen EA Incremental
Frequency
Pause time (milliseconds)

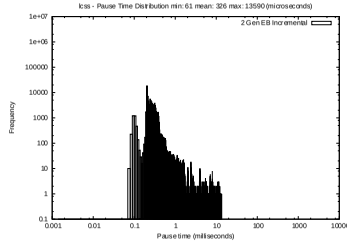

lcss - Pause Time Distribution min: 84 mean: 303 max: 12328 (microseconds)
1 Gen EB Incremental
Frequency
Pause time (milliseconds)

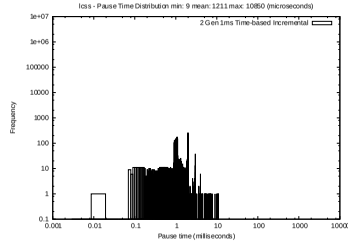

lcss - Pause Time Distribution min: 66 mean: 1237 max: 13959 (microseconds)
1 Gen 1ms Time-based Incremental
Frequency
Pause time (milliseconds)

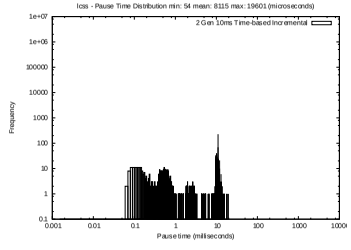

lcss - Pause Time Distribution min: 63 mean: 8814 max: 20202 (microseconds)
1 Gen 10ms Time-based Incremental
Frequency
Pause time (milliseconds)

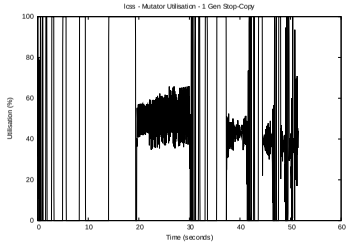

lcss - Pause Time Distribution min: 632 mean: 10026 max: 3211055 (microseconds)
2 Gen Stop-Copy
Frequency
Pause time (milliseconds)

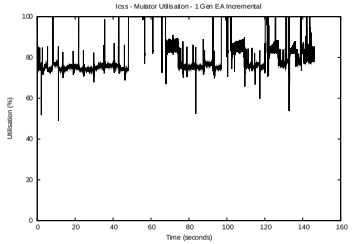

lcss - Pause Time Distribution min: 1 mean: 4 max: 13996 (microseconds)
2 Gen EA Incremental
Frequency
Pause time (milliseconds)

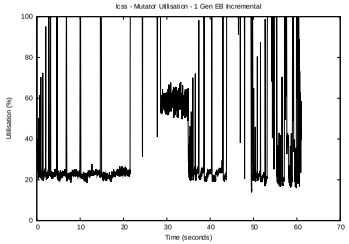

lcss - Pause Time Distribution min: 61 mean: 326 max: 13590 (microseconds)
2 Gen EB Incremental
Frequency
Pause time (milliseconds)

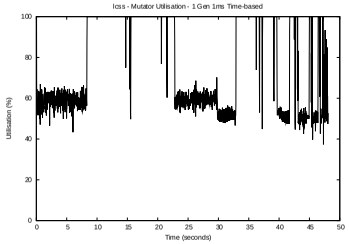

lcss - Pause Time Distribution min: 9 mean: 1211 max: 10850 (microseconds)
2 Gen 1ms Time-based Incremental
Frequency
Pause time (milliseconds)

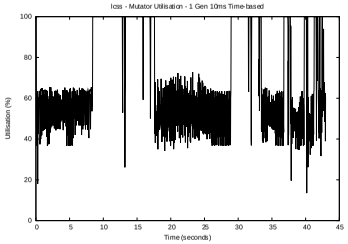

lcss - Pause Time Distribution min: 54 mean: 8115 max: 19601 (microseconds)
2 Gen 10ms Time-based Incremental
Frequency
Pause time (milliseconds)

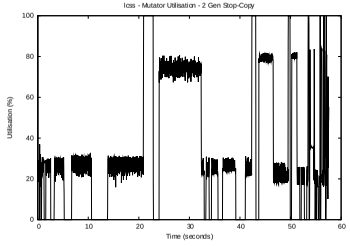

lcss - Mutator Utilisation - 1 Gen Stop-Copy
Utilisation (%)
Time (seconds)

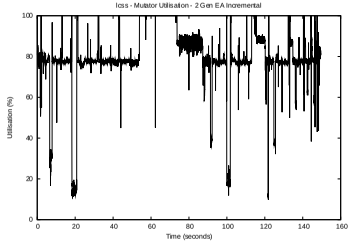

lcss - Mutator Utilisation - 1 Gen EA Incremental
Utilisation (%)
Time (seconds)

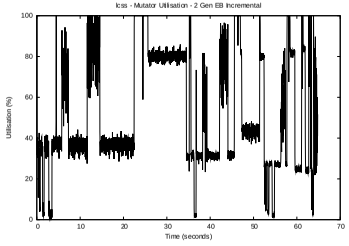

lcss - Mutator Utilisation - 1 Gen EB Incremental
Utilisation (%)
Time (seconds)

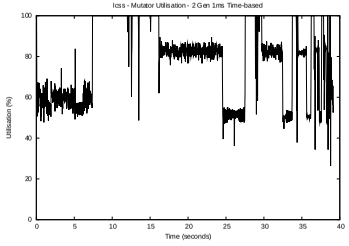

lcss - Mutator Utilisation - 1 Gen 1ms Time-based
Utilisation (%)
Time (seconds)

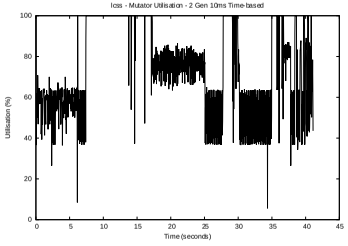

lcss - Mutator Utilisation - 1 Gen 10ms Time-based
Utilisation (%)
Time (seconds)

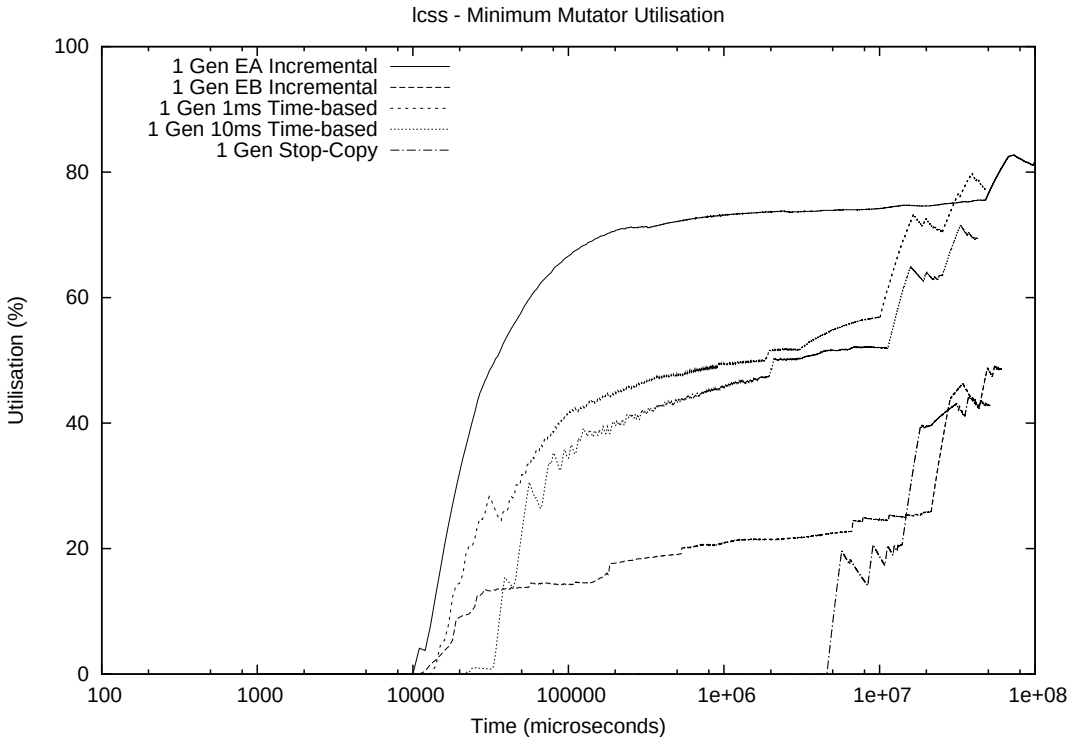

lcss - Mutator Utilisation - 2 Gen Stop-Copy
Utilisation (%)
Time (seconds)

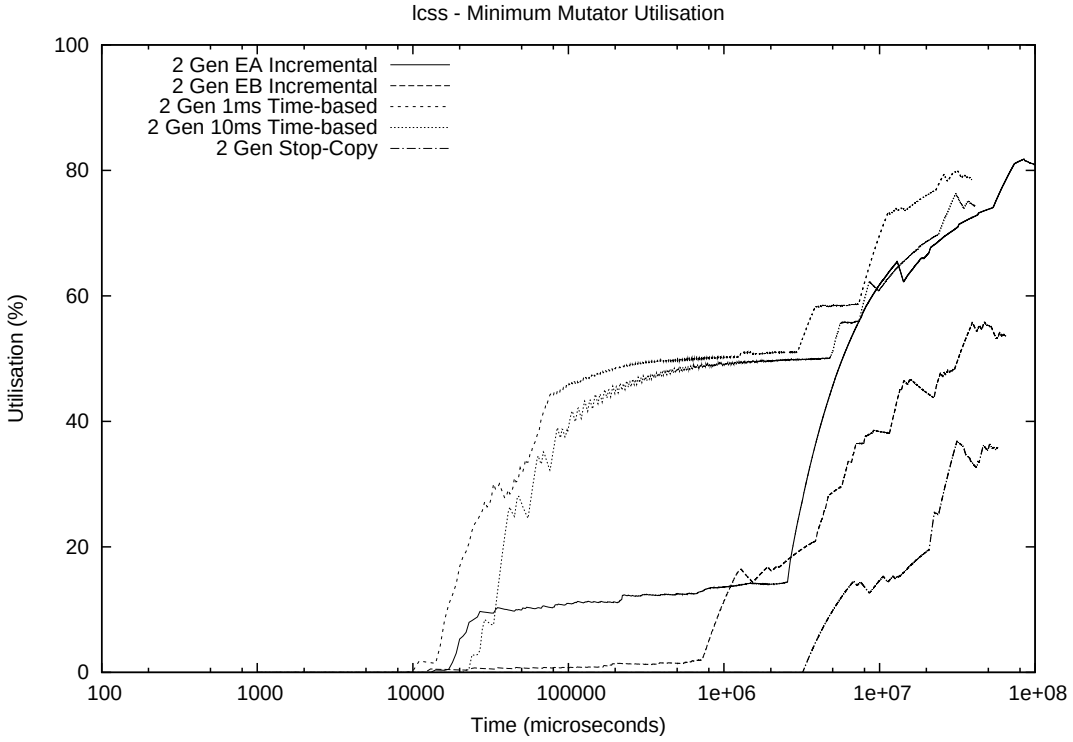

lcss - Mutator Utilisation - 2 Gen EA Incremental
Utilisation (%)
Time (seconds)

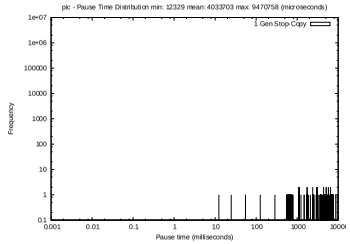

lcss - Mutator Utilisation - 2 Gen EB Incremental
Utilisation (%)
Time (seconds)

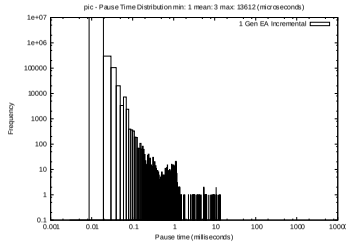

lcss - Mutator Utilisation - 2 Gen 1ms Time-based
Utilisation (%)
Time (seconds)

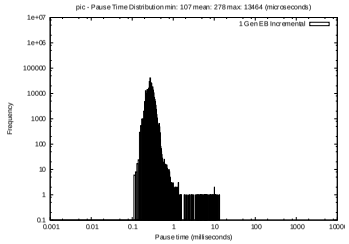

lcss - Mutator Utilisation - 2 Gen 10ms Time-based
Utilisation (%)
Time (seconds)

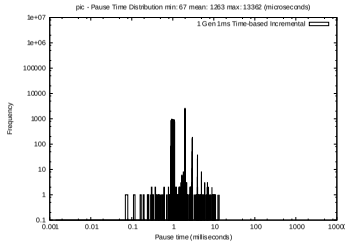

lcss - Minimum Mutator Utilisation
1 Gen EA Incremental
1 Gen EB Incremental
1 Gen 1ms Time-based
1 Gen 10ms Time-based
1 Gen Stop-Copy
Utilisation (%)
Time (microseconds)
0
20
40
60
80
100
100
1000
10000
100000
1e+06
1e+07
1e+08

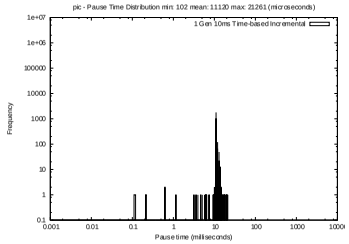

lcss - Minimum Mutator Utilisation
2 Gen EA Incremental
2 Gen EB Incremental
2 Gen 1ms Time-based
2 Gen 10ms Time-based
2 Gen Stop-Copy
Utilisation (%)
Time (microseconds)
0
20
40
60
80
100
100
1000
10000
100000
1e+06
1e+07
1e+08

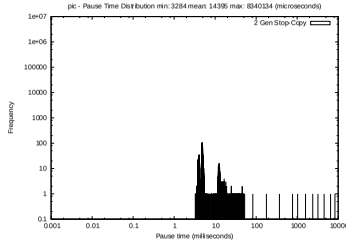

pic - Pause Time Distribution min: 12329 mean: 4033703 max: 9470758 (microseconds)
1 Gen Stop-Copy
Frequency
Pause time (milliseconds)

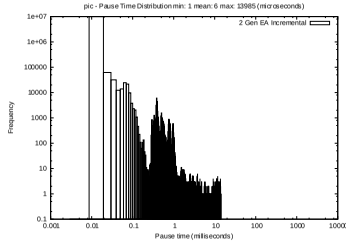

pic - Pause Time Distribution min: 1 mean: 3 max: 13612 (microseconds)
1 Gen EA Incremental
Frequency
Pause time (milliseconds)

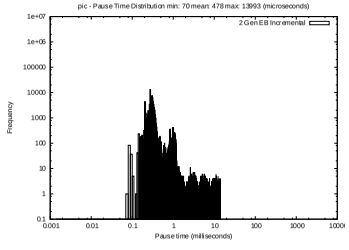

pic - Pause Time Distribution min: 107 mean: 278 max: 13464 (microseconds)
1 Gen EB Incremental
Frequency
Pause time (milliseconds)

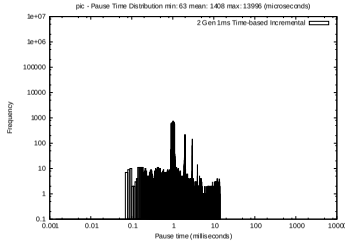

pic - Pause Time Distribution min: 67 mean: 1263 max: 13362 (microseconds)
1 Gen 1ms Time-based Incremental
Frequency
Pause time (milliseconds)

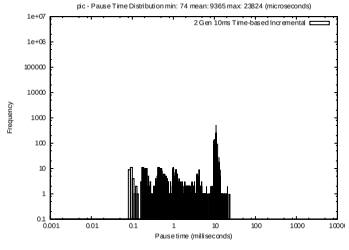

pic - Pause Time Distribution min: 102 mean: 11120 max: 21261 (microseconds)
1 Gen 10ms Time-based Incremental
Frequency
Pause time (milliseconds)

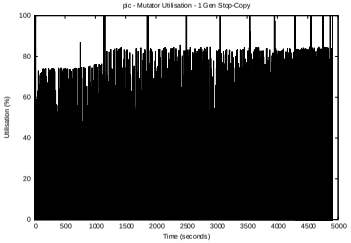

pic - Pause Time Distribution min: 3284 mean: 14395 max: 8340134 (microseconds)
2 Gen Stop-Copy
Frequency
Pause time (milliseconds)

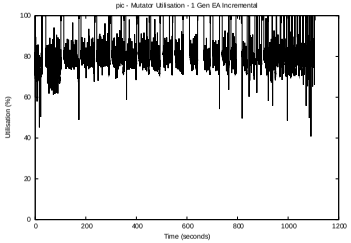

pic - Pause Time Distribution min: 1 mean: 6 max: 13985 (microseconds)
2 Gen EA Incremental
Frequency
Pause time (milliseconds)

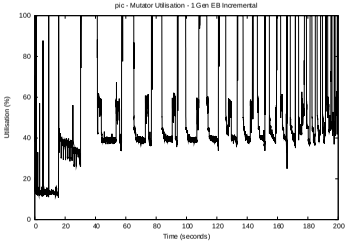

pic - Pause Time Distribution min: 70 mean: 478 max: 13993 (microseconds)
2 Gen EB Incremental
Frequency
Pause time (milliseconds)

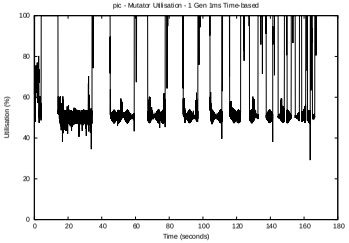

pic - Pause Time Distribution min: 63 mean: 1408 max: 13996 (microseconds)
2 Gen 1ms Time-based Incremental
Frequency
Pause time (milliseconds)

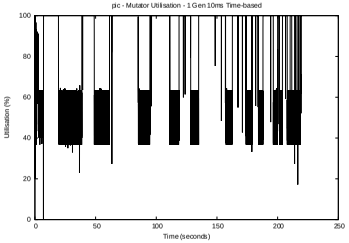

pic - Pause Time Distribution min: 74 mean: 9365 max: 23824 (microseconds)
2 Gen 10ms Time-based Incremental
Frequency
Pause time (milliseconds)

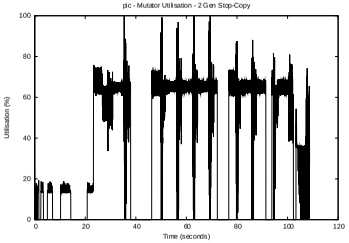

pic - Mutator Utilisation - 1 Gen Stop-Copy
Utilisation (%)
Time (seconds)

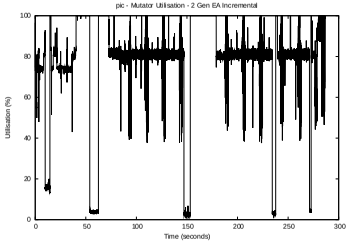

pic - Mutator Utilisation - 1 Gen EA Incremental
Utilisation (%)
Time (seconds)

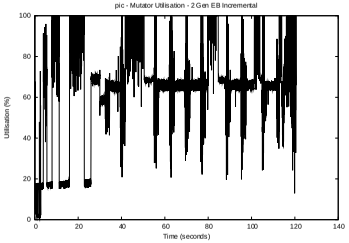

pic - Mutator Utilisation - 1 Gen EB Incremental
Utilisation (%)
Time (seconds)

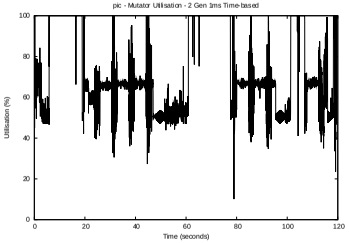

pic - Mutator Utilisation - 1 Gen 1ms Time-based
Utilisation (%)
Time (seconds)

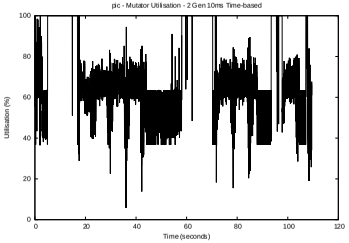

pic - Mutator Utilisation - 1 Gen 10ms Time-based
Utilisation (%)
Time (seconds)

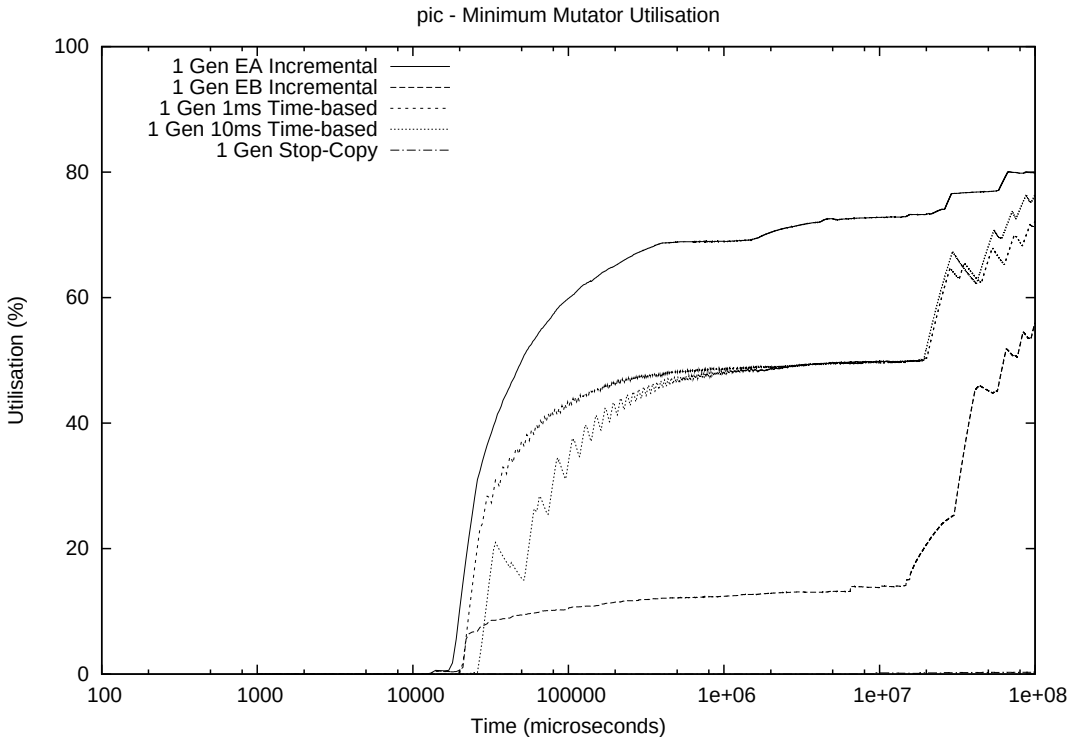

pic - Mutator Utilisation - 2 Gen Stop-Copy
Utilisation (%)
Time (seconds)

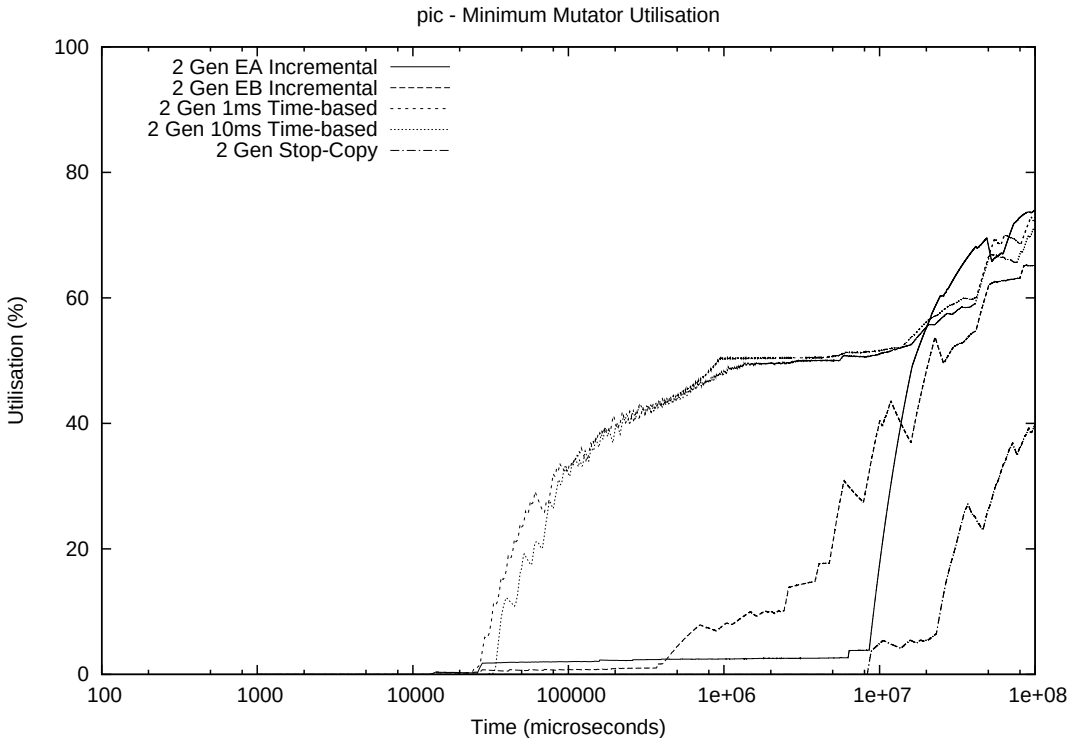

pic - Mutator Utilisation - 2 Gen EA Incremental
Utilisation (%)
Time (seconds)

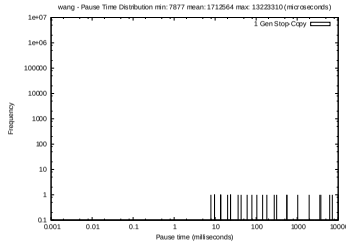

pic - Mutator Utilisation - 2 Gen EB Incremental
Utilisation (%)
Time (seconds)

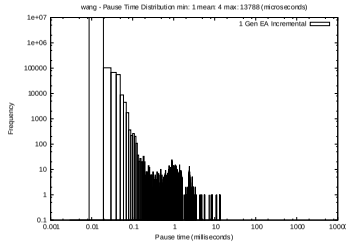

pic - Mutator Utilisation - 2 Gen 1ms Time-based
Utilisation (%)
Time (seconds)

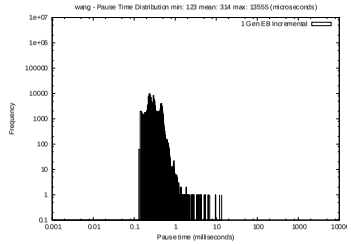

pic - Mutator Utilisation - 2 Gen 10ms Time-based
Utilisation (%)
Time (seconds)

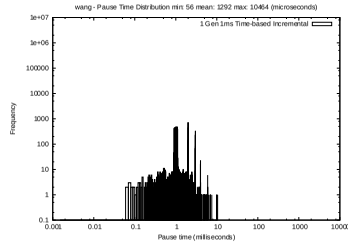

pic - Minimum Mutator Utilisation
1 Gen EA Incremental
1 Gen EB Incremental
1 Gen 1ms Time-based
1 Gen 10ms Time-based
1 Gen Stop-Copy
Utilisation (%)
Time (microseconds)
0
20
40
60
80
100
100
1000
10000
100000
1e+06
1e+07
1e+08

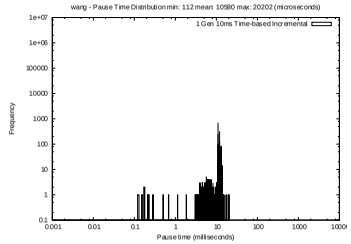

pic - Minimum Mutator Utilisation
2 Gen EA Incremental
2 Gen EB Incremental
2 Gen 1ms Time-based
2 Gen 10ms Time-based
2 Gen Stop-Copy
Utilisation (%)
Time (microseconds)
0
20
40
60
80
100
100
1000
10000
100000
1e+06
1e+07
1e+08

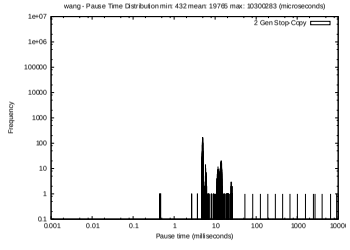

wang - Pause Time Distribution min: 7877 mean: 1712564 max: 13223310 (microseconds)
1 Gen Stop-Copy
Frequency
Pause time (milliseconds)

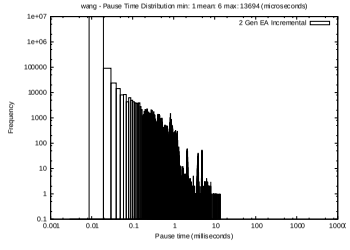

wang - Pause Time Distribution min: 1 mean: 4 max: 13788 (microseconds)
1 Gen EA Incremental
Frequency
Pause time (milliseconds)

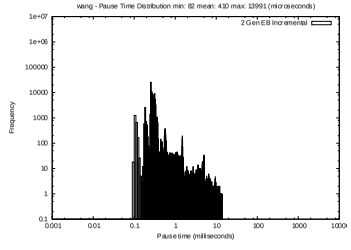

wang - Pause Time Distribution min: 123 mean: 314 max: 13555 (microseconds)
1 Gen EB Incremental
Frequency
Pause time (milliseconds)

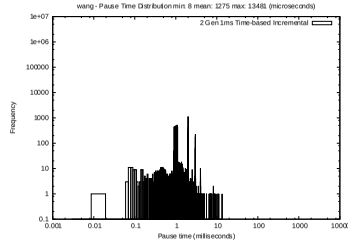

wang - Pause Time Distribution min: 56 mean: 1292 max: 10464 (microseconds)
1 Gen 1ms Time-based Incremental
Frequency
Pause time (milliseconds)

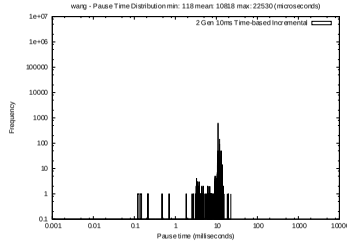

wang - Pause Time Distribution min: 112 mean: 10580 max: 20202 (microseconds)
1 Gen 10ms Time-based Incremental
Frequency
Pause time (milliseconds)

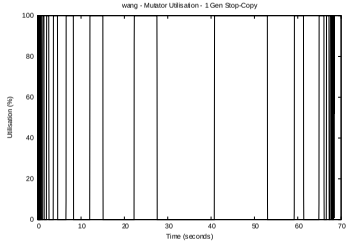

wang - Pause Time Distribution min: 432 mean: 19765 max: 10300283 (microseconds)
2 Gen Stop-Copy
Frequency
Pause time (milliseconds)

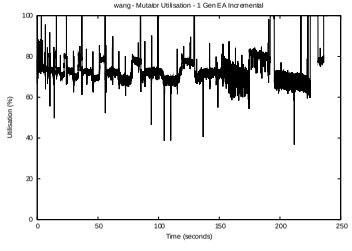

wang - Pause Time Distribution min: 1 mean: 6 max: 13694 (microseconds)
2 Gen EA Incremental
Frequency
Pause time (milliseconds)

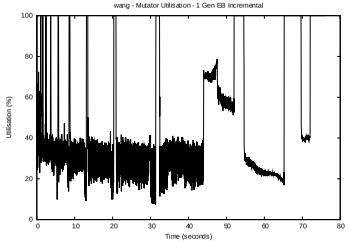

wang - Pause Time Distribution min: 82 mean: 410 max: 13991 (microseconds)
2 Gen EB Incremental
Frequency
Pause time (milliseconds)

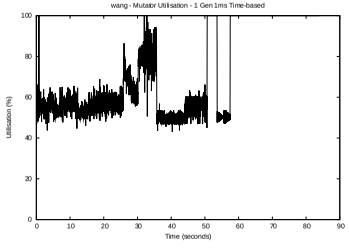

wang - Pause Time Distribution min: 8 mean: 1275 max: 13481 (microseconds)
2 Gen 1ms Time-based Incremental
Frequency
Pause time (milliseconds)

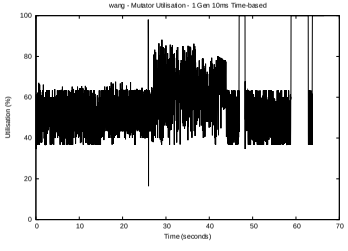

wang - Pause Time Distribution min: 118 mean: 10818 max: 22530 (microseconds)
2 Gen 10ms Time-based Incremental
Frequency
Pause time (milliseconds)

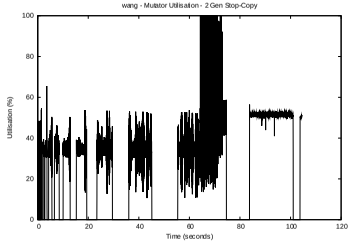

wang - Mutator Utilisation - 1 Gen Stop-Copy
Utilisation (%)
Time (seconds)

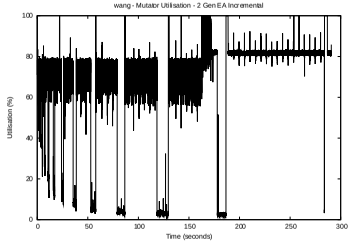

wang - Mutator Utilisation - 1 Gen EA Incremental
Utilisation (%)
Time (seconds)

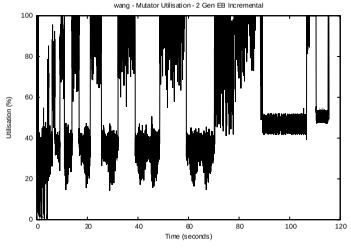

wang - Mutator Utilisation - 1 Gen EB Incremental
Utilisation (%)
Time (seconds)

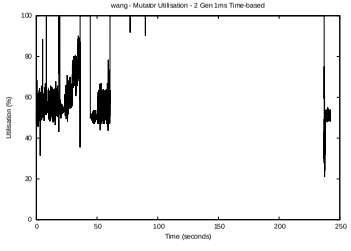

wang - Mutator Utilisation - 1 Gen 1ms Time-based
Utilisation (%)
Time (seconds)

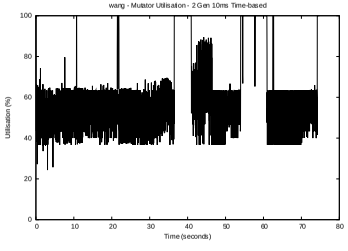

wang - Mutator Utilisation - 1 Gen 10ms Time-based
Utilisation (%)
Time (seconds)

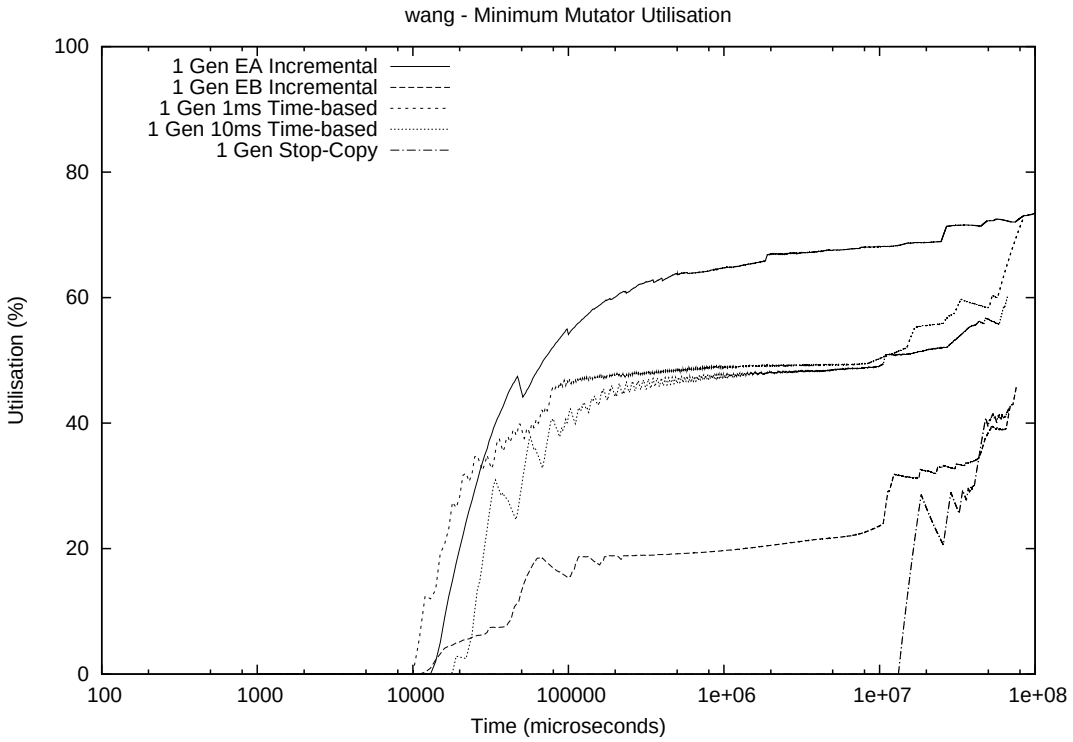

wang - Mutator Utilisation - 2 Gen Stop-Copy
Utilisation (%)
Time (seconds)

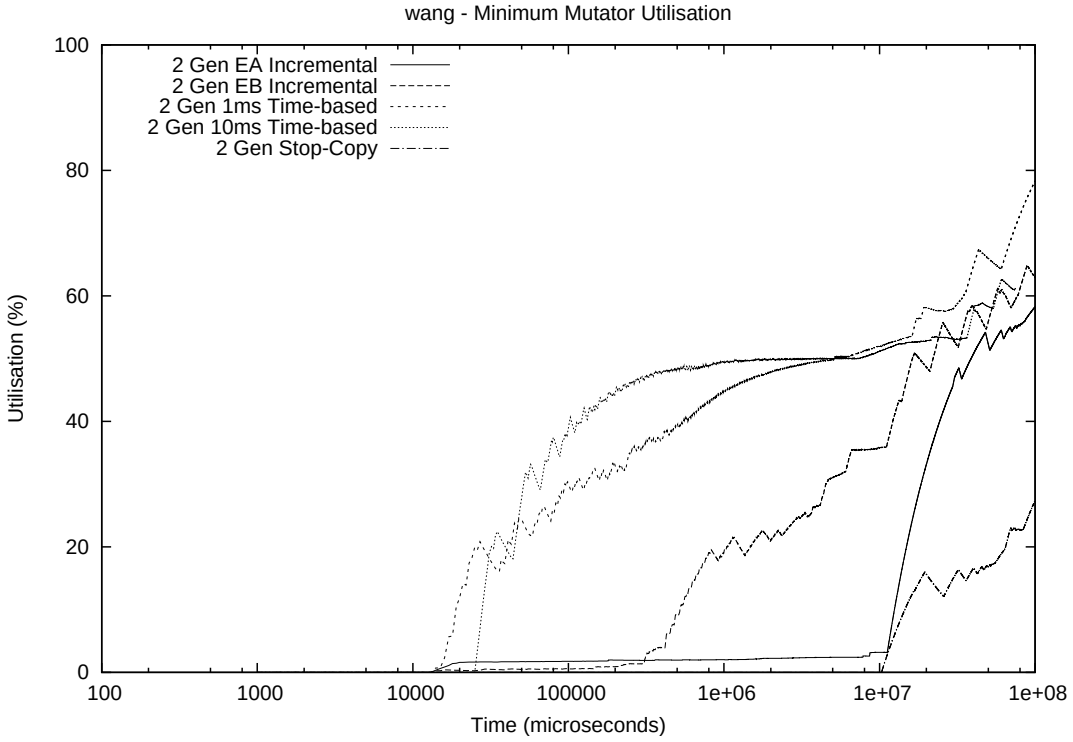

wang - Mutator Utilisation - 2 Gen EA Incremental
Utilisation (%)
Time (seconds)

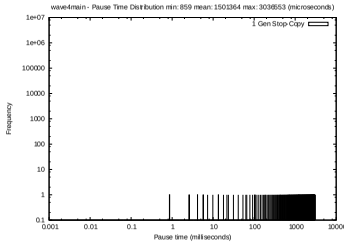

wang - Mutator Utilisation - 2 Gen EB Incremental
Utilisation (%)
Time (seconds)

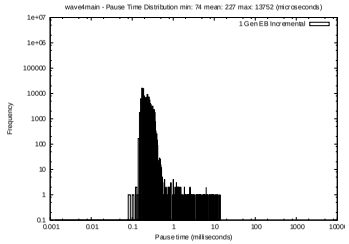

wang - Mutator Utilisation - 2 Gen 1ms Time-based
Utilisation (%)
Time (seconds)

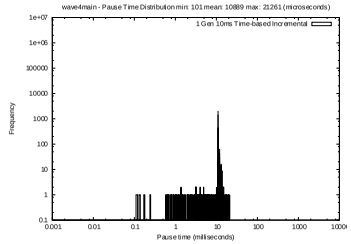

wang - Mutator Utilisation - 2 Gen 10ms Time-based
Utilisation (%)
Time (seconds)

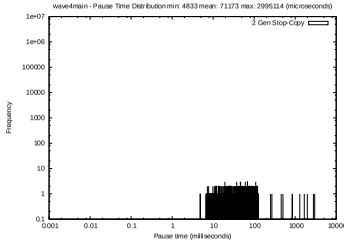

wang - Minimum Mutator Utilisation
1 Gen EA Incremental
1 Gen EB Incremental
1 Gen 1ms Time-based
1 Gen 10ms Time-based
1 Gen Stop-Copy
Utilisation (%)
Time (microseconds)
0
20
40
60
80
100
100
1000
10000
100000
1e+06
1e+07
1e+08

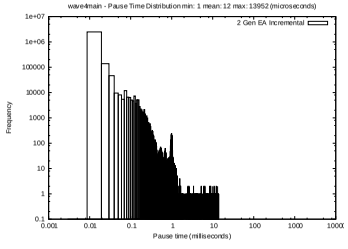

wang - Minimum Mutator Utilisation
2 Gen EA Incremental
2 Gen EB Incremental
2 Gen 1ms Time-based
2 Gen 10ms Time-based
2 Gen Stop-Copy
Utilisation (%)
Time (microseconds)
0
20
40
60
80
100
100
1000
10000
100000
1e+06
1e+07
1e+08

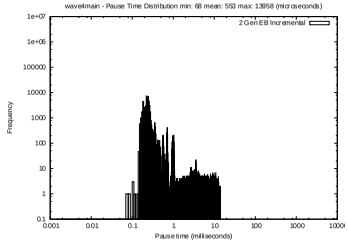

wave4main - Pause Time Distribution min: 859 mean: 1501364 max: 3036853 (microseconds)
1 Gen Stop-Copy
Frequency
Pause time (milliseconds)

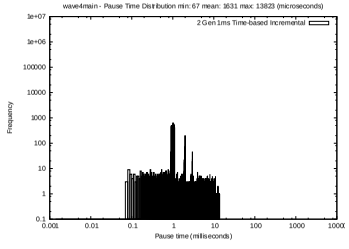

wave4main - Pause Time Distribution min: 74 mean: 227 max: 13752 (microseconds)
1 Gen EB Incremental
Frequency
Pause time (milliseconds)

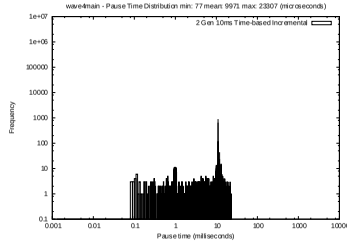

wave4main - Pause Time Distribution min: 101 mean: 10889 max: 21261 (microseconds)
1 Gen 10ms Time-based Incremental
Frequency
Pause time (milliseconds)

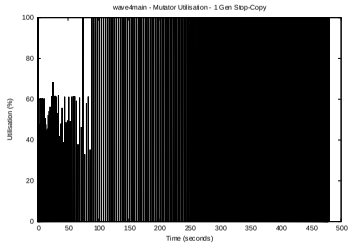

wave4main - Pause Time Distribution min: 4833 mean: 71173 max: 2995114 (microseconds)
2 Gen Stop-Copy
Frequency
Pause time (milliseconds)

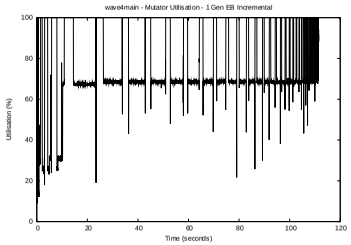

wave4main - Pause Time Distribution min: 1 mean: 12 max: 13952 (microseconds)
2 Gen EA Incremental
Frequency
Pause time (milliseconds)

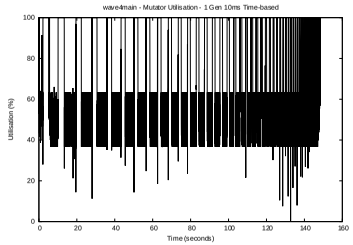

wave4main - Pause Time Distribution min: 68 mean: 553 max: 13958 (microseconds)
2 Gen EB Incremental
Frequency
Pause time (milliseconds)

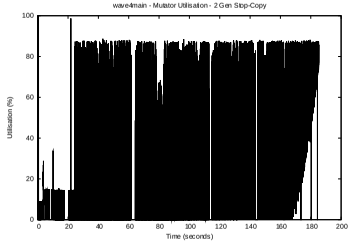

wave4main - Pause Time Distribution min: 67 mean: 1631 max: 13823 (microseconds)
2 Gen 1ms Time-based Incremental
Frequency
Pause time (milliseconds)

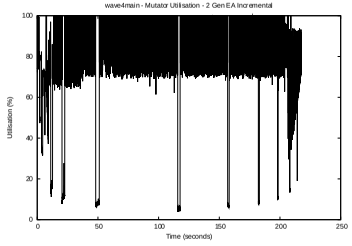

wave4main - Pause Time Distribution min: 77 mean: 9971 max: 23307 (microseconds)
2 Gen 10ms Time-based Incremental
Frequency
Pause time (milliseconds)

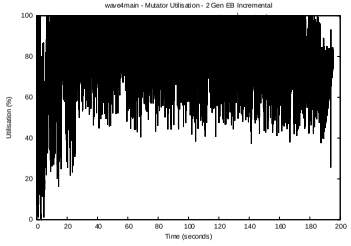

wave4main - Mutator Utilisation - 1 Gen Stop-Copy
Utilisation (%)
Time (seconds)

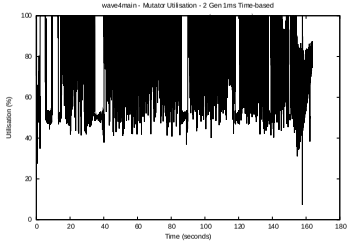

wave4main - Mutator Utilisation - 1 Gen EB Incremental
Utilisation (%)
Time (seconds)

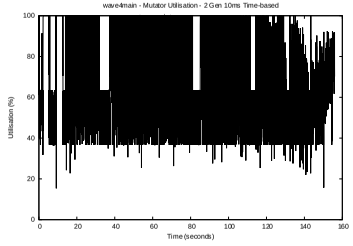

wave4main - Mutator Utilisation - 1 Gen 10ms Time-based
Utilisation (%)
Time (seconds)

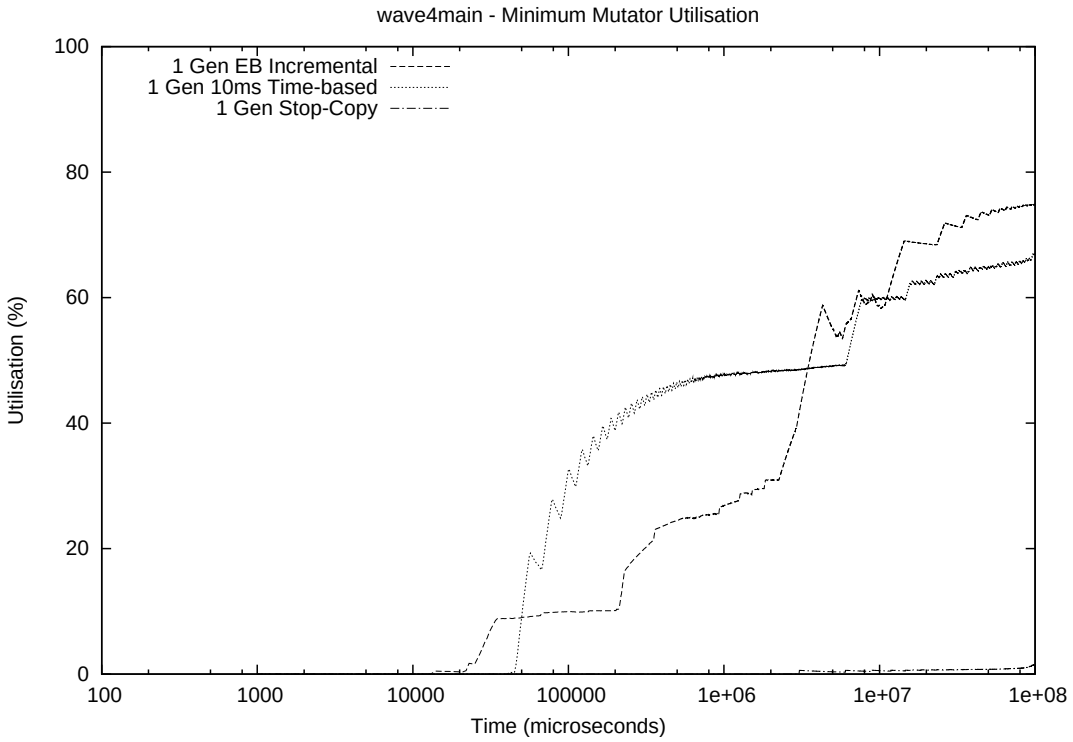

wave4main - Mutator Utilisation - 2 Gen Stop-Copy
Utilisation (%)
Time (seconds)

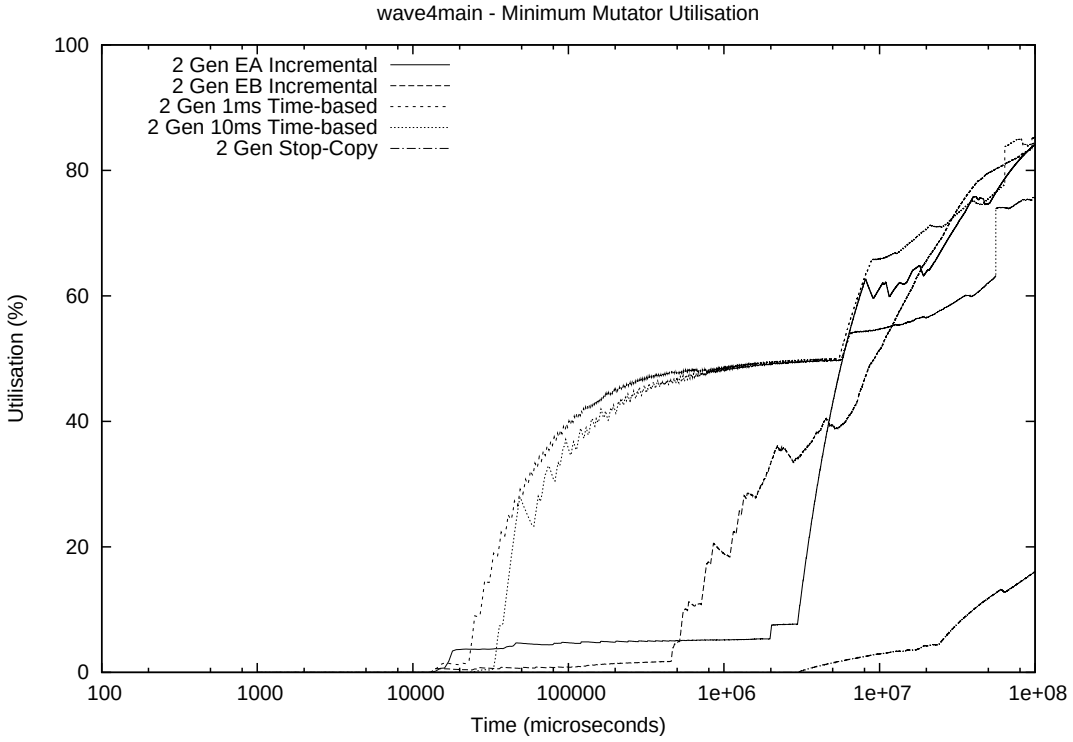

wave4main - Mutator Utilisation - 2 Gen EA Incremental
Utilisation (%)
Time (seconds)

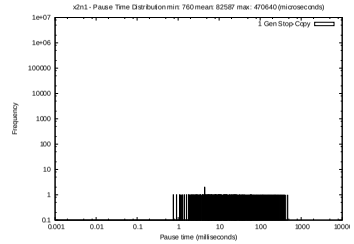

wave4main - Mutator Utilisation - 2 Gen EB Incremental
Utilisation (%)
Time (seconds)

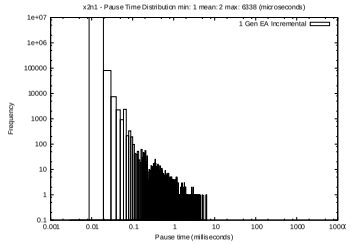

wave4main - Mutator Utilisation - 2 Gen 1ms Time-based
Utilisation (%)
Time (seconds)

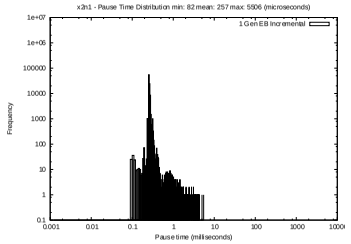

wave4main - Mutator Utilisation - 2 Gen 10ms Time-based
Utilisation (%)
Time (seconds)

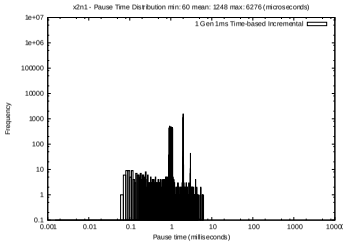

wave4main - Minimum Mutator Utilisation
1 Gen EB Incremental
1 Gen 10ms Time-based
1 Gen Stop-Copy
Utilisation (%)
Time (microseconds)
0
20
40
60
80
100
100
1000
10000
100000
1e+06
1e+07
1e+08

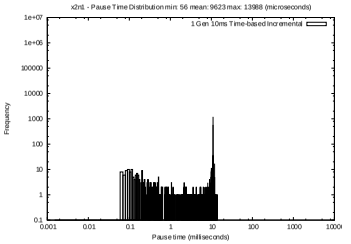

wave4main - Minimum Mutator Utilisation
2 Gen EA Incremental
2 Gen EB Incremental
2 Gen 1ms Time-based
2 Gen 10ms Time-based
2 Gen Stop-Copy
Utilisation (%)
Time (microseconds)
0
20
40
60
80
100
100
1000
10000
100000
1e+06
1e+07
1e+08

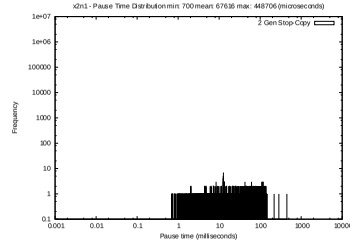

x2n1 - Pause Time Distribution min: 760 mean: 82587 max: 470640 (microseconds)
1 Gen Stop-Copy
Frequency
Pause time (milliseconds)

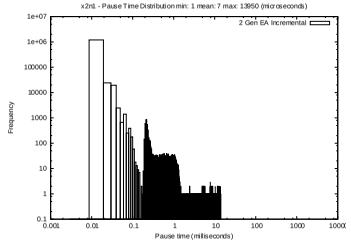

x2n1 - Pause Time Distribution min: 1 mean: 2 max: 6338 (microseconds)
1 Gen EA Incremental
Frequency
Pause time (milliseconds)

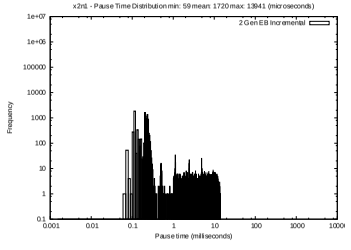

x2n1 - Pause Time Distribution min: 82 mean: 257 max: 5906 (microseconds)
1 Gen EB Incremental
Frequency
Pause time (milliseconds)

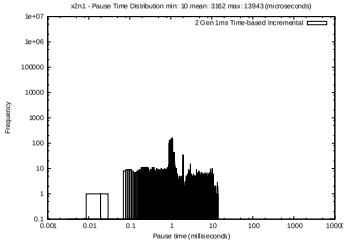

x2n1 - Pause Time Distribution min: 60 mean: 1248 max: 6276 (microseconds)
1 Gen 1ms Time-based Incremental
Frequency
Pause time (milliseconds)

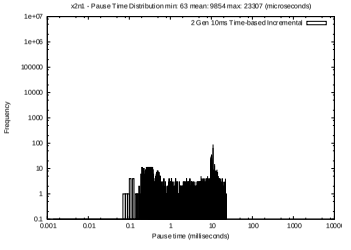

x2n1 - Pause Time Distribution min: 56 mean: 9623 max: 13988 (microseconds)
1 Gen 10ms Time-based Incremental
Frequency
Pause time (milliseconds)

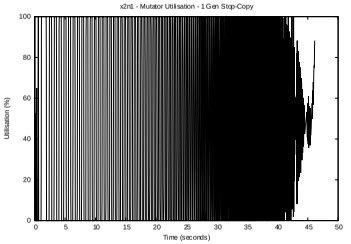

x2n1 - Pause Time Distribution min: 700 mean: 67636 max: 448706 (microseconds)
2 Gen Stop-Copy
Frequency
Pause time (milliseconds)

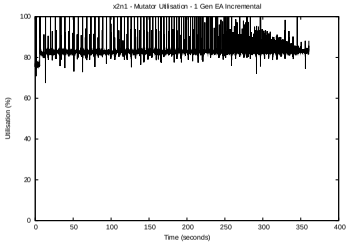

x2n1 - Pause Time Distribution min: 1 mean: 7 max: 13950 (microseconds)
2 Gen EA Incremental
Frequency
Pause time (milliseconds)

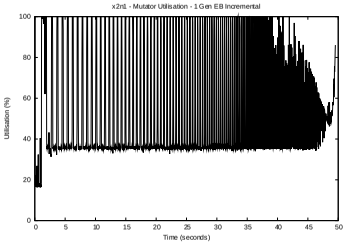

x2n1 - Pause Time Distribution min: 59 mean: 1720 max: 13941 (microseconds)
2 Gen EB Incremental
Frequency
Pause time (milliseconds)

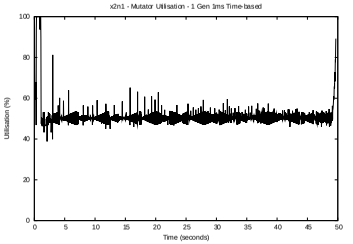

x2n1 - Pause Time Distribution min: 10 mean: 3352 max: 13943 (microseconds)
2 Gen 1ms Time-based Incremental
Frequency
Pause time (milliseconds)

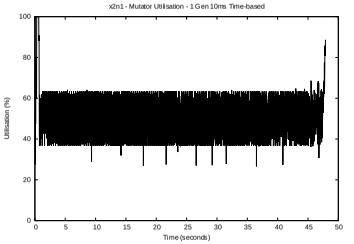

x2n1 - Pause Time Distribution min: 63 mean: 9854 max: 23307 (microseconds)
2 Gen 10ms Time-based Incremental
Frequency
Pause time (milliseconds)

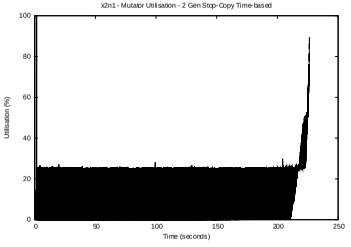

x2n1 - Mutator Utilisation - 1 Gen Stop-Copy
Utilisation (%)
Time (seconds)

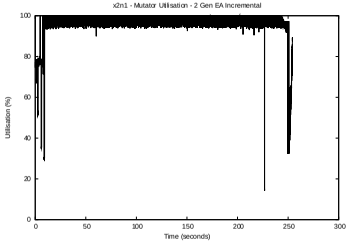

x2n1 - Mutator Utilisation - 1 Gen EA Incremental
Utilisation (%)
Time (seconds)

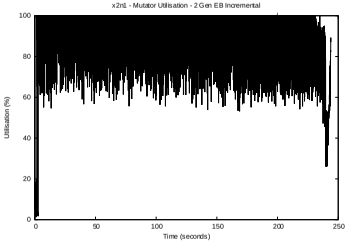

x2n1 - Mutator Utilisation - 1 Gen EB Incremental
Utilisation (%)
Time (seconds)

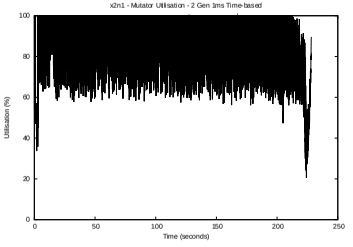

x2n1 - Mutator Utilisation - 1 Gen 1ms Time-based
Utilisation (%)
Time (seconds)

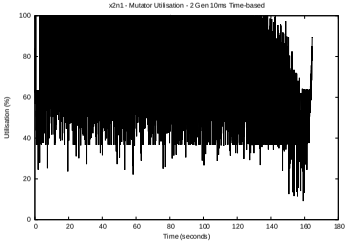

x2n1 - Mutator Utilisation - 1 Gen 10ms Time-based
Utilisation (%)
Time (seconds)

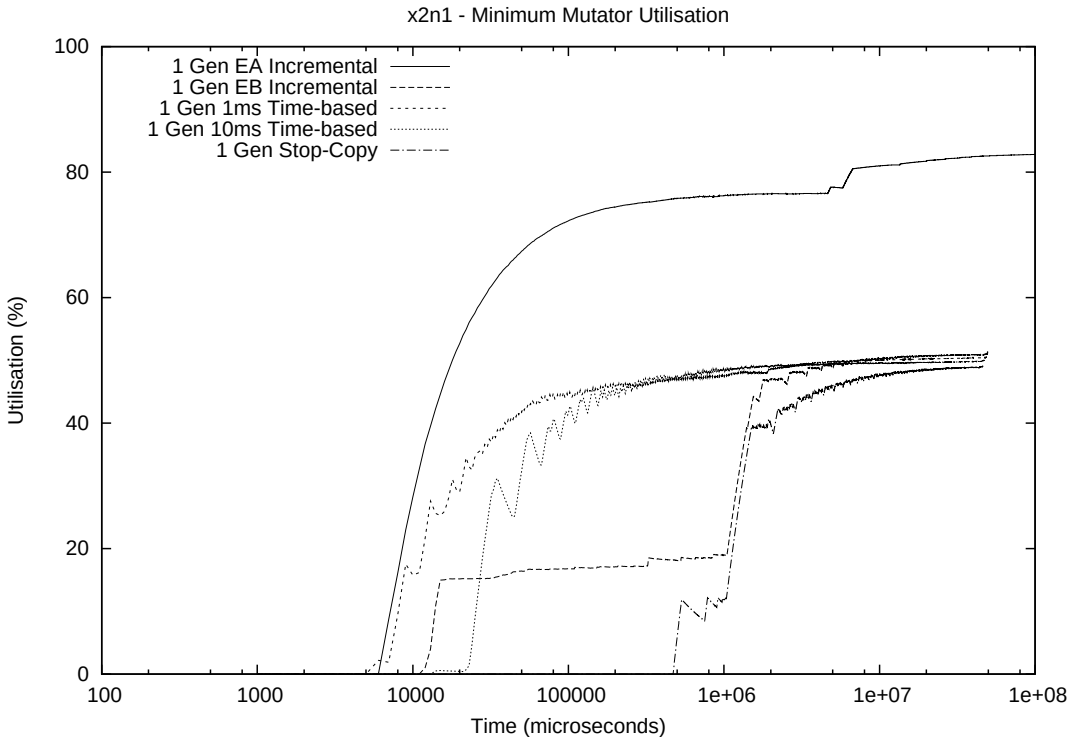

x2n1 - Mutator Utilisation - 2 Gen Stop-Copy Time-based
Utilisation (%)
Time (seconds)

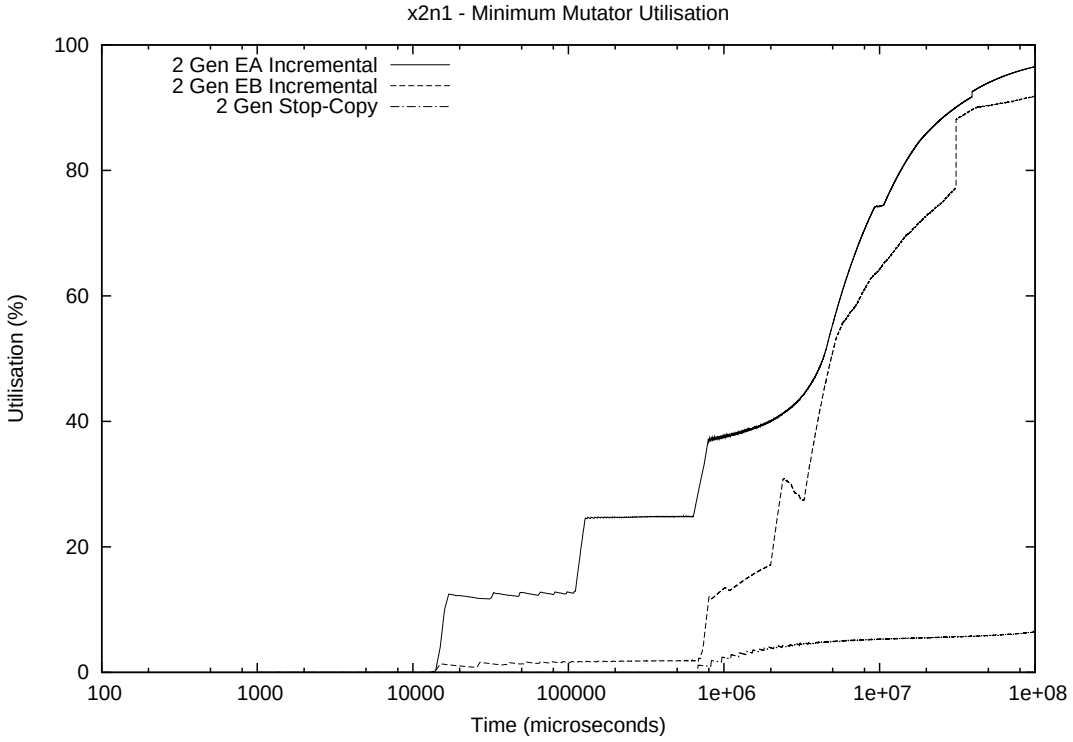

x2n1 - Mutator Utilisation - 2 Gen EA Incremental
Utilisation (%)
Time (seconds)


x2n1 - Mutator Utilisation - 2 Gen EB Incremental
Utilisation (%)
Time (seconds)


x2n1 - Mutator Utilisation - 2 Gen 1ms Time-based
Utilisation (%)
Time (seconds)


x2n1 - Mutator Utilisation - 2 Gen 10ms Time-based
Utilisation (%)
Time (seconds)


x2n1 - Minimum Mutator Utilisation
1 Gen EA Incremental
1 Gen EB Incremental
1 Gen 1ms Time-based
1 Gen 10ms Time-based
1 Gen Stop-Copy
Utilisation (%)
Time (microseconds)
0
20
40
60
80
100
100
1000
10000
100000
1e+06
1e+07
1e+08


x2n1 - Minimum Mutator Utilisation
2 Gen EA Incremental
2 Gen EB Incremental
2 Gen Stop-Copy
Utilisation (%)
Time (microseconds)
0
20
40
60
80
100
100
1000
10000
100000
1e+06
1e+07
1e+08

## 8.6 The Write Barrier

Historically, within GHC implementations employing contiguous heaps, the write barrier has been found to have a small overhead on execution time [Sew92, SP93]. The write barrier implementation is potentially more interesting in the current version of GHC because of its block-allocated heap. The object's address must first be mapped to a block identifier and the block header then examined to determine in which generation the object sits. The fast-path cost of the write barrier is 15 instructions on a Pentium III, compared with just two instructions for a contiguous heap. Does this significantly affect overall execution time?

To find out we executed the benchmarks with a 1GB heap so that no garbage collection, and hence no promotion, takes place. We used the baseline generational collector (i.e. with a write barrier) and compared it with our barrierless collector with specialised thunk code, but running in a non-incremental mode (`SPI-SC`).

Actually, only the young generation variant of the thunk code will be executed as no objects are ever promoted. Since neither collector executes, the difference is in the mutator code to implement the barrier. We then re-ran the benchmarks with the same collectors, but using the standard heap sizing policy, two generations and two tenuring steps, so that garbage collection and object promotion now occur. The results are shown in Table 8.6. Note, the `REF` times differ to those of Table 8.4 because different machines were used for the execution of the benchmarks (see Section 8). The same is true for the code bloat results presented in Table 8.7.

| | 1GB Heap | | Standard Heap | |
|---|---|---|---|---|
| Application | `REF` (s) | `SPI-SC` (%) | `REF` (s) | `SPI-SC` (%) |
| `anna` | 168.49 | –3.0 | 133.69 | –1.56 |
| `constraints` | 13.79 | +1.96 | 53.96 | +4.78 |
| `circsim` | 14.06 | –0.07 | 38.68 | +0.34 |
| `lambda` | 22.26 | +0.04 | 44.27 | +5.35 |
| `lcss` | 17.77 | –4.11 | 53.95 | +1.72 |
| `pic` | 38.45 | –1.47 | 93.24 | –1.18 |
| `scs` | 18.86 | +2.01 | 27.19 | –0.59 |
| `symalg` | 35.31 | –0.04 | 28.04 | +0.11 |
| `wang` | 23.93 | –3.01 | 101.90 | +1.68 |
| `wave4main` | 41.61 | –1.86 | 167.52 | –1.39 |
| `x2n1` | 11.90 | –3.87 | 71.70 | –3.38 |
| ALL 36 | | -1.67 | | -0.51 |

Table 8.6: Costing the write barrier

Running in a 1GB heap the `SPI-SC` figures indicate the maximum benefit that can be expected from removing the write barrier – if all objects sit in the young generation then the write barrier is wholly superfluous. The figures show that removing the barrier altogether

would only gain around 1.7% in performance averaged over all benchmarks. The figures for the standard configuration show the benefit in more typical situations.

| Application | REF (KB) | SPI-WB (%) | SPI-A (%) | SPI-B (%) |
|---|---|---|---|---|
| anna | 969 | 22.03 | 126.22 | 117.58 |
| constraints | 260 | 13.43 | 103.48 | 90.28 |
| circsim | 291 | 15.40 | 109.12 | 102.24 |
| lambda | 286 | 15.68 | 111.71 | 101.86 |
| lcss | 241 | 13.52 | 103.18 | 93.86 |
| pic | 357 | 15.81 | 106.41 | 91.46 |
| scs | 494 | 16.07 | 105.85 | 95.09 |
| symalg | 429 | 15.52 | 103.26 | 89.88 |
| wang | 269 | 13.13 | 92.63 | 83.66 |
| wave4main | 200 | 11.91 | 96.49 | 90.54 |
| x2n1 | 323 | 14.58 | 101.57 | 88.25 |
| ALL 36 | | 14.15 | 102.86 | 93.31 |

Table 8.7: Code bloat reported as a percentage overhead on the stop-copy collector

Table 8.7 compares the code bloat for our collectors implementing specialisation to varying degrees. Once again, REF, is the baseline stop-the-world generational collector. In Section 6.2.3 we described our preferred approach to write barrier optimisation where we arrange for the object to be added to the remembered set when it is updated. This *add-on-update* scheme requires a second (generic) variant of the update frame for the old generation which additionally adds the *updated* thunk to the immutable list. The entry code for the old generation is the same as for the young generation except that it pushes the modified update frame. SPI-WB is a stop-the-world generational collector that implements this scheme. Across all benchmarks, thunk and (single) update frame specialisation result in an increased binary size of, on average, 14.15% over those of the baseline generational collector.

Although for many applications removal of the write barrier doesn't pay, it is a worthwhile optimisation for the generational collector of "Eager Haskell" [Mae02], or more write-intensive applications. We have therefore developed a combined implementation of our incremental SPI-A and SPI-B collectors (see Section 7.2.4) with this "specialised add-on-update" scheme and an additional self-scavenging specialisation of the *special update frame* (that facilitates incremental stack scavenging as described in Section 7.1.4).

Table 8.7 reports the code bloat for these two barrierless incremental collectors. The code bloat is (unsurprisingly) substantial, pretty much doubling the binary sizes of the baseline collector. However, we can again choose to share the entry code that is common to each (as per the incremental SPS variants). Our previous evaluation shows that these give almost identical performance with a projected reduction in code bloat by between 20–40%.

## 8.7 Conclusions and Future Work

Our experiments have shown that closure code specialisation can buy performance when it is used to remove dynamic space overheads. A 30% code bloat over stop-and-copy (an additional 15% over our previous Non-stop Haskell collector) buys us a time-based incremental generational collector whose mutator overhead is only 5% slower than stop-and-copy when averaged over our chosen benchmarks. Furthermore, not only does the collector achieve consistent utilisation at around 10 to 15 milliseconds, but when averaged over the benchmarks, results in a performance increase of around 5% at the expense of a small additional amount of memory. The work-based collectors employing specialisation perform between 2–5% faster than our previous "Non-stop Haskell" collector (of course the benefits are greater when operating with limited memory, i.e. in the confines of a fixed-size heap). Finally, while the mutator overheads of the time-based collectors exceed those of the work-based collectors by between 2–3% the minimum mutator utilisation characteristics are at least an (and often several) order of magnitude better.

Specialisation opens up a number of additional optimisation opportunities, although, for the additional example we investigate, that of generational write barrier elimination, it appears to buy very little in practice. The conclusion is that, despite the relatively expensive write barrier in GHC, the write barrier overheads are actually very small. As such, although we have developed a fully barrierless incremental collector, we choose for our production incremental collector the low-overhead time-based `SPS` collector that exhibits very good minimum mutator progress characteristics but does not implement the write barrier optimisation via specialisation. The added complexity of such an implementation and the subsequent impact on its maintainability, coupled with the substantial increase in binary size, does not warrant the small performance improvement.

### 8.7.1 Honouring Time Quanta and Strictly Bounding Pause Times

Provided a program does not make use of large objects (mostly occurring as arrays allocated in separate blocks to the rest of the objects in the heap), we can, in theory, provide *hard* bounds on the pause time in keeping with the work of [BC99], for example. We have demonstrated a collector with consistent utilisation when both collector and mutator are scheduled using a time quantum of around 5 milliseconds. We have bound the unit of work performed by the stack scavenger, and given sufficient memory headroom, eliminated forced completions which lead to an unbounded pause at the end of those collection cycles where the collector cannot keep up with the mutator.

We have developed dynamic dispatching schemes for the handling of large closures (see Section 5.1.5 and Section 7.1.1), however, large unpointed objects that are not accessed and subsequently manipulated via dynamic dispatch, still present a problem. Although we have incorporated a lightweight read barrier into the the primitives that manipulate them, we have not broken them up into smaller fixed-sized structures [Sie00, BCR03b]. These structures

would then be accessed by more expensive primitives with more complex read barriers. While our approaches reduce the likelihood of having to scavenge these objects in one go, such cases do exist. An investigation into the incorporation of these smaller fixed-sized structures is an area of future work.

# Chapter 9

# Non-stop Java

The majority of the dissertation has been concerned with the exploitation of the virtualisation that is inherent to a *pure* dynamic dispatch environment to efficiently implement runtime services that most usually incur significant overhead when implemented in software. In particular, we have demonstrated the effectiveness of this with application to incremental read barrier and generational write barrier optimisation for collectors operating within the context of the Spineless Tagless G-machine. Furthermore, we demonstrated how to restrict the object header to a single word whilst avoiding destructive updates during the forwarding of copied objects *without* incurring the overhead of the tests usually required to support such an implementation. We believe these techniques can be applied to environments with equivalent 'purity', such as Smalltalk, Cecil and Lua, and will yield comparable benefits. What is not however clear, is how effective such techniques can be in environments where dynamic dispatch is less pervasive or devirtualisation is an aggressively employed optimisation. Throughout the following chapter we explore exactly this, within the context of a Java virtual machine, that utilises an aggressive build-time optimising compiler and an adaptive optimising compiler for run-time hotspots and methods.

The idea is to exploit the VM's dynamic dispatch mechanism, in a similar manner to Non-stop Haskell.

At run-time many object accesses in Java are made via method calls that are routed through a method table. This adds one level of indirection to the object access. Our first step is to rewrite a program's bytecode so that *all* object accesses are routed through the table. We refer to this as *full virtualisation*. With this done we can then 'hijack' certain accesses to the method table so that the access is directed to a *specialised* version of the method that performs some additional garbage collection task before invoking the original method code.

In the context of incremental garbage collection, one such specialisation automatically scavenges the object as a side effect of invoking one of its methods. Here the hijack is planted at the point where the object has been copied, but not yet scavenged by the collector. Any method invocation from outside the original object causes the "self-scavenging" code to be

executed, thus ensuring that all objects to which this object refers also sit in to-space. This preserves the read barrier invariant. After executing this extra code it resets the method table pointer so any subsequent accesses are directed to the standard method code.

Once the full virtualisation cost has been paid, the additional overheads are very low as the insertion and removal of the hijacker happens only once for each object per garbage collection cycle. At all other times the object access carries no additional overhead. The same idea can be used to eliminate the additional space overhead incurred by the forwarding pointer also associated with incremental copying collectors.

The tasks of virtualisation and method specialisation are managed by our *Class Transform Toolkit* (CTTk), which is built on top of the Jikes RVM [AAB+00a] and the Byte Code Engineering Library (BCEL) [Pro].

In order to buy back some of the overheads of full virtualisation we allow the RVM's dynamic optimiser to apply run-time optimisations such as method inlining. However, we have to be careful to ensure that this preserves correctness. In particular, if a method has one or more additional specialisations, we must arrange for *guard* code to be added that ensures the inlined code corresponds to the required specialisation and a branch taken if not. This inlined code, together with the guard, is similar in character to what would have been produced had a read barrier been implemented by hand.

An important additional advantage of our approach is that, in principle, we require only a single header word in each object[1]. The different states that the object can be in for the purposes of garbage collection are essentially encoded by the different specialisations of the method table. A single-header model reduces the object management overheads when compared with two-word object models, such as that of the standard Jikes RVM, as previously explored in [BFG02]. It also reduces the per-object footprint which increases the mean time between garbage collections and hence reduces the number of collections in a given period of time, as explored in Chapter 7 and in [CFS+04].

## 9.1 Chapter Overview

This chapter combines and expands the work described in [CFNP08, CFNP07], and makes the following contributions:

In Section 9.4 we present the design and implementation of a framework for efficient method specialisation within the Jikes RVM. Furthermore we provide: i) a framework in which arbitrary and user definable method specialisations may be specified and ii) a public API for switching between the specialisations.

Section 9.5 details how the framework is used to provide a set of method specialisations to eliminate an incremental collector's read barrier in a similar way to Non-stop Haskell except we now hijack methods instead of entry code.

[1] Our experimental evaluation is actually performed with respect to a two-header model in order to obtain an accurate measure of the mutator overheads.

We describe in Section 9.6 the use of the framework and method specialisation for the *non-destructive* installation of an incremental copying collector's forwarding pointer obviating the need for additional dynamic space overhead for *all* objects — we are able to support the one word header object model described in [BFG02] *without* incurring an overhead on type checking and virtual method invocation.

We present the trivial API invocations that must be made within MMTk to install: i) the self-scavenging objects that eliminate the need for an explicit read barrier and ii) the self-forwarding objects that avoid destructive forwarding pointer updates.

We describe in Section 9.4.4 how general purpose method specialisations may be developed and employed whilst allowing the method inlining and devirtualisation optimisations of an adaptive dynamic recompilation system to proceed unhindered.

Finally, in Section 9.7, we evaluate: i) the overheads that the method specialisation framework places on the infrastructure of the virtual machine; and ii) the costs associated with our self-scavenging objects in comparison to those of an explicit conditional read barrier. We perform our evaluation using the SPEC JVM98 and DaCapo [BGH+06] benchmark suites. We stop short of implementing an incremental collector in full as we are interested only in measuring the overheads placed on the mutator by the specialisation framework and barrier code.

For reference, the (unmodified) architecture of the Jikes Research Virtual Machine is documented in Appendix B.

## 9.2 Background

The power and expression of object-oriented languages such as Java are realised predominantly by promoting and maximising object extensibility and code re-use and are facilitated by abstract language features such as class local methods (enabling encapsulation and overridden re-implementation), *inheritance* and *message dispatch*. Message dispatch is the general mechanism by which a message sent to a *receiver* object results in the execution of a specific code sequence. Within object-oriented languages, message dispatch, termed *dynamic dispatch*, is the invocation of a *virtual* method. A virtual method is defined locally within its *base* class and can be subsequently *overridden* in its subclasses. At a particular call site of the virtual method any of the base or sub- class implementations may be invoked, depending on the dynamic type of the receiver. Such call sites are said to be *polymorphic*. For overridden method invocation, the method signatures are all identical. However, virtual methods may also be *overloaded* — multiple methods with the same name but differing argument signatures may be provided by both base and sub- classes. The process of method resolution for these *parametrically* polymorphic call sites depends on the semantics of the particular programming language — not only is the method choice dependent upon the receiver's dynamic type but also the number of method arguments and potentially the dynamic types of one or more of these arguments.

For *multi-method* dynamic dispatching languages such as Cecil, method resolution is purely dynamic and occurs at run-time based on the dynamic run-time types of the receiver and method arguments. In Java however, method targets are partially resolved at compile-time using the declared types and the number of arguments to the method as well as the declared type of the receiver. This compile-time pre-selection may result in i) a single method target, in which case the call site is *monomorphic* — the same virtual method is always invoked, ii) a set of potential method targets, in which case the *most specific matching* method must be sought for use as a *template*. Although such a call site is most generally polymorphic, it may be classed as *almost monomorphic* — even though the call site has multiple method targets the *majority* of calls resolve to a single target.

For those call sites that are verifiably monomorphic the target method body can be *inlined*, not only eliminating the overhead of virtual method invocation, but also facilitating wider application of classical code optimisations that, in the absence of inlining, would be restricted intra-procedurally. A method target can still be inlined at an almost monomorphic call site but a *guard* must be generated. The guard is a test that encloses the inlined code and verifies the dynamic type of the receiver is as expected and falls back to the standard virtual method invocation mechanism if it is not. The performance benefits of inlining especially in conjunction with the application of code optimisations can be massive. Furthermore, the presence of dynamic dispatch, whilst a powerful programmer available abstraction, is viewed at the language implementation level as performance crippling and therefore undesirable — it must be efficiently implemented and where possible, aggressively removed. As a result, inlining [BS96, DA99], method *devirtualisation* [IKY+00, CG94, GDGC95, AH96, SHR+00] and techniques for the efficient implementation of message dispatch [ZCC97, DHV95] have been extensively researched.

However, it is also possible to exploit dynamic dispatch. Notable exploitation of dynamic dispatch has been for the provision of language features and runtime services most usually by employing *proxy objects*. Examples include Parasitic Methods and type parameterisation [BC96, BC97, AFM97, BD98, TT99]; persistent object systems [Mos90] and Orthogonally Persistent Java [MZB00, MBMZ01]; distributed virtual machine implementation [AFT99, ZS03]; object caching and checkpointing; and hot swapping of class local features without class loader invocation, such as debugging trace and message logging.

This creates a dilemma — naïvely, if virtualisation is preserved, the performance benefits obtained from inlining and devirtualisation are sacrificed. On the other hand if they are employed then we lose the performance gained from the ability to implement cheaply runtime features such as a read barrier. In this chapter we explore how these techniques may be made to coexist and whether the results warrant the trade-offs and sacrifices that must be made. We do this within the context of a Java virtual machine, that utilises an aggressive build-time optimising compiler and an adaptive optimising compiler for run-time hotspots and methods.

### 9.2.1 Message Dispatch Implementation Techniques

Driesen et al. [DHV95] comprehensively review implementation techniques for general purpose message dispatch on pipeline processors and evaluate each of the techniques within a common framework. We briefly summarise the most common techniques.

Dispatch Table Search is the most naïve implementation technique. When a 'most specific matching' method must be sought, search proceeds through the class local methods of the receiver and widens in scope through those of its superclasses until a match is found. This is the most general mechanism that implementations fall-back to as a last resort for dynamic run-time resolution.

The remaining techniques are either static compile-time techniques or dynamic run-time techniques. The following are the primary static techniques:

- Selector Table Indexing (STI) — A single global two dimensional table containing an entry for *all* classes and *all* methods (or in Smalltalk, message *selectors*) is constructed. The target method entry is located simply by indexing on class and selector. Of course this strategy can be particularly space inefficient with the creation of incredibly large and sparse tables.

- Virtual Function Tables — Instead of employing a single global two dimensional selector table, a one dimensional table containing an entry for each method is associated with each class. The selector index for each superclass method inherited by a subclass is preserved within the subclass local table. Obviously the selector index of subclass local methods monotonically increments from the index of the superclass's last selector index (this is of course slightly more complex for systems implementing multiple inheritance). This approach is used in the implementation of dynamic dispatch for C++ and the Jikes RVM.

- Selector Colouring — A hybridised approach of Selector Table Indexing and Virtual Function Tables trivially supporting multiple inheritance and resulting in more compact tables. Each class is allocated a selector table in which a (global) graph colouring algorithm is used to assign colours instead of indices. A colour is unique within classes where the selector can be determined and can be shared by those selectors which never occur together within a single class.

- Row Displacement — Like Selector Colouring is a strategy that works to minimise the entries that result from the Selector Table Indexing approach. The rows of the STI table are flattened into a per-class one dimensional array with the populated entries overlapping the empty ones.

- Compact Selector-Indexed Dispatch Tables — The scheme implements two dispatch table types, the first for the set of *standard selectors* while the second is for *conflict selectors*. The standard selector table is as per the STI approach except the table

is more compact with sparse trailing entries removed and entries *aliased* by assigning the same offset to different selectors that are unrelated in the class hierarchy (i.e. via inheritance). Aliasing of standard selector offsets only leads to problems when two such unrelated classes implement a message with the same selector (method name and signature). For such cases conflict tables are constructed to resolve the conflict. The compiler is then responsible for generating the dispatch sequence that indexes either the standard or conflict table — there is no performance hit requiring two-level lookup, however like standard STI tables, the conflict tables are sparse and can't be compacted.

The following are the key dynamic techniques that attempt to either eliminate or optimise dispatch table search:

- Lookup Caching — The receiver's class is used in combination with the message selector to hash into a *global cache* containing entries consisting of the class, selector and address of the method to execute. If the current class matches that in the cached entry for the selector, then a jump to the cached method address results in immediate method execution. If it does not, then dispatch table search is used to resolve the correct method for execution and the cache entry is updated with its address and the receiver type.
- Inline Caching — A generalisation of guarded inlining applied most usually at almost monomorphic call sites to cache the code sequence to which the lookup for the current receiver's class and selector resolve. The code sequence is installed inline along with a guard on the receiver's type at run-time. When the guard fails, the code sequence for the method found as a result of performing dispatch table search is inlined at the call site replacing the previous 'cached' method.
- Polymorphic Inline Caching — A variation of inline caching used for polymorphic call sites that installs multiple guards (as if-elsif-else conditionals) and caches inline the $n$ most recently executed method bodies.

### 9.2.2 Method Inlining and Devirtualisation

We have already introduced guarded inlining / devirtualisation where a class test of the dynamic type of the receiver object guards the inlined method at the call site. This is a widely employed technique and has been studied in [CG94, GDGC95, AH96, HCU92]. In [DA99], Detlefs and Agesen present a variation of the guard using a method test instead of a class test which whilst incurring the overhead of an extra load they suggest is more robust and suited to dynamic recompilation systems. Many researchers have proposed whole program analyses that avoid the use of guards [GDGC95, Fer95, Bac97, TS97, SHR+00, ZCC97] but these are more suited to static compilation environments that lack the added complexity of dynamic recompilation and class loading. Such *direct devirtualisation* techniques can however be employed in dynamic systems by employing *on-stack replacement* [HCU92] or the *code patching* mechanism of Ishizaki et al. [IKY+00].

### 9.2.3 Persistent Object Systems

Persistent Object Systems focus on architectures and implementation techniques for instantiated object storage (to disk), where the objects outlive the execution of the application program that created them. The distinguishing feature from a classic database system is that the objects are involved in program execution and control flow. For example in an object-oriented environment they may have their virtual methods invoked, rather than serving as data records involved purely in program data flow. Implementation techniques [KDS01, CW00] are concerned with persistent object layout and pointer representation, marshalling the objects to (including checkpointed and transactional update techniques) and from disk and pointer *swizzling* — the conversion of pointers between their on-disk and in-memory representations. *Orthogonally* Persistent Object Systems [AM95], for example Orthogonally Persistent Java [ADJ+96] (OPJ), promote the power of abstraction over persistence by working to eliminate the programmer distinction between persistent and transient objects thereby presenting complete object transparency.

OPJ implementation techniques have explored source and byte-code transformations and compiler and runtime system modifications [MH96]. Marquez et al. [MZB00, MBMZ01] implement *portable* OPJ by subclassing Java's standard class loader to implement semantic extension mechanisms for persistence using bytecode transformation. These transformations insert read and write barriers into the generated code that ensure the object has been i) *faulted* into memory from disk before it is used and ii) to ensure that updates to it are propagated to the persistent store. They propose the use of *façade* objects that are fully virtualised objects, implementing the same methods as the in-memory objects, but are empty. These façade objects therefore proxy the real objects by faulting and swizzling them into memory before dispatching to the target method invocation. This exploitation of dynamic dispatch and proxy objects eliminates the persistent read barrier in a similar way to which we will demonstrate the elimination of an incremental collector's read barrier. Interestingly their measurements suggest that the use of façades results in a slow-down on the order of 20%.

### 9.2.4 Distributed Java Virtual Machines

Distributed Java Virtual Machine designs have employed similar implementation techniques to those that have been discussed for Persistent Object Systems. A distributed JVM must present a *single system image* (SSI) to the Java applications, hiding the distributed infrastructure upon which they run. Such an infrastructure may be built: i) as a layer external to and sitting on top of the JVM — Java application code as implemented in its non-distributed form is augmented with bytecodes that manage the distribution of and communication between objects; ii) as a layer on top of cluster enabled hardware, for example, a distributed shared memory environment; iii) as a cluster aware JVM — the JVM itself is modified to provide the services and infrastructure required to seamlessly distribute the application across the cluster. This third approach is adopted by both IBM's *cJVM* [AFT99] and ANU's *dJVM* [ZS03].

cJVM presents an object model incorporating both *master* and *proxy* objects. The master object is the 'real' object as used by the non-distributed application form. A proxy object is a *delegate* object to a remote object and through which the remote object is accessed — the proxy is used for location-transparent access to remote objects. However, proxy objects are further developed into *smart proxies* that provide different proxy implementations for different instances of the same class in order to improve performance. This is similar to the use of façade objects in OPJ — each smart proxy instance has an identical object header and method table, but the methods dispatch to a particular code sequence unique to the code proxy instance. This is achieved by replacing the class's virtual method table with an array of method tables, the first element of which refers to the standard virtual method table (VMT), while subsequent entries refer to specific smart proxy implementations. Depending on the object state it should be apparent that it is possible to switch between smart proxy implementations dynamically, on-the-fly, at run-time.

ANU's dJVM is similar to cJVM in its approach to presenting an SSI, however, it does not go that extra step of implementing smart proxy objects. Instead objects have a single method table, but for each declared method, a *proxy* and a *stub* method are also generated. The proxy method is responsible for marshalling the request for a remote method invocation, while the stub method is responsible for demarshalling this request and effecting the 'actual' method invocation locally on the appropriate node. The dJVM implementation is interesting for two reasons: i) Like our framework, it too is implemented within the Jikes RVM and so many of challenges faced and benefits realised are common; ii) The proxy and stub method generation occurs during class loading resulting from the use of bytecode transformation tools that have been 'hooked' into the RVM's *system* (not just *user*) class loader. The patch that hooks these bytecode transformation tools into Jikes RVM has been made publicly available by ANU and as a result we use these tools as a key component of our framework — we discuss the use of the tools in detail in Section 9.4.1.

Although similar in design to the *smart proxy* approach presented in the design of the cJVM's object model, our framework was independently developed without prior knowledge of cJVM, or even the façade objects of ANU's OPJ, but as a logical extension of our work on Non-stop Haskell. It is worth noting that unlike cJVM we provide a framework in which arbitrary and user definable method specialisations may be specified and a public API for switching between the specialisations.

### 9.2.5 Type Parameterisation and Parasitic Methods

Preceding Java 2 Platform SE 1. 5 there have been many proposals for the addition of varying degrees of support for parameterised types and generic classes to the Java programming language. Many of these proposals suggest syntactic extensions to the Java source language and generate instance classes and auxiliary methods for the parameterised type during class loading using bytecode rewriting tools [TT99, AFM97, BD98]. Parasitic Methods for Java [BC97]

augment their *host* methods with additional functionality for particular argument cases. By design, Java allows *invariant* method specialisation where a method is only overridden by methods with identical (widened) argument types. However, a parasite implemented to handle a subset of the hosts cases, can now *covariantly* specialise the host method. Furthermore, if it handles the full set, effectively overriding and hijacking the host (but perhaps delegating to it later), then it is said to *contravariantly* specialise it. Parasitic methods can be implemented in standard Java via bytecode rewriting and translation — the host method body is prepended with guard code that delegates to the appropriate parasite method.

## 9.3 Conventional Read Barriers in Java

Like Non-stop Haskell, we require a 'vanilla' read barrier implementation against which a dynamic dispatching read barrier can be benchmarked and compared. We have already introduced the Metronome collector of Bacon et al. and explained that their read barrier code is unavailable to us. This is most likely a blessing in disguise! The Metronome uses a Brooks-style indirection-based read barrier whose implementation carries a *mutator* overhead of a mere 4% on average, and a maximum of 10% across the SPECjvm98 benchmark suite. While this is on a par with the barrier overheads of our GHC collector implementation, given that the environment is now less 'pure' and aggressively devirtualised, it sets a hard target to beat! However, in order to achieve these results, the Java implementation of the RVM, the inline optimisation of the JVM core with user code, and furthermore barrier-specific optimisations are exploited. An example of one such optimisation is barrier-sinking where the barrier is moved to its point of use, allowing the barrier's null check to be combined with that of the memory load operation — because the pointer may be null, its forwarding cannot be unconditional. This is important because a) there is no introduction of additional null checks and b) because it ensures that any arising null pointer exception occurs at the same program point regardless of the presence of an incremental collector.

In [BH04] Blackburn and Hosking perform a study of the overhead placed solely on the mutator by both conditional and unconditional read barrier implementations within Jikes RVM. The study does not analyse the effect such barriers have on (increased) collection time. Their related work section recognises that "the spectre of high barrier costs has driven diverse work on elimination of barriers through compiler optimisation or synthesis of new collector algorithms that forgo reliance on barriers". Indeed, the focus of this dissertation is one such work. They go on to say, "Our results question the motivation for such efforts (if their point is only to avoid barrier costs), since we show such costs to be surprisingly low". 'Low' is on average 8.05% (AMD), 5.04% (Pentium 4), and 0.85% (PowerPC) for an unconditional barrier with an AMD worst case of 13.03%. For a conditional barrier it is on average 21.24% (AMD), 15.91% (Pentium 4), and 6.49% (PowerPC). Not only do we dispute this interpretation of 'low', but our experience in a range of incremental collector implementations leads us to also dispute that works focus solely on avoiding barrier costs since the techniques and such avoidance

are often coupled with significant effects on collection time. We do however recognise their contribution as valuable, benefiting those researchers who are in a position to compare apples with apples and oranges with oranges — that is to say that the implementations differ only in barrier implementation while the associated supporting infrastructure within the collector, and any additional run-time space overheads, is identical or at least comparable. As a result, neither our own work nor that of Blackburn and Hosking should be compared with the results reported by Bacon et al.— the Brooks-style indirection is a handle-based approach that carries an associated run-time space overhead — each object allocates extra space for a forwarding pointer that, when the collector is off or the object is in from-space points to itself, and when it has been moved, points to the relocated object. The mutator overhead is not therefore, meaningfully comparable with that of a conditional Baker-style barrier that carries no such space cost. Indeed, it is for exactly this reason that we worked to remove this overhead in the migration from GHC's prototype semi-space incremental collector (Chapter 5) to its production incremental generational collector (Chapter 7) — we recognised the increased collection costs and effective reduction in available memory.

The collector of Chapter 7 does not incur any run-time space overhead to achieve its incrementality. If the technique is successfully applied to Jikes RVM, then the same is also true. For this study, we are not interested in the overhead that a work or time based scheduler places on the mutator nor the effects that early reclamation or floating garbage have on collection time, but solely the cost of the dynamic dispatching read barrier and the infrastructure that supports it. Therefore, our results are directly comparable with those of Blackburn and Hosking's conditional read barrier and it follows that we benchmark in comparison to the implementation that they make available for Jikes RVM [BH04].

### 9.3.1 Implementation

Implementing a read barrier for the Java object model is in essence very simple. Objects are accessed and manipulated by reference and for safety, the language semantics disallow pointer arithmetic. How then are pointer loads from the garbage collected heap performed in Java? Such an operation is one that obtains a reference to an object. This is most usually achieved via (either implicit/explicit) assignment of a reference obtained using field access: `<object instance>.field`. The field may turn out to be an unboxed primitive type and not 'classically' require read barrier application. However the `getfield` bytecode that implements Java's field access operation does not necessarily know this at the point of barrier emission. Whether or not this barrier may be elided will be determined by the type and complexity of the analyses that the compiler and its code generator implement. Furthermore, with literals and numeric constants residing in the JTOC, depending on whether the JTOC is heap allocated or not, also determines the necessity of the read barrier. In addition, operations that manipulate the object or its contents directly (e.g. primitives internal to the JVM and its runtime system), avoiding such a dereference, must also contain the barrier code. Therefore,

the maximal set of access operations requiring read barrier placement (although as explained above some may be unnecessary or can subsequently be elided) are:

- Object field access
- Static field access
- Array element access
- Array copying operations
- JVM-internal object manipulation primitives

For clarification, it is worth noting that virtual method invocation on an object does not require incorporation of a read barrier since the read barrier invariant has already ensured that the object has already been evacuated when its reference was loaded. Furthermore method invocation does not require the implicit manipulation of the object's fields.

Like our previous incremental collector implementations the to-space invariant is only partially enforced by the read barrier, in addition, the root set must be evacuated at initiation of the collection cycle and the stack must either be treated as part of the root set or a mechanism enabling incremental scavenging applied. Obviously, scavenging the entire stack in one go may result in a mutator pause that violates real-time constraints. The incremental scavenging technique presented in Section 7.1.4 is applicable to any stack-based execution environment and in such an environment is similar in essence to the *stacklet* approach of Cheng et al. [CHL98]. A register must be reserved for the hijacked return address and then the return address to generic stack frame scavenging code is patched in. Once this scavenging code has been executed, control returns to the hijacked return address. It is worth noting that our "self-scavenging" approach can easily be implemented using our specialisation framework by exploiting Jikes RVM's on-stack replacement mechanism. Unfortunately, using either approach, care must be taken in the scheduling of stack scavenging to ensure that it is not performed solely by this incremental stack barrier which would result in the aggregation of pauses at an aggressive stack pop rate. These approaches are not specific to a dynamic dispatch environment nor to any read barrier infrastructure we develop. As a result, like Bacon et al. [BCR03b], we defer the implementation and benchmarking of incremental stack scavenging and the remedy of associated pause aggregation to an investigation proposed as further work (see Section 9.8).

Blackburn and Hosking provide read barrier infrastructure that supports a broad range of read barriers that are either:

- Conditional — The barrier performs a test on the tag bits of the object being read or on its address (to determine its from-space / to-space location) and control flow branches accordingly.

- Unconditional — High- or low-order address bits of the object reference are masked during the read. This type of barrier is employed when object references must be tagged transparently with respect to the mutator. Such a technique is used by collectors that perform per-object marking of references.

and either:

- Substituting — The barrier takes the place of, and is therefore responsible for, the load operation.
- Non-substituting — The barrier code is executed after the field has been loaded. Optionally, *pre-barrier* code can also be placed before the load so that it is 'wrapped' by barrier code. A non-substituting barrier is used in situations where it is desirable for a specific sequence of instructions be used to implement the load, for example in the case of hand-optimised assembler.

The `MM_Interface` is the high-level "wrapper" implementation that therefore exposes i) the need for the Jikes RVM compilers to compile and emit read barrier code; ii) whether the barrier is substituting (`readBarrier()`) or non-substituting (`postReadBarrier()`); and iii) in the case of a non-substituting barrier, whether or not pre-barrier (`preReadBarrier()`) code is required. Each of these "wrapper" methods operate on Java `Object`s and converts them to and from the raw addresses upon which the RVM internally operates. Importantly, they wrap the actual implementation exposed by MMTk's currently selected local plan. Each local plan provides access to and implements the thread-local operations and data structures, that each specific collector implementation requires. For example, Listing 9.1 lists the `MM_Interface readBarrier()` implementation that takes an object reference from which a field is to be read, applies the read barrier operations and returns a reference to the actual field read, where: `object` is the reference to the parent object of the field being read; `slot` is the address of the actual field to be read; `context` is an integer identifier used as auxiliary metadata to describe the operation in which the barrier is being emitted, for example, in a `getfield` operation versus a `getstatic` operation; and finally, the method returns the reference to the field that was read.

```
1 public static Object readBarrier(Object object, Address slot,
2                                  int context) throws InlinePragma {
3     return VM_Magic.addressAsObject(VM_Magic.getMemoryAddress(
4                 SelectedPlanLocal.get().readBarrier(
5                     ObjectReference.fromObject(object), slot, context)));
6 }
```

Listing 9.1: MMTk's `MM_Interface readBarrier()` implementation

Listing 9.3 lists the local plan read barrier implementation that we use in our evaluation. The `readBarrier()` implementation simply chains pre- and post-barrier operations

```
1 mov     edx eax
2 and     edx 1
3 cmp     edx 1
4 mov     edx 0
5 cmovne  edx eax
6 mov     eax edx
```

Listing 9.2: x86 Assembler for Blackburn and Hosking's conditional read barrier

together and substitutes the load of the desired field value. The `preReadBarrier()` implementation is simply a no-op, leaving the address of the field to be read unchanged. The `postReadBarrier()` implementation applies a conditional test to the field value simulating the "fast-path" portion of the from-space / to-space test that always evaluates to `true` — the object is always in to-space and so no action needs to be taken. This is the minimum mutator overhead that a conditional read barrier test must apply. Notice that the inlining of the barrier methods is forced so as to avoid the overhead of stack frame pushes and pops and to facilitate straight-line code optimisations. As a result, this code compiles to the six x86 assembler instructions of Listing 9.2. Furthermore the methods are specified as uninterruptible so that the compiler verifies that no operations which allow interruption by the collector are used within the methods. Were such an operation to be used and a GC interruption to occur, the GC methods will recurse indefinitely until either hardware registers are exhausted or the stack overflows.

```
1  public Address readBarrier(ObjectReference objectRef, Address slot,
2                             int context) throws InlinePragma {
3      Address rtn = preReadBarrier(objectRef, slot, context);
4      return postReadBarrier(rtn, context);
5  }
6
7  public Address preReadBarrier(ObjectReference objectRef, Address slot,
8                                int context) throws InlinePragma {
9      return slot;
10 }
11
12 public Address postReadBarrier(Address value,
13                                 int context) throws InlinePragma {
14     // Blackburn's conditional read barrier
15     if (value.toWord().and(Word.one()).NE(Word.one())) {
16        return value;
17     }
18
19     // Never reached
20     return Address.max();
21 }
```

Listing 9.3: MMTk's local plan `preReadBarrier()`, `readBarrier()`, and `postReadBarrier()` implementations

#### Barrier Tests

For a full read barrier implementation, the first barrier test determines whether a garbage collection cycle is in progress. If so, it then tests the object header of the field being accessed in order to determine whether the object resides in to-space or from-space. For some heap layouts it is possible to combine the two tests, specifically if all the addresses of one space differ from those of the other space in one or more bits. In general, however, a separate test is required, for example where the heap is configured via a block-allocated memory system.

A final test determines whether the object has already been evacuated. If the collector is on and the object has yet to be evacuated then it is copied to to-space and a *forwarding* pointer to the to-space copy is installed in the original from-space copy. A reserved 'forwarding' bit in the header is usually then set to indicate that the object has been evacuated. If a from-space access finds this bit is set the location of the copy in to-space (the installed *forwarding address*) is extracted and used.

Notice that the explicit *conditional* barrier of [BH04] as implemented in `postReadBarrier()` of Listing 9.3 is particularly efficient. The test merges the 'collector on' test with the 'from-space / to-space' test by assuming that from- and to-spaces can be distinguished by a single bit in an object's address and by flipping a reserved word between 0 and 1 each time a garbage collection cycle is initiated.

## 9.4 A Framework for Efficient Method Specialisation and Virtualisation

In principle, the implementation of a method specialisation framework is relatively simple — the virtual method table must be replaced by an array of method tables, with the first element providing access to the 'standard' method table, and each subsequent element providing access to each VMT specialisation. In addition, the virtual machine must be modified to access the VMT via this *specialisation* array. Finally, a mechanism must be provided to switch, arbitrarily, between method specialisations.

However, our framework not only enables method specialisation, but also *full* virtualisation and the ability to define *user specified* specialisations, not just intra-VM specialisations specified at VM build-time and internal to the VM.

The implementation is further complicated by method recompilation either by the RVM's adaptive optimisation subsystem or via the invocation of its class loading mechanism.

The following section describes the design choices and implementation challenges faced in assembling the full framework.

### 9.4.1 CTTk — The Class Transform Toolkit

Our approach is based on the ability to invoke different versions (*specialisations*) of a method, depending on context. In order to achieve this we have developed a *Class Transform Toolkit* (CTTk) for transforming classes as they are loaded by the RVM. We use CTTk to beanify field accesses (generate getter / setter methods) where necessary and to enable user-defined specialisations to be defined and integrated into a common *Type Information Block* (TIB) structure (see Section B.4.4) in a modified version of the RVM.

CTTk builds on Zigman's Bytecode Transformation Tools for Jikes RVM [Zig] originally developed for effecting distribution within ANU's distributed JVM, dJVM [ZS03]. Zigman's patch hooks the Apache Jakarta Project's Byte Code Engineering Library (BCEL) [Pro] onto the RVM's class loading mechanism. BCEL enables the analysis, creation, and binary manipulation of Java class files — in particular, APIs are provided to access and modify a class's methods, fields and bytecode instructions. By sitting at the end of the class loading chain the BCEL patch is able to transform both user and system loaded classes. Furthermore, recall that the RVM is itself written in Java, and if so desired, its classes too can be subjected to the transforms.

A CTTk transform is a class that implements the `Transform` interface's `process` method:

```
public interface Transform {
  public JavaClass process(JavaClass javaClass);
}
```

The `process` method is passed a `JavaClass` object that is the BCEL representation of a Java class. It returns a potentially new or mutated `JavaClass` object that has been subjected to the BCEL APIs as defined by the user defined implementation of the method.

We modify the signature of the `process` method to take an instance of a `TransformClass`. The `TransformClass` is a wrapper class that encapsulates the `JavaClass` undergoing transformation, and an array of `Method` object arrays, and provides getter and setter methods for these objects.

```
public interface Transform {
  public TransformClass process(TransformClass c) ;
}

public class TransformClass {
  public TransformClass(JavaClass javaClass,
                        Method[][] specMethods) {
    ...
  }
  ...
}
```

The `Method` class is the BCEL representation of a class method and is one of the basic types used by the manipulation APIs. The array of `Method` object arrays allows an arbitrary number of `Method` array specialisations to be presented to the transformation framework; each element provides a specialisation of the `javaClass`'s methods. It is from this object that the array of VMTs is constructed that replace the single VMT in the original RVM TIB structure. Specifically, the first `Method[][]` index identifies the TIB specialisation, while the second indexes the specialised methods for that particular TIB.

Figure 9.1 depicts the class loading process that leads to class resolution in both the original RVM and our fully virtualised RVM. During class loading the minimal amount of work necessary for the user program to start executing is performed. At this point the resulting `VM_Class` object is only partially initialised: superclasses are yet to be resolved, methods and fields are neither laid out nor resolved and the TIB is neither created nor instantiated. The class is only fully resolved and the `VM_Class` object fully initialised on first use of the class, either for object access or instantiation. By the time a class is fully resolved for use, the RVM must have built its internal representation for class instances (`VM_Class`), arrays (`VM_Array`), and primitives (`VM_Primitive`). These internal representations all subclass `VM_Type`, and correspond to run-time types. Similarly, the RVM defines internal representations for method (`VM_MethodReference`) and field (`VM_FieldReference`) references, which subclass `VM_MemberReference` and represent member references in the class files.

The process of class loading is initiated on invocation of the RVM's `VM_ClassLoader.defineClassInternal` method. In the original RVM, only a `VM_TypeReference` and a `VM_Class` instance are created while only the names of fields, methods, and superclasses are read from the class file. Nothing more is done until subsequent invocation of the `resolve` method. However, in our virtualised RVM, in addition to the above, a BCEL `ClassParser` is constructed that *fully* parses the class file, creating a corresponding `JavaClass`.

In the original RVM class loading is completed on invocation of `resolve`, whereupon superclasses are resolved, fields and methods are laid out, and the Type Information Block is created and populated. In our RVM the BCEL patch arranges for the CTTk internal transformations and user-defined transformations to be applied to the `JavaClass` that has been previously created. The `VM_Class` is updated, and its field and method definitions are updated to reflect the corresponding changes to the transformed `JavaClass`. Finally, methods and fields are laid out and the TIB is constructed. If specialised methods exist, then these are similarly processed and their associated TIB specialisations are constructed.

### 9.4.2 Specialisation

To implement method specialisation the standard virtual method table is replaced by an array of method tables. Within Jikes RVM this is achieved by providing duplicated Type Information Blocks that differ primarily in the instruction code arrays to which each virtual

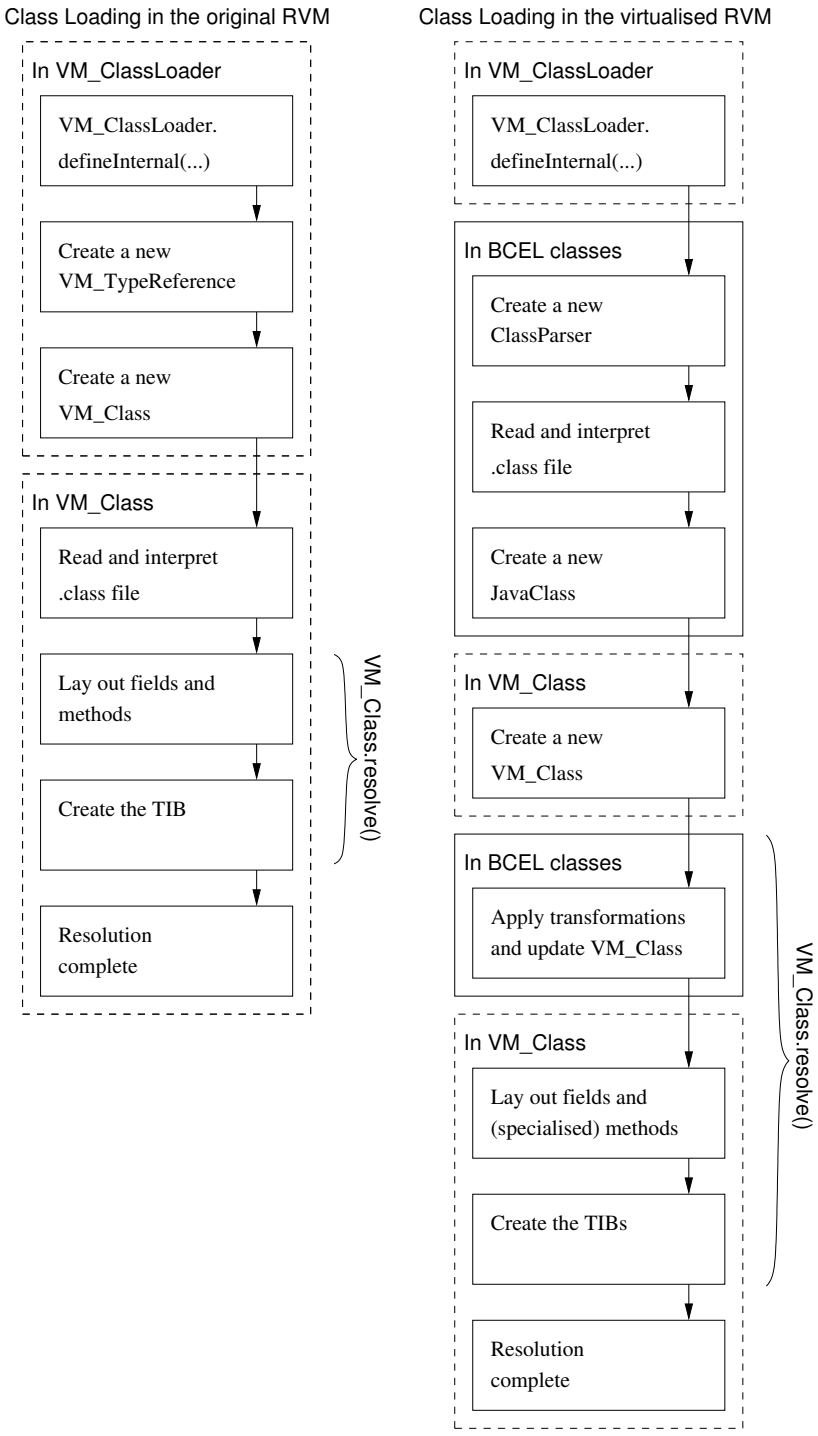



Figure 9.1: Class loading in the original RVM and in the fully virtualised RVM.

method slot points. As a result the TIB must be extended with a *TIB array pointer* which references the array of TIBs which carry the required method specialisations.

Figure 9.2 shows the modified TIB structure, with all additions to Figure B.6 shown shaded on the diagram. If there are $S$ specialisations then there are $S + 1$ TIBs which differ only in their method code. The diagram shows the primary TIB and one of the specialisations (Specialisation 1). A heap object is shown whose header field points to the primary TIB.

Each TIB is augmented with two additional fields. The first of these, occurring at TIB index 2, is the *current TIB index*, which identifies which index in the TIB array this TIB is. The second, at TIB index 3, is the *primary TIB pointer* and points to the type's primary (or default) TIB. This is used in operations such as type checking.

Most usually the primary TIB is at index zero and is the 'standard' TIB that is created regardless of whether specialisations are present. Furthermore, it is the TIB for which we *expect* to invoke methods on more frequently than the other specialisations.

Type checking within the RVM is implemented by the comparison of TIB object addresses: two objects are deemed to have the same type if their TIB addresses match. In our modified RVM we have a problem: there may be several specialisations of the same object, i.e. each with the same type. We get round this by using the additional primary TIB pointer field. This points to the TIB array, but could equally well point to the primary TIB, for example. In principle, two objects now have the same type if their primary TIB fields match. However, this requires an extra level of indirection as we have to index the TIBs to obtain the pointers.

Our experiments have shown that these extra references can prove expensive in some applications. We therefore code for the common case: we envisage that in general, the current TIB only rarely corresponds to a specialisation. A type check thus proceeds by comparing TIB addresses, as in the original RVM. If this fails we fall back and reapply using the more expensive primary TIB pointer comparison.

We prevent a loss in performance by enabling explicit (and therefore faster) access to the primary TIB element within the TIB array for use by type check operations — whenever a type check operation is performed, the primary TIBs are *always* obtained (regardless of whether the current TIB is a specialisation) and compared.

In order to access these extra fields we introduce two new low-level intermediate representation (IR) operators, `GET_OBJ_PRIMARY_TIB` and `GET_TIB_FROM_TIB_INDEX`. `GET_OBJ_PRIMARY_TIB` is similar to the existing IR operator `GET_OBJ_TIB` but first dereferences the primary TIB pointer to retrieve the primary TIB. In order to preserve operational correctness within the RVM, (as previously explained for type checking operations), we are forced to replace some uses of `GET_OBJ_TIB` with `GET_OBJ_PRIMARY_TIB`. `GET_TIB_FROM_TIB_INDEX` simply obtains the current TIB index from the object reference at TIB index two.

It is important to note that, because TIBs are shared by objects of the same type, the addition of these fields does not affect the size of the objects themselves.

#### TIB Flipping

A particular specialisation is invoked by *flipping* the TIB of an object among the various specialisations, thus *hijacking* accesses that are directed via the method table. In Figure 9.2, flipping the TIB of the object shown will involve overwriting the TIB pointer of the header field with a reference to one of the TIB specialisations located in the TIB array. It should be apparent that there is sufficient information in each TIB variant to move arbitrarily among the specialisations.

In order to implement TIB flipping, we augment the RVM's object model APIs for getting and setting TIBs:

```
Object[] getTIB(Object o)
void setTIB(Object o, Object[] tib)
```

with three additional (public static) functions:

```
Object[] getTIB(Object o, int tibSpecNum)
void hijackTIB(Object o, int tibSpecNum)
void restoreTIB(Object o)
```

The additional `getTIB` method retrieves the TIB at index `tibSpecNum` from the TIB array. The `hijackTIB` method updates the TIB of the given object/object reference to the TIB retrieved from the TIB array at the index specified by `tibSpecNum`. Finally, the `restoreTIB` method restores the TIB of the given object to its primary TIB.

### 9.4.3 Virtualisation of Field and Method Accesses

In order to provide a full virtualisation framework that exploits dynamic dispatch we must *naïvely* ensure that all operations on a Java object are performed using the virtual machine's `invokevirtual` instruction, i.e. a virtual method call — this enforces 'JavaBean' compliance where a class is compliant if *all* field (including `private`) access and assignment operations are performed via getter and setter methods. To achieve this, we build a `BeanifierTransform` transform using CTTk. The beanifier is invoked during class loading and uses the BCEL APIs to generate getter and setter methods for each field in a class. The Beanifier is able to detect existing getter and setter methods and avoids generating duplicates. Additional control is provided over the beanification of fields based on their access level modifiers. In the majority of cases, these accessor methods inherit the field's *access level modifier*. However, the Java Language Specification requires class initialiser methods, `private` methods, and directly inherited superclass methods to use the `invokespecial` instruction which avoids virtual method invocation via the VMT/TIB. As a result, `private` methods, and directly inherited superclass methods must have their access level modifier switched to either `public` or `protected`. Again, this is achieved using the BCEL APIs.

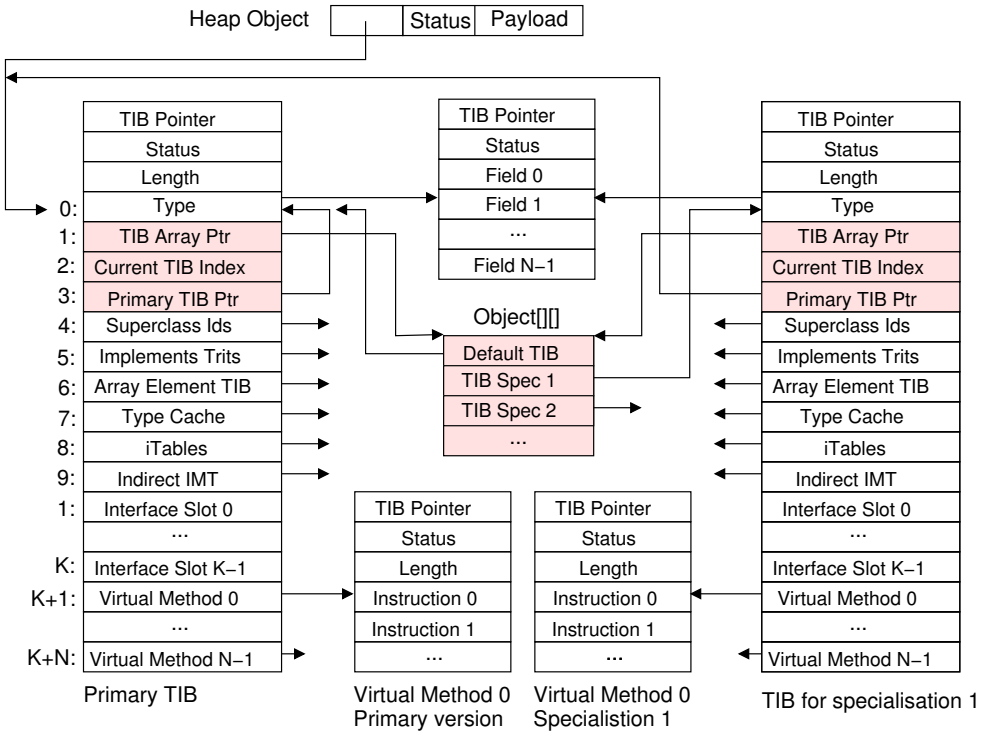



Figure 9.2: The Type Information Block augmented with TIB specialisations.

The *Java Native Interface* is a foreign function interface providing a set of primitives that allow Java operations, such as method invocation and field access from native languages such as C and C++. In order to preserve full virtualisation, the primitives, defined as methods within the RVM's `VM_JNIFunctions` class must be guarded to ensure the correct TIB method specialisation is invoked for the object being accessed. The operation of this guard is similar to the `ig_tib_guard` introduced in Section 9.4.4.

The final challenge in achieving full virtualisation is to ensure that the RVM does not perform inlining optimisations, inlining the 'standard' method body, avoiding `invokevirtual` method invocation via the VMT/TIB and thus preventing the invocation of specialised methods. The RVM provides a `NoInlinePragma` pragma that can be used to annotate a method declaration that explicitly prevents a method from being inlined. CTTk provides a `NoInlineTransform` transform that runs after the BeanifierTransform and adds the `NoInlinePragma` statement to *all* method declarations.

### 9.4.4 Recovering Performance via the Adaptive Optimisation Subsystem

Full virtualisation enables the exploitation of dynamic dispatch. However it imposes an additional overhead on some object accesses. In order to buy back this performance we

allow the AOS and the optimising compiler to perform a range of optimisations, most notably method inlining.

The inlining of 'hot' methods by the optimising compiler of the AOS at run-time is one of the most effective optimisations that Jikes RVM employs. It enables the most frequently executed methods to undergo a) (guarded) devirtualisation and b) inter-method code motion enabling subsequent optimisation resulting from context sensitive control and data flow analyses. Furthermore, because the resulting profile information from the AOS is context sensitive with respect to the dynamic call graph, the weights associated with each call site, that guide inlining decisions are more accurate than context insensitive inlining performed by a compile-time only optimiser.

The Jikes RVM implements a relatively sophisticated *speculative* inlining framework that performs guardless inlining of monomorphic call sites and guarded inlining of polymorphic and almost monomorphic call sites. The AOS periodically samples the currently executing methods on the top of the RVM's stack to determine which methods consume the majority of the execution time on each of the virtual processors. From the profiling data, weights are attached to each call site. If a call site accounts for a large enough proportion of processor time, then a more expensive inlining action is allowed. The weight of the callee as a fraction of the total weight of all the candidates for inlining is then used to determine the cost of inlining that particular call site and to determine whether or not the inlining action will be profitable.

The inlining guards already present in the RVM are `ig_class_test`, `ig_method_test` and `ig_patch_point`. `ig_class_test` simply compares the TIB of the receiver against the TIB encoded in the guard, testing the type of the receiver. This is relatively inexpensive. By contrast, `ig_method_test` examines the candidate's method pointer against the method pointer encoded in the guard. This is more expensive since sub-fields of the TIB must be examined, but permits inlining in more cases than for the `ig_class_test`. Finally, `ig_patch_point` emits assembly code that can be overwritten at run-time to update the guard condition without recompiling the method. These guards are selected by the *inlining oracle*, which is responsible for inlining decisions and determining the appropriate guard to deploy at a call site.

The RVM categorises methods according to the number of instructions that it projects will be generated when the method is inlined. *Tiny* methods are those methods with an instruction count less than twice the number of instructions required to invoke a method. *Small* methods have an instruction count between two and five times the method invocation instruction count. *Medium*-sized methods have an instruction count of between 5 and 25 times. *Huge* methods have an instruction count of greater than 25 times. Huge methods are never inlined. Tiny methods are always inlined if they are provably monomorphic, as determined statically. Small methods are inlined, with a guard if necessary, based on a heuristic that combines the instruction count and inlining depth. Medium-sized methods are considered for inlining only by the profile directed inlining oracle of the AOS. Furthermore the AOS may inline small methods that have exceed the instruction count or inlining depth limits and tiny

and small methods that were not statically determined to be monomorphic at compile-time.

There is a final method categorisation which is unrelated to the instruction count of the method body. A *forbidden speculation* is a method which cannot be inlined speculatively into its caller, because the caller cannot be recompiled if the callee gets invalidated during subsequent execution. Such a classification is most usually applied to those call sites where the caller method is a part of the RVM infrastructure that resides in a static, immortal area of the heap, and as such cannot be recompiled and the callee is user code that is not a part of the RVM.

The RVM also provides pragmas with which to annotate method definitions in order to explicitly force, (`InlinePragma`), and explicitly prohibit, (`NoInlinePragma`), method inlining. In addition, CTTk provides a further pragma, `DoNotSpecialisePragma` that can be used to specify that a class is not to be subjected to the transformations that the user has specified and allows the method to be subjected to inlining as described above.

As previously described, while virtualisation enables the exploitation of dynamic dispatch, it is undesirable because of the performance slowdown it imposes on the RVM. As a result our framework aims to provide all the benefits of virtualisation whilst recovering performance by allowing devirtualisation to be performed and inlining optimisations to proceed. The optimising compiler of the AOS must often selectively recompile methods, frequently discarding compiled methods and replacing them with new, more aggressively optimised ones. As a result, the method table entries within the TIB must be patched to point at the newly compiled code for the recently optimised replacee method body. In the presence of specialised TIBs, we need to ensure that the correct TIB specialisation is patched during method recompilation.

Recall from Section 9.4.1 the use of the RVM's `VM_MethodReference` type during class loading. They are uniquely identified by their fully qualified name and their method descriptor. In order to facilitate recompilation and ensure that the correct TIB is patched we augment this identifier with the associated TIBs. This is achieved through the introduction of a *TIB map* for each `VM_MethodReference` so that it is easy to determine in which TIBs a method reference or its method resides. It is worth noting that this is a one to many relation since, for example, it makes sense to share non-specialised static methods. The TIB map is simply a bitmap, an integer word, whose bits are set according to which TIBs the method reference is valid for[2].

When a method undergoes recompilation the optimising compiler first produces a new instance of `VM_CompiledMethod`. Then, `replaceCompiledMethod` is invoked for its corresponding `VM_Method`. `VM_Method` instances persist for the entire execution of the RVM, however, `VM_CompiledMethod`s are continuously invalidated and discarded with each new method recompilation. `updateMethod` is then invoked for the corresponding `VM_Class` and if the method is virtual, `updateVirtualMethod` is called, otherwise static method tables and interface in-

[2]The use of a 32-bit integer places an upper limit of 32 on the number of permitted TIB specialisations. However, if more are required an integer array could be used although there would be performance implications in doing so.

vocation slots are updated. Finally, in the unmodified RVM, the method slot in the TIB corresponding to the recompiled method is overwritten. In our RVM, we extend this final step to update the corresponding method slot and interface invocation slots for all those TIBs in which the method resides, as determined by the TIB map.

With the infrastructure in place to enable method recompilation in the presence of method and TIB specialisation it is now possible to enable devirtualisation and the inlining of specialised methods — we take a similar approach to the inlining of polymorphic call sites and exploit the speculative inlining infrastructure of the RVM. We introduce a new type of guard, `ig_tib_test` that determines if the inlined method is a method belonging to the current TIB index of the caller. If it is the inlined code is executed, if not the appropriate method must be invoked using `invokevirtual` on the virtual method slot of the current TIB. Because each compiled method has a TIB map that determines which TIBs it belongs to, emitting this guard is relatively simple and a bitwise AND operation is sufficient for the test. The full sequence of operations implemented by `ig_tib_test` are therefore as follows: i) The TIB of the caller is obtained; ii) the TIB index from the TIB using the new low-level IR operand `GET_TIB_INDEX_FROM_TIB`; and iii) the TIB map is used to determine if the inlined method is valid for this TIB index. When the guard is emitted into the compiled code, the TIB map of the method that is inlined is supplied as an immediate operand, for efficiency, so that it need not be retrieved from the method's `VM_MethodReference`. The `ig_tib_test` adds at most three native code instructions, (`mov`, `cmp`, and `jne`), and in many cases only two (the `mov` instruction is not emitted when the TIB pointer is already in a register).

There are two final modifications required to the inlining infrastructure. Previously, it was only ever necessary to emit a single guard at a call site, however we add additional functionality that allows multiple guards to be emitted at the same call site — when inlining polymorphic specialisations it is necessary to emit both the `ig_class_test`/`ig_method_test` and the `ig_tib_test` guards. The final modification is to the inlining oracle — we work to preserve the original inlining behaviour of the standard oracle and differ only in the following ways:

- We choose to only ever inline callees that reside in the primary TIB. Other specialisations are assumed to be called only a fraction of the time. However, it is trivial to define other oracles that do not make this assumption.

- We incorporate a heuristic for factoring in the cost of emitting the `ig_tib_test` guard within the oracle's `inliningActionCost` method, that estimates the total cost of an inlining action.

- The notion of a "complex" guard has been introduced. A complex guard is any guard other than `ig_tib_test`. Where the default inlining oracle tested whether a guard was needed and based decisions on that, in some cases it is now only testing whether complex guards are required.

- Whenever an inlining decision has been made, we test the callee to determine if it has any specialisations. If it does, we emit the `ig_tib_test` guard to the final inlining decision. Thus, a complete inlining decision can have 0, 1 or 2 guards, where previously it could only have 0 or 1.

### 9.4.5 Thread-safe Execution and Concurrent Operation

The ability to flip the TIB at any point within both user code and the RVM's runtime system raises some interesting issues relating to thread-safe operation within our environment. Code emitted by the compilers for the `invokevirtual` instruction *always* reloads the current TIB pointer before invoking a method. As a result there are no memory ordering issues resulting from the use of a cached, out-of-date, TIB pointer. However, the TIB pointer could change at any point during the execution of a method which means that a subsequent method call may invoke any of the available specialisations — method specialisations for the *current* TIB are always executed. Consider an object `O`, its member methods `m` and `p` and their specialisations `m'` and `p'`. The method body of `m'` is written with the intention that `p'` is always invoked. During the execution of `m'`, another thread hijacks the TIB of class `O` *before* the execution of `p'`, such that the original TIB is restored or an alternate TIB specialisation is installed. Although execution of the instructions within the body of `m'` continues, *any* specialisation of `p` may be invoked depending on the hijacking TIB. As a result, when writing specialisations, care must be taken to reason about the effect and interaction of all specialisations of a method on the current object state for all potential callers. It is worth noting that, in this trivial example, if it is desirable for method `m'` to always execute the instructions of method `p'` this can be achieved either via the forced inlining of `p'` or the construction of an appropriate set of specialisations that adhere to the corresponding state transition design. We do however acknowledge that such reasoning for larger programs involving numerous specialisations that interact at multiple call sites could potentially be much more complex.

Similarly, we delegate the responsibility of object locking and thread synchronisation to the method specialisation developer. This provides maximum flexibility, arguably, at the expense of additional complexity. In practice, we expect specialisations to be constructed based on reasoning about global invariants on when TIB replacement may occur and when particular specialisations may execute; indeed, we do exactly this when constructing our garbage collector method specialisations (Section 9.5 and Section 9.6). Note that where atomic execution is necessary for correctness, the RVM's `UNINTERRUPTIBLE_PRAGMA` can be used to ensure the execution of a method body cannot be preempted by another thread.

### 9.4.6 VM 'Neutral' Implementation

Although our implementation uses the Jikes RVM, much of the design is VM 'neutral' and can be applied to VM implementations where the runtime systems are written in native languages such as C or C++. The use of BCEL and the CTTk interface would remain

unchanged. CTTk must however be modified to drive the CTTk and BCEL APIs *solely* by overriding the `findClass()` method of one of the standard class loader extension mechanisms (via a bootstrap, system or context classloader). Having read the class file and effected the transformations (including the beanification transformation for full virtualisation) the BCEL `JavaClass.getBytes()` method can hand off the class as a byte array to the standard `defineClass()` implementation of the classloader. Because method specialisations would now appear as standard methods in the mutated class, CTTk must generate the specialisations using a unique naming scheme, such as: `__spec_<specNum>_<methodName>` where `specNum` is the VMT (previously the TIB) index number and `methodName` is the original method name. The VM must now be modified to:

- Implement an array of VMTs not just a single VMT.
- Recognise class method definitions of the form: `__spec_<specNum>_<methodName>` as specialisations and populate the appropriate VMT for the associated specialisation index.
- Expose `hijackTIB` and `restoreTIB` APIs as `hijackVMT` and `restoreVMT` as publicly accessible APIs via JNI VM extensions.
- Emit an `ig_vmt_test` corresponding to the `ig_tib_test` at polymorphic call sites that determines if the inlined method that is about to be executed is the appropriate method for the currently active VMT specialisation.

The obvious advantage in having implemented our framework in a VM written in Java is that full virtualisation and method specialisation can be exploited within the implementation of the VM itself. Within the framework implementation as envisaged above, only user code and system libraries such as GNU Classpath [GP] libraries as loaded by the various class loaders may be transformed.

One of the difficulties of working with a Java-based VM is that some of the run-time structures are attributable to the VM itself, rather than the executing program. The RVM developers dedicated considerable effort to removing such overheads from the original RVM. We have not attempted any such optimisations in our modified RVM. As a result, we would expect a performance hit, even from the use of the BCEL patch alone, as we end up adding intermediate VM data structures that are themselves subject to garbage collection. We revisit this in Section 9.7.

## 9.5 A Dynamic Dispatching Read Barrier

In our fully virtualised RVM object access can only be via method invocation so it is natural to consider placing the read barrier at each method prologue that accesses a field, i. e. the getter methods of the object. The barrier operates as per the eager barrier described in

Section 9.3.1. Notice that this is a *lazy* read barrier — it is applied to field accesses. There is however an *eager* variant where the from-space / to-space test is not applied to the field undergoing access, but to the accessing object. In this case, the read barrier must be applied to all operations that the object may perform. For our fully virtualised objects this is a trivial modification, we simply place this modified read barrier at *all* method prologues.

However, using specialisations, we can do better! The idea is to use our virtualisation framework to build an *implicit* read barrier. Rather than statically inserting barrier code, we instead virtualise all object accesses and build a second, specialised, version of each access method that first scavenges the associated object before executing the original method body. Recall that the read barrier need *only* be applied to unscavenged to-space objects, and neither to from-space objects nor to *any* objects when the collector is off. The trick is to arrange for the specialised version to be executed only when the object is known to reside in to-space. That way, the act of accessing an object has the side effect of copying it to to-space, *but only when it has yet to be copied.*

By specialising *every* method of the object we a) do not need to place the read barrier at every method prologue, b) the barrier no longer needs to test whether or not the collector is on, and c) there is no longer a need to reserve the extra GC 'forwarding' or 'from-space / to-space distinguishing' bits in the object header. How so? The method body of each specialised method is common between all specialisations for that object — it invokes `restoreTIB` to flip the object's TIB back to the primary TIB, it scavenges the object, evacuating each of its non-primitive fields to to-space, and then it reinvokes itself. This now results in invocation of the primary TIB's version of the specialisation, and so standard execution resumes. Evacuation of each field simply copies the field to to-space and hijacks the TIB, flipping it to its *self-scavenging* specialisation. Notice that the invocation of a specialised method occurs only in to-space when the collector is on (obviating the need for the explicit 'collector on' test), and at most once per collection cycle per object. Furthermore, the scavenge state of the object is implicit, based on whether or not the specialised TIB is installed, as a result no GC bit is needed to distinguish from-space from to-space.

As we demonstrated in Non-Stop Haskell, restricting object headers to a single word is *extremely* desirable, as more of the heap is available for program execution and less time is spent garbage collecting. Eliminating the need for additional GC bits goes some way toward achieving this — ideally the object header is *just* the TIB pointer. For an explicit read barrier, use of the one word object model makes it more expensive: *all* operations, in both the barrier and RVM, that access and manipulate the TIB, must first determine whether the TIB word *actually* represents the TIB pointer or the forwarding address pointer. Also, if necessary, they must dereference the forwarding pointer to obtain the TIB pointer. We need no such tests with our dynamic dispatching read barrier — Section 9.6 describes how we avoid the forwarding test using a dynamic dispatching forwarding pointer. The penalty we pay for all these benefits is that the self-scavenging of all fields in the object make the barrier coarser than its explicit counterpart. However because many fields can be copied and their TIBs

flipped in microseconds, the individual and aggregated (where the barrier is tripped in quick succession) mutator pause quanta that result seldom violate the soft real-time MMU bounds, doing so with negligible probability, as demonstrated in Section 8.5.

### 9.5.1 What to Specialise

Intuitively, the overhead of fully virtualising object accesses is prohibitively expensive. As previously described, we have eliminated much of this overhead that this and the framework imposes by allowing devirtualisation optimisations through guarded inlining of specialised methods. In the context of a dynamic dispatching read barrier there is one further optimisation that reduces this overhead even further — if we are careful, `private` methods do not need to be specialised and `private` fields need not be accessed and therefore wrapped by getter and setter methods. If `protected` methods are invoked only by subclasses (and not publicly at the package level) then they too can be treated in the same way. To determine this, as (re-)compilation proceeds, we flag whether or not a protected method is to be invoked publicly at the package level in order to determine whether this optimisation can be performed. The self-scavenging read barrier need be effected on inter- and not intra-object accesses, i.e. only those operations that provide external access to the object and are therefore initial entry points to the object. In order to ensure correctness, we must arrange for the currently active activation record on the top of each thread stack to be scavenged in its entirety at the initiation of the collection cycle. However, this is no extra work, it is done as standard *anyway* within an incremental collector and is unrelated to the issue of how the stack is subsequently incrementally scavenged.

Recall that Jikes RVM treats the static methods and fields of an object differently — because they are allocated in an immortal area of the heap that is neither garbage collected nor (initially) requires access to the object instance, we do not need to specialise static methods nor provide access to static fields via getter and setter methods.

In summary, we do not need to specialise:

- Static methods including constructors and class initialisers
- Private or solely subclass invoked Protected methods
- Abstract methods — this is non-sensical as there is no implementation.
- Methods that have been marked as being pure access methods by the `BeanifierTransform`

### 9.5.2 Implementation

The complete source of the `IncrGCmethodTransform` is presented in Listing 9.4. This code builds an additional self-scavenging specialisation of all methods that require it and installs the necessary specialisations in the modified TIB structure. We briefly outline how it works:

```
1  public class IncrGCmethodTransform extends Transform {
2    public IncrGCmethodTransform() {}
3
4    public TransformClass transform( TransformClass transformClass ) {
5      JavaClass javaClass = transformClass.getJavaClass();
6
7      // Get the class constant pool
8      ConstantPoolGen constantPoolGen = new ConstantPoolGen( javaClass.getConstantPool() );
9
10     Method[] methods = javaClass.getMethods();
11     int numMethods = methods.length;
12     Method[][] specialisedMethods = new Method[1][numMethods];
13
14     for ( int methodIndex = 0; methodIndex < numMethods; methodIndex++ ) {
15       String name = methods[methodIndex].getName();
16       if (methods[methodIndex].isAbstract() || methods[methodIndex].isStatic()
17                                             || methods[methodIndex].isPrivate()
18                                             || name.equals("<init>")
19                                             || name.equals("<clinit>")) {
20         specialisedMethods[0][methodIndex] = methods[methodIndex];
21       } else {
22
23         MethodGen methodGen = new MethodGen( methods[methodIndex],
24                                              javaClass.getClassName(), constantPoolGen );
25
26         // The method is a GC method, don't let it be interrupted!
27         methodGen.addException(UNINTERRUPTIBLE_PRAGMA);
28
29         // Method body -> invoke restoreTIB, scavenge the object, invoke the method again
30
31         InstructionList instructionList = new InstructionList();
32         int methodIdx = 0;
33
34         methodIdx = constantPoolGen.addMethodref( "org.jikesrvm.VM_ObjectModel",
35                                                   "restoreTIB", "(Ljava/lang/Object;)V" );
36         instructionList.append( new ALOAD( 0 ) ); // load this
37         instructionList.append( new INVOKESTATIC( methodIdx ) );
38
39         methodIdx = constantPoolGen.addMethodref( "org.jikesrvm.mm.mmtk",
40                                                   "scavenge", "(Ljava/lang/Object;)V" );
41         instructionList.append( new ALOAD( 0 ) ); // load this
42         instructionList.append( new INVOKESTATIC( methodIdx ) );
43         instructionList.append( new ALOAD( 0 ) ); // load this
44
45         // Push all arguments required to call a function onto the stack
46         Type[] argumentTypes = methodGen.getArgumentTypes();
47         for ( int argIndex = 0, numArgs = argumentTypes.length, localVarIndex = 1;
48               argIndex < numArgs; argIndex++ ) {
49           LocalVariableInstruction load =
50                    InstructionFactory.createLoad( argumentTypes[argIndex], localVarIndex );
51           instructionList.append( load );
52           localVarIndex += argumentTypes[argIndex].getSize();
53         }
54
55         // Invoke the virtual method
56         methodIdx = constantPoolGen.addMethodref( methodGen );
57         instructionList.append( new INVOKEVIRTUAL( methodIdx ) );
58
59         Type returnType = methodGen.getReturnType();
60         instructionList.append( InstructionFactory.createReturn( returnType ) );
61         specialisedMethods[0][methodIndex] = finaliseMethod( methodGen, instructionList );
62       }
63     }
64
65     // Update the constant pool
66     javaClass.setConstantPool( constantPoolGen.getFinalConstantPool() );
67     javaClass.setMethods( methods );
68
69     transformClass.setSpecialisedMethods( specialisedMethods );
70
71     return transformClass;
72   }
73 }
```

Listing 9.4: The `IncrGCmethodTransform` — The Self-scavenging Specialisation Transform

- Lines 1 to 11 and 64 - 73 encode class definition, initialisation and finalisation of the transform and the class constant pool. Line 12 declares the array of method specialisations, and in this case there is only a single TIB specialisation. It is worth noting that transforms can be chained in a pipeline. Were this to be the case then Line 12 would read: `Method[][] specialisedMethods = transformClass.getSpecialisedMethods()`.

- Lines 14 to 63 form a loop that iterates over each of the original methods defined for the class and provides a specialisation for each of them, if appropriate.

- Lines 16 to 19 determine whether a specialisation is required. If not then the entry in the specialisation array points to the standard method body.

- Lines 23 to 24 create a BCEL method generator allowing us to generate bytecode instructions for the method body of the specialisation.

- In line 27 we add a Jikes RVM-specific pragma, implemented as an exception, to the method. Because the self-scavenging method specialisation is essentially an RVM internal method that affects the state of the RVM and the consistency of the heap, it must not be preempted by another thread. The `UNINTERRUPTIBLE_PRAGMA` prevents this.

- The remaining lines build the list of instructions for the method body. Lines 34 to 37 add the instructions to invoke our static `restoreTIB` method on its owning object, i.e. `this`.

- Lines 39 to 42 invoke the RVM's object scavenging garbage collection routine, defined in its memory management toolkit, MMTk.

- Lines 43 to 57 push all the arguments necessary to reinvoke the method, but in its standard, non-specialised form, onto the stack and reinvokes the method, now on the restored TIB.

- Lines 59 to 61 add the instructions to effect the method return. The entry in the method specialisation array is then set. For brevity we omit the definition of `finaliseMethod` — it associates the instruction list with the BCEL method generator, generates the method, sets the stack limits and issues BCEL cleanup operations.

- Finally, we append to the evacuation routines, `copyScalar` and `copyArray`, within MMTk (`org.mmtk.vm.ObjectModel`) with `VM_ObjectModel.hijackTIB( to, 1 )` to flip the TIB to the incremental GC specialisations on object evacuation. The first parameter, `to`, is a reference to the to-space copy of the object, while the second parameter indicates the TIB specialisation to use.

## 9.6 A Dynamic Dispatching Forwarding Pointer

We have already described the motivation and challenges associated with implementing a one word object header model in combination with an incremental collector, highlighting in particular the per-object space saving whilst incurring an overhead on TIB access and manipulation operations that must distinguish the TIB pointer from the forwarding pointer.

Using almost identical code to the `IncrGCmethodTransform`, we can build a second specialisation of an object that implements automatic forwarding via dynamic dispatch, and eliminates the need for logic that distinguishes the TIB pointer from the forwarding pointer. When an object is evacuated to to-space the idea is to leave behind a modified version of the object whose payload contains the address of the object in to-space and whose TIB references the forwarding specialisation. A method invoked on such an object causes the invocation to be applied to the to-space copy of the object.

Using the `IncrGCmethodTransform` as a template, the implementation of the `IncrGCforwardingPointerMethodTransform` in Listing 9.5 is identical, except for a few minor modifications:

- Line 12 of `IncrGCmethodTransform` is modified to allow transform chaining with the `IncrGCforwardingPointerMethodTransform` by declaring two TIB specialisations: `Method[][] specialisedMethods = new Method[2][numMethods];`.
- Line 12 is modified to retrieve the specialised methods of the `IncrGCmethodTransform` within the transform pipeline: `Method[][] specialisedMethods = transformClass.getSpecialisedMethods()`.
- Line 20 is modified to assign the standard method, since no forwarding is required for these particular methods, to the second TIB specialisation: `specialisedMethods[1][methodIndex] = methods[methodIndex]`
- Lines 34 to 37 are removed since we do not restore the TIB for an object undergoing forwarding — it may subsequently be accessed many times before from-space is condemned.
- Line 39 (now Line 34) does not invoke MMTk's scavenge method, but its forwarding pointer dereference method:

```
methodIdx = constantPoolGen.addMethodref( "org.jikesrvm.mm.mmtk",
                                           "dereferenceForwardingPointer",
                                           "(Ljava/lang/Object;)Ljava/lang/Object;");
```

- Line 43 (now Line 40) is modified to load the object reference returned by `dereferenceForwardingPointer` so that the reinvocation of the method

```
public class IncrGCforwardingPointerMethodTransform extends Transform {
  public IncrGCforwardingPointerMethodTransform() {}

  public TransformClass transform( TransformClass transformClass ) {
    JavaClass javaClass = transformClass.getJavaClass();

    // Get the class constant pool
    ConstantPoolGen constantPoolGen = new ConstantPoolGen( javaClass.getConstantPool() );

    Method[] methods = javaClass.getMethods();
    int numMethods = methods.length;
    Method[][] specialisedMethods = transformClass.getSpecialisedMethods();

    for ( int methodIndex = 0; methodIndex < numMethods; methodIndex++ ) {
      String name = methods[methodIndex].getName();
      if (methods[methodIndex].isAbstract() || methods[methodIndex].isStatic()
                                            || methods[methodIndex].isPrivate()
                                            || name.equals("<init>")
                                            || name.equals("<clinit>")) {
        specialisedMethods[1][methodIndex] = methods[methodIndex];
      } else {

        MethodGen methodGen = new MethodGen( methods[methodIndex],
                                             javaClass.getClassName(), constantPoolGen );

        // The method is a GC method, don't let it be interrupted!
        methodGen.addException( UNINTERRUPTIBLE_PRAGMA );

        // Method body -> invoke restoreTIB, scavenge the object, invoke the method again

        InstructionList instructionList = new InstructionList();
        int methodIdx = 0;

        methodIdx =
                constantPoolGen.addMethodref( "org.jikesrvm.mm.mmtk",
                                              "dereferenceForwardingPointer",
                                              "(Ljava/lang/Object;)Ljava/lang/Object;" );
        instructionList.append( new ALOAD( 0 ) ); // load this
        instructionList.append( new INVOKESTATIC( methodIdx ) );
        instructionList.append( new ALOAD( 1 ) ); // load the forwarding pointer value

        // Push all arguments required to call a function onto the stack
        Type[] argumentTypes = methodGen.getArgumentTypes();
        for ( int argIndex = 0, numArgs = argumentTypes.length, localVarIndex = 1;
              argIndex < numArgs; argIndex++ ) {
          LocalVariableInstruction load =
                  InstructionFactory.createLoad( argumentTypes[argIndex], localVarIndex );
          instructionList.append( load );
          localVarIndex += argumentTypes[argIndex].getSize();
        }

        // Invoke the virtual method
        methodIdx = constantPoolGen.addMethodref( methodGen );
        instructionList.append( new INVOKEVIRTUAL( methodIdx ) );

        Type returnType = methodGen.getReturnType();
        instructionList.append( InstructionFactory.createReturn( returnType ) );
        specialisedMethods[1][methodIndex] = finaliseMethod( methodGen, instructionList );
      }
    }

    // Update the constant pool
    javaClass.setConstantPool( constantPoolGen.getFinalConstantPool() );
    javaClass.setMethods( methods );

    transformClass.setSpecialisedMethods( specialisedMethods );

    return transformClass;
  }
}
```

Listing 9.5: The `IncrGCforwardingPointerMethodTransform` — The Dynamic Dispatching Forwarding Pointer

is on the forwarded copy of the object and is the non-specialised version: `instructionList.append( new ALOAD( 1 ) ); // load to-space object reference`.

- Line 61 (now Line 58) is modified to assign the specialise method to the second TIB specialisation: `specialisedMethods[1][methodIndex] = finaliseMethod(methodGen, instructionList);`.

We next append to MMTk's evacuation routines, `copyScalar` and `copyArray`, after our previous `hijackTIB` call that flips the to-space object's TIB to its self-scavenging specialisations, by calling `VM_ObjectModel.hijackTIB( from, 2 )`. This flips the TIB of the from-space object to the incremental GC forwarding pointer specialisations on object evacuation. Finally we corrupt the from-space copy of the object, writing the forwarding address into the first word of the object's payload. Notice again that this supports forwarding without an additional space overhead.

Because we are focusing on the barrier overheads we do not attempt to evaluate the relative performance of our forwarding mechanism. In any case, the majority of the overheads result from the virtualisation and specialisation framework, and are shared with the self-scavenging specialisations. Meanwhile, the tradeoffs of employing both single- and two-word object headers have been extensively studied by Bacon et al. [BFG02].

### 9.6.1 The Final Story

At the end of the specialisation exercise there are three variants of each TIB: one referencing the original bodies of associated class methods, one containing self-scavenging variants of those methods and one containing forwarding variants. Although there is some duplication in the TIB structure, there is only one copy of the original method body code: both the self-scavenging and forwarding variants ultimately invoke the original method body code via a separate invocation. This limits the extent of the static "code bloat".

## 9.7 Evaluation

We have conducted a series of experiments in order to quantify the overheads of the virtualisation framework and the cost of the implicit read barrier when compared with a conventional, explicit barrier, as previously studied for the Jikes RVM by Blackburn and Hosking [BH04]. Recall that the explicit *conditional* barrier of [BH04] is particularly efficient as it merges the 'collector on' test with the 'from-space / to-space' test by assuming that from- and to-spaces can be distinguished by a single bit in an object's address and by flipping a reserved word between 0 and 1 each time a garbage collection cycle is initiated.

Our framework is integrated with Jikes RVM version 2.4.3 (CVS repository version 16 January 2006). Similarly, Blackburn and Hosking's read barrier patch for version 2.3.3 of the

RVM is ported and applied to this 2.4.3 version of the RVM.

We use a subset of the SPEC JVM98 and the DaCapo 'dacapo-2006-10-MR2' benchmarks for evaluation. As we are investigating mutator overhead we chose a subset of the benchmarks that perform minimal amounts of garbage collection. The reference execution time for each benchmark is measured by running it on an unmodified RVM (no read barrier) with its generational copying mark-sweep hybrid (GenMS) collector and the default heap size of initially 50MB growing to a maximum of 100MB. The overheads of virtualisation, and the cost of the explicit and implicit read barriers, are measured by executing the same benchmark on modified versions of the RVM, again with the 'stop-the-world' collector enabled. Because the amount of garbage collection performed is identical in each run, the difference between the average observed execution time and the average reference time provides the average overhead with the garbage collection time factored out. Note that we are only interested in measuring the cost of the barrier, not the cost of any garbage collection operations (e.g. evacuation) that may follow it.

Measuring the cost of the implicit barrier is not easy because the specialised access methods can only be invoked during incremental garbage collection, specifically when the program tries to access an uncopied object. Although we do not have a complete incremental collector, we can nonetheless compute bounds on the barrier cost, based on two extremes.

At one extreme (lower bound), the specialised methods may not be invoked at all — this will happen if the program only touches objects that have already been copied by the scavenger. The additional cost includes that of full virtualisation and one TIB hijack and one TIB restore for each object copied during garbage collection. Note that the hijack, which replaces the object's original TIB pointer with that of the specialised version, will happen even though the program will never get to invoke the specialised method.

At the other extreme (upper bound), all evacuations may take place as a result of the program touching all currently uncopied objects (i.e. no objects are collector-scavenged) — this will only happen if the program performs no memory allocation at all during the collection cycle (assuming a work-based rather than a time-based collector). The overhead in this case is as above, but with one additional call from the specialised method to the original.[3]

We calculate both bounds by artificially adding code to the scavenger of the original 'stop-the-world' collector. The lower bound execution time is calculated by adding a TIB hijack and restore operation each time the scavenger scavenges a live object, thus yielding the overhead that would be imposed on an incremental scavenger in this extreme case. For the upper bound, we do the same but add an additional method invocation on each scavenge representing the additional method call above. The measurements are estimates of performance bounds for the extreme cases, but we can reasonably expect the measured barrier cost in a 'production' incremental collector to lie somewhere between the two.

[3]Note that this extra call could be eliminated by duplicating the original method's body, but at the expense of substantial code bloat.

### 9.7.1 Results

Experiments were conducted on a 2.8GHz Pentium IV with 1GB RAM, a 512K level-2 cache, a 16KB instruction cache and a 16KB, 4-way set-associative level-1 data cache with 32-byte lines. The system runs Mandrake Linux 10.1 with a 2.6.15.1 kernel in single-user mode. All results reported are average overheads taken over five runs.

We use the *pseudo adaptive* driver for the Jikes RVM compiler which applies compiler optimisations deterministically according to an advice file computed ahead of time, so as to avoid variations in those methods that timer-based sampling identifies as 'hot'. We measure both the initial performance and that after optimisation in order to measure the extent to which the gross virtualisation overheads can be recovered by the optimiser. When applied, the optimisations are run to convergence, i.e. they are applied until no further improvement in execution time is observed.

#### Virtualisation Overhead

Table 9.1 shows the initial and fully optimised (converged) execution times for each benchmark for the original RVM implementation (Original) without the read barrier code. The overheads of the fully virtualised implementation in each case (Virtual) is shown as a percentage overhead on the original RVM execution time (for example a 50% overhead on a 10s run equates to a 15s run). The overheads are like-for-like with respect to optimisation (Init and Conv). Without

| Benchmark | Original(s) | | Virtual (% o/head) | |
|---|---|---|---|---|
| | Init | Conv | Init | Conv |
| _201_compress | 6.17 | 5.02 | 42.63% | 8.37% |
| _202_jess | 3.54 | 2.83 | 90.40% | 2.83% |
| _209_db | 13.72 | 13.40 | 11.88% | 2.69% |
| _213_javac | 8.83 | 5.87 | 91.50% | 21.64% |
| _222_mpegaudio | 6.68 | 3.72 | 28.44% | 5.65% |
| _227_mtrt | 5.08 | 2.53 | 30.31% | 7.11% |
| antlr | 5.57 | 4.46 | 328.17% | 5.38% |
| bloat | 15.02 | 11.51 | 165.79% | 11.99% |
| fop | 5.33 | 3.21 | 711.82% | 3.74% |
| luindex | 19.36 | 15.40 | 273.46% | 21.49% |
| pmd | 15.14 | 9.78 | 360.69% | 20.96% |
| Min | 3.54 | 2.83 | 11.88% | 2.69% |
| Max | 19.36 | 13.4 | 711.82% | 21.49% |
| Geo. mean | 8.23 | 5.82 | 102.97% | 7.75% |

Table 9.1: Virtualisation overheads

optimisation, full virtualisation adds between around 12% and 700% to the execution time (gross overhead). The overheads come from the use of BCEL (see below), the cost of additional beanification and the cost of maintaining the modified TIB structure. After optimisation, these overheads are reduced substantially: the net overhead is reduced to between around

2.7% and 22% after convergence. The benefits that accrue from using the AOS are thus substantial, although this is perhaps not surprising. Note that use of the AOS targets long-running (e.g. server) applications where the full benefits of feedback-directed profiling and optimisation are obtained. For this reason, we do not concern ourselves with the time taken for the optimisations to converge.

### Read Barrier Cost

Table 9.2 shows percentage overheads, when compared to the original execution times in Table 9.1, for the lower and upper bound execution times for the implicit read barrier (Implicit) and Blackburn and Hosking's explicit read barrier (Explicit).

| Benchmark | Implicit (Lower) | | Implicit (Upper) | | Explicit | |
|---|---|---|---|---|---|---|
| | Init | Conv | Init | Conv | Init | Conv |
| _201_compress | 42.79% | 8.37% | 42.86% | 8.37% | 24.47% | 43.63% |
| _202_jess | 90.89% | 2.95% | 91.11% | 3.00% | 7.62% | 7.42% |
| _209_db | 12.10% | 2.86% | 1.12% | 2.93% | 12.39% | 12.31% |
| _213_javac | 59.52% | 23.27% | 64.35% | 23.99% | 5.55% | 4.94% |
| _222_mpegaudio | 28.62% | 5.66% | 28.70% | 5.66% | 48.20% | 34.95% |
| _227_mtrt | 42.79% | 8.21% | 31.34% | 8.70% | 2.17% | 32.02% |
| antlr | 328.17% | 5.62% | 328.17% | 5.72% | 11.67% | 8.30% |
| bloat | 165.79% | 12.27% | 165.79% | 12.41% | 21.17% | 18.11% |
| fop* | 711.83% | 4.41% | 711.83% | 4.41% | 32.80% | 28.40% |
| luindex* | 273.46% | 21.49% | 273.46% | 21.49% | 22.74% | 19.03% |
| pmd* | 360.70% | 21.0% | 360.70% | 21.04% | 8.11% | 7.22% |
| Min | 12.10% | 2.86% | 1.12% | 2.93% | 2.17% | 4.94% |
| Max | 711.83% | 23.27% | 711.83% | 23.99% | 48.20% | 43.63% |
| Geo. mean | 102.49% | 8.15% | 80.87% | 8.27% | 13.13% | 15.56% |

Table 9.2: Read barrier overheads. *Note that Blackburn and Hosking's explicit barrier code causes run-time errors on `bloat`, `fop`, `luindex` and `pmd` benchmarks – as reported in the original paper, the read barrier is not completely 'robust' [BH04]. The figures reported here are based on measurements taken before the programs crashed.

The cost of the explicit read barrier varies between around 5% and 44% in the fully optimised case; the results are consistent with the reported measurements in [BH04] to within a few percent.

The implicit read barrier achieves significantly lower overheads than the explicit barrier, except for `_213_javac`. The lower bound figures all show slightly increased overhead when compared to those for virtualisation alone, as is expected. The difference between the two represents the additional overhead of hijacking and restoring the TIB during incremental garbage collection.

The upper bound measures the effect of an additional method invocation when scavenging objects (the cost of invoking the original method body having restored the object's TIB). Notice that even the upper bounds on overheads are still substantially lower than for the

explicit barrier, again with the exception of `_213_javac`.

| Benchmark | Virt$^n$s | Devirt$^n$s | Code Bloat(%) |
|---|---|---|---|
| _201_compress | 5189 | 2989 | 10.1 |
| _202_jess | 5977 | 3684 | 13.7 |
| _209_db | 5078 | 2564 | 23.2 |
| _213_javac | 9159 | 2668 | 11.8 |
| _222_mpegaudio | 6172 | 3673 | 32.2 |
| _227_mtrt | 5499 | 2635 | 28.1 |
| antlr | 14025 | 4716 | 21.2 |
| bloat | 9871 | 9781 | 12.1 |
| fop | 9295 | 3431 | 24.2 |
| luindex | 6661 | 2513 | 18.2 |
| pmd | 13898 | 2928 | 26.3 |

Table 9.3: (De)Virtualisation counts and (static) code bloat

In order to quantify the effects of the CTTk transformations when generating the implicit read barrier implementations, Table 9.3 shows counts of: i) The number of getter/setter methods generated for beanification by our framework (Virt$^n$s); ii) The number of such methods that are eventually inlined by the AOS, but guarded by `ig_tib_test` guards (Devirt$^n$s); and iii) The code bloat, which is the percentage increase in the original RVM code size attributable to virtualisation (additional beanifier methods and TIB structure in particular) and specialisation (self-scavenging method variants). The average code bloat for the chosen benchmarks is around 20%.

Note that in the case of `bloat` nearly all of the compiler-generated methods of i) end up being inlined. However, this does not necessarily guarantee to maximise the performance improvement obtained from the AOS. Recall that inlining adds a cost, namely that of the additional `ig_tib_test`. At the same time, there may be virtual methods in the original code (i.e. not those added by the virtualisation framework) that will be executed with little or no additional virtualisation cost; these methods may or may not end up being inlined themselves. If a majority of virtual method calls end up being made to these existing virtual methods then inlining of the compiler-generated getter/setter methods may yield proportionally less benefit in performance. This explains why increased devirtualisation does not necessarily yield a corresponding increase in performance after optimisation (Table 9.2).

### BCEL Overheads

Table 9.4 reports the percentage overheads on the original initial and fully optimised (converged) execution times (Table 9.1, columns 2 and 3) for each application when compiled with Zigman's BCEL patch, but *without* the TIB and method specialisation modifications of CTTk. The results show that use of the BCEL patch alone adds quite significantly to the execution time; this is a result of additional VM data structures that have to be maintained and garbage collected (see Section 9.4.6). The BCEL overheads account for a substantial pro-

portion of the net overheads observed from the virtualisation and read barrier experiments. The results of Table 9.4 suggest that, if it were possible to eliminate the BCEL overheads

| Benchmark | BCEL only | |
|---|---|---|
| | Init | Conv |
| _201_compress | 33.06% | 7.57% |
| _202_jess | 37.85% | 2.47% |
| _209_db | 6.12% | 1.27% |
| _213_javac | 38.51% | 13.12% |
| _222_mpegaudio | 9.88% | 2.42% |
| _227_mtrt | 30.12% | 3.95% |
| antlr | 189.77% | 4.26% |
| bloat | 111.32% | 8.95% |
| fop | 552.72% | 3.12% |
| luindex | 58.73% | 14.09% |
| pmd | 283.16% | 14.83% |
| Min | 6.12% | 1.27% |
| Max | 552.72% | 14.83% |
| Geo. mean | 56.02% | 5.19% |

Table 9.4: BCEL overheads

altogether, the virtualisation costs reported above could be reduced to a maximum of around 7%, and the read barrier costs to around a maximum of around 11%. This could be achieved, for example, by working with a native-code VM, rather than the Java-based RVM, or by working to remove the additional overheads in the RVM, as discussed in Section 9.4.6. There is clearly scope for additional work in this area.

## 9.8 Conclusion and Future Work

We have presented a virtualisation and method specialisation framework for Java that facilitates efficient, dynamic modification, of the behaviour of object accesses at run time. The framework can be used to simplify the often complex and error-prone task of efficiently implementing a range of common runtime services. The dynamic, adaptive, optimisation system is exploited to improve performance, partially recovering the overheads of full virtualisation based upon run-time profiling.

Our results suggest that we have reached an excellent compromise between efficiency and ease of use. Both the implicit read barrier and the dynamic dispatching forwarding pointer implementations, presented in Section 9.5 and Section 9.6, proved relatively easy to build. More importantly, the optimisation framework eventually delivered pleasing and extremely promising performance benefits, when compared with an existing conventional barrier implementation. Our experiments also suggest that further, substantial performance improvements could be achieved either by eliminating the overheads that the BCEL library imposes, or by

implementing the framework within a native VM implementation.

There are two obvious areas for further development. We are currently implementing a fully operational incremental collector for the Jikes RVM, building on our existing read barrier implementation. A crucial feature of the implementation is the adoption of the more optimal single-word object model that is facilitated by the use of a "dynamic dispatching forwarding pointer" which uses additional specialisations to direct object accesses in from-space automatically to their copies in to-space [CFNP07]. This allows us to reduce the space overhead that is usually incurred when supporting non-destructive object forwarding within incremental copying collectors. In addition, we are investigating the use of the RVM's on-stack replacement mechanism for the hijacking of an unscavenged stack frame of a currently executing method. The return address is "replaced" with its specialisation counterpart that incrementally scavenges the stack frame and hijacks the frame beneath it before resuming execution of the original method target, as described in Section 7.1.4. The framework lacks expressiveness and flexibility in the way in which the CTTk transforms bytecode and the way specialisations are defined. The addition of a domain specific language (DSL), that provides syntactic sugar for (and therefore hides) many of the BCEL API sequences that are commonly used, will add substantially to expressive power of the framework. In particular, a DSL would eliminate the rather cumbersome way in which individual transformations are currently sequenced within the transformation pipeline.

### 9.8.1 Other Applications

The exploitation of dynamic dispatch and proxy objects in OPJ [MBMZ01] eliminates the persistent read barrier in a similar way to our approach to incremental garbage collection. It would appear to be straightforward to build an OPJ implementation within our framework, building on our current work. Interestingly their measurements suggest that the use of façades results in a slow-down on the order of 20%. It would be interesting to explore the extent to which our approach might reduce these overheads.

The cJVM [AFT99] shares many features in common with our infrastructure except that we support arbitrary and user definable method specialisations and a public API for switching between them. Our framework is eminently applicable to distributed shared memory, and our approach would follow closely that taken by both cJVM and dJVM [ZS03].

An obvious application of our framework is for the rapid deployment and hot swapping of multiple versions of instrumented debug code or enablement of message logging facilities at different trace levels. Previous instrumentation profiling techniques have either relied upon bespoke implementation within the VM, direct modification of application code, or have been aspect oriented [PWBK07] in nature. These are seldom hot swappable, therefore requiring (repeated) class loader invocation for enablement and deployment.

# Chapter 10

# Conclusion

## 10.1 Summary of Contributions

The motivation for this work (see Chapter 1) was to investigate techniques that reduce and eliminate the costs of mutator-collector synchronisation for incremental real-time garbage collectors, within the context of dynamic dispatch languages. To this end, basic garbage collection techniques (see Chapter 2) and soft and hard real-time algorithms were introduced and the current state-of-the-art surveyed (see Chapter 3).

Our key objective was to explore the real-time collector design space for a range of benchmark applications in order that the tradeoffs of read barrier, write barrier, and barrierless schemes could be better understood in dynamic dispatch languages. Armed with such understanding, we sought to develop a real-time collector that can be scheduled to meet soft real-time constraints on the order of 10 milliseconds.

Mechanisms for the optimisation of Baker's incremental read barrier and a remembered set generational write barrier were presented. The optimisation and elimination of these barriers was realised in a fully functional, barrierless, incremental multi-generation garbage collector for Haskell. The performance characteristics of the collector were evaluated, with particular focus on the reduction in mutator overhead resulting from barrier elimination and also on the mutator's minimum utilisation. We also briefly discussed the challenges in efficiently implementing a replicating or Brooks-style collector in Haskell. In addition, the study highlighted and measured the punitive effects on collection time incurred by the space overhead required to support a Brooks-style indirection barrier. To our knowledge, this is the first time such a study has been performed, and our barrier is the most efficient implementation of a conditional software read barrier to date.

There is much debate within the real-time collector community as to whether time-based collectors are the necessary emerging technology for real-time collector implementations. As discussed in Section 3.3.3, the MMU for a work-based collector, at an interval of $\Delta t$, is determined by the program's allocation rate. Critically, bursty allocation often drives the

MMU so low as to force the collector outside of 'real-time' classification. Our analyses in Section 8.5 of both pause time distributions and minimum mutator utilisation, demonstrate that aggregated pause times of work-based collectors striving to meet pause times between 10 and 50 milliseconds result in a high risk of real-time bounds violation. In an attempt to achieve consistent mutator utilisation of 50% for time quanta in the range of 1 to 10 milliseconds, we exploited GHC's lightweight thread scheduler and its block allocator to create a time-based collector that achieves this, albeit at the expense of a potentially larger virtual memory footprint. In defence of a work-based collector, by adapting to an application's allocation rate it requires no complex analyses or application dependent parameterisations and very little tuning. As a result, for interactive applications, the incrementality it provides can, on average, often be within acceptable bounds. Siebert's body of research [Sie02, Sie07] implements an incremental mark-sweep collector based on the on-the-fly collector of Dijkstra et al. [DLM$^+$76] for the Jamaica Virtual Machine. The collector employs "automatic work-based pacing" and processes a block-allocated heap in 32-byte block increments to "fight" external fragmentation whilst achieving microsecond pause times. This work reports impressive results in support of work-based scheduling. However, we contend that it is the non-moving nature of the collector, and the finer grained atomicity of mark and sweep operations, that limits bounds violation due to the aggregation of pauses that we have demonstrated to be so problematic for copying collectors. Our experience in developing both work-based and time-based collectors and the results that they yield are exactly in line with those stated by Detlefs [Det04] and by Bacon et al. [BCR03b]. This experience also leads us to advocate, at least within collectors that compact and defragment through object copying, the use of time-based mechanisms.

Finally, we investigated the effectiveness of our barrier elimination techniques for environments where dynamic dispatch is less pervasive and where devirtualisation is an aggressively employed optimisation. This culminated in a promising prototype implementation for Java. This 'barrierless' scheme provides an interesting starting point for an incremental collector for Java that avoids the Brooks-style indirection. The performance results for the read barrier from Chapter 8 and Section 9.7 provide some convincing evidence to suggest that the overhead of such a collector may be significantly lower than existing schemes (e.g. the Metronome), were such a collector to be implemented (see Section 10.2).

The infrastructure we provide to realise our dynamic dispatching barriers is not limited solely to garbage collection. Additionally, it provides a generic framework where virtualisation, via method specialisation can be exploited, while preserving optimisation opportunities resulting from devirtualisation. As a result, the framework enables low-overhead implementation of the key mechanisms that support, such mechanisms as i) orthogonal persistence, ii) instrumentation-based profiling, iii) logging and iv) transparent virtual pointer implementation for distributed and cached environments.

These results, in combination with our conclusions, validate the thesis of this dissertation, defined in Chapter 1:

*Devirtualisation and compile-time dynamic dispatch elimination are assumed optimisations. They should not be considered so. The presence of dynamic dispatch enables a class of runtime system optimisations, particularly, but not exclusively, suited to efficient garbage collector implementation, that outweigh the benefits of their indiscriminate removal.*

## 10.2 Future Work

The experiments of Section 8 demonstrate that the smallest quantum that our time-based collectors are currently able to achieve lies somewhere between 5 and 10 milliseconds. Our schemes for: i) incremental scavenging of the stack; ii) minimal processing of the roots and remembered set lists (using self-evacuating-and-scavenging specialisations); and iii) the handling of large unpointed objects, all reduce the chance of overrunning collector quanta during initiation, termination and collector-scavenging phases, but such overruns are still possible. This is reflected in experimental results where the minimum mutator utilisation is reduced and consistent utilisation is only achieved at between 10 and 20 milliseconds. The main objective for our uniprocessor implementation is to further reduce these overruns, thus lowering and tightening this range, in order to target a quantum of, for example, 1 millisecond. This can only be achieved with additional research that investigates the use of techniques such as *remembered set triggers*; and the enhancement of the scheduling heuristics and a reduction in the granularity at which yield points are emitted.

Another key challenge is in the reduction of the footprint of the generational collector so that memory is made more readily available to the block-chaining copy reserve; this prevents cycles having to be forced to completion. To achieve this, an algorithm that allows condemned from-space blocks to be reused for dynamic heap expansion of the nursery, as soon as they become available, must be developed. To support this, the write barrier becomes more complex and must now be bi-directional. Adding support for a bi-directional remembered set write barrier will incur additional dynamic space overheads, potentially on *all* objects, which may prove to be prohibitive. This additional overhead may, however, facilitate further write barrier optimisation and elimination opportunities using our dynamic-dispatching write barrier techniques. An alternative approach could use a card marking scheme which would result in an algorithm that would have much in common with an *incremental* version of the Garbage-First algorithm of Detlefs et al.

When operating in a memory constrained environment with a high residency, the cost of using a purely copying generational collector is high. It is for these scenarios that GHC enables its stop-the-world, in-place, compacting collector for a major collection. An obvious extension to our collector is the incorporation of an analogous incremental, in-place, compactor. The current algorithm employs pointer reversal. Thus, modifying it to run incrementally with the mutator would most likely incur a prohibitively high overhead. A better choice of algorithm, as a starting point for such an investigation, would be those of the stop-the-world *table-based* compactors [HW67, Weg72, FN78].

Haskell has emerged as one of the fastest, state-of-the-art concurrent programming languages, as a result of the incorporation of *Software Transactional Memory* into recent versions of GHC. While the latest version of GHC employs a parallel stop-the-world collector that exploits the block-allocated heap to perform load-balancing [MHJJ08], the addition of a concurrent collector to support the concurrency of Multicore Haskell is an obvious goal. An exciting area of further work, therefore, is the combination of our incremental collector with this parallel collector and its enhancement for concurrent execution. Concurrent copying collectors typically incur high overheads in order to coordinate and synchronise collector copying and scavenging operations with mutator update operations. We speculate that such overheads in Haskell are much lower, and the synchronisation requirements looser, due to its (largely) immutable update semantics. Closure evacuation by multiple threads simply results in the loss of sharing, repeated evaluation, and increased heap usage, but this is not erroneous with respect to mutator execution. Furthermore, by using the processor's compare-and-swap instruction, info table (and TIB) specialisations can be atomically installed. This appears to open up a range of additional applications of our dynamic-dispatching techniques for reducing and eliminating these synchronisation overheads within concurrent algorithms. For example, on evacuation, a collector thread can install a variant of the dynamic-dispatching forwarding pointer specialisation, in the from-space object, that "locks out" and blocks any mutator threads that attempt to access this object, until it has been fully copied and its self-scavenging variant deployed in to-space. This latter technique, and the implementation of an incremental, parallel and concurrent collector for Jikes RVM are currently being investigated within the Department of Computing at Imperial College London using the framework we have developed in Chapter 9.

# Appendix A

# The Glasgow Haskell Compiler

Over the following section we detail the structure of GHC's internals, describing: i) the compiler; ii) the implementation strategy for the evaluation model; iii) the runtime system, its scheduler, storage manager, and garbage collector interface. We pay particular attention to those components that be must modified for our incremental collector implementations.

## A.1 The Compiler

GHC's compilation pipeline makes extensive use of external tools, most usually, those of the GNU `gcc` tool chain. Compilation of a Haskell module proceeds with processing by the `unlit` pre-processor to remove any literate programming style markup and then the standard C pre-processor `cpp`. Then the 'GHC compiler proper' runs, the output of which is then passed to the `gcc` C compiler. A Perl script, the *Evil Mangler*, then post-processes the generated assembly file before invoking the GNU `asm` assembler and `ld` object linker.

The *Evil Mangler* is a Perl script, and a necessary evil — it eliminates the compiled prologue and epilogue of each C procedure (GHC compiled code manages its own explicit stack and the C stack is used only for runtime system return addresses and foreign language calls) and moves a closure's entry code adjacent to its info table (see Section 4.5).

The compiler 'proper' consists of about ten key modules that span approximately 250 source files and in excess of 130,000 lines of Haskell code:

- A recursive-descent parser that produces an abstract syntax tree that represents every construct in the Haskell source language, even where there is only a syntactic distinction — thus improving the readability of error messages relating to type checking.
- A *renamer* that resolves scoping and naming issues, more specifically, those relating to module imports and exports.
- A *type inference* module that annotates the program source with type information, and removes overloading via transformation.

- A *desugarer* that converts the Haskell abstract syntax into a very much simpler functional language called the *Core language*.
- A module containing a variety of optional *Core-language transformation passes* that improve and optimise the code.
- A module that converts Core to the *Shared Term Graph (STG) language*, a simpler and more concise purely functional language (Section 4.4).
- A module containing a variety of optional *STG-language transformation passes* that improve and optimise the code.
- A *code generator* that converts the STG language to either *Abstract C* and more recently straight to C−− [JRR99]. Abstract C is an intermediate C representation and is simply a set of GHC's internal data types that can be printed in C. C−− is the final intermediate language that performs an equivalent role to `gcc`'s *Register Transfer Language*.
- A *target-code* printer prints Abstract C or C−− in a form acceptable to a C or C−− compiler.
- The target-code printer can be replaced on some platforms by a *native code generator* (NCG) that produces architecture specific assembly code. A significant portion of the NCG is concerned with register allocation.

Two key strategies are fundamental to the design of GHC. In the first, compilation proceeds as the application of a series of correctness preserving transformations. In the second, type information is preserved throughout the entire compilation process right up until code generation. Not only does this allow type errors to be generated that relate to the original source, but the strictness analysis algorithm requires it in order to reach fixed-point.

## A.2 The Evaluation Model

In Section 4.4 the basics of GHC's *push/enter* evaluation model were introduced. When entering a closure, the evaluator loads abstract machine register $arg_1$, or `Node`, with the address of the closure and jumps to its entry code. The entry code accesses the environment for the closure, its payload, via $arg_1$ (Section 4.5). Historically, push/enter has been the favoured evaluation model of lazy language implementations — the evaluation of unknown (curried) higher order functions is simple and intuitive. Since our prototype implementation of Nonstop Haskell (Section 5) the *eval/apply* evaluation model has also been incorporated. This has allowed the GHC team to compare quantitatively the performance of both implementations, and also to compare them qualitatively for ease of maintenance [MJ04]. Surprisingly, such an evaluation has not previously been performed. The conclusion from [MJ04] is that in terms of both performance (on average, 2–3% better), and the ease of its implementation and

maintainability eval/apply is the superior evaluation model. For reasons that will become apparent, this is also our conclusion with respect to the support of incremental garbage collection. Before summarising the key differences between the models we briefly describe GHC's evaluation stack.

Each Haskell thread has its own explicit stack upon which continuation frames are pushed as each expression undergoes evaluation. Once evaluation completes the popping of the top continuation frame determines how the flow of execution proceeds. Unlike the abstract machines described in Chapter 4, GHC combines the update and return stacks into a single evaluation stack. Two types of continuation frame are common to both the evaluation models, *update frames* and *case continuations*. The layout of a stack frame is identical to that of a closure (recall Section 4.5). The return address *is* the info pointer and the info table describes the active pointer/non-pointer slots of the frame. Update frames(see Section 4.3 and Section 5.1.3) contain the address of the thunk that is to be updated with its WHNF. Its return address, i.e. info pointer, points at shared entry code that simply performs the update, pops the frame and returns to / enters the topmost stack frame.

Recall that pattern matching is transformed into a 'case' expression encapsulating the alternatives: `case` *scrutinee* `of` *alternatives*. Furthermore, the evaluation of a thunk is always triggered by a 'case' expression. For example, when evaluating an expression representing an if-else conditional, the alternatives encode the two possible branches:

$$\begin{aligned} \texttt{case}\ e_1\ \texttt{of} \quad & \texttt{True} \rightarrow e_2 \\ & \texttt{False} \rightarrow e_3 \end{aligned}$$

The evaluation code for this expression must push a *case continuation* frame onto the stack. This frame encapsulates the free variables of each alternative, $e_2$ and $e_3$, that must be saved across the evaluation of the scrutinee, $e_1$, and a return address, an info pointer, pointing to entry code for the alternatives. Having pushed the frame the scrutinee is evaluated by entering $e_1$. $e_1$ will eventually reduce to either a constructor or an unboxed data type, at which point the return address on the top of the stack is entered. The benefit of incorporating a return address into an update frame should now be apparent — the evaluator can return to the topmost stack frame without having to test whether it is an update frame or a case continuation.

GHC supports both register- rich and poor architectures. Abstract machine registers $arg_1..arg_n$ are therefore mapped directly to machine registers while $arg_{n+1}..arg_m$ are passed on the top of the stack, where $n = 0$ is supported. When the evaluator returns to the top stack frame it does so according to a return convention based on the type of the scrutinee. Data types often have more than one constructor, so how does the return address encode which one is being returned? Before evaluating the scrutinee, a pointer to a *return vector* is pushed by the case expression onto the stack. This is a static structure containing code pointers for each of the alternatives juxtaposed with an info table that describes its layout. Considering

the example above, the constructor `True` returns by jumping to the first code pointer in the return vector, whilst `False` returns by jumping to the second. In both cases, $arg_1$ has been loaded with the closure's address. Like update frames and case continuations, no test needs to be made to discriminate between the two constructors; execution just resumes at the correct branch. For single constructor data types, or data types with many constructors, for which the return vector will be large (and also sparse), it is more appropriate to use a *direct return*. In these cases, the return address is an info pointer to either to the entry code for the single alternative, or to entry code that explicitly performs the conditional test that was previously avoided, by testing the tag of the closure (in $arg_2$) to determine which constructor is being returned. It is worth noting that stack frames are not popped until the last possible moment so that a valid frame is encountered on top of the stack should a garbage collection occur.

## A.3 Push/Enter versus Eval/Apply

Consider the higher-order function `flip`, that flips the order of its arguments before applying the *unknown function* `f`.

```
flip :: (a → b → c) → b → a → c
flip f x y = f y x
```

The two evaluation models differ in how they evaluate the application of `f y x`.

### A.3.1 *Push/Enter*

In the push/enter model the *pending arguments* (here x and y) of the argument function `f` are pushed onto the stack and the function's *entry* code is *entered*. At the prologue of the entry code is a function's *argument satisfaction check* that inspects the number of arguments on the top of the stack, comparing them with the function's (compile-time) arity. If there are a sufficient number of arguments, then evaluation of the function body proceeds, binding the arguments with their formal parameters. If the function is saturated and there are extra arguments, these are left in place for consumption by the function that results from the evaluation. If however, there are too few arguments, the function must then construct a partial application (PAP), that captures the available stacked arguments, and return it as a value to the caller, awaiting the availability of further stacked arguments. Subsequent entry of the PAP unpacks its arguments onto the stack and enters the function.

Unlike the other stack-based continuation types, pending arguments do not incorporate a return address — they are never 'returned to' by the evaluator, but are grabbed when entering the function closure as described above. As such there is no info table or entry code associated with them and they do not follow the uniform representation presented in Section 4.5. In order to determine how many arguments are resident on the stack, an abstract machine

register, *Su*, is maintained that points to the topmost update or case continuation frame. The argument satisfaction check deduces the number of arguments by taking the difference between the `Su` register and the top-of-stack pointer. The compiler generates two entry points for each function. The *slow entry point* is used to evaluate unknown functions as described above, where all (available) arguments reside on the stack and the argument satisfaction check determines how evaluation proceeds. If there are a sufficient number of arguments, the slow entry point 'falls through' to the *fast entry point* to perform the evaluation proper. The fast entry point avoids the argument satisfaction check and is used for the evaluation of known functions with its arguments pre-loaded into the correct registers.

Consider this function (taken from [CFS+04]):

$$\texttt{f x} = \texttt{let } \{\texttt{g y} = \texttt{x} + \texttt{y}\} \texttt{ in } (\texttt{g 3},\ \texttt{g 4})$$

The function `g` will be represented by a dynamically-allocated function closure, capturing its free variable `x`. At the call sites, GHC "knows" statically what code will be executed, so instead of entering the closure for `g` via the slow entry point, dereferenced from the `Node` register, it simply loads a pointer to `g` into a register, and the argument (3 or 4) into another, and *jumps directly* to `g`'s fast entry point.

The pending argument continuations of the stack that intersperse update frames and case continuations make accurate walking of the stack by the garbage collector difficult — there is no info table that describes the layout and allows non-pointer arguments to be discriminated from the pointer arguments that must be scavenged. Furthermore, the collector must be able to distinguish the last pending argument from the return address of the next (regular) continuation frame. GHC addresses these problems using tag words that describe the size of the non-pointer pending arguments sections and which are easily distinguishable from pointer arguments and return addresses using address-based tests. Pointer arguments occur much more frequently than non-pointer arguments and so, as a performance optimisation, they are not tagged. Instead a more complicated address-based test is employed by the collector.

### A.3.2 *Eval/Apply*

Consider again the example function `flip` above. In the eval/apply model, the caller *evaluates* the unknown function (`f`) and then *applies* it to the correct number of arguments. Thus the caller is responsible for the arity matching of the argument satisfaction check. This requires the run-time extraction of the arity for the function from the associated closure, in this case, that of `f`. If the arity of the function matches the exact number of arguments on the stack (in our example, this is two) then a tail-call to code for the function's body (`f`) is made, passing the arguments (`y` and `x`) and the address of the closure for access to the function's free variables. If the arity is less than the number of available arguments (the saturated case), then a *call continuation* must be pushed onto the stack which encapsulates the excess argument(s) (`x` and `y` if `f` has arity zero and `x` if `f` has arity one). The code for the function (the fast entry

point above) is then entered. When this call completes, the process is repeated, proceeding with the inspection of the resulting function's arity before application to the argument(s) from the continuation frame. If the arity is greater than the number of arguments available (`f` has arity greater than two), then a closure for the partial application `f y x` must be constructed and returned.

To simplify the implementation, return addresses for the most common call continuation code sequences are pre-generated (as `stgApply` functions) and built into the runtime system, each of which represents a call sequence for $1..n$ arguments. If more then $n$ arguments are required, a sequence of call continuations are pushed. It should also be noted that for $n$ arguments, $n^3$ code sequences are required that combine the possible argument permutations for pointers, and 32- and 64-bit non-pointers.

One of the most significant benefits of adopting eval/apply over push/enter is that all stack continuations are represented by stack frames employing the uniform representation described in Section 4.5. In particular the problematic pending arguments sections are replaced by call continuation frames that incorporate the arguments along with info pointers for the return addresses and corresponding info tables describing their layout. As a result, the garbage collector can now accurately and trivially walk the stack. Furthermore, the runtime cost of tagging non-pointer arguments and distinguishing pointers from return addresses is eliminated.

## A.4 The Runtime System

GHC's runtime system is written predominantly in C and spans fifty nine source files and ninety one header files. In total, there are just under 60,000 lines of code, the majority of which makes extensive use of macros. Macros are used for efficiency, eliminating the overheads of functions (register storage and stack frame pushes) that can be used to achieve the same task. The object code that results, whilst more efficient is also larger — the result of macro expansion and their implicit inlining. Therefore, there is a trade-off of efficiency against readability, maintainability, and code size that must be made. Predominantly, efficiency is chosen, the unfortunate result of which is that the code can be difficult to read and debug.

### A.4.1 The Scheduler

GHC supports a lightweight, *green threads*, concurrency model [PJGF96] in which many Haskell threads are multiplexed for scheduling against a single operating system thread thus avoiding the use of expensive OS synchronisation primitives to coordinate access to the Haskell heap. GHC's thread scheduler runs within the context of a single OS thread and is responsible for the selection of, and transfer of control to, the next runnable Haskell thread. Each Haskell thread has its own explicit evaluation stack that is separate from the C stack. The C stack is used to invoke foreign function calls from Haskell, and by the support services of the runtime

system and, in particular, the scheduler and garbage collector. A Haskell thread's stack along with the relevant abstract machine registers are allocated within a *Thread State Object* (TSO), a contiguous block of memory in the Haskell heap, and as such it adopts the uniform representation described in Section 4.5. A TSO is an *unpointed* object because its info table has no entry code associated since it is never entered by the evaluator.

The scheduler manages the global program state of a Haskell program, linking each thread's TSO onto one of several lists, depending on its current execution state. This state and these lists constitute most of the garbage collector's root set. The majority of the lists are typical of a standard thread scheduler, representing queues of: runnable threads, and blocked threads, either sleeping on a timeout, or awaiting completion of outstanding I/O operations. A key non-standard list contains those threads executing external foreign function calls. Such threads could call back into the Haskell "world" or the runtime system itself (for example by triggering a garbage collection) and so the scheduler must be reentrant. To support the foreign function interface (FFI) the scheduler also maintains a *stable reference table* for the collector's root set. Stable references are pointers to Haskell heap objects that have been passed to a foreign language "world", such as C, via the FFI, and maintaining their liveness prevents the collector from collecting the object if they are not referenced from within the heap but solely via these external references.

GHC's runtime system incorporates two evaluators, a *machine code evaluator* for the execution of compiled code, and a *bytecode evaluator* that executes as GHC's interpreted interactive environment executes. Execution of the evaluators may be interleaved so that compiled code may be invoked from interpreted code and vice versa. When the current evaluator returns to the scheduler it pushes a closure onto the stack. The scheduler runs in a loop, selecting a runnable thread and resuming its execution, passing control back to the evaluator, by entering the closure on the top of the thread stack. The evaluator executes the current reduction operations for the thread, before returning to the scheduler. In keeping with convention, a Haskell thread returns to the scheduler when it terminates, becomes blocked, or is 'preempted'. Haskell threads run, by default, for a time-slice of 10 milliseconds. Haskell threads cannot be actively preempted; instead they yield to the scheduler, at its request, through the periodic checking of a global preemption flag. During interpreted execution, the bytecode evaluator checks the preemption flag on entry to a closure. However, the machine code evaluator only checks the preemption flag on failure of a closure's *heap check* or *stack check*, the point at which the thread must return to the scheduler to allow memory management activities to occur. The two checks are a part of every closure's standard entry code and, in the case of a function closure, follow the argument satisfaction check of the slow entry point. Failure of a heap check indicates that the current nursery block (4KB in size) is full and the bump pointer must be moved to the next block in the chain or that the nursery is full and that the garbage collector must be triggered. Failure of a stack check indicates exhaustion of the contiguous memory allocated within the TSO to a thread's stack. The scheduler then allocates a new TSO for the thread in the heap doubling its previous size. This 'preemption'

strategy is simple and limits the code size because the check occurs from within the generic shared nursery extension code, but the downside is that if a thread performs little allocation then it may exceed its allotted time-slice.

In addition to the above, a Haskell thread will also return to the scheduler if it must make a foreign function call or if it enters a closure, or returns to a continuation, built by the other evaluator.

### A.4.2 The Storage Manager

The motivating design decision that underpins GHC's Storage Manager is that of *block allocation* where memory is managed as discontiguous blocks of 4 KB in size. There are several benefits that make a block allocated memory manager more flexible than its contiguous counterparts. Firstly, the heap can grow dynamically, as needed, without the need for an explicit *a priori* parameterisation at runtime. If more memory is required from the operating system, it can just chain it on, without the need for it to be contiguous. Secondly, segregated heap regions may be allocated and managed according to different policies, not just in the style of a generational collector, but any multi-partition collector (see Section 2.4.2). Similarly, objects may be type or size segregated or allocated within blocks that are "pinned" and thus undergo scavenging but not copying. Thirdly, the nursery can be allocated so as to fit entirely within the cache. Finally, programs can run with a smaller memory footprint. For example when heap residency is 50% the total memory requirement of a copying collector is only three times the amount of live memory, as opposed to four. Furthermore, blocks that become free are immediately available for re-use. As we shall see in Section 8.3, such flexibility of block re-use is crucial in providing additional headroom for meeting soft real-time goals that would otherwise be missed.

Figure A.1 shows the segmented, layered architecture of GHC's storage manager that, along with the garbage collector, comprises a significant proportion of the runtime system. The *Megablock Allocator* is responsible for obtaining contiguous chunks of memory, in multiples of a (default) 1 megabyte unit, from the host operating system. The allocator encapsulates operating system and architecture specific methods for requesting these *megablocks*, primarily using the `mmap()`, or if unsupported, the `malloc()` system calls. The allocator does not currently return memory to the operating system, nor does it manage its own free list of megablocks. It simply passes megablocks to the next layer of the Storage Manager, the *Block Allocator*, when requested by the `getMBlocks()` API.

The Block Allocator splits megablocks into power-of-two sized and aligned *blocks*, which are by default 4 KB in size, but which can be *grouped* contiguously. It is at this point that the block descriptors, as described in Section 6.1.7, are contiguously allocated at the beginning of the megablock and are initialised and associated with each consecutive 4KB block within the megablock. When free, these 4 KB blocks are managed on a conventional address-ordered free list. The garbage collector allocates the heap by chaining these 4 kilobyte blocks together,

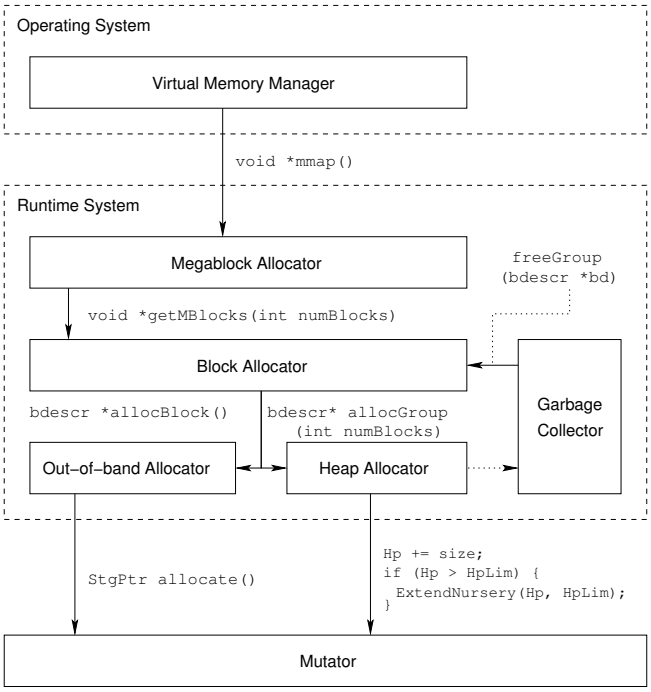



Figure A.1: Architecture of GHC's Storage Manager

using the Block Allocator's `allocBlock()` and `allocGroup()` APIs, to create the nursery, the older generations, and when necessary, to-space for each, as described in Section 6.1.2. The garbage collector returns the 4KB blocks from the condemned heap to the free list using the `freeGroup()` API.

The mutator allocates its heap objects sequentially using the fast "in-band" bump pointer allocation of the *Heap Allocator*'s heap pointer (a reserved machine register designated `Hp`) within those blocks linked to form the heap. An additional machine register, `HpLim`, marks the end of the current heap block and when the bump pointer, `Hp`, meets it, the Heap Allocator's `ExtendNursery()` macro is invoked to move the two pointers to their respective places within the next linked heap block. If there are none, the macro forces the mutator to return to the scheduler and the garbage collector is invoked. It is worth noting that it is the heap check that precedes each basic block of compiled code that i) determines whether `Hp` and `HpLim` will collide based on the total memory requirements of its basic block; and ii) if necessary, subsequently invokes `ExtendNursery()`.

There are cases, primarily within the runtime system itself and its supporting libraries (such as GNU's GMP library for arbitrary precision arithmetic), where the heap allocator cannot be used — the garbage collector cannot identify those live pointers whose roots are held solely by the support libraries. In such cases, the `allocate()` API of the *Out-of-band* allocator is used to request block(s) directly from either the Block Allocator or, if necessary,

the Megablock Allocator. Small objects that fit within a block are bump pointer allocated "out-of-band" in an identical way to the Heap Allocator and so the Out-of-band Allocator manages its own equivalent pointers for `Hp` and `HpLim`, `alloc_Hp` and `alloc_HpLim`. This is also the interface by which both large objects that exceed the size of a single heap block, or "address-pinned" blocks can be allocated. Examples of such are TSO objects, containing thread stacks, and large arrays.

### A.4.3 The Garbage Collector

Section 6.1.7 outlined the architecture and described the basic implementation of GHC's generational collector, which is, on the whole, fairly conventional. The collector does however provide some unconventional features that support the Haskell language.

Firstly, in order to optimise heap space, the collector eliminates indirection objects, removing them and in their place, evacuating the indirectee to the step of the generation to which the indirection object would have been tenured or promoted. Secondly, when the collector encounters a *thunk selector* closure it attempts to perform the selection, and is an important mechanism by which space leaks are eliminated. A thunk selector is a heap-allocated closure whose entry code selects a data member from a data constructor based on a single-constructor type. For example, the following is a thunk selector, `x = case y of (a,b)` $\rightarrow$ `a`, that follows the uniform closure representation and its first payload field is a pointer to the selectee. The collector inspects the selectee's tag to determine if it's evaluated, and if it is, the selection is performed and the thunk selector is eliminated as if it were an indirection object, if it is not, it is evacuated as per any other closure.

Thirdly, evaluated *Constant Applicative Forms* (CAFs) (also known as *static thunks*) are "garbage collected", so as to eliminate potential space leaks. A CAF is a top-level expression that takes no arguments, for example: `intListPortion = [0..100000]`. A CAF is represented by a static closure allocated in a static region of memory that is separate from the heap. On most architectures, this region is the text segment, where code is stored. In the example above, `IntListPortion` is represented by a CAF that on evaluation will be overwritten with a (special *static*) indirection to a heap-allocated list. Once a CAF is evaluated the garbage collector must be able to treat it as a root in order to keep its heap-allocated indirectee alive, update frames for CAFs therefore place the static indirection on the immutable list of the oldest generation. Garbage collecting CAFs is tricky because pointers to static closures are never stored in the heap since their address is known at compile-time. The problem then, is in identifying when each evaluated CAF is no longer referenced by the current expression and can therefore be "reverted" to its unevaluated form. This saves the memory occupied by the evaluated CAF at the expense of possible recomputation if the value of the CAF is once again demanded. The solution adopted is to generate, at compile-time, a *static reference table* (SRT) for each closure type containing the addresses of those CAFs it references. As each live closure is scavenged its SRT is traversed and the immutable list entry of any evaluated

CAFs are preserved and its indirectee is evacuated.

Finally, at the end of a collection cycle, the collector executes the finalisation routines for those garbage objects which have had a finalisation handler associated. Such handlers most usually clean up and deallocate external resources[1].

The interface to the garbage collector is defined by the following five functions:

`void GarbageCollect(void (*get_roots)(evacuation_function), rtsBool force_major_gc)`
is the top-level garbage collection routine whose pseudo-code was described in Function `generationalGC()` of Section 6.1.7. The garbage collector invokes the `get_roots()` function to locate and subsequently scavenge those roots that are external to its own root set. The `get_roots()` function is defined within the scheduler and simply registers the TSOs, (i.e. the stacks) of each thread residing on one of its queues, with the collector.

`rtsBool doYouWantToGC(void)`
is the interface by which external FFI code can determine whether a collection cycle must be triggered as the result of the memory available to the Out-of-band Allocator exceeding its defined threshold.

`void newCAF(StgClosure *caf)`
places the CAF `caf` on the immutable list of the oldest generation.

`void recordMutable(StgMutClosure *closure)`
registers the previously immutable `closure` object as mutable thus placing it on the mutable list of the generation to which it belongs. Note that if the object is only temporarily immutable (for example a *frozen array*) then when its payload has been scavenged, it can be removed from the list.

`void recordOldToNewPointers(StgMutClosure *closure)`
places `closure` on the immutable list of the generation to which it belongs.

[1] These routines are generally written in C

# Appendix B

# The Jikes Research Virtual Machine

The Jikes Research Virtual Machine (RVM) originated out of the Jalapeño project at IBM T J Watson Laboratories. The primary goals of the project were: a) to create, from scratch, a server Java virtual machine (JVM) with competitive features and performance to the state-of-the-art JVMs; and b) to determine if it was possible to write the JVM itself in Java as opposed to C / C++ given the restrictions imposed by the language and runtime system. While the RVM was developed primarily as a research platform to explore, measure, and evaluate, novel virtual machine features and implementation techniques, by the time JDK 1.3 was released its performance was comparable and, in some cases, better than both Sun and IBM's production JVMs. This is particularly impressive given that the choice of development language, which counters intuition and the conventional choice of a low-level language, has yielded substantial payoffs through the use of a modern, object-oriented, type-safe programming language with automatic memory management. The development methodology adopted promoted the initial implementation of simple mechanisms and avoided premature optimisation. Only when a mechanism was observed to be a performance bottleneck was it subsequently refined — the adaptive optimisation engine not only optimises the user program but is also free to act on the JVM itself and can compile frequently executed runtime services inline with user code.

Over the following section we summarise the key architectural features of the RVM as described in [AAB+00b].

While the runtime systems of the majority of mainstream JVMs have been implemented as native methods in C, C++ and even assembler, in Jikes RVM, exception handling, dynamic type checking, dynamic class loading, interface invocation, reflection, input/output, memory allocation and garbage collection are all implemented primarily in Java.

## B.1 Bootstrapping the RVM

In order to bootstrap the RVM, a minimal set of resolved classes and fully compiled methods belonging to the JVM's core runtime services is stored as a *bootimage* in native form on disk. The bootimage is a snapshot of the executing RVM — it contains a `Thread` object and thread stack ready to run the RVM's `boot` method and a small number of supporting 'live' Java objects. The *bootimage writer* uses a slightly modified implementation of the RVM's reflection mechanism to transform the layout of live objects and translate object references, rewriting them as machine addresses into the runnable bootloader disk image. When the RVM is started from the command-line, a C++ program, the *bootimage runner* loads the bootimage and a small piece of architecture specific assembler transfers control to the thread which executes the `boot` method and initialises the RVM's runtime system. In addition, it also installs interfaces to operating system services such as hardware interrupt handlers.

## B.2 Bytecode Compilation

All JVMs operate on bytecodes that result from compilation of Java source statements. However, to achieve its high performance goals, Jikes RVM does not interpret Java bytecodes, instead they are compiled to native machine code prior to execution. The line between bytecode 'compile-time' and 'run-time' is therefore rather fuzzy — compile-time starts when a class is loaded in bytecode form into the JVM and ends with the production of a machine code representation ready for native execution. Native execution then proceeds — this is 'run-time'. However, to the user, 'run-time' is the moment at which the JVM is started and classloading invoked and execution time is the time taken to perform both compile-time and run-time phases. It should be clear, that there is a trade-off that must be balanced to achieve the best execution time. The trade-off is between the time spent compiling bytecodes and the application of optimisation phases for native code generation and 'just getting on' with executing the user program.

Jikes RVM employs three interoperable compilers at various stages of development through to deployment. The *base* compiler is a fast non-optimising compiler to assist rapid development and prototyping yielding a fast write-execute-debug development cycle. The *quick* compiler is a *Just-in-Time* (JIT) compiler that compiles a method when it is executed for the very first time. It applies only those optimisations deemed to be most effective for the architecture the RVM is running on so as to balance compile-time and run-time, optimising for good average-case execution time. The optimisations performed include temporary variable elimination via copy propagation, method inlining of short methods that are final or static and register allocation. The *optimising* compiler produces high-quality machine code for computationally intensive and frequently executed methods by applying traditional static compiler optimisations as well as some which are specific to a Java execution environment. As of the Jikes RVM 2.3.4 release the base compiler is seldom used (except in initial ports to new

architectures), and the quick compiler is now employed as the standard *baseline* compiler.

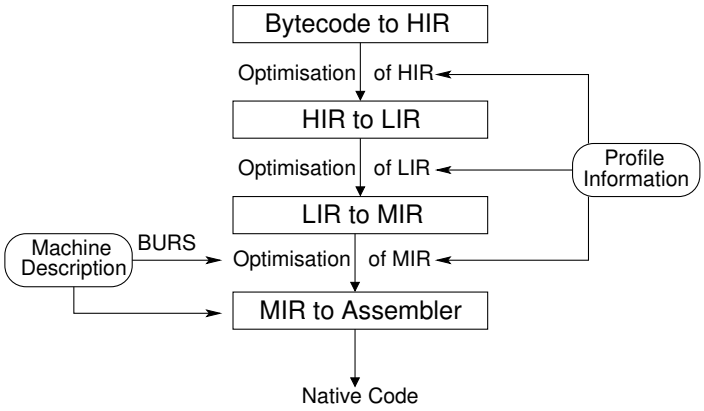



Figure B.1: Phases of the Jikes RVM Optimising Compiler

Figure B.1 depicts the compilation and optimisation phases that Java bytecode instructions undergo in the process of bytecode to native machine code compilation by the Jikes RVM optimising compiler. The bytecodes undergo three register-based intermediate representation translations before a bottom-up rewrite phase that results in native machine code generation:

**High-level Intermediate Representation** (HIR) — The translation of bytecodes to HIR is performed by abstract interpretation, operating over the type and compile-time values, if known, of local and stack stored variables. The compilers store and continually update the symbolic state which drives the abstract interpreter. As abstract interpretation proceeds, the extended-basic-blocks of a method, along with type information, is discovered and stored for subsequent optimisation operations, and the generation of the method's *exception table* and GC *reference map*. At the same time, HIR instructions are emitted and 'simple' optimisations, such as copy and constant propagation, dead-code elimination, and local variable register renaming are performed.

With the generation of HIR instructions complete, a HIR optimisation pass is made. The key difference between Java bytecode and HIR instructions is that the former is a stack-based representation while the latter is register-based. Register-based representations are chosen i) for the ease with which code motion and code transformation operations can be performed over stack-based representations; and ii) for the comparative ease with which the native architecture instruction sets can be targeted.

The optimisation phase first applies local extended-basic-block optimisations, such as common subexpression elimination, redundant load elimination and the elimination of redundant exception checks. Flow-insensitive optimisations are then applied that optimise across the basic block boundaries. Variable definition-usage chains are built and copy propagation and dead-code elimination performed along with conservative escape analysis for scalar replace-

ment of aggregates, without the need for expensive control-flow or data-flow analyses[1]. Finally, a static size-based heuristic is applied to method callsites to determine whether static and final methods should be inlined. For the callsites of non-final virtual methods, the object on which the virtual call is invoked, its *receiver*, is predicted to be the declared type of the object and the method inlined. The callsite is then *guarded* with a run-time test that verifies the prediction is correct and if it is not, falls back to virtual method invocation.

Because the RVM is written in Java, this inlining mechanism is used to inline user code with Java libraries and also the RVM's runtime services such as object allocation and synchronisation primitives. Furthermore, the use of guards supports class loading and the extent of this inlining provides excellent optimisation opportunities.

**Low-level Intermediate Representation** (LIR) — Translation of HIR into LIR instructions introduces operations that encode the RVM's calling and parameter passing conventions and manipulate its representation of objects and their layout. For example, the HIR instruction corresponding to `invokevirtual` method execution is expanded to three operations that: i) retrieve the object's *Type Information Block* (TIB); ii) lookup the address of the method in the TIB; iii) transfer control to the method body.

Having expanded the HIR instructions into LIR, local common subexpression elimination is performed, and a *dependence graph* for each extended basic block is created. The edges link the LIR instruction nodes as shared dependencies between a pair of instructions. These edges include control, synchronisation and exception dependencies.

**Machine-specific Intermediate Representation** (MIR) — The MIR instructions encode the architecture to which the bytecode is targeted, initially this was simply a register-rich PowerPC instruction set, but over time additional architecture ports of the RVM have been performed, most notably, to the register-scarce x86 architecture. The MIR is produced by partitioning the dependence graphs, constructed in the previous phase, into trees and passing them to a *Bottom-Up Rewriting System* (BURS). The BURS is a code-generator generator that creates tables that drive instruction selection from a tree grammar for each target architecture. Each grammar rule has an associated code-generation action and a cost that accounts for the size of the instructions generated and their cycle times. The tables are constructed by traversing the tree with paths selected based upon the least aggregate cost. The dependence graph allows the application of aggressive code reordering optimisations to the MIR while preserving correctness. With the MIR generated:

- Live variable analysis is performed to identify i) the live register ranges and ii) stack variables that hold object references at GC safe points.
- A global, linear scan, register allocation algorithm assigns symbolic registers to physical registers.

[1]The Java Virtual Machine Specification mandates that every variable must have a value before it is used.

- Method prologues and epilogues are created for each method. The prologue is responsible for pushing a stack frame and saving registers that will be *clobbered* by the method, and a thread yield point is generated. This yield point checks to see if a yield has been requested — the RVM's thread model is discussed in Section B.4. The epilogue restores the saved registers and pops the stack frame. If the method is `synchronised`, the prologue is responsible for locking the associated object and the epilogue for unlocking it.

- Native code is emitted as an array of `int`s for the method bodies and their exception tables and reference maps are finalised.

Although we have discussed the optimisations that the compiler applies within the context of the intermediate representation translations, the compiler applies the optimisations according to an *optimisation plan* that is built depending on the optimisation level specified. Each level augments the optimisations of previous levels [AFG+00a]:

- Level 0 — On-the-fly optimisations are applied during bytecode to IR translation that do not require multiple or subsequent passes over the IR. The optimisations tend to reduce the size of the IR and therefore the overall compilation time — constant, type, non-null and copy propagation; constant folding and arithmetic simplification; dead-code elimination; and redundant null check, `checkcast` and array store check elimination, are all performed.

- Level 1 — Local extended basic block optimisations are applied: common subexpression elimination; array bounds check and redundant load elimination; global-flow-insensitive copy and constant propagation and dead assignment elimination; StringBuffer synchronisation optimisations, and scalar replacement of aggregates and short arrays.

- Level 2 — Static Single Assignment form (SSA) based flow-sensitive optimisations are performed. In addition to traditional SSA optimisations for scalar variables, redundant load elimination and array check elimination is performed.

The optimising compiler can be employed in four usage scenarios. The first is as a static compiler that compiles bytecode instructions and emits machine code to the *bootimage*. Here, the bootimage is not just the minimal set of core runtime services, but can also be fully resolved and compiled user code. This mode of operation is depicted in Figure B.2. In the second, when an unresolved object reference is encountered during execution, resulting in a new class being loaded, the class loader invokes the compiler to compile the class and execute its class initialiser. At this point, the compiled code for all class methods is initialised to a *lazy compilation stub*. In the third, the compiler is invoked on execution of a lazy compilation stub as the result of invocation of a method that has not yet been compiled. In scenarios two and three, the application thread is paused until compilation finishes. In the final scenario, the *adaptive optimisation system* invokes the compiler when the recompilation of a method

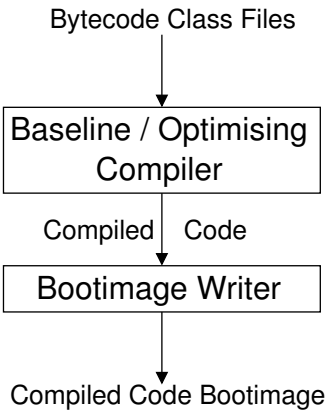



Figure B.2: Jikes RVM compilers employed as static compilers.

with a more aggressive optimisation plan is suggested. The suggestion is the result of online analysis of sample-driven profiling data collected as the RVM runs. In this case, recompilation proceeds in another thread and the reference to the new native code array for the method body is only installed in the class's type information block once compilation has completed.

## B.3 Adaptive Optimisation Subsystem

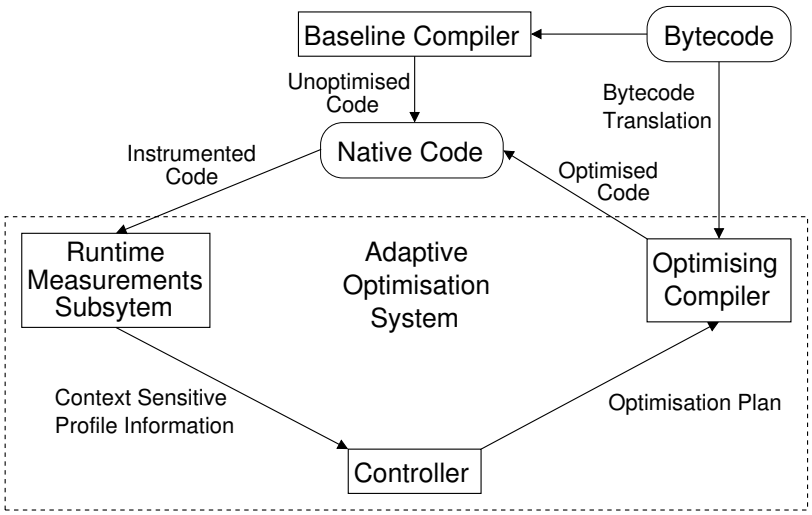



Figure B.3: The Adaptive Optimisation System of Jikes RVM.

The adaptive optimisation system (AOS) is the online feedback-directed optimisation engine of the Jikes RVM, that selectively optimises both the user code and the JVM, and is

guided by profiling data gathered by statistical sampling at run-time. In the compile-only approach of Jikes RVM this subsystem is responsible for dynamic run-time optimisation that is normally delegated to a JIT or Hotspot optimiser of a more conventional JVM. The system consists of three key components which are shown in Figure B.3:

**Runtime Measurements Subsystem** — The system is responsible for obtaining measurements of the running code using a variety of online techniques, such as the sampling of hardware performance counters, call stack sampling, and instrumentation-based profiling such as basic block invocation counters, edge, path and value profiles. These measurement techniques have the potential to generate an overwhelmingly large amount of raw data, of which only a very small amount can be analysed on-the-fly. As a result, additional *organiser* threads are scheduled periodically to analyse and process this raw data. They produce data that is either passed directly to the *controller* via the *organiser event queue* or stored in the *AOS database* for subsequent query by either the controller or the *recompilation subsystem*.

**Controller** — This component is responsible for coordinating the entire AOS, determining where to focus the profiling effort of the runtime measurements subsystem and for how long, and creates recompilation plans which it passes to the recompilation subsystem. It makes decisions based on AOS events, previous decisions, and static analysis results, queried from the AOS database or received on the organiser event queue as determined by a cost-benefit model that guides the courses of action. In the first, the strategy in progress by the runtime measurement subsystem can be terminated, extended or modified. In the second, an optimisation plan may be sent to the *compilation queue* for subsequent execution by the recompilation subsystem in order to improve performance.

**Recompilation Subsystem** — The subsystem consists of a set of threads that execute compilation plans scheduled on the priority compilation queue by the controller. These compilation plans are made up of two plan components, optimisation plans, which may be augmented with profiling data, and *instrumentation plan*s. Optimisation plans are used to invoke the optimising compiler and have already been previously discussed. Any associated profiling data is used to guide the optimising compiler's feedback-directed optimisations. Instrumentation plans instruct the compiler where to insert calls to instrumentation-based profiling mechanisms.

Organiser threads of the AOS focus primarily on *hot* methods: those methods that are frequently executed, passing hot method events via the organiser event queue to the controller for possible recompilation. These events contain a method identifier and the methods relative 'hotness'. Decay organiser threads are responsible for reducing, over time, the counters of the measurements subsystem including hotness. A hot method is recompiled using a cost model, described in detail in [AFG+00b], that determines the potential benefit based on: an estimate for the total time the user program will spend executing the method if it is not

recompiled; the cost of recompiling the method at each optimisation level; and an estimate, for each optimisation level, for the total time the program will spend executing the method in the future, having been compiled at that level.

We have already mentioned that the ability to inline user code and core JVM services enables more aggressive optimisation opportunities than are normally available within a JIT or Hotspot environment. The AOS employs *feedback-directed* inlining in order to realise these opportunities at run-time. The AOS maintains an approximation for the dynamic call graph of the running program by statistically sampling method calls and running methods. 'Hot' call graph edges are identified and passed to the optimising compiler, and in many cases, already optimised methods undergo recompilation in order to inline them. One minor, but particularly important implementation detail, is that when a method undergoes recompilation, all previous inlining decisions within that method are preserved / reapplied. This must be done because call edges that have been inlined may no longer contribute entries to the dynamic call graph or 'hot' profiling data. As a result, method recompilation could lead to rapid oscillation between two different inlining plans, with a degradation in performance as a result of increased compilation time and no increasing benefit (over time), in the optimised code.

## B.4 Runtime System Overview

The Jikes RVM runtime system provides hooks for the run-time invocation for the majority of the compilation components and the AOS subsystem — the dynamic compilers, linker, classloader, and AOS profiler. It incorporates the object allocator, the garbage collector, thread scheduler, and support for exception handling and dynamic type testing.

### B.4.1 Thread Scheduler

Unlike many production JVMs, the task of thread scheduling is not delegated to the operating system scheduler using a 1:1 thread model (Java thread to operating system thread). Operating system interfacing does however utilise the POSIX `pthread` library which allows user space applications to use 1:1 or M:N threading models to interface to the OS scheduler. The Jikes RVM implements its own *quasi-preemptive* thread scheduler in order to achieve the scalability required to implement a high performing SMP JVM for server applications. As a result multiple Jikes RVM Java threads are multiplexed for execution within a single `pthread`. Jikes RVM runs as a user space application and `pthread` library settings can be configured to determine the thread model that is adopted between itself and the OS (1:1 or M:N). Currently, a *virtual processor*, a single `pthread`, is created for each physical processor, in a 1:1 mapping. A quasi-preemptive scheduler is chosen for two reasons. Firstly, allowing threads to "run-until-blocked" using explicit thread requested yields is neither fair, nor can progress guarantees be made which are required in a server environment — threads may suffer CPU starvation. Secondly, a fully preemptive environment severely complicates type-accurate

garbage collection and arbitrary collector invocations can be problematic. Instead, the compiler inserts yield points into method prologues and on the "back edges" of loops. These yield points provide a check for preemption by other threads. The scheduler sets a bit indicating that a thread context switch is to occur and it is this that is checked at each yield point.

### B.4.2 Exception Handling

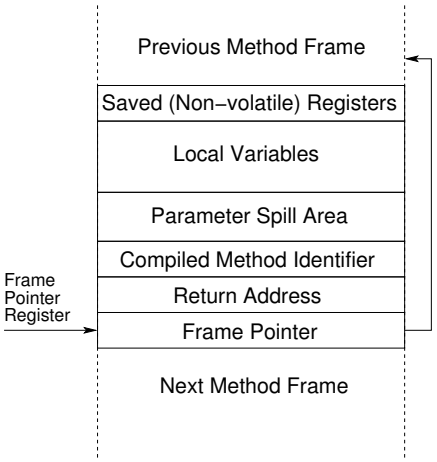



Figure B.4: Jikes RVM stack layout.

Recall that the RVM is bootstrapped by the bootimage runner that installs OS interfacing signal handlers. In particular, a signal handler for a timed periodic interrupt which drives the thread scheduler is installed, as is an interrupt handler that catches the following hardware based exceptions: a null pointer dereference, indexing an array beyond its defined bounds, integer division by zero and the overflow of a thread's stack on method invocation. In the exception cases the handler invokes a Java method that builds the appropriate Java exception and passes it to the generic hardware / software exception delivery mechanism of the RVM. Figure B.4 shows the layout of Jikes RVM stack frames. The delivery mechanism, the `deliverException` method, then walks the stack, storing method identifiers and return addresses within the exception object so that, if requested, a stack trace can be produced. The stack walk proceeds, popping frames, until an appropriate *catch* block is found and execution is resumed, within the block's context. If no appropriate block is found, the thread is terminated.

The structure of the RVM's stack frames is such that it violates Java's type system. Within the stack frames, it must be possible to differentiate object references from the immediate values of primitive types. The compilers must therefore generate for each *safe point* within the compiled method where garbage collection can occur, a *stack map* that identifies the location of references within the stack frame. Similarly, *register maps* must be generated that identify

object references that reside in registers at each safe point. Together the stack and register maps combine to form a safe point's *reference map*.

### B.4.3 Dynamic Class Loading

We have previously described the class loading mechanism within the context of optimising compiler invocation. More generally, for any of the Jikes RVM compilers, the lazy compilation stub is first emitted when the class is loaded, and subsequent execution results in the actual compilation and native code generation of the method. At this point the compiled code must be 'backpatched' with the callsite of the lazy compilation stub overwritten by this new code, and then branched to for execution. This is a relatively complex procedure, especially within a multi-threaded SMP environment, further complicated by the register allocation and resulting optimisations performed by the optimising compiler. The implementation is fully described in [ACC+99].

### B.4.4 Object Allocation and Object Layout

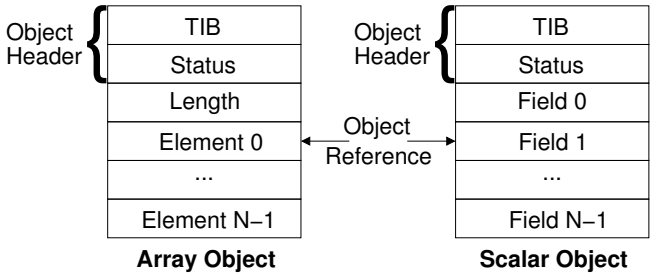



Figure B.5: Jikes RVM object layout.

The Jikes RVM lays objects out in memory so as to achieve fast field and array access and to exploit a hardware null-pointer check. Figure B.5 shows the layout of both an array object and a scalar object. An array reference is a pointer to the 0th index, the *array base*, into the array. The element at the $i$th index is obtained by offset from the array base, calculated as $i$ times the size of the base array element. The length of the array is stored in a field prepended to the array base, i.e. at a negative word offset. The length of the array must be examined on every array index operation in order to perform a bounds check. As a result, the effect of attempting to load the array length for an array reference that is now a null pointer results in a hardware trap — access of 'high' memory (via a small negative offset from address zero), falling outside of user space generates this. The benefit is a null pointer check with no associated cost. Scalar objects are laid out in a similar way to arrays so that for both object types, the header elements are at an identical offset from the object reference. As a result, the first field of a scalar object is at a negative offset, while the rest are at positive offsets. Some

operating systems, such as Linux, generate hardware traps on attempts to address either very high or very low addresses, others such as AIX generate them for high address accesses only. As a result, on Linux all null pointer checks are performed implicitly via hardware traps. On AIX however, explicit checks must be inserted for the majority of scalar field accesses because access of 'low' memory via a small positive offset from address zero is permitted and not trapped.

Scalar objects of the Jikes RVM have a two word object header, while array objects have three. The first word of the object header is a reference to the object's *Type Information Block* (TIB). For each class in the RVM there is a unique TIB. The TIB is an array of references to Java `Object`s which describe the object's type and provide the infrastructure that support Java's dynamic operations on objects, such as virtual method dispatch, interface method invocation, and dynamic type checking. The second field of the object header is the `status` word, a bit field, with bits reserved for object locking, the object hash value, and garbage collector support — mark bits, etc. For array objects, the third field stores the length of the array. The remainder of the object contains its payload — its fields, i.e. its unboxed primitive values and field references.

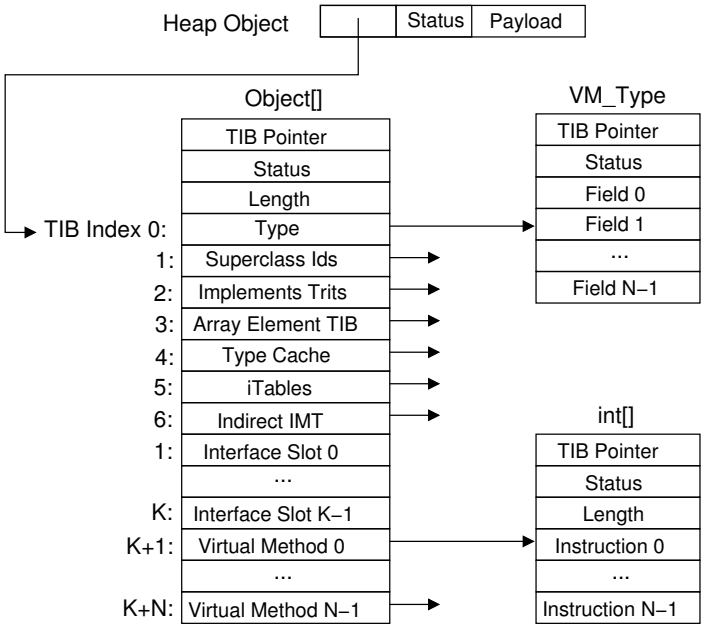



Figure B.6: The Jikes RVM Type Information Block.

Figure B.6 shows the layout and contents of the TIB. Notice that because the TIB structure is a valid Java array object, it too has a 'TIB pointer' field, that references the TIB for the `Object` array class. The 'Type' field refers to a `VM_Type` (and ultimately `VM_Class`) object, which is an internal RVM object that stores metadata about the object's class, such

as its superclass, the interfaces it implements and method and field offsets. This object is constructed when the class is loaded and subsequently accessed during method compilation by the lazy compilation stub, and by some of the more expensive and less frequently invoked run-time mechanisms of the RVM.

The next three fields assist in dynamic type checking. Dynamic type checking is performed by evaluating the condition:
*Can an instance of the right-hand-side (RHS) class be stored in a variable of the left-hand-side (LHS) class or interface?*
and can involve four bytecodes: `instanceof`, `checkcast`, `aastore`, and `invokeinterface`.

Field '1' references a vector of *superclass identifiers* (SIDs). Java objects are *extended* via single inheritance semantics. Inheritance is therefore from a single superclass. However, the superclass can of course inherit from its own superclass. A single-chain class inheritance hierarchy is therefore built. Each class is associated with a unique type identifier, and it is these identifiers, for all preceding classes in the inheritance hierarchy, that are stored in this array — the array element order reflects the chain ordering and class depth. For the implementation of the `instanceof` keyword, object instance `b` is an instance of class `A`, if the SID at the index of `A`'s depth is equal to A's type identifier (found in its `VM_Type` class). Note that the uniformity of the dynamic type condition is counter-intuitive with respect to `instanceof` where `b` is interpreted as the RHS and `A` the LHS.

Field '2' is the *implements trits vector* (ITV) and is indexed by an *interface* identifier (IID). Each interface that the class can possibly implement is assigned a unique IID. Elements of the ITV can take values: 'yes', 'no' and 'maybe'. These values indicate that for the associated IID, whether the class definitely implements the interface, definitely does not implement it, or it is yet to be determined whether the class implements it. The ITV must be a dynamically resizable structure because there is no a priori bound on the number of interfaces that may be encountered during execution. It is reasonable sized so that those interfaces with indices less than the size do not require a bounds check on access. Initially, all entries are set to maybe. If a class does not implement any interfaces then it shares the ITV of its superclass. During a dynamic type checking operation involving an interface type, if a value of 'maybe' is retrieved for the associated IID then the entry is resolved to either 'yes' or 'no', the ITV updated, and the result of the subsequent type check operation returned.

Field '3' references the type information block for the elements if the object is an array type. Type checking arrays uses this TIB reference and the dimensionality, $k$, specified in the associated `VM_Type` instance and covers three cases where the LHS is an array of dimension $k$:

1. If the LHS is a primitive array then the RHS is verified as having: i) dimensionality $k$ and ii) the same primitive baseclass.

2. If the LHS array is a sub-type of `Object` then the RHS is verified as having: i) dimensionality greater than $k$ or ii) dimensionality $k$ and baseclass is not primitive.

3. If the LHS is neither a primitive array or a sub-typing of `object` then the RHS is verified as having: i) dimensionality, $k$, and ii) a baseclass assignable with the LHS class.

Field '4' references a class that caches type *reference* metadata if necessary — currently it is unused and therefore not allocated.

The majority of the remaining TIB elements are references to integer arrays, one for each of the class's methods including those methods it inherits and interfaces it implements. These arrays contain the native code for the compiled method bodies. The TIB therefore implements the class's *Virtual Method Table* (VMT) and *Interface Method Table* (IMT). As a result, method dispatch is achieved via one level of indirection as opposed to two, where the method table would be accessible only via the TIB's `VM_Class` object. Other JVM's, for example Sun's ExactVM / ResearchVM [WG98], have previously implemented the object header as a reference to a class object that then references the method tables.

The fields 'K+1' to 'K+N' form the inline VMT and it is indexed by a unique *method identifier* associated with each method. Each virtual method that is inherited resides at the same location in the VMT as in the superclass's VMT. The effect is to clone a local copy of the the superclass VMT and append the class's additional virtual method table entries to the end. Obviously, the reference to a code array for a method that overrides a superclass's method resides at the index of that overridden method. Compilation of the `invokevirtual` bytecode instruction then simply involves the method identifier and correctly calls the overridden method, or the *inline* method inherited from the superclass, if it hasn't been overridden. Method invocation of an overridden method by upcalling to the superclass using `super.method` indexes the VMT from `super`'s TIB.

Unfortunately interface method invocation is not as elegant as that of virtual methods. It is complicated by the multiple inheritance semantics for Java interfaces. The significant difference is that interface implementing methods do not necessarily reside at the same VMT offset. For example, if two unrelated classes, `A` and `B`, both implement interface `I`, and provide a method `foo()`, because there is no inheritance relationship between `A` and `B`, `foo()` will not (unless by luck) reside at the same VMT offset. The result of this is that there is usually (unless determined statically at compile-time) an element of search in locating the method for invocation based on the class and the interface. The details of the efficient implementation of interface method invocation is discussed in [ACFG01]. In summary, Jikes RVM caters for two implementation techniques. The first uses either an embedded inline ('Interface Slot 0' to 'Interface Slot K-1') or *indirect* IMT. The second uses *iTable* structures reachable from an *iTable dictionary* referenced from TIB field '5'. So, depending on the choice of technique the TIB array contains either field '5', field '6' or fields 'Interface Slot 0' to 'Interface Slot K-1'. When using an IMT whether embedded inline or indirect, it is fixed-size allocated. A hash function maps a *method descriptor* (a combination of interface and method signature) to an entry in the IMT. In case of a hashing conflict, the IMT entry is a pointer to a *conflict resolution stub* that resolves the desired compiled interface method code array. When using

an iTable implementation, field '5' references the iTable dictionary, of which there is one for each class, that contains an entry for each loaded interface. It is indexed by IID and each entry points to an iTable that houses the compiled interface method code arrays for that class.

The final issue in data layout is where to allocate and how to access a class's static fields and methods. While it is tempting to have static fields accessible via offset to a fixed location from the `VM_Class` of an object, and static methods accessible from the TIB, there are several drawbacks to these approaches — there are unnecessary levels of indirection in field and method access; a mechanism to access them when a class instance is unavailable is required; and finally, the class objects would have to be of different types since different classes have static fields of different types.

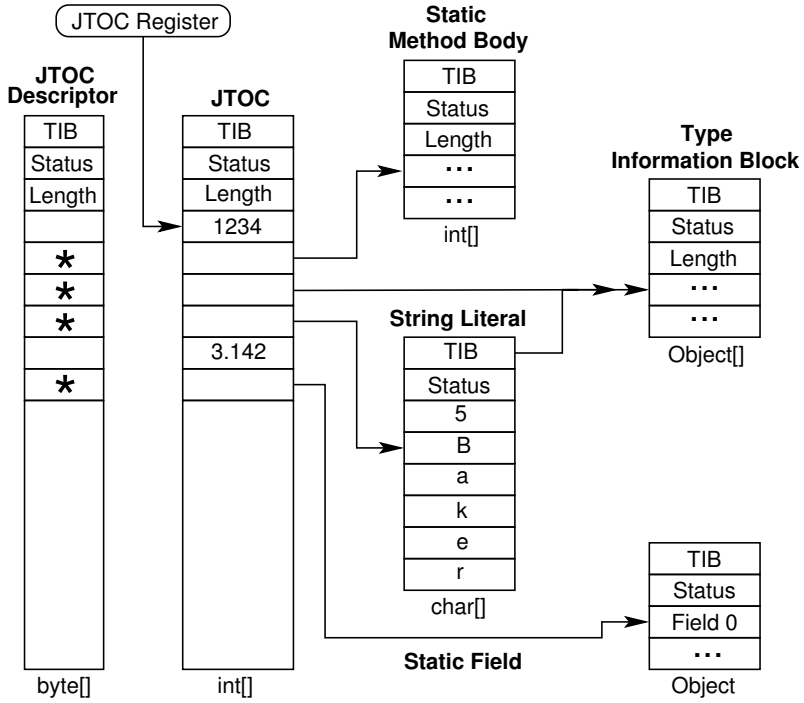



Figure B.7: The Jikes RVM Table of Contents.

The approach taken is simply to store all static fields and all references to static method bodies in the *Jikes RVM Table of Contents* (JTOC), a single array that parallels the Java Language Specification's Constant Pool in housing all literals, numeric constants, and references to string constants. However, unlike the constant pool, all of the RVM's global data structures, or references to them, are stored in the JTOC, including references to *all* TIBs in the RVM, to allow fast common-case dynamic type checking. There are significant benefits in having all global structures in the RVM accessible via the JTOC — both the bootimage writer and the garbage collector use the JTOC as part of the root set of objects to either be written to the bootimage or marked as live during the collection process (recall that with the

majority of the RVM implemented in Java its internal objects will also be under consideration by the collector). The JTOC is however stored in an 'immortal' part of the heap that is not subject to garbage collection (except for root scanning). Figure B.7 shows the JTOC, the dedicated machine register, the JTOC register, that maintains the reference to the JTOC array. Primitive types reside unboxed embedded within the JTOC, the diagram shows this with examples of integer and floating point values. Also depicted are JTOC references to a string literal, a static field, a static method body, and a type information block. The JTOC is the only structure, other than method invocation stacks, that violates Java's type system within the RVM — although specified as an integer array, it contains object references for non-primitive types. The bootimage writer and garbage collector must be able to identify object references to static data. The *JTOC Descriptor Array*, a byte array, is used to differentiate the embedded unboxed values of primitives from object references. For each entry in the JTOC an entry is made in the JTOC descriptor array. If the entry is for an object reference then the byte value for ASCII character code '*' is made, non-primitive entries contain a byte value of '0'.

### B.4.5 The `Magic` of the Jikes RVM

The benefit of implementing the majority of the RVM in Java should be clear — in addition to greater optimisation opportunities, the costs associated in the bridging of Java and native language calling conventions are avoided. These costs usually result from the maintenance of a native call stack in addition to the JVM (thread) stack(s). However, a few of the RVM's internal low-level operations cannot be implemented without violating, and therefore circumventing, the semantics of the Java programming language, its object and its memory models. During object allocation and garbage collection, the memory manager must manipulate and compute 'raw' memory addresses. The garbage collector, exception handling, and reflection mechanisms must be capable of walking or constructing thread stack frames. The exception handling and dynamic linking mechanisms must be able to modify registers, synchronise and flush instruction and data caches and transfer control to the appropriate `catch` block or the backpatched call site.

The RVM employs the use of its `VM_Magic` class in order to circumvent the language restrictions of Java. The class implements the methods that must be used in the above operations. However, their method bodies are empty. A source to bytecode compiler compiles and generates the bytecodes for these methods as is standard. However, the Jikes RVM compilers are aware (through the use of pragmas implemented as 'special' exceptions) of these methods, and ignore the associated bytecodes, substituting them inline, for the native machine code that actually implements the services. Furthermore, to prevent user code from circumventing the language restrictions using these facilities the compilers verify the methods into which the services are being compiled belong to the RVM core.

The challenge in implementing these services and mechanisms is to do so without being

dependent upon themselves — a circular dependency. The implementing code fragments are 'critical sections' from which the standard runtime services such as garbage collection, Java-level object allocation, class loading and dynamic type checking must be excluded. In actual fact, their disablement results primarily from the need to work with raw memory addresses. Code manipulating such addresses must do so with garbage collection disabled (the collector suite supports only type-accurate collection) — for example, a copying collector must update object references during relocation, but not the raw addresses that are undergoing manipulation. With collection disabled, the code must be careful not to allocate a new Java object. The inability to allocate new objects inhibits class loading. The disablement of class loading prohibits use of both the dynamic linker and type cast mechanism. Finally, thread context switches at yield points must be deferred — the collector won't be re-enabled until the yielding thread resumes and completes this 'critical section', and of course, execution of the interrupting thread could well result in the need for a collection cycle to be performed.

### B.4.6 MMTk - Memory Management Toolkit

Early releases of Jikes RVM employed 'monolithic' implementations of both mark-sweep and (generational) copying collectors. These collectors were highly-tuned and tightly integrated with the virtual machine, and as such there was little re-use between collectors with the implementation of new algorithms requiring, in many cases, significant reimplementation.

MMTk has replaced this monolithic set of collectors with an object-oriented toolkit that provides a composable, extensible, and portable framework for building efficient garbage collectors. The toolkit makes extensive use of design patterns to efficiently implement succinct policy and correctness composition in the presence of concurrency. The result is a toolkit that allows fast implementation of new algorithms that not only contain less defects than the monolithic suite but also show, for SPEC benchmarks, on average, between a 5% and 25% total performance improvement over them [BCM04].

The toolkit provides three key compositional sets of components and abstractions. *Mechanisms* are highly-tuned, allocator- and collector-neutral implementations of: bump pointer, free list and large object allocators; finalisation; soft, weak and phantom reference implementations; data structures for parallel load balancing; trial-deletion cyclic garbage collection; and finally thread-safe and GC-safe I/O routines. *Policies* employ mechanisms to combine object specific allocator and collector implementations for a *space*. Contiguous regions of virtual memory are segregated into *spaces*, a set of which, are managed by a single policy. *Plans* sit at the highest level of composition and define the rules by which policies are combined. Plans therefore determine the implementation of the entire collection algorithm across multiple spaces and object types, and operate by delegating work to the appropriate policy based on the type of the object. More concretely, policies provide spaces for immortal object allocation, and copying, mark-sweep, reference counting, and treadmill collection. The `SemiSpace`, `MarkSweep`, `RefCount`, `GenCopy`, and `NoGC` plans are straightforward implementa-

tions for these. The `CopyMS` and `GenMS` are copying-mark-sweep hybrid plans employing (generational) copying in young spaces and mark-sweep in old. `GenRC` and `UlteriorRC` [BM03], are hybridised reference counting collector plans. Hybridised plans for the Beltway suite of collectors [BJMM02] and a Mark-Copy [SM03] collector are also provided. The majority of the non-reference counting algorithms inherit from the `StopTheWorld` plan. Currently, there is no `Incremental` plan provided by MMTk, and a fully operational incremental collector is yet to be developed. Bacon et al. have implemented an incremental collector for Jikes RVM [BCR03b] but it remains a proprietary codebase at IBM's T J Watson laboratories and unavailable to the larger research community.

MMTk supports multi-threaded, parallel, type-accurate garbage collectors. Parallel phases require phase coordination and synchronised access to shared data. MMTk provides *local* and *global* scopes in order to achieve this. A single global scope coordinates synchronised access to global data and the phase transitions of each thread, while a local scope, uniquely associated with each thread, governs unsynchronised thread-local operations. The threads therefore operate concurrently with respect to their local scope. In practice, global and local scopes are implemented by global and local plans. MMTk collection algorithms operate in three phases at both scoping levels and the execution order is as follows: `prepareGlobal`, `prepareLocal`, `processAllWork`, `releaseLocal`, and finally `releaseGlobal`. The addition of each new collector implements new prepare and release phases, while the `processAllWork` phase is common to all collectors and simply processes work queues populated during the prepare phase.

Unlike the 'monolithic' collector suite, the collector interface to the RVM is thin and self-contained, although it is also bidirectional. The RVM exposes the unboxed (atomic) address manipulation mechanisms of its *Magic* class to MMTk via the `VM_Interface` class, so that the memory managers may implement low-level operations that must directly manipulate virtual memory. In addition, the RVM exposes inlining and interruptibility pragmas. MMTk methods that require forced inlining (such as object allocation) are annotated as throwing the `PragmaInline` exception. The RVM compilers recognise this exception and force the inlining of the method before removing the exception from the method's exception table. Similarly methods that must not be preempted or interrupted by a garbage collection (we have discussed the reasons for this above), throw the `UninterruptiblePragma`. The compilers then reflect uninterruptibility by omitting the yield points that result in preemption and return to the scheduler. The thread scheduler is not just responsible for scheduling the order in which blocked threads run, but also for ensuring that when a garbage collection cycle is required, i) the threads are stopped at safe points, where each thread stack is in a consistent state; ii) the state of the running threads are saved; and iii) collector threads are scheduled and dispatched. The scheduler invokes the collector through the `MM_interface` class, which also exposes allocation, finalisation, barrier, and heap statistics accounting services. The RVM compilers are responsible for emitting calls to the `MM_interface`'s write barrier implementation for every pointer store when utilising a generational collector. The write barrier conditionally records

pointers using either card marking or a comprehensive remembered set list.

The main drawback in writing the RVM and MMTk in Java is that their internal data structures are allocated in the RVM's heap along with those of user code — this severely complicates garbage collection and the collector must be prevented from scavenging itself. To resolve this, the RVM allocates, via MMTk, an *immortal* space whose object's are neither collected, nor movable. Into the immortal space certain key data structures are allocated. However, not all of the RVM's or collector's structures need to be, nor should they be treated as, immortal — to do so requires either an unnecessarily large or potentially unbounded immortal space. Instead, the collector *pre-copies* the relevant portions of its internal state and executes with respect to that, leaving the original structures as garbage which can undergo collection.

# Bibliography


[AAB+00a] Bowen Alpern, Dick Attanasio, John Barton, Michael Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Ton Ngo, Mark Mergen, Vivek Sarkar, Mauricio Serrano, Janice Shepherd, Stephen Smith, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño virtual machine. *IBM Systems Journal, Java Performance Issue*, 39(1), 2000.

[AAB+00b] Bowen Alpern, Dick Attanasio, John Barton, Michael Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Ton Ngo, Mark Mergen, Vivek Sarkar, Mauricio Serrano, Janice Shepherd, Stephen Smith, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño virtual machine. *IBM Systems Journal, Java Performance Issue*, 39(1), 2000.

[AB98] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *RTSS '98: Proceedings of the IEEE Real-Time Systems Symposium*, page 4, Washington, DC, USA, 1998. IEEE Computer Society.

[ACC+99] Bowen Alpern, Mark Charney, Jong-Deok Choi, Anthony Cocchi, and Derek Lieber. Dynamic linking on a shared-memory multiprocessor. In *IEEE PACT*, pages 177–182, 1999.

[ACFG01] Bowen Alpern, Anthony Cocchi, Stephen Fink, and David Grove. Efficient implementation of java interfaces: Invokeinterface considered harmless. *SIGPLAN Notices*, 36(11):108–124, 2001.

[ACHS88] Karen Appleby, Mats Carlsson, Seif Haridi, and Dan Sahlin. Garbage collection for Prolog based on WAM. *Communications of the ACM*, 31(6):719–741, 1988.

[ADJ+96] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent java. *SIGMOD Record*, 25(4):68–75, 1996.

[AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

[AFG+00a] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalapeño jvm. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, New York, NY, USA, 2000. ACM Press.

[AFG+00b] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, 2000.

[AFM97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the java language. *SIGPLAN Not.*, 32(10):49–65, 1997.

[AFT99] Yariv Aridor, Michael Factor, and Avi Teperman. cjvm: A single system image of a jvm on a cluster. In *ICPP '99: Proceedings of the 1999 International Conference on Parallel Processing*, page 4, Washington, DC, USA, 1999. IEEE Computer Society.

[AG00] Ole Agesen and Alex Garthwaite. Efficient object sampling via weak references. In Chambers and Hosking [CH00].

[AH96] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in c++ programs. In *ECCOP '96: Proceedings of the 10th European Conference on Object-Oriented Programming*, pages 142–166, London, UK, 1996. Springer-Verlag.

[AK91] Hassan Aït-Kaci. The WAM: A (real) tutorial. In *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991. Also Technical report 5, DEC Paris Research Laboratory, 1990.

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. *ACM SIGPLAN Notices*, 26(4):96–107, 1991. Also in SIGARCH Computer Architecture News 19 (2) and SIGOPS Operating Systems Review 25.

[AM87] Andrew W. Appel and David B. MacQueen. A standard ml compiler. In *Proceedings of a Conference on Functional Programming Languages and Computer Architecture*, pages 301–324, London, UK, 1987. Springer-Verlag.

[AM95] Malcolm Atkinson and Ronald Morrison. Orthogonally persistent object systems. *The International Journal on Very Large Data Bases*, 4(3):319–402, 1995.

[AP87] Santosh Abraham and J. Patel. Parallel garbage collection on a virtual memory system. In *International Conference on Parallel Processing*, pages 243–246, University Park, Pennsylvania, USA, August 1987. Pennsylvania State University Press. Also technical report CSRD 620, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development.

[App89] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[AR98] Saleh E. Abdullahi and Graem A. Ringwood. Garbage collecting the Internet: a survey of distributed garbage collection. *ACM Computing Surveys*, 30(3):330–373, September 1998.

[Bac97] David Francis Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, University of California, Berkeley, 1997. Chair-Graham,, Susan L.

[Bak78] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[Bak95] Henry Baker, editor. *Proceedings of the International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, September 1995. Springer-Verlag.

[BB99] Emery D. Berger and Robert D. Blumofe. Hoard: A fast, scalable, and memory-efficient allocator for shared-memory multiprocessors. Technical Report UTCS TR99-22, University of Texas at Austin, November 1999.

[BC92] Yves Bekkers and Jacques Cohen, editors. *Proceedings of the International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St Malo, France, 16–18 September 1992. Springer-Verlag.

[BC96] John Boyland and Giuseppe Castagna. Type-safe compilation of covariant specialization: A practical case. In *ECCOP '96: Proceedings of the 10th European Conference on Object-Oriented Programming*, pages 3–25, London, UK, 1996. Springer-Verlag.

[BC97] John Boyland and Giuseppe Castagna. Parasitic methods: An implementation of multi-methods for java. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 66–76, New York, NY, USA, 1997. ACM.

[BC99] Guy E. Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 104–117, Atlanta, May 1999. ACM Press.

[BCM04] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *ICSE 2004, 26th*

*International Conference on Software Engineering*, pages 137–146, Edinburgh, May 2004.

[BCR03a] David F. Bacon, Perry Cheng, and V.T. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In LCTES [LCT03].

[BCR03b] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collecor with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, New Orleans, LA, January 2003. ACM Press.

[BCR04] David F. Bacon, Perry Cheng, and V. T. Rajan. A unified theory of garbage collection. In OOPSLA [OOP04], pages 50–68.

[BCRU83] Yves Bekkers, B. Canet, Olivier Ridoux, and L. Ungaro. A short note on garbage collection in Prolog interpreters. *Logic Programming Newsletter*, 5, 1983.

[BD98] Boris Bokowski and Markus Dahm. Poor man's genericity for java. In *ECOOP '98: Workshop on Object-Oriented Technology*, page 552, London, UK, 1998. Springer-Verlag.

[BD02] Hans-J. Boehm and David Detlefs, editors. *Proceedings of the Third International Symposium on Memory Management*, volume 38(2) of *ACM SIGPLAN Notices*, Berlin, Germany, June 2002. ACM Press.

[BD04] David F. Bacon and Amer Diwan, editors. *Proceedings of the Fourth International Symposium on Memory Management*, Vancouver, Canada, October 2004. ACM Press.

[BDS91] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

[BFG02] David F. Bacon, Stephen Fink, and David Grove. Space- and time-efficient implementation of the Java object model. In *Proceedings of 16th European Conference on Object-Oriented Programming, ECOOP 2002*, Lecture Notes in Computer Science, pages 111–132. Springer-Verlag, 2002.

[BGH+06] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[BH04] Stephen M. Blackburn and Tony Hosking. Barriers: Friend or foe? In Bacon and Diwan [BD04], pages 143–151.

[BJMM02] Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In PLDI [PLD02], pages 153–164.

[BL94] Johan Bevemyr and Thomas Lindgren. A simple and efficient copying garbage collector for Prolog. In *PLILP94 International Symposium on Programming Language Implementation and Logic Programming*, pages 88–101, 1994.

[BM03] Stephen M. Blackburn and Kathryn S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In OOPSLA [OOP03].

[BPJR88] G. L. Burn, S. L. Peyton Jones, and J. D. Robson. The spineless g-machine. In *Conference Record of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 244–258, Snowbird, Utah, July 1988. ACM Press.

[BR01] David F. Bacon and V.T. Rajan. Concurrent cycle collection in reference counted systems. In Jørgen Lindskov Knudsen, editor, *Proceedings of 15th European Conference on Object-Oriented Programming, ECOOP 2001*, volume 2072 of *Lecture Notes in Computer Science*, Budapest, June 2001. Springer-Verlag.

[Bro84] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In Steele [Ste84], pages 256–262.

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA'97 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, pages 324–34, San Jose, CA, October 1996. ACM Press.

[BSH+01] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinley, and J. Eliot B. Moss. Pretenuring for Java. In OOPSLA [OOP01], pages 342–352.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[BZ93a] David A. Barrett and Benjamin Zorn. Garbage collection using a dynamic threatening boundary. Computer Science Technical Report CU-CS-659-93, University of Colorado, July 1993.

[BZ93b] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In PLDI [PLD93], pages 187–196.

[BZM01] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing high-performance memory allocators. In PLDI [PLD01].

[BZM02] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *OOPSLA'02 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Seattle, WA, November 2002. ACM Press.

[CB01] Perry Cheng and Guy Blelloch. A parallel, real-time garbage collector. In PLDI [PLD01], pages 125–136.

[CFNP07] A. M. Cheadle, A. J. Field, and J. Nystrom-Persson. Method specialisation and incremental garbage collection in java. Technical report, Department of Computing, Imperial College, May 2007.

[CFNP08] 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.

[CFS+00] Andrew M. Cheadle, Anthony J. Field, Marlow Simon, Simon L. Peyton-Jones, and R.L While. Non-stop Haskell. In *Proceedings of International Conference on Functional Programming*, Montreal, September 2000. ACM Press.

[CFS+04] Andrew M. Cheadle, Anthony J. Field, Marlow Simon, Simon L. Peyton-Jones, and Lyndon While. Exploring the barrier to entry — incremental generational garbage collection for Haskell. In Bacon and Diwan [BD04].

[CG94] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in c++ programs. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 397–408, New York, NY, USA, 1994. ACM.

[CGS+99] Jong-Deok Choi, M. Gupta, Maurice Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 1–19, Denver, CO, October 1999. ACM Press.

[CH00] Craig Chambers and Antony L. Hosking, editors. *Proceedings of the Second International Symposium on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*, Minneapolis, MN, October 2000. ACM Press.

[Che70] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[Che01] Perry Cheng. *Scalable Real-Time Parallel Garbage Collection for Symmetric Multiprocessors*. PhD thesis, Carnegie Mellon University, 2001.

[CHL98] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Montreal, June 1998. ACM Press.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[CKV+03] Guangyu Chen, Mahmut Kandemir, Naraya Vijaykrishnan, Mary Jane Irwin, Bernd Mathiske, and Mario Wolczko. Heap compression for memory-constrained Java environment. In OOPSLA [OOP03].

[Coh81] Jacques Cohen. Garbage collection of linked data structures. *Computing Surveys*, 13(3):341–367, September 1981.

[Col60] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[Cou88] Robert Courts. Improving locality of reference in a garbage-collecting memory management-system. *Communications of the ACM*, 31(9):1128–1138, 1988.

[CW00] Douglas E. Comer and Andy J. Wellings, editors. *Persistent Object Systems*, volume 30, New York, NY, USA, 2000. John Wiley & Sons, Inc.

[DA99] David Detlefs and Ole Agesen. Inlining of virtual methods. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 258–278, London, UK, 1999. Springer-Verlag.

[DB76] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[Det04] David Detlefs. A hard look at hard real-time garbage collection. In *Proceedings of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, pages 23–32, Vienna, May 2004. Invited paper.

[DFHP04] David Detlefs, Christine Flood, Steven Heller, and Tony Printezis. Garbage-first garbage collection. In Bacon and Diwan [BD04].

[DG94] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In POPL [POP94].

[DHV95] Karel Driesen, Urs Hölzle, and Jan Vitek. Message dispatch on pipelined processors. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 253–282, London, UK, 1995. Springer-Verlag.

[Dij75] Edsgar W. Dijkstra. Notes on a real-time garbage collection system. From a conversation with D. E. Knuth (private collection of D. E. Knuth), 1975.

[DKL+00] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Elliot E. Salant, Katherine Barabash, Itai Lahan, Erez Petrank, Igor Yanover, and Yossi Levanoni. Implementing an on-the-fly garbage collector for Java. In Chambers and Hosking [CH00].

[DKL+02] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-local heaps for Java. In Boehm and Detlefs [BD02], pages 76–87.

[DKP00] Tamar Domani, Elliot Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Vancouver, June 2000. ACM Press.

[DL93] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 113–123. ACM Press, January 1993.

[DLM+76] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Language Hierarchies and Interfaces: International Summer School*, volume 46 of *Lecture Notes in Computer Science*, pages 43–56. Springer-Verlag, Marktoberdorf, Germany, 1976.

[DW97] Peter Dickman and Paul R. Wilson, editors. *OOPSLA '97 Workshop on Garbage Collection and Memory Management*, October 1997.

[Edwwn] Daniel J. Edwards. Lisp II garbage collector. AI Memo 19, MIT AI Laboratory, Date unknown.

[ELA88] John R. Ellis, Kai Li, and Andrew W. Appel. Real-time concurrent collection on stock multiprocessors. Technical Report DEC–SRC–TR–25, DEC Systems Research Center, Palo Alto, CA, February 1988.

[Fer95] Mary F. Fernández. Simple and effective link-time optimization of modula-3 programs. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference*

*on Programming Language Design and Implementation*, pages 103–115, New York, NY, USA, 1995. ACM.

[FF81] John K. Foderaro and Richard J. Fateman. Characterization of VAX Macsyma. In *1981 ACM Symposium on Symbolic and Algebraic Computation*, pages 14–19, Berkeley, CA, 1981. ACM Press.

[FH88] A. J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, July 1988.

[FN78] John P. Fitch and Arthur C. Norman. A note on compacting garbage collection. *Computer Journal*, 21(1):31–34, February 1978.

[FY69] Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[GDGC95] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-guided receiver class prediction. *SIGPLAN Notices*, 30(10):108–123, 1995.

[GM04] Samuel Guyer and Kathryn McKinley. Finding your cronies: Static analysis for dynamic object colocation. In OOPSLA [OOP04].

[GP] Free Software Foundation GNU Project. Gnu classpath.

[GS98] David Gay and Bjarne Steensgaard. Stack allocating objects in Java. Technical report, Microsoft Research, October 1998.

[GS00] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *International Conference on Compiler Construction (CC'2000)*, volume 1781 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[HÖ3] Urs Hölzle. A fast write barrier for generational garbage collectors. In Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

[Han90] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software Practice and Experience*, 20(1):5–12, January 1990.

[Han94] Chris Hankin. *Lambda Calculi: A Guide for Computer Scientists*. Clarendon Press (Oxford University Press), October 1994.

[Har00] Timothy Harris. Dynamic adaptive pre-tenuring. In Chambers and Hosking [CH00].

[Hay91] Barry Hayes. Using key object opportunism to collect old objects. In Andreas Paepcke, editor, *OOPSLA'91 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 26(11) of *ACM SIGPLAN Notices*, pages 33–46, Phoenix, Arizona, October 1991. ACM Press.

[HCU92] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pages 32–43, New York, NY, USA, 1992. ACM.

[HD90] Richard L. Hudson and Amer Diwan. Adaptive garbage collection for Modula-3 and Smalltalk. In Eric Jul and Niels-Christian Juul, editors, *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, October 1990.

[HDH03] Martin Hirzel, Amer Diwan, and Matthew Hertz. Connectivity-based garbage collection. In OOPSLA [OOP03].

[Hen94] Roger Henriksson. Scheduling real-time garbage collection. In *Proceedings of NWPER'94*, Lund, Sweden, 1994.

[Hen96a] Roger Henriksson. Adaptive scheduling of incremental copying garbage collection for interactive applications. Technical Report 96–174, Lund University, Sweden, 1996.

[Hen96b] Roger Henriksson. Scheduling real-time garbage collection. Licentiate thesis, Department of Computer Science, Lund University, 1996. Lund technical report LU-CS-TR:96-161.

[Hen97] Roger Henriksson. Predictable automatic memory management for embedded systems. In Dickman and Wilson [DW97].

[Hen98] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.

[Hen02] Fergus Henderson. Accurate garbage collection in an uncooperative environment. In Boehm and Detlefs [BD02], pages 150–156.

[HHDH02] Martin Hirzel, Johannes Henkel, Amer Diwan, and Michael Hind. Understanding the connectivity of heap objects. In Boehm and Detlefs [BD02], pages 36–49.

[HHMN98] Michael Hicks, Luke Hornof, Jonathan T. Moore, and Scott Nettles. A study of Large Object Spaces. In Jones [Jon98], pages 138–145.

[HL93] Lorenz Huelsbergen and James R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Fourth Annual ACM Symposium on Principles and Practice of Parallel Programming*, volume 28(7) of *ACM SIGPLAN Notices*, pages 73–82, San Diego, CA, May 1993. ACM Press.

[HM92] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In Bekkers and Cohen [BC92].

[HM01] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying GC without stopping the world. In *Joint ACM Java Grande — ISCOPE 2001 Conference*, Stanford University, CA, June 2001.

[HM03] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying garbage collection without stopping the world. *Concurrency and Computation: Practice and Experience*, 15(3–5):223–261, 2003.

[HMS92] Anthony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In Andreas Paepcke, editor, *OOPSLA'92 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 27(10) of *ACM SIGPLAN Notices*, pages 92–109, Vancouver, British Columbia, October 1992. ACM Press.

[HSaC04] Wei Huang, W. Srisa-an, and J. M. Chang. Adaptive pretenuring for generational garbage collection. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-04)*, pages 133–140, Austin, TX, March 2004.

[Hug82] R. J. M. Hughes. Super-combinators a new implementation method for applicative languages. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10, Pittsburgh, PA, August 1982. ACM Press.

[HW67] B. K. Haddon and W. M. Waite. A compaction procedure for variable length storage elements. *Computer Journal*, 10:162–165, August 1967.

[IKY+00] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a java just-in-time compiler. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 294–310, New York, NY, USA, 2000. ACM.

[JBM04] Maria Jump, Stephen M. Blackburn, and Kathryn S. McKinley. Dynamic object sampling for pretenuring. In Bacon and Diwan [BD04].

[JK04] Richard E. Jones and Andy C. King. Collecting the garbage without blocking the traffic. Technical Report 18–04, Computing Laboratory, University of Kent, September 2004. This report summarises [Kin04].

[JK05] Richard E. Jones and Andy C. King. A fast analysis for thread-local garbage collection with dynamic class loading. In *Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 129–138, Budapest, September 2005. This is a shorter version of [JK04].

[Joh84] Thomas Johnsson. Efficient compilation of lazy evaluation. *SIGPLAN Notices*, 19(6):58–69, 1984.

[Joh88] Douglas Johnson. Trap architectures for Lisp systems. Technical Report UCB/CSD/88/470, University of California, Berkeley, November 1988.

[Joh91] Douglas Johnson. The case for a read barrier. *ACM SIGPLAN Notices*, 26(4):279–287, 1991.

[Joh97] Mark S. Johnstone. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, University of Texas at Austin, December 1997.

[Jon79] H. B. M. Jonkers. A fast garbage compaction algorithm. *Information Processing Letters*, 9(1):25–30, July 1979.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[Jon92] Richard E. Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–79, January 1992.

[Jon96] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[Jon98] Richard Jones, editor. *Proceedings of the International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, Vancouver, October 1998. ACM Press.

[JRR99] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. C–: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, September 1999.

[JW97] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In Dickman and Wilson [DW97].

[JW98] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In Jones [Jon98], pages 26–36.

[KCKS99] Taehyoun Kim, Naehyuck Chang, Namyun Kim, and Heonshik Shin. Scheduling garbage collector for embedded real-time systems. In *ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'99)*, pages 55–64, Atlanta, GA, May 1999. ACM Press.

[KCS00] Taehyoun Kim, Naehyuck Chang, and Heonshik Shin. Bounding worst case garbage collection time for embedded real-time systems. In *Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000)*, 2000.

[KCS01] Taehyoun Kim, Naehyuck Chang, and Heonshik Shin. Joint scheduling of garbage collector and hard real-time tasks for embedded applications. *Journal of Systems and Software*, 58(3):247–260, September 2001.

[KDS01] Graham N. C. Kirby, Alan Dearle, and Dag I. K. Sjøberg, editors. *Persistent Object Systems, 9th International Workshop, POS-9, Lillehammer, Norway, September 6-8, 2000, Revised Papers*, volume 2135 of *Lecture Notes in Computer Science*. Springer, 2001.

[Kin04] Andy C. King. *Removing Garbage Collector Synchronisation.* PhD thesis, Computing Laboratory, The University of Kent at Canterbury, 2004.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume I: Fundamental Algorithms, pages 602–603. Addison-Wesley, second edition, 1973.

[KP06] Haim Kermany and Erez Petrank. The Compressor: Concurrent, incremental and parallel compaction. In PLDI [PLD06], pages 354–363.

[KS77] H. T. Kung and S. W. Song. An efficient parallel garbage collection system and its correctness proof. In *IEEE Symposium on Foundations of Computer Science*, pages 120–131. IEEE Press, 1977.

[Lan64] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, January 1964.

[LCT03] *ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'2003)*, San Diego, CA, June 2003. ACM Press.

[Lea97] Doug Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html, 1997.

[LH83] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[Lin92] Rafael D. Lins. Cyclic reference counting with lazy mark-scan. *Information Processing Letters*, 44(4):215–220, 1992. Also Computing Laboratory Technical Report 75, University of Kent, July 1990.

[LP01] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. In OOPSLA [OOP01].

[LSR94] S.S. Lui Sha ; Rajkumar, R. ; Sathaye. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. *Proceedings of the IEEE*, 82(1):68–82, 1994.

[Mae02] Jan-Willem Maessen. *Hybrid Eager and Lazy Evaluation for Efficient Compilation of Haskell*. PhD dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2002.

[MBMZ01] Alonso Marquez, Stephen Blackburn, Gavin Mercer, and John N. Zigman. Implementing orthogonally persistent java. In *POS-9: Revised Papers from the 9th International Workshop on Persistent Object Systems*, pages 247–261, London, UK, 2001. Springer-Verlag.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, April 1960.

[MH95] Boris Magnusson and Roger Henriksson. Garbage collection for control systems. In Baker [Bak95].

[MH96] J. Eliot B. Moss and Antony L. Hosking. Approaches to adding persistence to java. In *First International Workshop on Persistence and Java*. Sun Microsystems, 1996.

[MHJJ08] Simon Marlow, Tim Harris, Roshan James, and Simon Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In Richard Jones and Steve Blackburn, editors, *Proceedings of the Seventh International Symposium on Memory Management*, pages 11–20, Tucson, AZ, June 2008. ACM Press.

[Min63] Marvin L. Minsky. A Lisp garbage collector algorithm using serial secondary storage. Technical Report Memo 58 (rev.), Project MAC, MIT, Cambridge, MA, December 1963.

[MJ04] Simon Marlow and Simon Peyton Jones. Making a fast curry: Push/enter vs. eval/apply for higher-order languages. *SIGPLAN Notices*, 39(9):4–15, 2004.

[MJR07] Sebastien Marion, Richard Jones, and Chris Ryder. Decrypting the Java gene pool: Predicting objects' lifetimes with micro-patterns. In Morrisett and Sagiv [MS07], pages 67–78.

[Moo84] David A. Moon. Garbage collection in a large LISP system. In Steele [Ste84], pages 235–245.

[Mor78] F. Lockwood Morris. A time- and space-efficient garbage compaction algorithm. *Communications of the ACM*, 21(8):662–5, 1978.

[Mos90] J. Eliot B. Moss. Working with objects: To swizzle or not to swizzle? Technical Report 90–38, University of Massachusetts, Amherst, MA, May 1990.

[MS07] Greg Morrisett and Mooly Sagiv, editors. *Proceedings of the Sixth International Symposium on Memory Management*, Montréal, Canada, October 2007. ACM Press.

[MVA05] Raymond Devillers Maxime Van Assche, Joël Goossens. Joint garbage collection and hard real-time scheduling. In *Proceedings of Thirteenth International Conference on Real-time Systems*, Paris, France, April 2005.

[MWL90] A. D. Martinez, R. Wachenchauzer, and Rafael D. Lins. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34:31–35, 1990.

[MZB00] A. Marquez, J. N. Zigman, and S. M Blackburn. Fast portable orthogonally persistent Java. *Software Practice and Experience*, 30(4):449–479, 2000.

[NO93] Scott M. Nettles and James W. O'Toole. Real-time replication-based garbage collection. In PLDI [PLD93].

[NOPH92] Scott M. Nettles, James W. O'Toole, David Pierce, and Nicholas Haines. Replication-based incremental copying collection. In Bekkers and Cohen [BC92].

[OBYG+02] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent GC for servers. In PLDI [PLD02], pages 129–140.

[OBYS04] Yoav Ossia, Ori Ben-Yitzhak, and Marc Segal. Mostly concurrent compaction for mark-sweep GC. In Bacon and Diwan [BD04].

[OOP01] *OOPSLA'01 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 36(10) of *ACM SIGPLAN Notices*, Tampa, FL, October 2001. ACM Press.

[OOP03] *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.

[OOP04] *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, volume 39(10) of *ACM SIGPLAN Notices*, Vancouver, Canada, October 2004. ACM Press.

[Par93] Will Partain. The nofib benchmark suite of haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, 1993. Springer-Verlag.

[PBW85] Edwin Pittomvils, Maurice Bruynooghe, and Yves D. Willems. Towards a real time garbage collector for PROLOG. In *1985 Symposium on Logic Programming. Boston, 1985 Jul 15–18*, pages 185–198. IEEE Press, 1985.

[PFPS07] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgard. STOPLESS: A real-time garbage collector for multiprocessors. In Morrisett and Sagiv [MS07], pages 159–172.

[Pir98] Pekka P. Pirinen. Barrier techniques for incremental tracing. In Jones [Jon98], pages 20–25.

[PJ92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[PJGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *Conference Record of the Twenty-third Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, 1996.

[PJL91] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 636–666, London, UK, 1991. Springer-Verlag.

[PJS89] Simon L. Peyton Jones and Jon Salkild. The spineless tagless g-machine. In *Record of the 1989 Conference on Functional Programming and Computer Architecture*, pages 184–201, Imperial College, London, August 1989. ACM Press.

[PLD93] *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, Albuquerque, NM, June 1993. ACM Press.

[PLD01] *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Snowbird, Utah, June 2001. ACM Press.

[PLD02] *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Berlin, June 2002. ACM Press.

[PLD06] *Proceedings of SIGPLAN 2006 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Ottawa, June 2006. ACM Press.

[POP94] *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, Portland, OR, January 1994. ACM Press.

[PPB+05] Harel Paz, Erez Petrank, David F. Bacon, V.T. Rajan, and Elliot K. Kolodner. An efficient on-the-fly cycle collection. In *Proceedings of the 14th International Conference on Compiler Construction*, Edinburgh, April 2005. Springer-Verlag.

[PPS08] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In *Proceedings of SIGPLAN 2008 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 33–44, Tucson, AZ, June 2008. ACM Press.

[Pro] The Apache Jakarta Project. The byte code engineering library.

[Pro00] Java Community Process. Jsr 1: Real-time specification for java, May 2000.

[Pro09] Java Community Process. Jsr 282: Rtsj version 1.1, May 2009.

[PS95] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In Baker [Bak95].

[PWBK07] David J. Pearce, Matthew Webster, Robert Berry, and Paul H. J. Kelly. Profiling with aspectj. *Software Practice and Experience*, 37(7):747–777, 2007.

[RÖ95] Niklas Röjemo. Generational garbage collection without temporary space leaks for lazy functional languages. In Baker [Bak95].

[RH03] Sven Gestegøard Robertz and Roger Henriksson. Time-triggered garbage collection — robust and adaptive real-time GC scheduling for embedded systems. In LCTES [LCT03].

[Rob02] Sven Robertz. Applying priorities to memory allocation. In Boehm and Detlefs [BD02], pages 1–11.

[RS05] A. Ravindar and Y.N. Srikant. Static analysis for identifying and allocating clusters of immortal objects. In *.NET Technologies 2005*, Plzen, Czech Republic, 2005.

[SAB+06] Daniel Spoonhower, Joshua Auerbach, David F. Bacon, Perry Cheng, and David Grove. Eventrons: A safe programming construct for high-frequency hard real-time applications. In PLDI [PLD06].

[San92] Patrick M. Sansom. Combining copying and compacting garbage collection. In Simon L. Peyton Jones, G. Hutton, and C. K. Hols, editors, *Fourth Annual Glasgow Workshop on Functional Programming*, Workshops in Computer Science. Springer-Verlag, 1992.

[SAr+04] Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2-3):101–155, 2004.

[Sch90] J. Schimpf. Garbage collection for Prolog based on twin cells. In *2nd NACLP Workshop on Logic Programming Architectures and Implementations*. MIT Press, 1990.

[SCN84] Will R. Stoye, T. J. W. Clarke, and Arthur C. Norman. Some practical methods for rapid combinator reduction. In Steele [Ste84], pages 159–166.

[Sew] Julian Seward. Valgrind.

[Sew92] Julian Seward. Generational garbage collection for lazy graph reduction. In Bekkers and Cohen [BC92].

[SG95] Jacob Seligmann and Steffen Grarup. Incremental mature garbage collection using the train algorithm. In O. Nierstras, editor, *Proceedings of 1995 European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, pages 235–252, University of Aarhus, August 1995. Springer-Verlag.

[Sha88] Robert A. Shaw. *Empirical Analysis of a Lisp System*. PhD thesis, Stanford University, 1988. Technical Report CSL–TR–88–351.

[SHB+02] Darko Stefanović, Matthew Hertz, Stephen Blackburn, Kathryn McKinley, and J. Eliot Moss. Older-first garbage collection in practice: Evaluation in a Java virtual machine. In *ACM SIGPLAN Workshop on Memory System Performance (MSP 2002)*, Berlin, June 2002.

[SHR+00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 264–280, New York, NY, USA, 2000. ACM.

[Sie98] Fridtjof Siebert. Guaranteeing non-disruptiveness and real-time deadlines in an incremental garbage collector. In Jones [Jon98], pages 130–137.

[Sie00] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Compilers, Architectures and Synthesis for Embedded Systems (CASES2000)*, San Jose, November 2000.

[Sie02] Fridtjof Siebert. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*. aicas Books, 2002.

[Sie07] Fridtjof Siebert. Realtime garbage collection in the JamaicaVM 3.0. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 94–103, Vienna, Austria, September 2007. ACM Press.

[SM03] Narendran Sachindran and Eliot Moss. MarkCopy: Fast copying GC with less space overhead. In OOPSLA [OOP03].

[Sob88] Patrick Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. Technical Report AITR-1417, MIT AI Lab, February 1988. Bachelor of Science thesis.

[SP93] Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for Haskell. In R. John M. Hughes, editor, *Record of the 1993 Conference on Functional Programming and Computer Architecture*, University of Glasgow, June 1993. ACM Press.

[Spr90] Brinkley Sprunt. *Aperiodic Task Scheduling for Real-Time Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1990.

[Ste75] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[Ste84] Guy L. Steele, editor. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, TX, August 1984. ACM Press.

[Ste99] Darko Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. PhD thesis, University of Massachusetts, 1999.

[Ste00] Bjarne Steensgaard. Thread-specific heaps for multi-threaded programs. In Chambers and Hosking [CH00].

[SW67] H. Schorr and W. Waite. An efficient machine independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

[SZ97] Matthew L. Seidl and Benjamin Zorn. Predicting references to dynamically allocated objects. Technical Report CU-CS-826-97, University of Colorado, January 1997.

[TBE+97] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with Regions in the ML Kit. Technical Report DIKU-TR-97/12, Department of Computer Science (DIKU), University of Copenhagen, April 1997.

[TICL] University of Tennessee The Innovatice Computing Laboratory. Performance application programming interface.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

[Tof98] Mads Tofte. A brief introduction to Regions. In Jones [Jon98], pages 186–195.

[Ton97] Guanshan Tong. *Leveled Garbage Collection For Automatic Memory Management*. PhD thesis, University of Chicago, November 1997.

[TS97] Frank Tip and Peter F. Sweeney. Class hierarchy specialization. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 271–285, New York, NY, USA, 1997. ACM.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, February 1997. An earlier version of this was presented at [POP94].

[TT99] Kresten Krab Thorup and Mads Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized classes. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 186–204, London, UK, 1999. Springer-Verlag.

[Tur79] David A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9, 1979.

[UJ88] David M. Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. *ACM SIGPLAN Notices*, 23(11):1–17, 1988.

[UJ92] David M. Ungar and Frank Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, 1992.

[Ung84] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

[Wad71] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University, 1971.

[War83] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, 1983.

[Weg72] B. Wegbreit. A generalised compactifying garbage collector. *Computer Journal*, 15(3):204–208, August 1972.

[WF92] R. Lyndon While and Tony Field. Incremental garbage collection for the Spineless Tagless G-machine. In Evan Ireland and Nigel Perry, editors, *Proceedings of the Massey Functional Programming Workshop 1992*. Department of Computer Science, Massey University, 1992.

[WF93] R. Lyndon While and A. J. Field. The non-stop spineless tagless g-machine. Departmental Report DoC 93/8, Imperial College, London, 1993.

[WG98] Derek White and Alex Garthwaite. The GC interface in the EVM. Technical Report SML TR–98–67, Sun Microsystems Laboratories, December 1998.

[Wil89] Paul R. Wilson. A simple bucket-brigade advancement mechanism for generation-based garbage collection. *ACM SIGPLAN Notices*, 24(5):38–46, May 1989.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Bekkers and Cohen [BC92].

[Wil94] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Baker [Bak95].

[WM89] Paul R. Wilson and Thomas G. Moher. A card-marking scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *ACM SIGPLAN Notices*, 24(5):87–92, 1989.

[Yua90] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990.

[ZCC97] Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient dynamic dispatch without virtual function tables: The smalleiffel compiler. *SIGPLAN Notices*, 32(10):125–141, 1997.

[Zig] J. N. Zigman. Bytecode transformation tools for jikes rvm.

[Zor89] Benjamin G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, March 1989. Technical Report UCB/CSD 89/544.

[Zor90] Benjamin Zorn. Barrier methods for garbage collection. Technical Report CU-CS-494-90, University of Colorado, Boulder, November 1990.

[ZS03] J. N. Zigman and R. Sankaranarayana. Designing a distributed jvm on a cluster. In *ESM 2003: Proceedings of the 17th European Simulation Multiconference*, Erlangen, Germany, 2003. SCS Europe Bvba.