An Examination of Deferred Reference Counting and Cycle Detection

Luke Nathan Quinane

A subthesis submitted in partial fulfillment of the degree of
Bachelor of Software Engineering (Research Project)
at
The Department of Computer Science
Australian National University

April 2004

Typeset in Palatino by TeX and LaTeX 2_ε.

Except where otherwise indicated, this thesis is my own original work.

Luke Nathan Quinane
30 April 2004

To my parents and step-parents, for helping me through my education.

Acknowledgements

Firstly I would like to thank my supervisor Steve Blackburn. You have helped throughout my project in many ways. You offered much assistance and guidance and your energy and enthusiasm for this area of research is contagious. I would also like to thank you for the usage of two machines, 'chipmunk' and 'raccoon' on which many of my results were produced.

Thanks also to Daniel Frampton for his help during the many late nights and white board sessions. Daniel was also kind enough to provide usage of 'toki' on which the remainder of my results were produced.

Thanks to Ian, Robin, Fahad and Andrew for listening and brain-storming during our weekly meetings.

Thanks to John Zigman for raising the humour level at our meetings and for reminding me that my problems were always small in comparison (. . . and only running on one computer).

I would also like to thank Daniel B. I couldn't have done the robotics lab and assignment without your help.

Thanks to Stephen Milnes from the Academic Skills and Learning Centre for reading over my drafts.

Thanks to my girlfriend Mira for working late and generally putting up with all my grumpiness.

Finally I would like to thank my parents and step-parents. They have provided me with support, understanding and caring.

Abstract

Object-oriented programing languages are becoming increasingly important as are managed runtime-systems. An area of importance in such systems is dynamic automatic memory management. A key function of dynamic automatic memory management is detecting and reclaiming discarded memory regions; this is also referred to as garbage collection.

A significant proportion of research has been conducted in the field of memory management, and more specifically garbage collection techniques. In the past, adequate comparisons against a range of competing algorithms and implementations has often been overlooked. JMTk is a flexible memory management toolkit, written in Java, which attempts to provide a testbed for such comparisons.

This thesis aims to examine the implementation of one algorithm currently available in JMTk: the deferred reference counter. Other research has shown that the reference counter in JMTk performs poorly both in throughput and responsiveness. Several aspects of the reference counter are tested, including the *write barrier*, *allocation cost*, *increment and decrement processing* and *cycle-detection*.

The results of these examinations found the *bump-pointer* to be 8% faster than the *free-list* in raw allocation. The cost of the reference counting write barrier was determined to be 10% on the PPC architecture and 20% on the i686 architecture. Processing increments in the write barrier was found to be up to 13% faster than buffering them until collection time on a uni-processor platform. Cycle detection was identified as a key area of cost in reference counting.

In order to improve the performance of the deferred reference counter and to contribute to the JMTk testbed, a new algorithm for detecting cyclic garbage was described. This algorithm is based on a mark scan approach to cycle detection. Using this algorithm, two new cycle detectors were implemented and compared to the original trial deletion cycle detector.

The semi-concurrent cycle detector had the best throughput, outperforming trial deletion by more than 25% on the *javac* benchmark. The non-concurrent cycle detector had poor throughput attributed to poor triggering heuristics. Both new cycle detectors had poor pause times. Even so, the semi-concurrent cycle detector had the lowest pause times on the *javac* benchmark.

The work presented in this thesis contributes to an evaluation of components of the reference counter and a comparsion between approaches to reference counting implementation. Previous to this work, the cost of the reference counter's components had not been quantified. Additionally, past work presented different approaches to reference counting implementation as a whole, instead of individual components.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

With industry leaders such as Sun, IBM and Microsoft pushing managed run-times such as *Java* and *.NET*, automatic dynamic memory management is becoming increasingly important. This is especially highlighted by papers which suggest that a program could spend up to a third of its running time performing *garbage collection* [Levanoni and Petrank 2001].

Study in the field of automatic dynamic memory management dates back to 1960 and much research has been conducted. While there are many papers and reports available on the performance of memory management algorithms and implementations, they are often implemented in different environments. Unfortunately this makes it difficult to compare different collection algorithms as other elements from the underlying runtime affect the comparison.

Reference counting is one approach to automatic memory management that has not had its differing implementations thoroughly compared and examined. For example, the *mark-scan* and *trial-deletion* approaches to *cycle detection* have been described and implemented but not compared with each other. Instead, implementations of reference counters and their components are presented as a whole when they are really several orthogonal components [Bacon et al. 2001; Levanoni and Petrank 2001].

The reference counter is also considered to perform poorly in throughput when compared to other collection algorithms [Blackburn and McKinley 2003; Blackburn et al. 2003]. However the reasons for this poor performance have not been examined.

The cost of maintaining reference counts for each object is also thought to be high [Wilson 1992; Jones 1996], but again the costs have not been quantified.

This thesis aims to dissect the costs involved in the components of the one reference counter implementation and, where possible, offer improvements.

1.2 Approach

The costs involved in allocation, the write barrier, processing increments, processing decrements, processing roots and cycle detection will be examined and their costs quantified.

Using these results and the knowledge of the algorithm, improvements to the current implementation will be proposed and tested.

All work will be done in JMTk, a flexible memory management toolkit written in Java [Blackburn et al. 2003]. This will allow any improvements to be compared against other collectors in the same platform.

1.3 Contribution

The main costs involved in reference counting are dissected. Allocation cost, the cost of the write barrier, different approaches for handling increments and the costs of processing decrements, roots and cycle detection are quantified.

A new algorithm for collecting cyclic garbage is described and discussed. Using this new approach to cycle detection, two new cycle detector implementations are provided to JMTk.

The new cycle detectors are compared to the original cycle detector. One of the cycle detector implementations performed well in throughput. However, both performed poorly in responsiveness.

1.4 Organisation

Chapter 2 gives an overview of the garbage collection problem domain. Some approaches to garbage collection are discussed, particularly those relating to reference counting.

Chapter 3 describes the environment in which the reference counter is currently implemented.

Chapter 4 examines the various costs involved in deferred reference counting garbage collection. Preliminary results indicate that cycle detection is a major cost in the reference counter.

Chapter 5 describes a new algorithm for collecting cyclic garbage. Based on this algorithm, two alternate cycle detectors are implemented. These implementations aim to improve the performance of the reference counter.

Chapter 6 presents results from the cost analysis and an evaluation of the new cycle detectors.

Chapter 7 critically examines the result obtain in Chapter 6.

Chapter 8 makes recommendations based on the findings from chapter 7. Further areas of research are also presented.

Chapter 2

Background

This chapter aims to provide background information on the general problem of memory management. The differences between traditional methods for memory management and automatic dynamic memory management are examined. Alternative approaches for memory allocation and reclamation are described with respect to garbage collection in object-oriented systems. Approaches to mark-sweep and reference counting collection are described in more detail.

2.1 Traditional Dynamic Memory Management

Traditional memory management places the onus of memory management on the programmer. The programmer must explicitly instruct the program when to allocate and release regions of memory as well as indicate the quantity of memory required by the program. The programmer must also ensure that any pointers in the program are valid. During the program execution and in a concurrent setting, the programmer must also ensure that the order in which the memory is accessed is legal and resulting data is consistent. If any of these components of the program's memory management is defective, undesirable and unpredictable behaviour can result [Jones 1996].

Special care must be taken when sharing data between modules [DeTreville 1990; Wilson 1992]. It must be checked that one module does not free the allocated memory while another module is using it. It must also be checked that the allocated memory is eventually freed when all modules have finished using it.

In addition, pointer arithmetic and pointers to arbitrary memory regions are allowed. Thus the programmer must check that memory locations reference by these pointers are only accessed when the pointers are referencing valid locations.

2.2 Automatic Dynamic Memory Management

Automatic dynamic memory management is a high level facility provided to the programmer that removes the need for explicit memory management. It allows memory regions to be allocated automatically as they are required and collected automatically as they become discarded [Wilson 1992].

Many object-oriented languages have automatic dynamic memory management as part of their specification. These languages include C# [ECMA 2002], Java [Joy et al. 2000] and Smalltalk [Committee 1997]. When referred to in this context, automatic dynamic memory management is called *garbage collection*. These languages also ensure that all references are either valid or null, thus removing another burden from the programmer.

Meyer considers that automatic dynamic memory management is an important part of a good object-oriented language. Further, he states that leaving developers responsible for memory management complicates the software being developed and compromises its safety [Meyer 1988].

It is possible to implement automatic dynamic memory management for most languages, even those that do not include garbage collection in their specification. This is the case for two popular languages, C and C++ [Jones 1996]. However, these implementations may need to operate without communication from the compiler. Additionally, these languages usually have very different allocation and collection patterns to languages commonly associated with garbage collection [Jones 1996].

2.2.1 Collector Evaluation

When evaluating the performance of garbage collection algorithms several qualities must be taken into consideration. These include the throughput and pause-times of the collector and the required meta-data.

The throughput of the collector indicates the speed at which the collector is able to reclaim garbage. The higher a collector's throughput the more efficient it will be able to complete processing its workload. High throughput is important for systems that generate large quantities of garbage. Throughput provides an indication of the overhead the collector places on total system running time [Jones 1996].

The collector's pause-times indicate how long a collection cycle takes. Low pause-times are important for interactive and hard real-time systems. In most cases there is a trade-off between short pause-times and high throughput [Blackburn and McKinley 2003].

Some collectors, such as reference counting collectors, require meta-data in order to operate. This data can be stored both in the object header or associated with the collector. Some meta-data may be required throughout the execution of the program while other meta-data is only required at collection time. The quantity of meta-data required by the collection algorithm is an important consideration for systems with limited memory [Jones 1996].

2.2.2 Allocation

Two approaches are used to allocate memory to newly created objects. One approach is to maintain a pointer to the end of the last allocated object. Newly requested space starts from the pointer position. After allocation the pointer is updated to the end of the new object. This pointer is known as a bump-pointer, see figure 2.1. Bump-pointers provide fast allocation routines as only one field needs to be read and updated. The drawback to using bump-pointers is that they are only able to allocate into an empty memory region. This means that the region of memory allocated by a bump-pointer must be freed completely before it can be reused.

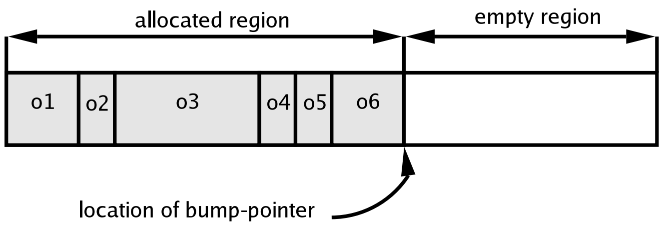


Figure 2.1: An example bump-pointer allocator.

The other approach is to allocate objects using a "free-list". A free-list indicates regions of memory not containing live objects, see figure 2.2. When the collector finds a garbage object, the object's space is recorded in the free-list. Free-lists are more complex than bump-pointers and are slower as a result. However, free-lists can allocate into a previously allocated space, making them essential in non-copying collectors.

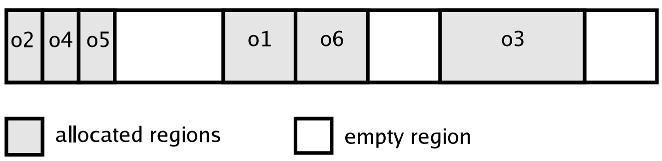


Figure 2.2: An example free-list allocator.

2.2.3 Copying Vs. Non-Copying Collectors

Garbage collection algorithms can either move objects found to be live or leave live objects fixed and maintain a free-list for the heap.

Copying collectors are able to utilise the fast allocation associated with a bump-pointer, as they are always allocating into unused space. However, they must pay the overhead of copying all live objects from the space being collected. In the worst case, all objects in the collected region will remain live and the collector must have enough free space to copy these objects [Blackburn et al. 2002]. In the most inefficient implementation, a copying collector will only be able to allocate half of the memory region it is managing.

Non-copying collectors avoid the costs associated with copying objects but have a more complicated allocation routine. Conversely, the free-list avoids the need for a "copy-to" space that copying collectors require.

2.2.4 Tracing Vs. Reference Counting Collectors

Garbage collection algorithms use two approaches to determine if an object is live. The first method is to trace through the graph of live objects. Any object reached in this traversal must be live due to the nature of garbage objects [McCarthy 1960]. The other approach is to record the number of references to an object [Collins 1960]. When an object has a reference count of zero it is garbage, as no live objects are referencing it.

One advantage of tracing algorithms is that unlike reference counting, no special action is required to reclaim cyclic garbage [Schorr and Waite 1967]. Since a garbage cycle only contains internal references, a trace of live objects will never reach the garbage cycle indicating that is not live. Reference counting algorithms must therefore employ additional approaches to detect garbage cycles.

Non-incremental, non-concurrent tracing collectors also avoid the overhead of a write barrier, which is required by reference counting algorithms [Jones 1996].

2.2.5 Write Barriers

A write barrier is a piece of code that is executed every time a reference mutation occurs. The write barrier can either be conditional or fixed. Conditional write barriers can either execute a “fast-path” or a “slow-path”. The fast-path checks if the reference mutation should be recorded, while the slow-path actually records the mutation. Fixed write barriers have no fast-path and execute the slow-path for every mutation.

Conditional write barriers are used in generational collectors, collectors that perform incremental collection and also other styles of collectors [Jones 1996]. Reference counting collectors can be implemented with a fixed write barrier, however coalescing and generational reference counting still use a conditional write barrier [Levanoni and Petrank 2001; Blackburn and McKinley 2003].

2.2.6 Object Life-Cycles

In order to improve the performance of garbage collection algorithms, research has been conducted to identify trends in object life cycles. One trend discovered is that most objects die young. To take advantage of this behaviour, some collectors take a generational approach to garbage collection [Lieberman and Hewitt 1983; Ungar 1984]. A nursery is employed into which all new objects are allocated. Objects in the nursery remain there until the next nursery collection. At this point they are copied over into the mature object space. The premise is that because the nursery will have a high mortality rate, only a minimal subset of the nursery object will need to be copied into the mature space.

An important property of the life cycle for all objects is that when the object dies and becomes garbage, it cannot be resurrected. Once an object has been lost from the system (all its references removed) there is no way of re-locating it again [Wilson 1992].

In modern languages the exception to this rule is finalisation. A *finaliser* is a piece of code that is executed after the object dies but before the object is collected. It is possible for this code to "resurrect" the object by restoring a reference to itself.

2.2.7 Locality of Reference

Locality of reference is the degree to which subsequent memory accesses are close to each other both temporally and spatially. In both cases, closer references have better locality. Memory accesses that are close temporally have better locality if a last recently used (LRU) policy is used for the cache or paging system [Tanenbaum 1999]. This is because they are less likely to be bumped out of the cache or have their page unloaded. Memory accesses which are spatially close will have better locality since they are more likely to be on the same page, reducing the number of page faults.

A program with good locality will frequently take advantage of objects in the cache and any cache pre-fetching mechanisms. Programs with good locality will also have a concentrated working set. As indicated above, a concentrated working set helps to reduce page faults since most required data is on a small subset of all pages.

Since the allocation strategy and garbage collection policy are both responsible for the placement of objects, they can both aid or hinder benefits gained from locality. For example, segregated free-lists group similar sized objects together and copying collectors compact objects into the "copy-to" space. Thus locality is an important consideration when dealing with memory management.

2.3 Mark-Sweep Collection

The mark-sweep garbage collection algorithm operates by tracing through all live objects in the heap and then sweeping the heap for any unmarked objects. Objects that are unmarked after a traversal of the heap have no live objects referencing them and are therefore garbage.

To prevent objects from being reclaimed incorrectly the algorithm must retain garbage until a complete scan of the heap is complete. In contrast, reference counting algorithms are able to collect garbage objects immediately.[1]

In basic implementation of the mark-sweep algorithm pause times are large when compared with some other algorithms, for example reference counting [Blackburn and McKinley 2003]. The algorithm's high pause times can be attributed to its need to traverse large portions of the heap at collection time. In the marking phase, all live objects in the heap must be traversed. In the sweeping phase, the heap is traversed again updating the free-lists.

To help minimise the length of the pause times and maximise throughput, various strategies are attempted in both the marking and sweeping phases of the algorithm.

[1] This is not the case for deferred reference counting, see section 2.4.1.

2.3.1 Marking Strategies

The goal of the marking phase is to detect all live objects in the heap and to flag them as live. The live objects are detected by tracing the graph of live objects from the roots. A sub-graph is only traced if the root of the sub-graph had not been previously marked. This prevents unneeded tracing and handles cyclic data structures.

The mark state is typically stored in two locations: in the header of the object or in a separate bitmap. Storing the mark state with the object is cheaper to access than storing it in a bitmap but this is not always possible, for example in C collectors [Jones 1996].

The space required to store the bitmap is small when compared to the entire heap. This means that there is a possibility that the bitmap will remain in the cache or at least that reading and writing mark bits should not cause as many page faults. The additional advantage is that large portions of the heap are not written to while marking occurs. This helps to avoid marking many pages as dirty, thus reducing the number of page writes the operating system must perform. The downside is that writing to the mark bit is more expensive [Zorn 1989].

The marking process is recursive in nature, however the marking program cannot use recursive calls to perform the mark because a stack overflow can result. To overcome this problem two approaches can be used: storing objects that need to be traversed in a marking queue or traversing the graph using pointer reversals.

Using a marking queue avoids the need to call the marking function recursively, however it requires additional space to store the queue. Research has been undertaken to approximate the space needed, and it was found that real applications tend to produce marking stacks that are shallow in proportion to the heap size [Jones 1996]. Even so, some collectors will push large objects onto the marking queue incrementally to save space[Jones 1996].

Using the pointer reversal technique does not require any additional space to perform the mark however it is suggested to be 50% slower than using a marking queue [Schorr and Waite 1967]. Thus it is only recommended for use when marking with a mark queue fills the heap.

2.3.2 Sweeping Strategies

The most basic sweeping strategy is to traverse the entire heap in a linear fashion and push unmarked objects back onto the free-list. The heap is scanned in a linear fashion to take advantage of any page pre-fetching mechanism in the hardware or operating system [Jones 1996].

Sweeping can also be performed concurrently to the mutator. An effective way to implement this policy is to perform a partial sweep each time an allocation is requested [Hughes 1982]. This is called *lazy* sweeping and it helps to reduce the collector's pause times.

2.4 Reference Counting

Reference counting garbage collection records the number of references to live objects. When a reference to an object is created, the reference count of that object is incremented. When a reference to an object is removed, the reference count of the object is decremented. Objects with a reference count of zero are not linked to any live object and are therefore garbage. When an object's reference count drops to zero, the reference counts of child objects are decremented before the object is reclaimed. This ensures that all objects will have a correct reference count during the execution of the program.

To update the reference counts of objects when references are altered, a write barrier is used. The write barrier increments the object receiving the reference and decrements the object losing the reference. Since garbage objects can be detected as soon as their reference counts drop to zero, the collection work is more evenly spread than a traditional tracing collector. This spread of the workload enables reference counting to achieve consistently small pause times.

As mentioned above, reference counting is not *complete*. It requires the assistance of a *cycle detector* to collect all garbage on the heap. It is possible to avoid the need for cycle detection by using weak pointer implementations, however this is suggested to be highly costly [Jones 1996].

2.4.1 Deferred Reference Counting

It is costly for the run-time to execute the write barrier for every reference mutation. To reduce this cost a reference counting algorithm can choose to ignore some sets of frequently mutated references [Deutsch and Bobrow 1976]. References from hardware registers or the stack are examples of frequently mutated references. To prevent incorrect collection of live objects, all objects must be checked for references residing in the stack and registers before they are reclaimed.

Deferred reference counting typically defers adjusting the reference counts in the write barrier. Instead increments and decrements are placed into a buffer and processed together. This approach requires more meta-data space since buffers for the increments and decrements are required. It also produces longer pause times but gains significant increases in throughput [Ungar 1984].

2.4.2 Coalescing Reference Counting

Coalescing reference counting is built on the idea that many updates to reference counts are unnecessary [Levanoni and Petrank 2001]. Between each collection, to maintain a correct reference count for an object, only previous reference and final references to that object need to be considered. The object can receive a decrement for any previous reference that is now missing and an increment for any final reference not previously there. Any intermediate changes in between the collections would cause both an increment and a decrement causing no net change to the reference count. This scenario is illustrated in figure 2.3. The object *o5* initially references object *o1*. Before a collection takes place the reference from *o5* changes to reference *o2*, then *o3* and finally *o4*. The coalescing collector only need remember the initial reference to *o1* and the final destination, *o4*. The objects in between each would receive an increment and a decrement resulting in no net change.

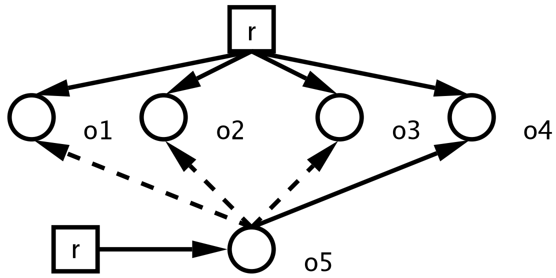


Figure 2.3: An example of several reference mutations.

2.4.3 Retain Events

To prevent deferred reference counting algorithms from collecting objects referenced from the stack or from hardware registers, *retain events* are used. There are three ways to implement retain events: using a zero count table, using temporary increments and using a flag in the object's header.

The zero count table method places all objects with a zero reference count into a table [Deutsch and Bobrow 1976]. Before the objects are reclaimed each object is checked for references in the stack and hardware registers. If such a reference exists the object is left in the zero count table until the next collection, otherwise the object is reclaimed. If the object's reference count increases again between collections it is removed from the zero count table.

The other two methods gather all roots from the stack and the hardware registers and indicate to the reference counter that these objects should be retained [Bacon et al. 2001; Blackburn and McKinley 2003]. This is achieved either by incrementing the reference count and decrementing it in the next collection or by setting a flag in the object's header.

2.4.4 Cyclic Garbage

A problem with reference counting as a collection algorithm is that it is not complete. In some cases garbage objects will not be detected and thus not collected using reference counting alone [McBeth 1963; Deutsch and Bobrow 1976; Lins 1992; DeTreville 1990]. Figure 2.4 describes this situation. When the reference *r1* is removed, both objects in cycle *c1* will have non-zero reference counts and the collector will not reclaim them. To ensure that all garbage objects are collected, a cycle detection algorithm must also be used. The two basic cycle detection algorithms are trial deletion and mark-scan.

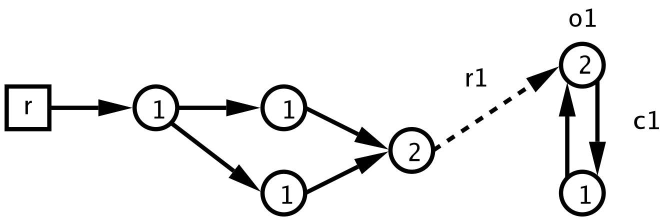


Figure 2.4: A cycle of garbage in a reference counted heap.

2.4.4.1 Trial Deletion

The trial deletion cycle detector operates by considering possible cycle roots. For each possible cycle root, it pretends that the object is dead and then decrements the reference counts of all child objects [Lins 1992; Bacon et al. 2001]. If the object is the root of a cycle of garbage, then this process will kill the child objects which will in turn decrement the parent and kill it. Consider the cycle *c1* in figure 2.5. If the possible cycle root *o1* were to decrement all its child objects, in this case just *o2*, the child object would have a reference count of zero. The child object *o2* would be marked as a possible member of a garbage cycle and its child nodes would be decremented, in this case just *o1*. This decrement would leave both objects with a reference count of zero indicating the two objects were members of a garbage cycle.

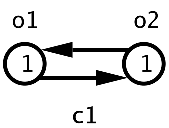


Figure 2.5: A garbage cycle.

Conversely, figure 2.6 describes a cycle *c2* that is not garbage (it is referenced from a root). In this case, if the object *o3* were considered a possible cycle root and its child nodes decremented, *o4* would still have a non-zero reference count. This in turn would prevent *o3* from obtaining a zero reference count. Therefore the cycle remains live. To maintain correct reference counts after a live cycle is traversed, it must be traversed again, restoring any decremented reference counts.

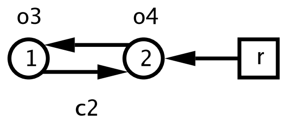


Figure 2.6: A non-garbage cycle.

2.4.4.2 Mark-Scan

An alternative approach to detect cycles is to perform an occasional mark sweep of the entire heap [DeTreville 1990; Levanoni and Petrank 2001]. Any garbage cycles uncollected from reference counting would not be reached in the mark phase and thus could be collected in the sweep phase.

In implementations where a limited number of bits are used to store the reference count, the mark sweep is also used to reset *stuck* reference counts back to their correct values [Jones 1996].

2.5 Summary

In this chapter various approaches to automatic dynamic memory management have been explored and the general problems associated with reference counting have been discussed. In the next chapter, the environment for the remainder of the work in this thesis will be described. Details of the current reference counting implementation will also be described.

Chapter 3

Jikes RVM, JMTk and the Reference Counter

The previous chapter looked at the general problem of memory management and garbage collection. Possible approaches to reference counting were also described.

In this chapter, the software used as a platform for research, the Jikes Research Virtual Machine is described. The Java Memory management Toolkit (JMTk) is also described along with details of its reference counter implementation. Important properties of the platform, such as the approach to compilation and the self-hosting nature of Jikes, are discussed.

3.1 Jikes RVM

The Jikes Research Virtual Machine (RVM) is an open source research Java Virtual Machine (JVM) written in Java [Alpern et al. 1999a; Alpern et al. 1999b; Alpern et al. 2000]. It is designed to be a high performance JVM and is intended to run on high end server machines. To support these goals it incorporates a high performance compiler system and a family of memory managers.

An interesting property of the RVM is that it is self hosting. This means that the RVM not only runs the application program it is executing, but also its own internal components. For example, the RVM's compiler system can recompile code that the RVM is itself using. This approach to design opens up many opportunities for optimisation unavailable in other JVMs. For example the RVM has the ability to inline parts of the supporting Java libraries and the runtime itself, such as the write-barrier and allocation fast-path, into the application code being executed.

While there are other virtual machines which run in the language they were programmed in [Ingalls et al. 1997; Taivalsaari 1998], Jikes RVM is the only such implementation that is both self hosting and aiming to be a high performance VM.

3.1.1 Boot Image

Since the RVM is self hosting, it requires a boot image to commence execution. This boot image is generated by running the RVM on another JVM. The RVM then uses a special boot writer class to write an image of the running RVM into a boot image. This boot image is then reloaded using limited quantities of C and assembly code.

When generated, the classes in the boot image can either be compiled by the baseline compiler or by the optimising compiler. A baseline compiled boot image is faster to generate but runs slower than a boot image generated from the optimising compiler.

3.1.2 Threading and Scheduling

When the RVM loads its boot image, it creates a native process for each processor available in the system. Using these native processes, Java threads are multiplexed using quasi-preemptive scheduling [Alpern et al. 2000].

Again since the RVM is self hosting threads for compilation, garbage collection and other services are scheduled using the same scheduling system that runs the applications threads.

3.1.3 Synchronisation and Locking

The RVM uses a locking mechanism based on thin locks to support Java synchronisation [Alpern et al. 2000]. Each object has information in its header detailing its current locking state. When there is no contention for the lock a thin lock is used. When contention occurs the thin lock is promoted into a thick lock. This approach avoids the need for the heavier thick locks in cases where there is no contention for the object in question.

The RVM also includes other lower level locking mechanisms only available inside the RVM [IBM Research 2003].

3.1.4 Compilers

Unlike other JVMs, Jikes RVM does not use an interpreter loop to run its applications. Instead the RVM takes a compile only approach to bytecode execution. Initially it compiles every executed method with a very fast "baseline" compiler. Later "hot" methods are re-compiled using a more aggressive optimising compiler.

The RVM also takes a lazy approach to compilation; when a class is loaded all methods are left uncompiled until they are called. This approach helps to reduce the initial time cost of compiling all bytecode.

3.1.4.1 Baseline Compiler

The baseline compiler is a rigorous implementation of a Java stack machine [Alpern et al. 1999a]. It is able to quickly compile Java bytecode into native machine code. However, the code produced by the baseline compiler executes much slower than code produced by the optimising compiler. The baseline compiler performs no code inlining optimisations.

3.1.4.2 Dynamic Optimising Compiler

The optimising compiler is more aggressive but also more expensive than the baseline compiler. It aims to generate the best possible code while at the same time remembering that compilation time is an important factor. The optimising compiler is able to produce code that performs at a speed comparable to production JIT compilers [Burke et al. 1999].

Code within the RVM is able to use a compiler pragma to instruct the optimising compiler how certain portions of code should be handled. For example, it is possible to instruct the optimising compiler to always inline a specific method [Blackburn et al. 2003].

3.1.4.3 Adaptive Optimisation System

The adaptive optimisation system controls the two compilers and attempts to make a good cost/performance trade-off. It uses metrics gathered at runtime to recompile "hot" methods using the optimising compiler. It can also recompile optimised methods using difference optimisations to achieve even greater performance [Arnold et al. 2000].

The behaviour of the adaptive optimisation system is non-deterministic which can affect benchmarking [Blackburn et al. 2003].

3.1.5 Object Model

Jikes RVM is implemented with an unusual object model that is inherited from early versions of the Jalapeño JVM. Object fields are laid out backwards in decreasing memory from the object reference. Additionally, the length of arrays are stored before the array reference and the array data starts at the array reference, see figure 3.1.

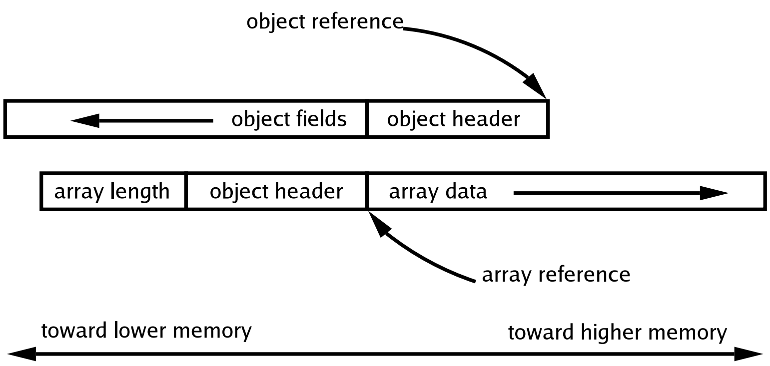


Figure 3.1: The layout of the object model implemented in Jikes RVM and used by JMTk.

For the hardware and operating system Jalapeño was initially implemented on, this layout had several benefits [Alpern et al. 2000]. Firstly, it allows cheap null pointer checks by protecting high memory to trigger a hardware exception. When the program attempts to access any field of a null object (object reference at 0x0), the field would be high memory (below 0xFFFFFFFF), triggering the exception. This approach makes the assumption that the field in the object being referenced will be in a protected region of memory, that is:

$$\textit{fieldoffset} > -(\textit{protectedpages}) \times (\textit{page size})$$

Another consequence is that the RVM cannot safely be loaded into high memory as the page protection system would not work in this case.

The Java language specification ensures that all array accesses are bounds checked, which implies that the array length needs to be read before an access is attempted. Again with the length laid out before the array reference, accessing a null array would trigger an exception.

The other advantage of this object layout is efficient array access. The ith element in the array is i times the element size from the object header reference. This allows fast and cheap access to array elements.

3.1.6 Object Headers

In the RVM all objects and arrays are stored with a header. This header contains information about the type and status of the object or array. The reference counting header also contains an additional word to store the reference count and data used for cycle detection.

3.1.7 Memory Management

The RVM supports a family of memory managers. This includes collectors from the original Jalapeño VM.[1] More importantly the RVM includes the versatile Java Memory management Toolkit (JMTk).

3.2 JMTk

JMTk is a memory management toolkit written in Java. Since Jikes RVM was designed to run on server machines that possibly have more than one CPU, JMTk is designed with concurrent operation in mind. Since Jikes RVM is also designed to be high a performance VM, JMTk is intended to contain high performance algorithms for both collection and allocation [Blackburn et al. 2003].

JMTk provides a testbed where different memory management strategies can be examined and compared. It improves upon previous memory management toolkits by providing both free-list and bump-pointer allocators [Blackburn et al. 2003]. It supports a range of collection algorithms including copying collectors, mark-sweep and reference counting. It also has support for generational collectors.

[1] These collectors are referred to as the Watson collectors, they are now deprecated.

3.2.1 Plans

JMTk contains a set of "plans" for its range of collectors. Each plan specifies how a certain memory management approach will allocate and collect objects. The plan also specifies if a write barrier is required by the collector.

3.2.2 Allocation Spaces

JMTk provides the VM with several different spaces to allocate objects. These spaces allow collectors and allocators to take different action for objects residing in different spaces.

Since Jikes RVM is self-running and requires a boot-image to start running some objects are allocated into the boot space. Objects in the boot space are never allocated at runtime, they are allocated at build time when the boot image is made.

JMTk also provides an immortal space. JMTk will allocate objects that never need to be collected into the immortal space, this removes some load off the collector. These objects are typically internal RVM data structures.

Most JMTk collectors also have a large object space (LOS). The large object space provides a space to deal with objects so large that they would otherwise hinder the normal memory management process. Consider a copying collector; during a collection it would be quite expensive to copy a large object into a new memory region. The LOS allows the collector to avoid this problem. Segregating large objects into a separate space may also help collectors by providing better object locality.

JMTk collectors also have a default space used by which ever collection algorithm is being used. For example the default space for a reference counting collector is the "RefCount" space.

3.2.3 Allocation Locals

For each space in the system, each processor can have a "local". Each processor allocates into and collects its own locals. Locals get their memory from their associated space in large chunks. This prevents the need for fine grain synchronisation during allocation and collection. Instead only coarse gain synchronisation is needed for access among processors.

3.2.4 Double Ended Queues (Deques)

JMTk provides a set of general purpose queues for storing data. These queues, as the name suggests, allow values to be inserted at either end.

These queues are implemented using coarse grained synchronisation. Each thread that accesses a shared queue has its own associated local queue that is not synchronised. When the local queue becomes filled or empties, data is transfered from a global pool. This global pool is synchronised between threads.

3.2.5 Approach to Marking

Marking can be done by changing a bit in the object's header. In the first mark of the graph, an object is marked by setting the mark bit in the object's header. Then instead of unmarking all objects after a mark and sweep, the meaning of the mark bit is flipped. Thus, in the next mark phase an object will be marked by clearing the mark bit.

3.2.6 Acyclic Objects

The compiler assists the reference counter by identifying some acyclic objects. Only classes that only reference scalars and final acyclic classes can be safely treated as acyclic. This is because a subclass of an acyclic object could be cyclic, causing the original analysis to be incorrect. Thus the compiler can only perform limited analysis of which objects are acyclic.

Even with these restrictions Bacon et al. [2001] found that some benchmarks from the SPECJVM98 suite contained high proportions of acyclic objects. In the two ray-tracing benchmarks, ray-trace and mtrt, 90% of objects allocated were acyclic.

The trial deletion cycle detector in JMTk never traverses acyclic objects. This is because an acyclic object cannot be part of a cycle.

3.3 The Reference Counter

The reference counter currently implemented in JMTk is a deferred reference counter.[2] This means that all reference mutations to registers and stack variables are ignored. All other mutations are handled by a write barrier.

3.3.1 Write Barrier

The write barrier in the reference counter handles all non-deferred reference mutations. For each such mutation the corresponding increment and decrement are placed into increment and decrement buffers. These buffered increments and decrements grow in size until a collection is needed either due to object allocation or excessive reference mutation.

3.3.2 Roots

At collection time all roots of the object graph from the previous collection are decremented. All new roots are located and their reference counts incremented. The incremented roots are also placed into a buffer to be decremented in the next collection. This approach avoids the need for a zero count table (ZCT).

[2]Ian Warrington, Honours student at Australian National University, has also implemented a coalescing reference counter.

3.3.3 Increments and Decrements

Increments are processed before decrements. Increments need to be processed first to avoid an object's reference count incorrectly dropping to zero or lower. Any object whose reference count drops (is decremented) to a non-zero value is coloured purple and placed into a buffer. These purple objects are considered possible roots of cyclic garbage and need to be handled by the cycle detector. Any object whose reference count drops to zero is considered garbage and is collected. This involves decrementing all reference objects which may in turn be killed.

3.3.4 Reference Counting Header

The reference counter uses an extra word on top of the standard header. This extra space is used to store cycle detection state and the reference count. Having this much space for the header removes the need to "un-stick" reference counts that have overflowed.

3.3.5 Allocation Strategy

The reference counter allocates objects using a segregated free-list and a large object allocator. The large object allocator rounds the object's size up to the nearest page.

3.3.6 Cycle Detection

The reference counter uses the trial deletion algorithm described by Bacon et al. [2001]. The cycle detector is called at each collection. Instead of performing cycle detection every time it is called, the implementation occasionally filters out any objects in the purple buffer that have either been incremented and are now black or have been decremented to zero. It also filters live purple objects into a "mature" purple buffer to help reduce the load on the cycle detection. The implementation uses time caps at each stage of the algorithm to help prevent large pause times.

3.4 Summary

In this chapter the research platform has been described. The details of the current reference counter implementation have also been documented. In the next chapter the costs of parts of this implementation will be dissected looking for areas that can be improved.

Chapter 4

Costs involved in Deferred Reference Counting

The previous chapters have described the issues surrounding reference counting. Chapter 2 has described some approaches to the components of the reference counter. The current implementation of the reference counter has been described in chapter 3.

This chapter aims to describe tests to dissect the costs in the current reference counter. Justifications for these tests are provided along with expected results.

4.1 Measuring Allocation Cost

As mentioned in chapter 2, the bump-pointer is considered to be the cheapest method of allocation. However, in JMTk it is unclear how the free-list allocator compares from a cost perspective. Measuring and comparing the cost of allocation for the different allocation approaches aims to determine whether the free-list is significantly slower than the bump-pointer.

While it is not possible to replace the free-list in the reference counter with a bump-pointer, if the difference in cost is significant future work could try to improve the free-list performance.

It is expected that free-list allocation will be more costly than allocation using the bump-pointer. There is no expectation of the degree of difference.

4.1.1 Approach

All automatic memory management systems need an allocator. This requirement makes determining the cost of the allocator a difficult task. While it is not possible to run the system without the allocator and determine the system speed-up, it is possible to change the allocation process and measure any performance improvement or degradation.

An allocation only plan[1] was used to measure the performance difference between the various allocator strategies available in JMTk. This plan was chosen to remove the

[1]The "NoGC" plan.

cost of collection from the total cost. While using the same collector for different allocators is possible, there is no collector that is designed to perform this task. Removing the collector from the costs removes the possibility of the collector's design favouring one allocator over another.

4.1.2 Bump-Pointer

The allocation only plan traditionally uses a bump-pointer to allocate objects. To measure the difference between allocators, this bump-pointer will be replaced by the other allocators and differences measured. During testing, a defect was uncovered in the bump-pointer implementation, thus the optimised version of the bump-pointer was also compared.

4.1.3 Segregated Free-list

The bump-pointer in the plan was replaced by the segregated free-list allocator used in the reference counter. This allocator does not handle allocation of large objects[2] so the reference counter's large object allocator was added.

4.1.4 Bump-Pointer With Large Object Allocator

The bump-pointer was also tested with the large object allocator from the reference counter and the allocation only free-list plan. This aims to uncover the benefits of adding the large object allocator.

4.1.5 Allocation Benchmark

While all benchmarks need to allocate objects to function, many are not aimed at testing allocation alone. Instead the benchmarks allocate some data and then perform a task involving the data. The execution of the task causes other factors, such as locality, to affect overall performance. To help remove factors like locality from affecting the results, an additional custom benchmark, *gc random alloc*, is used.

The *gc random alloc* benchmark initially allocates a set of data to fill part of the heap, then allocates sets of garbage objects. The objects allocated are all arrays of random length.[3] This approach is aimed at testing the various size buckets in the free-list as well as any boundary issues in the bump-pointer. Issues like locality are avoided since the benchmark does not actually operate on the objects it allocates. Any locality issues that arise are purely allocation related.

4.1.6 Compiler Settings

When factoring out the cost of the allocator the behaviour of the adaptive optimising system (AOS) and the optimising (OPT) compiler will have a large impact on the

[2] Objects greater than 8Kb.

[3] The random sizes are always generated using the same seed, however they may vary between JVMs

final results. The runtime must be allowed to perform some optimisations, such as inlining allocation fast-paths. However, running the system in without a collector also prevents the possibility of running without the AOS enabled. This is because OPT compiling every method uses too much memory.

4.2 Approaches for Processing Increments

The reference counter must increment an object's reference count for every new reference that the object receives. The reference counter records these changes to a reference by using a write barrier.

Increments must be processed before decrements to avoid incorrectly lowering an objects reference count to zero. Aside from this constraint the increments can be processed at any point before the decrements. This includes the possibility of processing the increments directly in the write barrier.

Processing the increments in the write barrier should move some of the collector's work load out of collection time and into mutator time. This should cause shorter pause times, a decrease in collection time, and an increase in mutator time.

This test aims to quantify the difference between processing the increments in the write barrier and buffering increments to be processed later.

4.2.1 Buffering the Increments

As mentioned in chapter 3, the current reference counter implementation buffers increments in the write barrier to be processed at collection time. This approach is thought to take advantage of locality benefits. It is hoped that mutations will mostly be to the working set, and that this set is concentrated in one region.

This method requires more meta-data to store the increments until collection time. This extra meta-data would cause collections to occur more frequently. Storing the increments in a buffer also incurs the cost of the push and pop associated with the buffer.

4.2.2 Incrementing in the Write Barrier

It is possible to avoid the cost of managing the increment buffer by processing the increments directly in the write barrier.

It also aims to achieve a locality gain as it is expected that in many cases an incremented object will belong to the mutator's workset helping avoid causing excessive page faults.

It is expected that fewer collections will be required because less meta-data will be used. However, having fewer collections is not expected to reduce total time significantly as the garbage collection workload is more closely related to the number of mutations than the state of the heap.

4.2.3 The Write Barrier

The write barrier for the reference counter is already quite large. Processing the increments in the write barrier has the potential to increase its size and as such affect code size and total running time. However, by selectively inlining only parts of the write barrier the impact of its size is reduced. This results in less load being placed on the compiler and code generated is smaller [Blackburn and McKinley 2002].

4.2.4 Approach

The benefit of both approaches will be compared by considering total system running time using the two configurations. Mutator time will be compared to see if some of the workload is displaced. The number of collections will be compared to highlight the meta-data impact of buffering all increments. Average and maximum pause times will also be considered to see if processing increments in the write barrier cause workload to become more distributed.

4.3 Write Barrier Cost

The cost of using a write barrier is traditionally thought to be non-trivial [Wilson 1992; Jones 1996]. This is because the write barrier for reference counting is quite large and must be executed for every non-deferred pointer mutation. Also, the write barrier in the reference counter is *fixed* (non conditional) and thus always executes an expensive *slow path*.

While there has been work done to examine the cost of conditional write barriers [Zorn 1990; Blackburn and McKinley 2002], those used in incremental and generational collectors, little research indicates the overhead of a reference counting write barrier.

This test seeks to determine the overhead of the reference counting write barrier used in JMTk. It also aims to determine how dependant the write barrier cost is on the system architecture.

4.3.1 Approach

The main problem with factoring out the write barrier is that it is an integral part of the reference counter and the reference counter will not operate with it removed. Also, the write barrier is executed at many points throughout the running of the VM and the program being executed. Unlike other components of the reference counter it is not possible to measure start and end times for each execution of the write barrier since it is executed too frequently and the timing operation may be more expensive than the write barrier. Thus to determine the cost of the write barrier it will be placed into a collector that does not use a write barrier and the overhead will be measured. A semi-space collector is used for this test since it is simple[4] and does not require a

[4]This will help reduce noise in the results.

write barrier.

4.3.2 Meta-Data

The write barrier used in the reference counter catches all non-deferred pointer mutations and places a reference to the involved objects into the increment and decrement buffers. When this meta-data grows too large, the reference counter processes these buffers.

To fully simulate the cost of the write barrier, the writes to the buffers must be made as they may cause new pages to be allocated as meta-data (a non-trivial cost if frequent enough). When placed into a semi-space collector this will uncover a new problem. Without processing, the meta-data will grow until there is no free space on the heap.

There are several solutions to this problem. The increment and decrement buffers can be periodically emptied, however this will incur additional costs. The increment and decrement buffers could be left unprocessed and the collector run in a large heap. Due to memory constraints this approach is not possible. Finally, the buffers can be modified to overwrite the same entry every time a push occurs, however this avoids the need to fetch extra meta-data pages. The last method will be used in testing since it avoids extra processing and is able to run within the memory constraints of the testing system.

4.3.3 Inlining the Write Barrier

As mentioned above, the amount of the write barrier that is inlined can affect the cost of both running time and compilation. Thus the write barrier will be tested completely out-of-line and with set portions inlined.

4.4 Processing Decrements, Roots and Cycle Detection

The reference counter must process decrements and perform cycle detection to reclaim garbage. Since the reference counter in JMTk uses temporary increments to ensure objects referenced from outside the heap are not collected, roots must also be processed at each collection.

Processing decrements and roots and collecting cycles happens once each collection at most. This property makes these components of the reference counter easy to measure using standard timers. Thus the costs of each item of work will be measured using the reference counter's built in statistics.

The number of mutations is fixed across heap sizes, thus it is expected that cycle detection and processing the roots will contribute to a larger percentage of the cost in smaller heap sizes.

This test aims to quantify the costs involved in processing the decrements and roots and also the cost of cycle detection.

4.5 Summary

In this chapter tests to factor out some areas of cost in the reference counter have been described. Initial results show that cycle detection is a key area of cost in the reference counter. In the next chapter a new cycle detector implementation for JMTk will be described.

Chapter 5

Mark-Scan Cycle Detection

The previous chapter looked at the cost of the reference counter's components. Initial results indicated that cycle detection was a major cost in the reference counter. Currently, the only cycle detection algorithm implemented in JMTk is trial deletion. This means there is no alternate cycle detector implementation against which to compare the performance of trial deletion.

This chapter describes the algorithm for an alternative approach to cycle detection. Two possible variants are described.

5.1 Properties of the Cycle Detector

The cycle detector must obey certain properties to function properly. These are described below.

5.1.1 Safety

During the cycle detection process, it is important that the cycle detector only references areas of memory that contain object headers for live or unclaimed objects. This property will be referred to as the *safety* property.

The safety property implies that the cycle detector will not end up accessing random memory regions when it attempts to traverse the object graph. If the cycle detector did not have this property, the virtual machine (VM) would enter an undefined state.[1]

For example, if the VM obtained a reference to the fields of an object, instead of an object header, an incorrect object type could be looked up, see figure 5.1. The VM would be tricked into thinking it was dealing with a different object and values on the heap would be treated as references to objects. When traversal of these references was attempted, access to "random" memory regions would result.

It is acceptable for the cycle detector to access a reference to a dead object provided it has not been collected and its space re-allocated or cleared. This is because the object header (and thus the object type) along with any references will remain on the heap

[1]Under Linux, most likely a segmentation fault would be generated, crashing the VM.

```
COLLECT-CYCLES()
  // Check if cycle detection is required
  if shouldCollectCycles()
    then
        mark()
        processPurples()
        scan()
        processKillQueue()

    else if shouldFilterPurple()
          then
              filterPurple()
```

Algorithm 5.1: Top level view of the mark scan cycle detection algorithm.

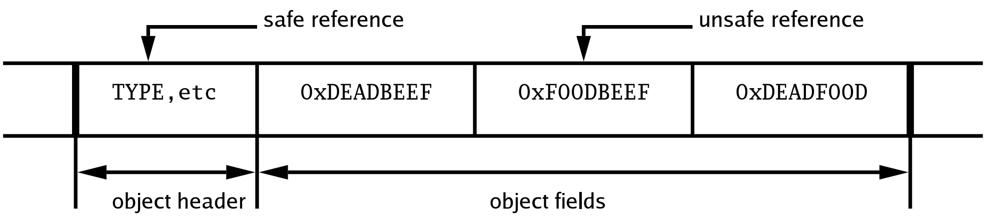


Figure 5.1: Example of a safe and an unsafe object reference.

until reallocation. To help ensure dead objects are not reallocated while the cycle detector is using them, flags are set in the object header indicating the the object is "in use". Since the normal collection process will retain flagged objects, it is up to the cycle detector to cleanup any objects that become (or already are[2]) garbage.

5.1.2 Completeness

The cycle detector must ensure that all cycles are eventually detected and reclaimed, this property will be referred to as *completeness*. If the cycle detector did not satisfy this property, the VM would leak memory and would eventually run out of memory.

Further, it implies that every possible cycle root considered is correctly identified the first time. This condition arises because a cycle with no remaining references will not receive any further pointer mutations due to the nature of garbage objects.

[2] Flagged objects may die before cycle detection starts.

```
MARK()
 // Add all roots into the marking queue
 markQueue ← roots

 while ¬markQueue.isEmpty()
 do
     object ← markQueue.pop()

     // Only traverse unmarked subgraphs
     if object.testAndMark()
       then
            for child in object.children()
            do
                markQueue.push(child)
```

Algorithm 5.2: Mark the graph of live objects.

5.1.3 Correctness

The cycle detector must only collect objects that are actually garbage; this property is the *correctness* property. If the cycle detector was not correct and reclaimed arbitrary objects from the program running on the VM, spurious `NullPointerException` or `ClassCastExceptions` could be thrown inside the program. Since the VM also performs garbage collection on its own data structures, core components of the VM such as compiled methods, could be reclaimed therefore causing the VM to crash.

5.2 Non-concurrent Cycle Detector

In the non-concurrent case the cycle detector runs in a stop-the-world setting. A top level overview is described in algorithm 5.1. The cycle detector performs a mark of all live objects in the heap starting from the roots, see algorithm 5.2. Since the garbage collector has stopped all other threads from running, no special precautions need to be taken during marking. It then examines all possible roots of cyclic garbage obtained after the previous collection, see algorithm 5.3. If any of these objects are unmarked it cannot be attached to the graph of live objects and is therefore a root of a garbage cycle, see figure 5.2.

This more conservative approach to scanning should be more efficient for the reference counter in JMTk. This is because in JMTk, there is no need to scan the entire heap restoring stuck reference counts. Instead, only possible cycles of garbage and interconnecting garbage need to be considered.

```
PROCESS-PURPLES()
while ¬purpleQueue.isEmpty()
do
    object ← purpleQueue.pop()

    // Unmarked purple objects are roots of cyclic garbage
    if ¬object.isMarked()
      then
            // Colour the object white to indicate it is a member of a garbage cycle
            object.colourWhite()
            // Queue the object to be reclaimed later
            killQueue.push(object)
            for child in object.children()
            do
                scanQueue.push(child)
```

Algorithm 5.3: Process purple objects.

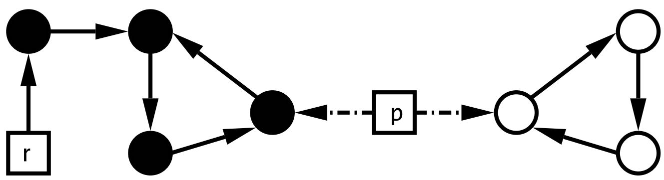


Figure 5.2: Example of a marked live cycle (left) and unmarked dead cycle of garbage (right).

5.2.1 Allocation

It is possible for a program to allocate a cycle that is never connected to the graph of live objects,[3] consider figure 5.3. In the depicted example it is possible for the created cycle to be considered live by the cycle detector if the objects allocated have the opposite mark state to the graph of live objects. In this case, the cycle detector would consider the cycle to be live after the next mark and leave it uncollected. To avoid this possibility, all objects must be marked with the current mark state after allocation. Then when the next mark occurs, any garbage cycle will always be coloured opposite to the graph of live objects.

[3]Except being stored in the stack or hardware registers.

```
SCAN()
while ¬scanQueue.isEmpty()
do
    object ← scanQueue.pop()

    // Unmarked purple are roots of cyclic garbage
    if object.isMarked()
      then
          // Decrement the object's reference count (a reference from cyclic garbage).
          decrementQueue.push(object)

      else  if ¬object.isWhite()
              then
                  // Colour the object as member of a garbage cycle
                  object.colourWhite()
                  // Queue the object to be reclaimed later
                  killQueue.push(object)
                  for child in object.children()
                  do
                      scanQueue.push(child)
```

Algorithm 5.4: Process the scan queue.

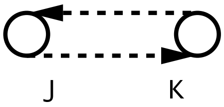
5.2.2 Filtering Possible Cycle Roots

It is possible that an object becomes a candidate cycle root, but before cycle detection can run it becomes garbage. To help reduce the load on the cycle detector, the buffer containing the possible cycle roots is filtered. The filtering process removes any dead objects from the buffer and places them onto a kill queue. If the size of this buffer is small enough after the filtering (if enough memory has been reclaimed), a full mark and scan of the heap can be postponed until a later collection.

5.2.3 Reclaiming Garbage Cycles

After a root of a garbage cycle has been identified, the algorithm traverses the garbage cycle and flags any unmarked objects as cyclic garbage, see algorithm 5.4. Flagging objects as cyclic garbage prevents them being traversed more than once (as will otherwise happen with cyclic garbage). The flagged objects are also added to a kill buffer to be reclaimed at the end of cycle detection.

The kill buffer is required to prevent reclaiming objects that may still be traversed by the cycle detector as this would violate the safety property, consider cycle *c1* in figure 5.4. Without a kill buffer, *o1* is killed and its reference enumerated. This in turn kills *o2* and its reference is enumerated. However *o1* has already been reclaimed and




```
...
new J();
...
```

```
class J {
  K myK;
  public J() {
    myK = new K();
  }
}
```

```
class K {
  J myJ;
  public K(J j) {
    myJ = j;
  }
}
```

Figure 5.3: A cycle that is created disconnected from the graph of live objects.

accessing it would violate the safety property.

If a marked object is encountered it is added to the decrement buffer but not traversed. Adding these objects to the decrement buffer ensures that all reference counts will remain consistent after cycle detection is complete, consider figure 5.5. When the cycle is collected the marked object's reference count should be decremented for it to be correct.

The cycle detector must avoid performing such a decrement operation on a cycle of garbage, consider figure 5.4 again. If the cycle *c2* was collected and *o2* received a decrement, the safety property would be violated when the decrement was processed. This is because the cycle *c1* would be reclaimed before the decrement to *o2* occurred.

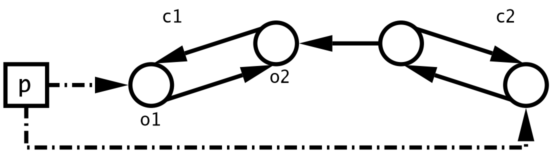


Figure 5.4: Configuration that could cause problems during collection of the cycle.

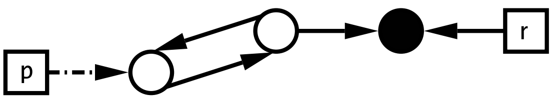


Figure 5.5: Situation where reference counts need to be updated.

5.2.4 Acyclic Objects

Acyclic objects can cause the mark scan cycle detector problems if dealt with in the same manner as trial deletion. Consider the situation in figure 5.6. Using trial deletion, the cycle *c1* could be reclaimed but the cycle *c2* could not (*o1* is keeping it alive). Reclaiming cycle *c1* would cause the acyclic object *o1* to be decremented. This would cause the cycle *c2* to be reconsidered by trial deletion and it would be reclaimed at a later collection.

Using a mark scan cycle detector and taking the same approach to acyclic objects, a different situation could arise. If the two cycles *c1* and *c2* were collected, *o1* would be placed on a decrement buffer. Then, when the buffer was processed, the safety property would be violated; *o1*'s reference to *c2* would no longer be valid.

To prevent this problem, the mark scan cycle detector must treat acyclic objects the same way as any other unmarked objects. This unfortunately means that the mark scan cycle detector cannot take advantage of acyclic objects.

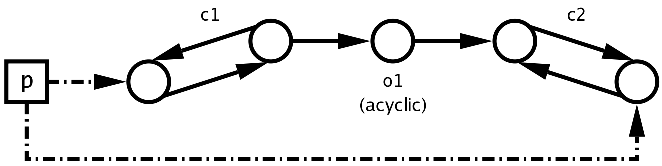


Figure 5.6: A possible problem caused by an acyclic object.

5.2.5 Properties

The algorithm ensures the correctness property by marking every live object. With every live object marked it is not possible to incorrectly collect a live part of the object graph. Since the algorithm runs in a stop-the-world setting no special precautions need to be taken while marking the graph.

The algorithm satisfies the completeness property by considering every possible cycle root before any mutator activity resumes. This means that any cycle that is unmarked is garbage and any marked cycle is not.

The safety property is enforced by using the kill buffer as mentioned above.

5.3 Parallel Non-concurrent Cycle Detector

The parallel non-concurrent cycle detector operates in a stop-the-world setting but collects on more than one processor. It uses the same algorithm as the non-parallel cycle detector with one small change. All CPUs rendezvous in between marking, scanning and processing the kill queue. This ensures that no CPU is still processing its marking queue and thus the entire heap is marked before scanning takes place.

5.4 Concurrent Cycle Detector

In the concurrent version of the algorithm, the heap is marked in a separate thread running concurrently with the mutator thread. The scanning of the cycles is still performed in a stop-the-world setting, and at this point the marking thread is also suspended.

This approach aims for smaller pause times compared with the non-concurrent version. This is because the marking work is performed concurrent to the mutator leaving only the scanning phase during collection time.

5.4.1 Mutation During Marking

The first problem that arises from marking the heap in a concurrent setting is mutations to the object graph. Without proper precautions, a mutation to the graph could trick the marking thread and prevent it from marking a subtree of the object graph, see figure 5.7. The reference from the marked object is added after its references are enumerated. Later, the reference from the unmarked object is removed before it is marked. This can leave the subgraph in an incorrectly unmarked state.

To avoid this problem all increments are recorded in the write barrier and these objects are added back onto the marking queue [Dijkstra et al. 1978].

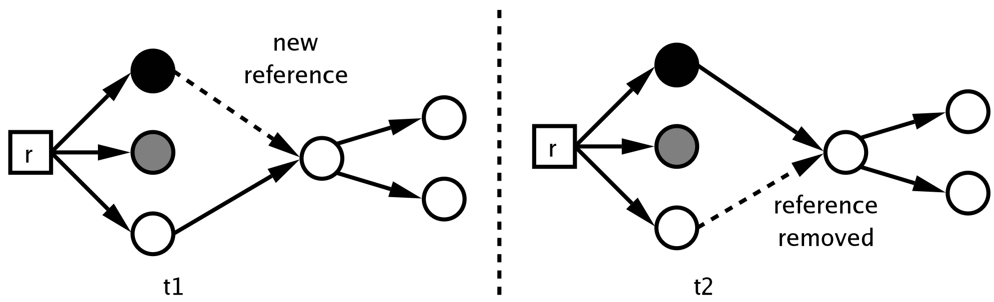


Figure 5.7: Example of how the mutator can cause problems when marking.

Another problem that arises due to marking concurrently is the issue of safety. Without precautions it is possible for the marking thread to perform an unsafe access, consider figure 5.8. In this case some objects could be collected while the marking thread is still accessing them. To prevent this situation a bit is set in the object's header to indicate that it is in the marking queue and should not be collected. As with the possible cycle roots, the cycle detector is then responsible for collecting any objects buffered in the mark queue which subsequently die.

5.4.2 Mutation After Marking

It is possible for the marking thread to empty the marking queue before the cycle detector is ready to perform the scanning. When this happens it is crucial to ensure that any mutations which occur in this time are processed. Without processing, subgraphs of objects in the queue could remain unmarked incorrectly.

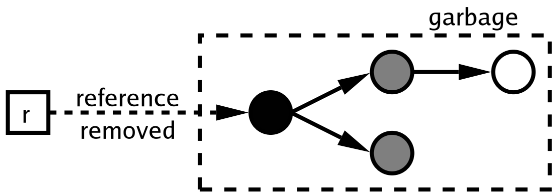


Figure 5.8: Potentially unsafe pointer access.

5.4.3 Roots

While the marking thread is marking the heap, it is important to add any new roots onto the marking queue. This ensures that when scanning occurs the entire graph of live objects is marked.

5.4.4 Allocation

The problem that the non-concurrent cycle detector has with allocation of objects does not apply to the concurrent version. This is because any new pointer that could form a cycle will be caught in the write barrier and the cycle will be marked to the correct state. Also, all new objects receive an initial reference count of 1 and a buffered decrement. This decrement would place the cycle into the buffer of possible cycle roots where it would be processed as usual.

5.4.5 Detecting Cycles

Cycles are detected in the same way as the non-concurrent cycle detector.

5.4.6 Filtering Possible Cycle Roots

Possible cycle roots are filtered in the same way as the non-concurrent cycle detector.

5.4.7 Acyclic Objects

The concurrent mark scan cycle detector suffers the same problems with acyclic objects as the non-concurrent version.

5.4.8 Reclaiming Garbage Cycles

With the marking concurrent to the scanning process, it is possible to mark a cycle and then for that cycle to become a possible root of cyclic garbage. To correctly detect if the possible root is the root of a garbage cycle, the cycle must be retained until after the next mark, see figure 5.9. In this example, a partial mark of the heap has occurred and before scanning takes place the cycle becomes garbage. The cycle detector must retain the cycle until the epoch when the graph is "unmarked" to determine if the cycle is actually garbage.

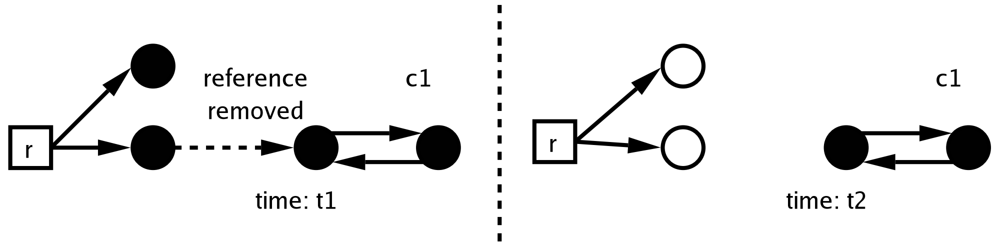


Figure 5.9: Cycle c1 is marked in time t1 and must be retained until "un-marking" is complete in time t2..

Conversely, if a cycle becomes garbage and it has not already been marked, it can be processed in the next scanning phase. Due to the nature of garbage, if the cycle is unmarked, it is definitely garbage, figure 5.10 illustrates the two different scenarios.

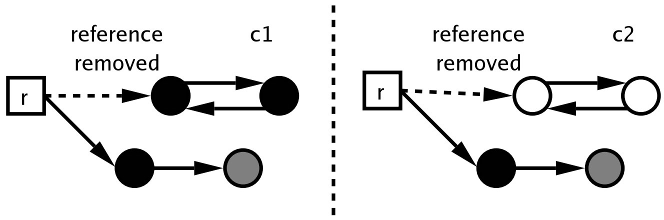


Figure 5.10: Cycle c1 will be processed after the current mark, cycle c2 needs to be retained till after the next mark.

5.5 Comparison to Trial Deletion

The main limitation of the mark scan cycle detector when compared to trial deletion is that it must perform a mark of the whole graph of live objects. Conversely, trial deletion only ever considers a local sub-graph when collecting cycles. The main benefit of the mark scan cycle detector is that after a mark is completed it only needs to traverse a cyclic object once to collect it. Trial deletion on the other hand must traverse a garbage cycle three times before it can be reclaimed. The pathological worst case for trial deletion is the best case for the mark scan cycle detector. Consider the case where most of the heap died as a large cycle. Trial deletion would need to traverse the whole cyclic structure repeatedly. However, the mark scan cycle detector would have very little to mark and only need to traverse the cycle once to collect it.

Another weakness of the mark scan cycle detector is that it does not make use of information regarding acyclic objects. However, acyclic objects can be problematic for trial deletion as well, consider figure 5.6. As mentioned in section 5.2.4, the cycle *c2* would be retained by the reference from *o1*. Instead of saving work, taking "advantage" of the acyclic object *o1* will cause the cycle *c2* to be traversed by trial deletion on two separate occasions.

5.6 Summary

This chapter has described an algorithm for a new style of mark scan cycle detection. Two possible implementations of this algorithm have also been described.

In the next chapter the performance of the cycle detectors will be compared against the original trial deletion cycle detector.

Chapter 6

Results

Chapter 4 described tests for factoring out cost components of the reference counter. Tests for comparing approaches to reference counter implementation were also described. Chapter 5 described two implementations of the mark scan cycle detector, an alternative to trial deletion.

In this chapter the results from the tests and the performance of the cycle detectors will be presented and briefly analysed. Only representative or interesting cases are displayed and discussed for brevity; complete results are available in appendix B.

6.1 Approach to Benchmarking

Each garbage collection algorithm is compiled using a "FastAdapative" build. This build configuration turns off assertion checking, optimises the boot image, and enables the adaptive optimising system (AOS). The AOS is chosen in perference to using the optimising (OPT) compiler exclusively since it provides the most realistic runtime setting.

Each benchmark was run two times in the same instance of the RVM. This allows the second run to be free of the cost of OPT compiling hot methods. This is referred to as a benchmark run.

Each benchmark run was executed five times and the best result is taken. This result was considered relatively free of system disturbances and the best result from the natural variation in the AOS.

The primary i686 benchmarking platform consisted of a single Intel Pentium 4 2.4GHz "C" with hyper-threading enabled. The platform had 1GB of primary memory, 512KB of L2 cache and 8KB of L1 cache. This hardware was running the Linux 2.4.20-19.9smp kernel that is packaged for Redhat 9.0. Even though the platform was running a SMP kernel, the RVM was instructed to use only one CPU.[1] All results were generated from this platform unless stated otherwise.

The secondary i686 benchmarking plaform was an AMD 1GHz Duron. The platform had 740MB of primary memory, 64KB L2 cache and 128KB L1 cache. The hardware was running the Linux 2.4.20-8 kernel that is packaged for Redhat 9.0. This

[1] Technically there is only one CPU, however a virtual processor is presented by the kernel.

platform is only used where indicated.

The Power-PC platform consisted of a single G3 PPC CPU running at 400Mhz. The platform had 256Mb primary memory, 1MB of L2 cache and 64KB of L1 cache. This hardware was running the Linux 2.4.19-4a kernel that is packaged for Yellow Dog 2.3.

These tests were run against Jikes RVM release 2.3.0 unless stated otherwise.

6.2 Reference Counting Costs

6.2.1 Allocation

The results for the allocation tests are shown in figure 6.1. Locality seems to play a large part in the total runtime, as there is no distinguishable trend in the data. The allocation only *gc random alloc* benchmark seems to support this claim with the optimised bump-pointer obtaining the best performance. The bump-pointer is 8% faster than the free-list in the allocation only benchmark.

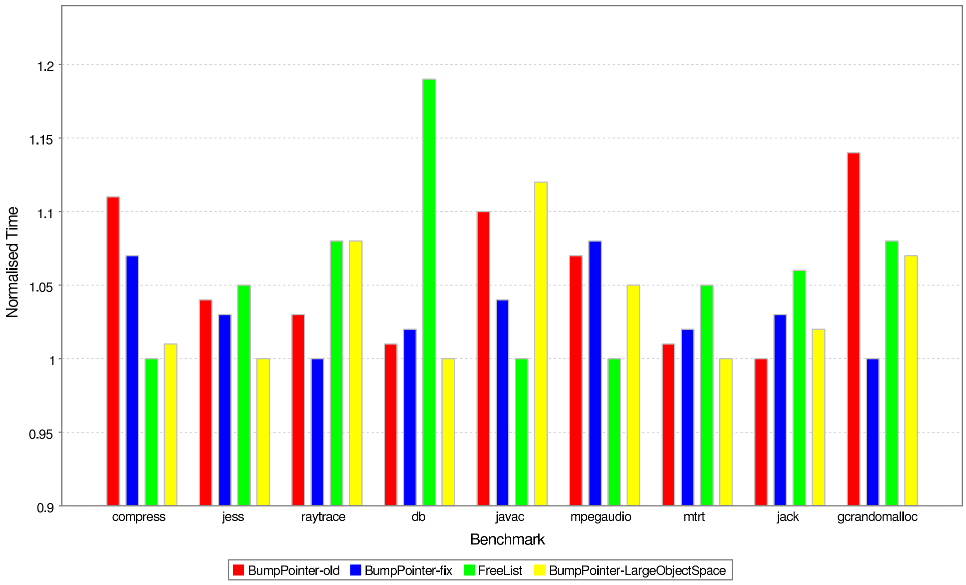


Figure 6.1: Allocation performance across benchmarks.

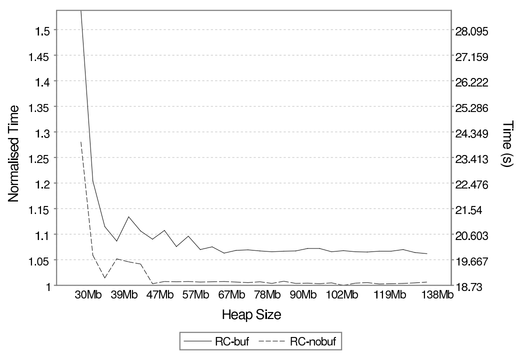
6.2.2 Processing Increments

The results for the two approaches to processing increments are shown in figure 6.2. In both examples processing the increments in the write barrier reduces total running time by around 5%. In the best case, processing increments in the write barrier was 13% faster than buffering them until collection time.

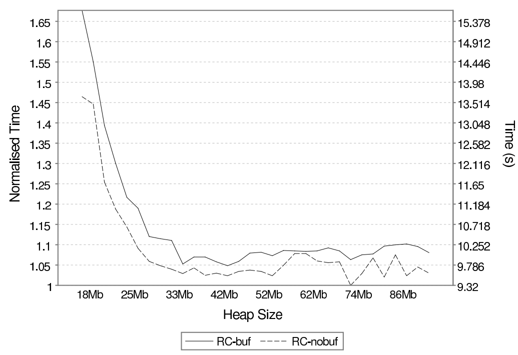
The mutator time for the two approaches is shown in figure 6.3. In 6.3(b) it is clear that mutator time is reduced by 4% when processing increments in the write barrier. In the best case, processing increments in the write barrier was 10% faster.

As expected, processing the increments in the write barrier triggers fewer garbage collections, this is shown in figure 6.4. Processing the increments in the write barrier requires fewer collections since it uses less meta data.

There was little difference in pause times for the two approaches. However, there were two exceptions to this: the *javac* benchmark and the *db* benchmark. The *javac* benchmark showed higher average pause times when processing the increments in the write barrier, this is shown in figure 6.5(a). The *db* benchmark showed lower maximum pause times when processing the increments in the write barrier, this is shown in figure 6.5(b).



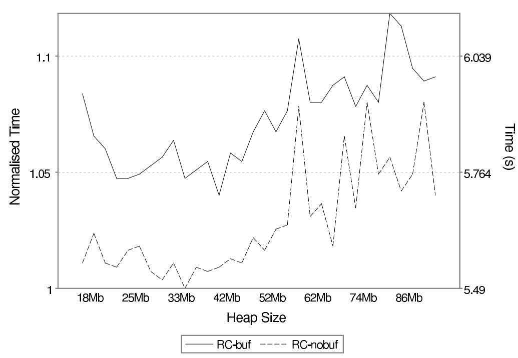
(a) _209_db



(b) _202_jess

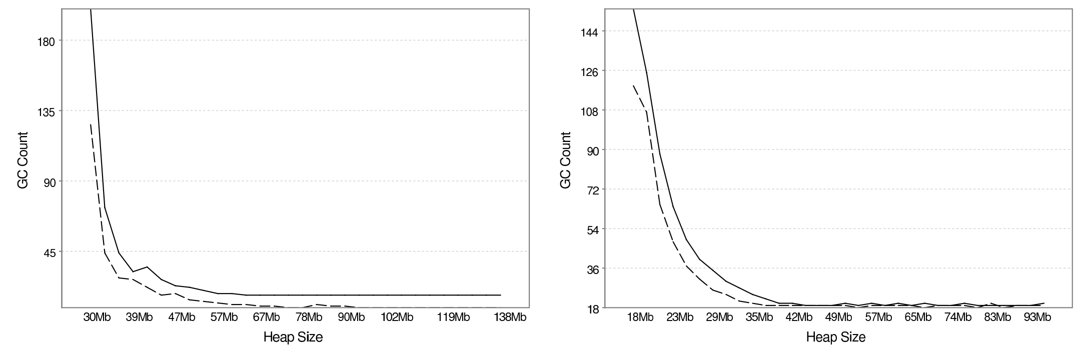
Figure 6.2: Total running time for different increment processing approaches.

(a) _209_db



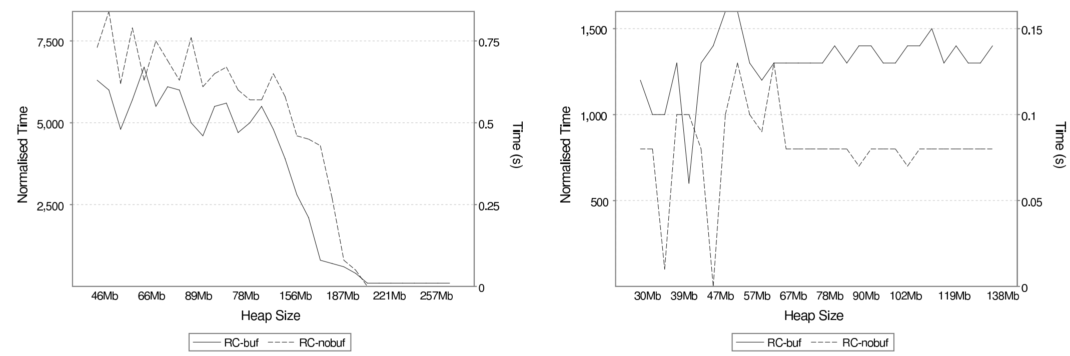
(b) _202_jess

Figure 6.3: Mutator time for different increment processing approaches.



(a) _209_db (b) _202_jess

Figure 6.4: Garbage collection count for different increment processing approaches.



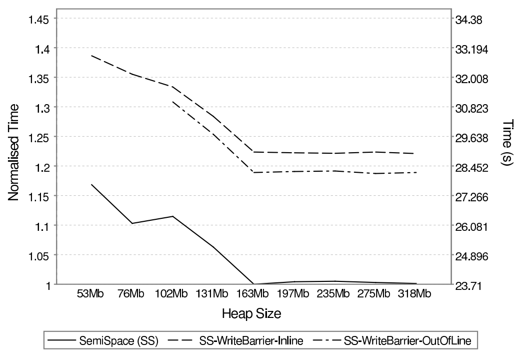
(a) _213_javac (Average pause times) (b) _209_db (Maximum pause times)

Figure 6.5: Pause times for different increment processing approaches.

6.2.3 Write Barrier

The results for the write barrier tests are shown in figure 6.6. While the cost of the write barrier is smaller on the PPC architecture, it still represents approximately 10% overhead. On the i686 architecture the write barrier costs around 20%. As expected this extra overhead is displayed in longer mutator time, see figure 6.7.

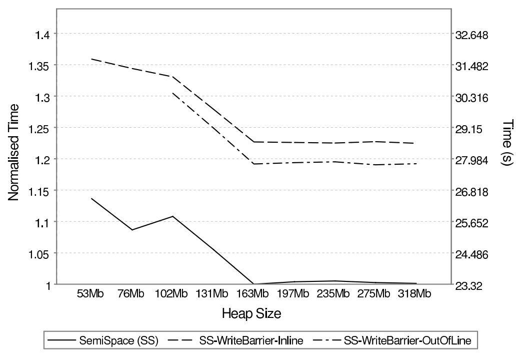
One benchmark that produced an interesting result was *jess*, see figure 6.8. Instead of the write barrier adding 10% overhead for the PPC architecture, as seen on all other benchmarks, in *jess* the write barrier cost 20%. This result is mirrored on the i686 architecture where the added overhead is 50%, more than double the expected 20% overhead.



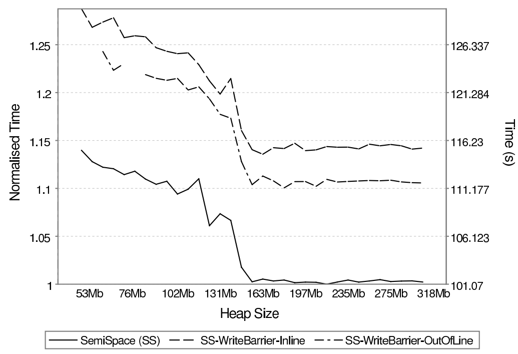
(a) i686 architecture

(b) PPC architecture

Figure 6.6: Total running time for _209_db.



(a) i686 architecture



(b) PPC architecture

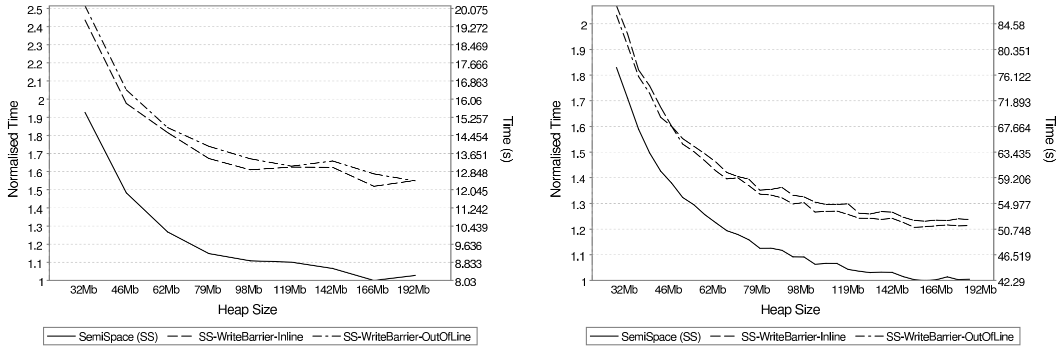
Figure 6.7: Mutator running time for _209_db.

6.2.4 Roots, Decrement Buffers and Cycle Detection

The cost of the roots, decrements and cycle detection is shown in figure 6.9. These results are not produced using the Jikes RVM 2.3.0 release. Instead a recent development build was used. In this development build, increments are processed in the write barrier instead of being buffered.

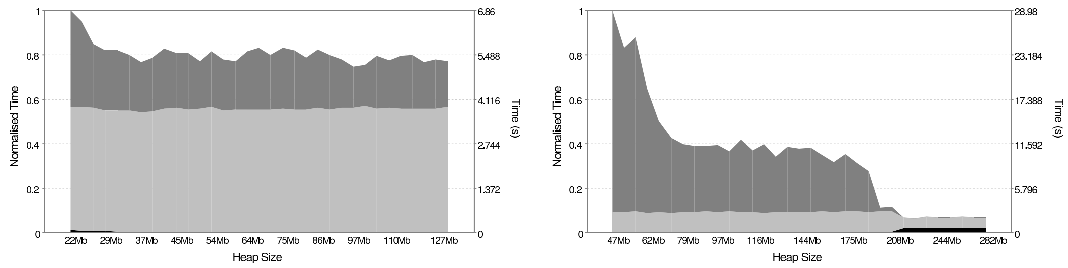
The cost of processing the roots is tiny compared with the cost of cycle detection and processing decrements. In most benchmarks cycle detection is a major cost especially at small heap sizes. In *jess*, cycle detection is a major cost at all heap sizes.

As expected the cost of processing decrements is fairly constant across heap sizes.



(a) i686 architecture (b) PPC architecture

Figure 6.8: Total running time for _202_jess.



(a) _202_jess (b) _213_javac

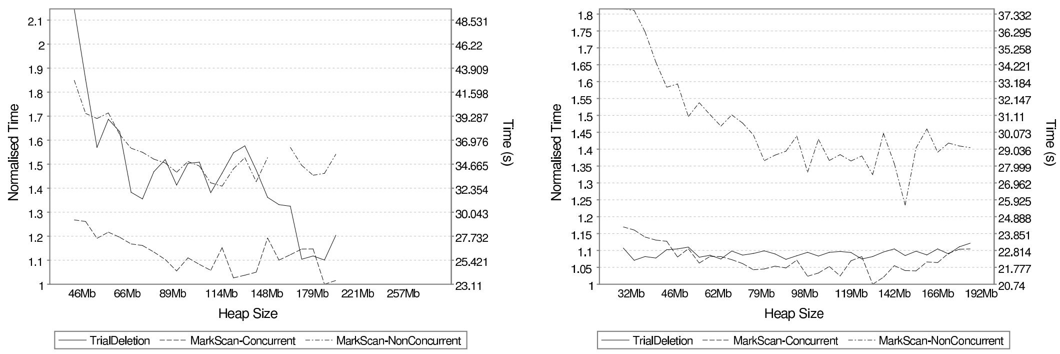
Figure 6.9: Breakdown of reference counter time costs.

6.3 Cycle Detectors

The results for cycle detection throughput are shown in figure 6.10. The concurrent mark scan cycle detector seems to have the best throughput for collecting cycles. On the *javac* benchmark it is able to outperform trial deletion by more than 25%. On the other hand, the non-concurrent mark scan cycle detector performs poorly in throughput.

The results for cycle detection latency are shown in figure 6.11. Both of the mark scan cycle detectors perform poorly in pause times. Even so, the cycle detector workload causes the non-concurrent mark scan cycle detector and trial deletion to obtain longer average pause times for the *javac* benchmark.

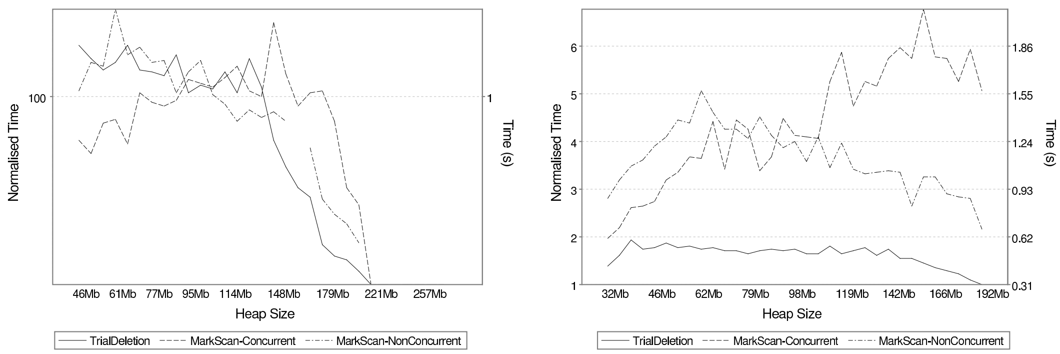
These results were run on the secondary i686 platform.



(a) _213_javac

(b) _202_jess

Figure 6.10: Total times for cycle detectors.



(a) _213_javac

(b) _202_jess

Figure 6.11: Average pause times for cycle detectors.

6.4 Summary

In this chapter results for the reference counter cost analysis were presented. Results from the comparison between trial deletion and the two new cycle detector implementations was also presented.

In the next chapter these results will be analysed in more detail and their implications discussed.

Chapter 7

Discussion

The previous chapter presented results from the tests performed and comparisons of approaches to reference counter implementation. Results from the two new cycle detector implementations were also presented. In both cases a brief analysis was given.

This chapter looks at the results in more detail and discusses their implications.

7.1 Allocation

7.1.1 Locality

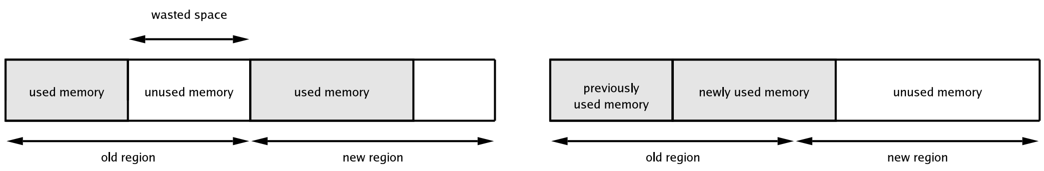
Surprisingly, the free-list outperforms the bump-pointer on some benchmarks. The main reason for these results appears to be locality. This is because the free-list and bump-pointer allocators order objects differently. The free-list groups similar sized objects together, giving a locality win on some benchmarks.

The bump-pointer with a large object space performs especially well on the *compress* benchmark (a benchmark that allocates and uses many large objects). Again, the benefit seems to arise from better locality; the large objects allocated by the *compress* benchmark spread the working set when they are allocated into the same space. With the free-list and bump-pointer with the large object space, the working set has better locality leading to better performance.

The results from the *gc random alloc* benchmark also support locality arguments. This benchmark only allocates and does not perform any work on its allocated objects. This makes it a better indication of allocation performance than other benchmarks. For this benchmark, the fixed bump-pointer outperformed all other allocators.

7.1.2 Bump-Pointer Fix

The original bump-pointer had a defect where the heap would become fragmented. When a new chunk of memory was acquired for allocation, the bump-pointer would always be updated to the start of the new region. In the case where the new memory region is contiguous with the old, the bump-pointer does not need to be moved, see figure 7.1. The optimised bump-pointer has improved performance since the slow allocation path needs to be executed less frequently.



(a) Original bump-pointer (b) Optimised bump-pointer

Figure 7.1: Bump-pointer behaviour during allocation.

This defect is especially highlighted on the *compress* benchmark. As mentioned above, the *compress* benchmark allocates large objects which cause the original bump-pointer to waste a large amount of space and call the more expensive slow path more frequently.

7.1.3 Summary of Allocation Findings

The bump-pointer is 8% faster than the segregated free-list in raw allocation. However, this difference in performance can be outweighed by locality benefits as seen in the results.

The use of a large object space helps to prevent large objects from dispersing the working set. This results in better performance due to a more concentrated working set and thus better locality.

High levels of fragmentation are expensive in terms of locality costs since it disperses the working set. However, fragmentation also causes added expense at allocation time as the allocation slow path must be called more frequently.

7.2 Processing Increments

Buffering the increments to be processed at collection time performs almost an identical quantity of work as processing increments in the write barrier. Neglecting locality effects, the only difference in cost is that buffering the increments incurs the cost of maintaining the buffer. Thus, processing increments in the write barrier performs better in total time either because of a locality gain, avoiding the cost of maintaining the increment buffer, or a combination of both factors.

Mutator time is reduced when processing the increments in the write barrier. This is because updating the reference count of a single object is less expensive than maintaining the increment buffer. However, when running the reference counter on a multiprocessor platform the cost of updating the increment will increase due to the required synchronisation overhead.

Also, as expected, the number of collections required when processing the increments in the write barrier decreased. As mentioned in chapter 6, this is because processing the increments in the write barrier requires less meta data. Having less metadata allows more decrements to accumulate before a collection is required, triggering collections less often.

7.2.1 Pause Times

Pause times remain similar for both strategies. The javac and db benchmarks both have slight deviations for average and maximum pause times respectively. While there is not much change in pause times, they should be easier to control when processing the increments in the write barrier.

As mentioned previously, reference counting must process all increments prior to processing of decrements to avoid incorrectly dropping a reference count to zero. This implies that when buffering the increments, the increment buffer must be emptied before the decrement buffer is processed. If memory is exhausted and decrements need to be processed to reclaim space, all the buffered increments need to be processed first.

This constraint means that pause times are only bound by the limit on meta data size. While it is possible to lower the size of the meta data, this causes more collections to occur and is more expensive. Conversely, processing the increments in the write barrier allows the decrements to be processed at any point in time, allowing strategies, such as time caps, to be used more effectively.

7.2.2 Summary of Processing Increments Findings

Processing the increments in the write barrier appears to be a superior approach for handling increments in a uni-processor setting. Total running time and mutator time both decreased. In the best case, processing increments in the write barrier was faster in total time and mutator time by 13% and 10% respectively.

Processing increments in the write barrier may lead to degraded performance in a multi-processor setting, as synchronisation is required among processors.

Additionally, processing the increments in the write barrier allows for easier control of pause times by relaxing a constraint surrounding the period when decrements can be processed.

7.3 Write Barrier Cost

Adding the write barrier to the semi-space collector increased total running time and mutator time as expected. The number of collections and pause times did not change when the write barrier was added, confirming that the addition of the write barrier did not interfere with the collection process.

The cost of the reference counting write barrier is dependent on architecture. On the PPC platform, the cost of the write barrier was 10%, whereas on the i686 platform the cost of the write barrier was 20%.

On both platforms, the tests where the write barrier was completely out-of-line performed slightly better than partially inlining the write barrier.

It is also important to note that the write barrier tests were performed with the referencing counting write barrier from the Jikes RVM 2.3.0 release, as such both increments and decrements were buffered[1] by the write barrier. Using the latest version of the reference counter, with increments processed in the write barrier, the costs could differ.

The *jess* benchmark produced an unusual result on both architectures. The *jess* benchmark produces the largest number of increments out of all the benchmarks in the SPECJVM98 suite. In the future, this would make it an interesting benchmark to test with the reference counter's updated write barrier.

7.3.1 Summary of Write Barrier Findings

The write barrier is a significant cost incurred by reference counting. The cost of the write barrier is highly dependent on the system architecture; the PPC architecture costing half the percentage of time required by th i686 architecture, making it a better candidate for reference counting.

7.4 Processing Decrements, Roots and Cycle Detection

Cycle detection is clearly a major problem for reference counting, especially at small heap sizes. The *javac* benchmark in particular spends up to 90% of collection time preforming cycle detection.

As expected, the time spent on decrements is fairly constant across heap sizes. This is because the number of decrements is tied to mutator activity and not to heap size.

Time spent on processing the roots decreases as heap size increases since there are fewer collections, thus the roots are scanned less frequently.

The javac benchmark showed some unusual behaviour when cycle detection stopped, the cost of processing roots increased and the cost of decrements decreased. These results occur because a large number of objects remain in the purple buffers and are unprocessed. This in turn means that some objects are retained and avoid being decremented.

7.4.1 Summary of Roots, Decrements and Cycle Detection Findings

As demonstrated by the results, cycle detection is a key area of cost in the reference counter. While other research [Levanoni and Petrank 2001] has been conducted to reduce the overhead of processing reference counter mutations, cycle detection remains a problematic component of reference counting.

[1] In the semi-space test no meta-data was *actually* buffered by the write barrier, see section 4.3.2.

7.5 Cycle Detection

The concurrent mark scan cycle detector has a much higher level of throughput than trial deletion. This high level of throughput is most likely due to the fact that after marking, the mark scan cycle detector only needs to traverse cyclic garbage once to collect it. Trial deletion, on the other hand, must traverse any cyclic structures multiple times in order to collect them. In addition, mark scan cycle detection does not retain cycles referenced by acyclic garbage, thus removing the chance that they will be unnecessarily traversed multiple times.

The non-concurrent mark scan cycle detector performed quite poorly in throughput when compared to the concurrent mark scan cycle detectors. This poor performance seems to be associated with the cost of invoking the marking phase too frequently. Since the marking operation can be quite expensive, the two mark scan cycle detectors should be invoked less frequently, thus reducing the load on the marking phase. This change should improve their performance.

The concurrent mark scan cycle detector also had the smallest average pause times for the *javac* benchmark. This appears to be because the other two cycle detectors were overwhelmed by their workload rather than the concurrent mark scan cycle detector performing well.

On the *jess* benchmark, both mark scan cycle detectors had poor pause times when compared to trial deletion. Responsiveness is obviously an area of improvement in both the mark scan cycle detectors.

7.5.1 Incremental Scanning

One possible improvement for both of the mark scan cycle detectors is to perform scanning of cycles incrementally. Performing incremental scanning could help to reduce pause times associated with cycle detection.

Unfortunately, as mentioned in chapter 5, cycles can only be collected after the entire cyclic structure has been added to a kill queue. This implies that while incremental scanning of the cyclic structures can be performed after a complete mark occurs, no cycles can safely be collected until scanning is finished.

7.5.2 Summary of Cycle Detection Findings

A new algorithm for cycle detection has been presented. Two implementations of that algorithm have been compared against the original trial deletion implementation.

The concurrent mark scan cycle detector performed well in throughput. On the *javac* benchmark it outperformed trial deletion by more than 25%. Conversely, the non-concurrent mark scan cycle detector performed poorly in throughput in comparison to its semi-concurrent counterpart. This poor performance was attributed to poor heuristics for triggering the marking phase of cycle detection.

Both mark scan cycle detectors performed poorly in pause times. The concurrent mark scan cycle detector still managed to obtain better pause times than the other two cycle detector implementations on the *javac* benchmark.

Incremental scanning is one strategy to help reduce the length of pause times in both mark scan cycle detectors.

7.6 Summary

This chapter has explored the results presented in chapter 6 in more detail. The implications of the results have also been discussed.

The bump-pointer has faster allocation than the free-list. It is cheaper to process increments in the write barrier on a uni-processor platform. The write barrier is dependent on the architecture of the platform on which it is running. Cycle detection is a major cost in reference counting.

The two new mark scan cycle detectors were compared against trial deletion. The semi-concurrent cycle detection has better throughput than trial deletion. Both new cycle detectors have poor pause times.

Chapter 8

Conclusion

8.1 Summary

Software engineering princples, such as modularity and re-usability, are driving forces behind the increasing shift towards object-oriented languages. Managed runtimes are also becoming increasingly attractive due to their improving compilation systems, such as the adaptive optimisation system in Jikes RVM, and advanced features, such as built in security. Given that previous studies have shown that managed runtimes can spend a third of their total execution time collecting garbage, dynamic automatic memory management is an important area of research. Further, it is currently a strong area of interest for both the research and industry groups.

While research in the area of automatic dynamic memory management dates back to 1960, adequate comparisons between garbage collection techniques and implementations are lacking.

Deferred reference counting is an example of an algorithm for which approaches to implmentation have yet to be compared or their costs quantified. Quantifying the costs of the reference counter's components and comparing implementation techniques was the first aim of this thesis. Tests for determining these values were described in chapter 4 and the results of these tests were presented in chapter 6.

The second aim of this thesis was to improve the performance of the reference counter where possible. The comparison between approaches for processing increments has lead to a marginal improvement in performance in a uni-processor platform. The cycle detection algorithm developed in chapter 5 is another step towards improving performance. While the two new cycle detector implementations perform poorly in pause times, there is room for improvement. Further, the performance of the semi-concurrent cycle detector has a significantly superior throughput when compared to the original trial deletion cycle detector.

8.2 Future Work

Processing Increments. The benefits of processing the increments in the write barrier have been explored. While this approach is clearly less expensive in a uni-processor setting, it is recommended that research is undertaken to compare the costs in a multiprocessor setting.

Reference Counter Implementation. While approaches to processing increments have been examined, approaches for implementing retain events have not. Furthermore the benefits of indicating acyclic objects to trial deletion have not been quantified. Research could be conducted to determine the costs and trade-offs involved in the three implementations of retain events and the benefits of flagging objects as acyclic.

Mark Scan Cycle Detection. The two new cycle detector implementations clearly require performance tuning. Triggering the marking phase less frequently should help to improve performance. Implementing incremental scanning may also help to reduce pause times.

8.3 Conclusion

The major work for this thesis has been completed in two areas. Firstly, key costs involved in reference counting garbage collection, including the write barrier, the cost of allocation, increment and decrement processing and cycle detection have been quantified. Secondly, having found cycle detection to be a key area of cost, a new algorithm for detecting and collecting cylces has been developed. Using this new algorithm allowed two new cycle detector implementations to be contributed to JMTk.

This work contributes to an evaluation of components of the reference counter where previous research has only looked at the performance of the reference counter as a whole. Additionally, different approaches to reference counting implementation, such as the approach used for cycle detection, have been compared. Again, this approach is a departure from previous work undertaken in this area.

Appendix A

Conventions

Throughout the document, figure A.1 will describe shapes used in figures unless indicated otherwise.

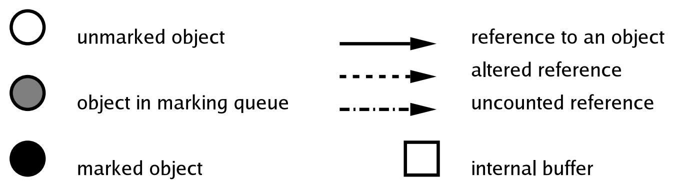


Figure A.1: A legend for figures in this document.

Appendix B

Complete Results

The complete results for the testing described in chapter 6 is listed below. An overview of the benchmarking approach along with the hardware and software used for each test is provided in chapter 6. This chapter also highlights the representative and distinguishing results.

For an overview of the behaviour of the benchmarks see [Standard Performance Evaluation Corporation 2003].

B.1 Allocation Costs

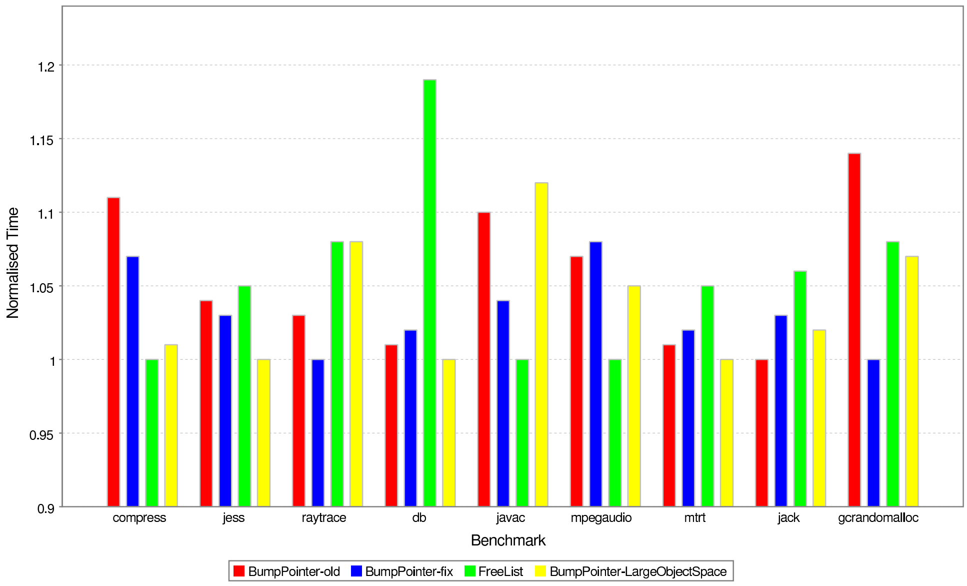
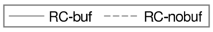
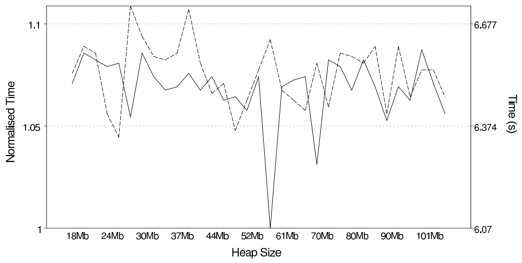


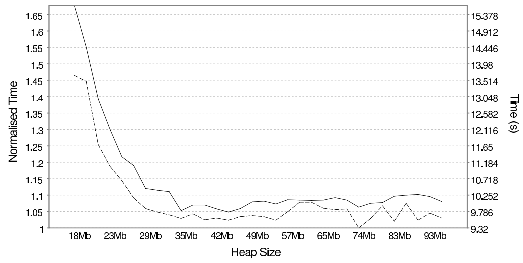
Figure B.1: Allocation performance across benchmarks.

B.2 Processing Increments



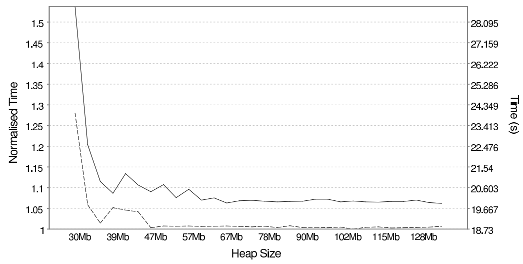


_201_compress

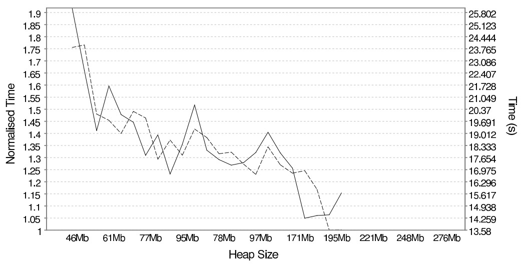


_202_jess

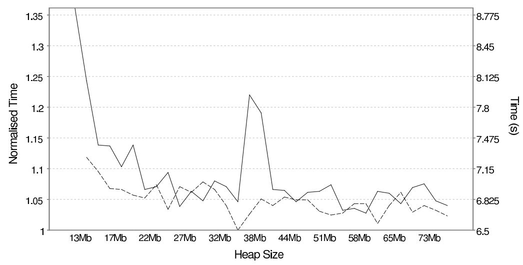
_205_raytrace



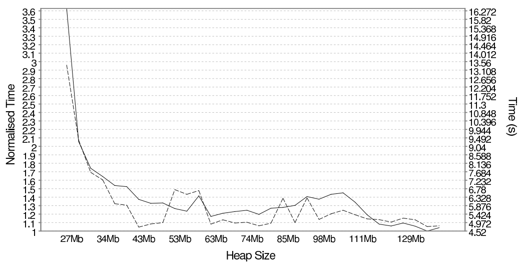
_209_db



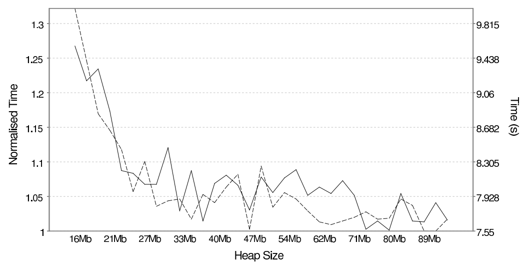
_213_javac



_222_mpegaudio



_227_mtrt



_228_jack

Figure B.2: Processing increments in write barrier vs. buffering till collection - total time.

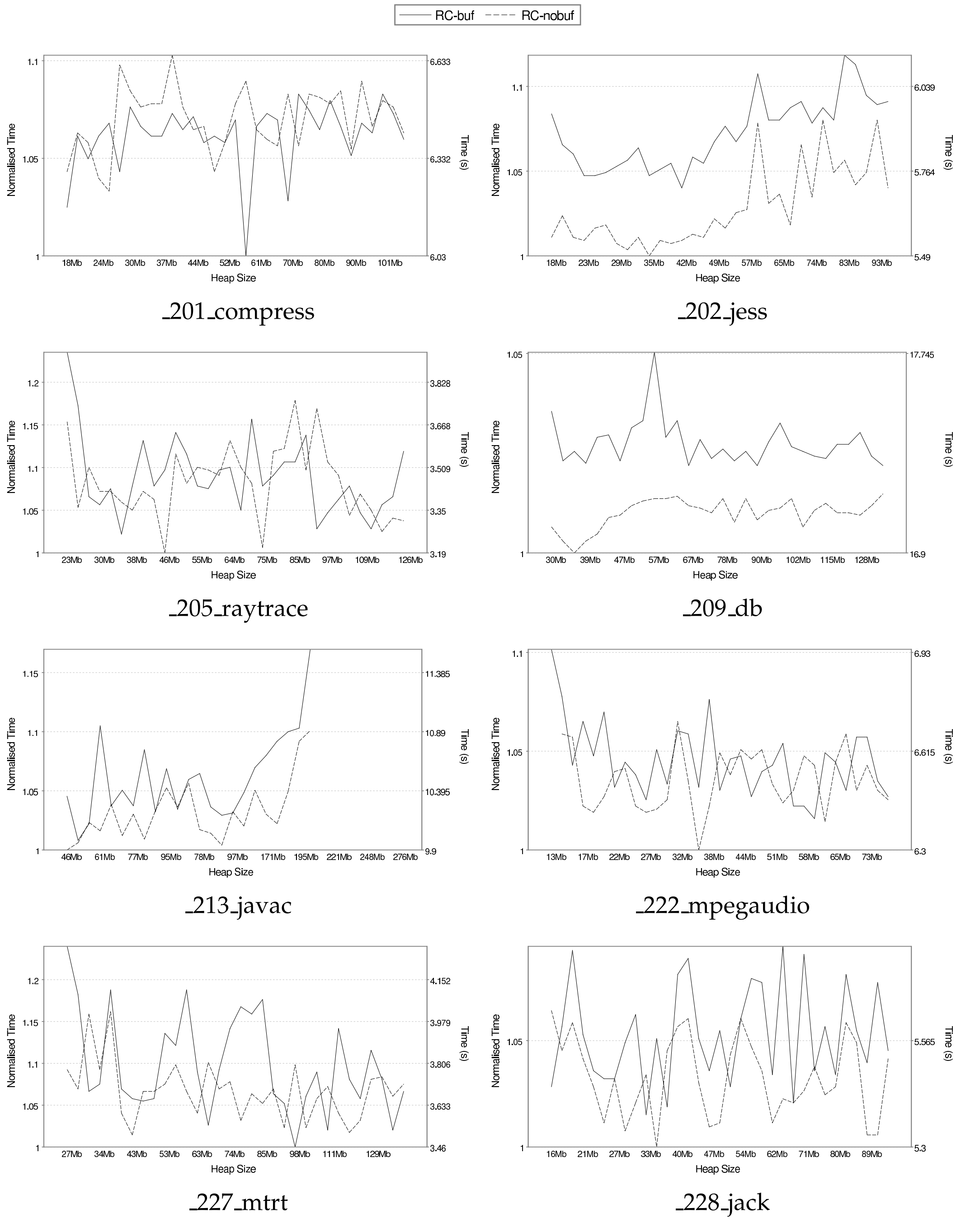
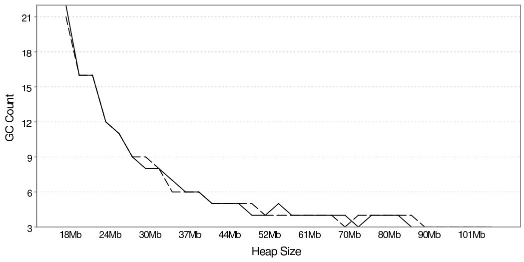
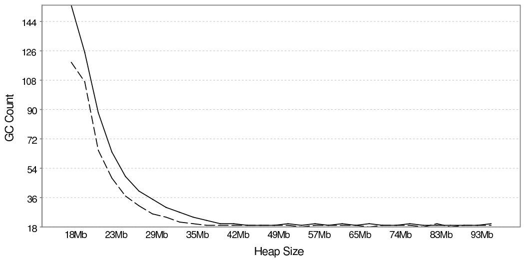


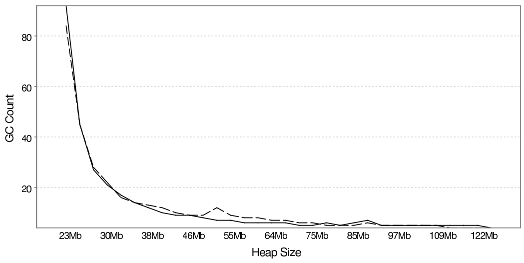
Figure B.3: Processing increments in write barrier vs. buffering till collection - mutator time.



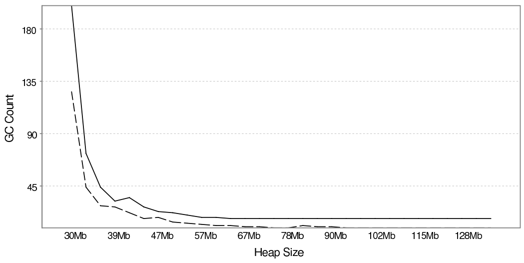
_201_compress



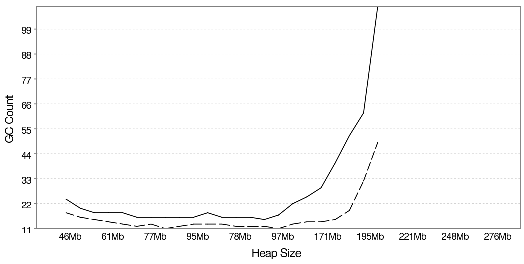
_202_jess



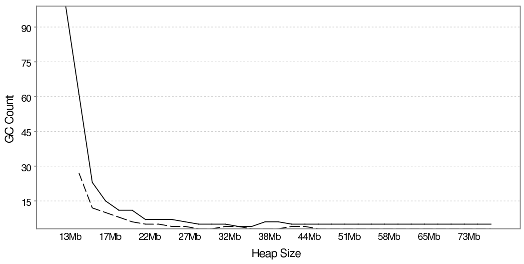
_205_raytrace



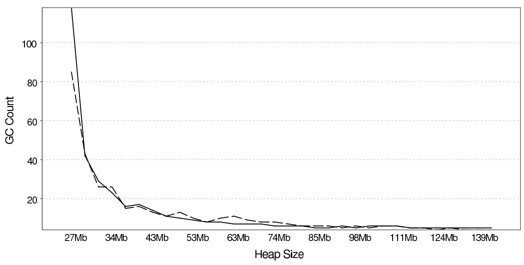
_209_db



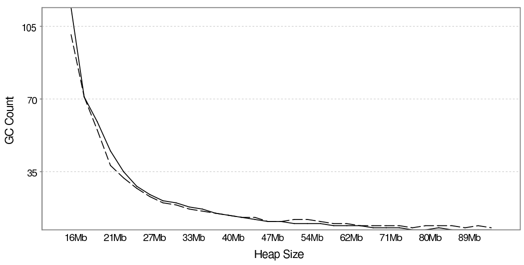
_213_javac



_222_mpegaudio



_227_mtrt



_228_jack

Figure B.4: Processing increments in write barrier vs. buffering till collection - collection count.

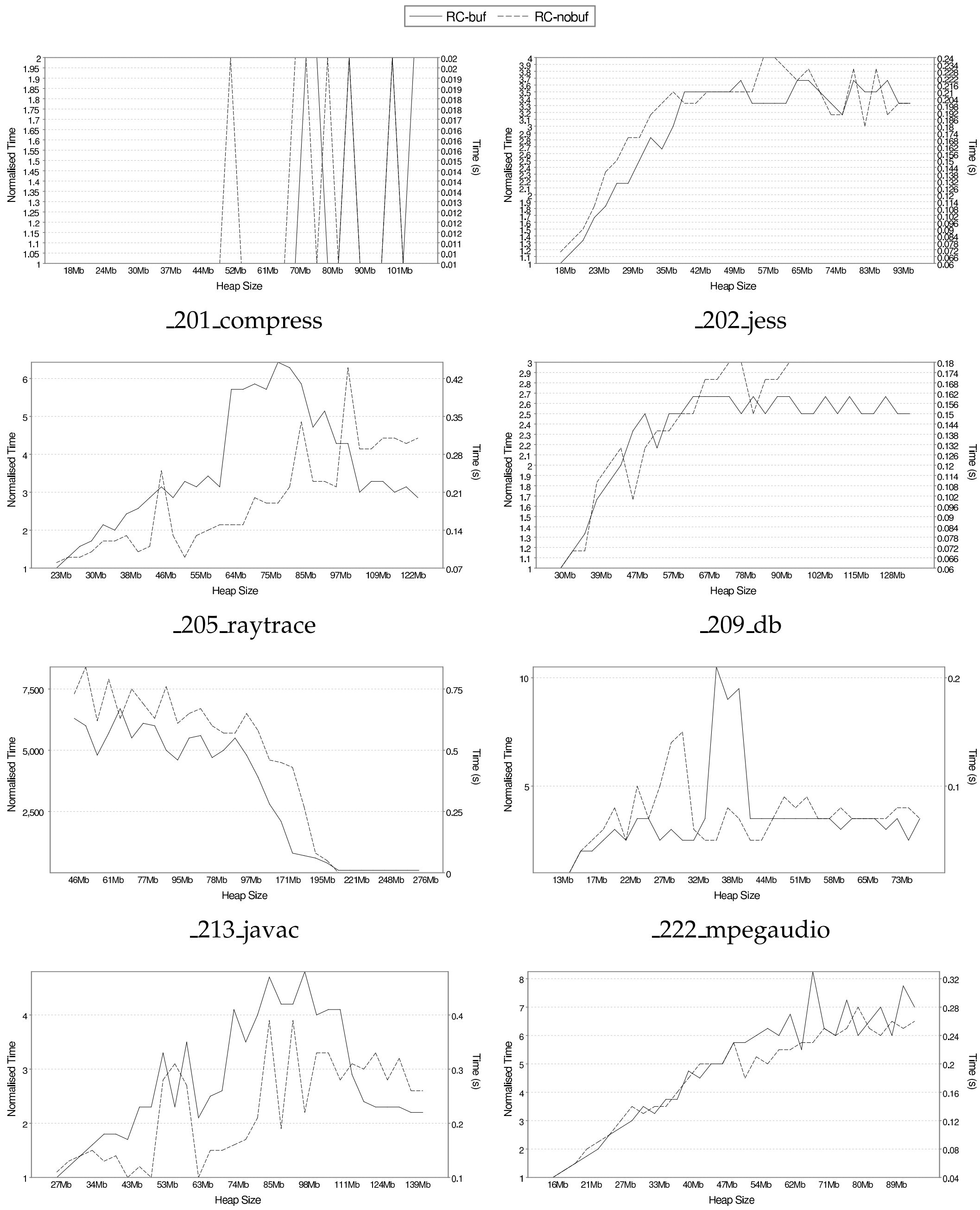


Figure B.5: Processing increments in write barrier vs. buffering till collection - average pause times.

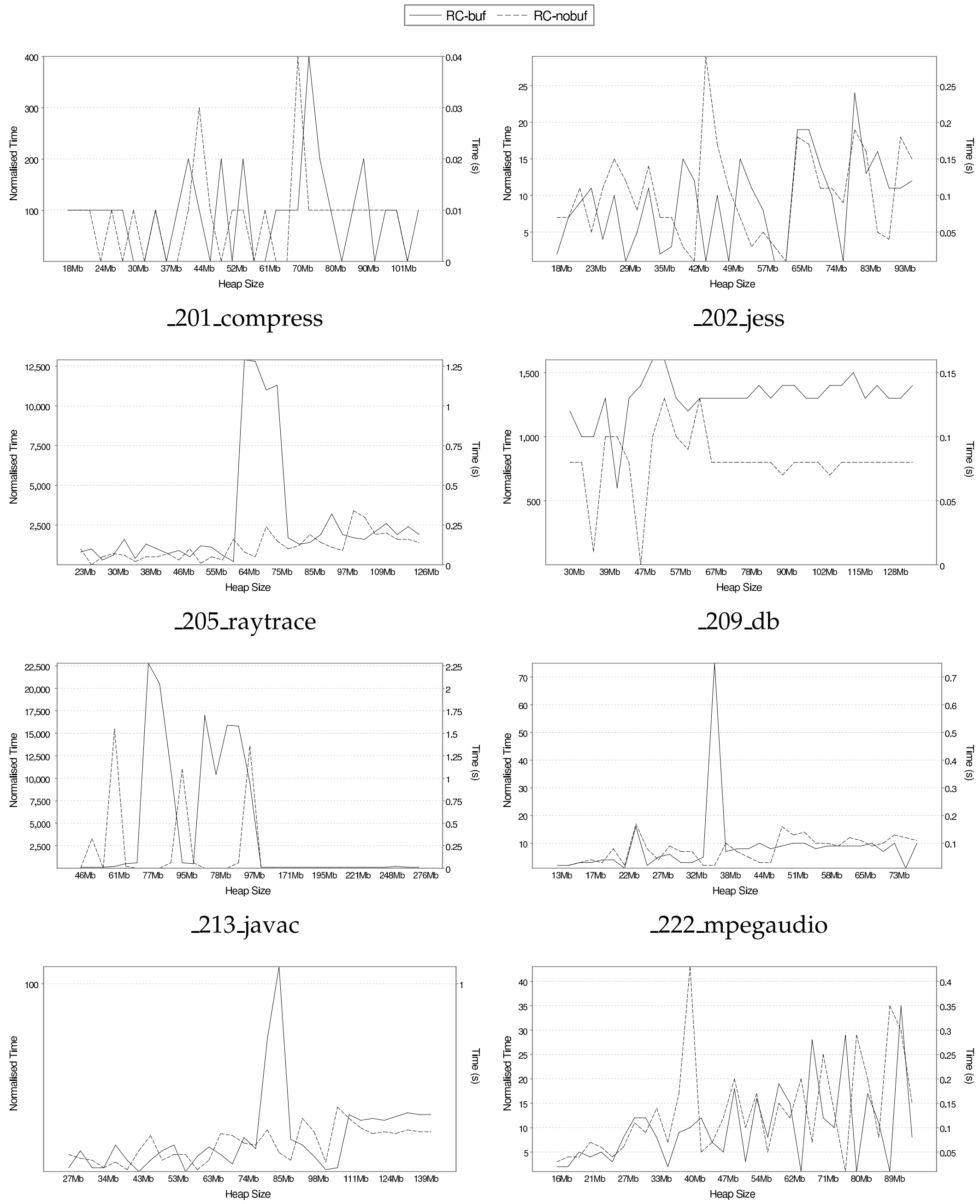
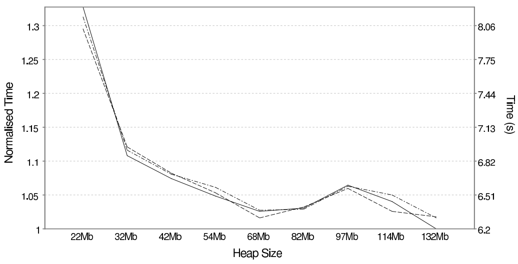


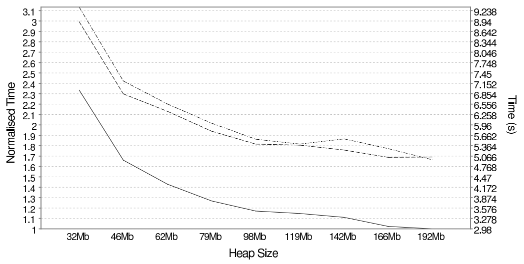
Figure B.6: Processing increments in write barrier vs. buffering till collection - maximum pause times.

B.3 Semi-Space with Write Barrier

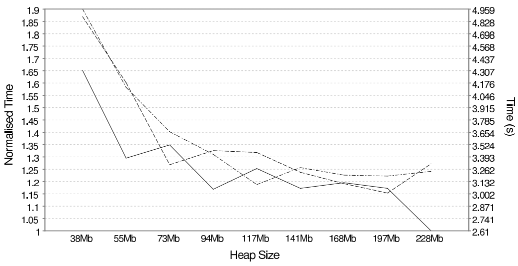
B.3.1 Intel i686 Architecture



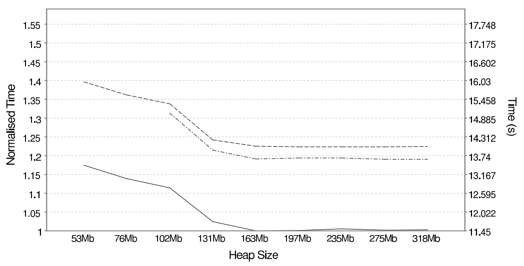
_201_compress



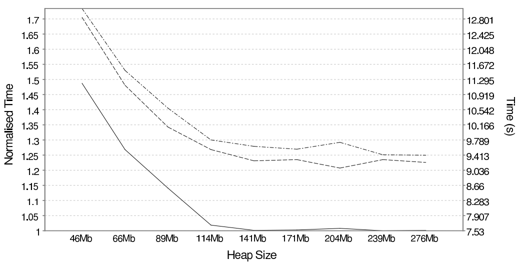
_202_jess



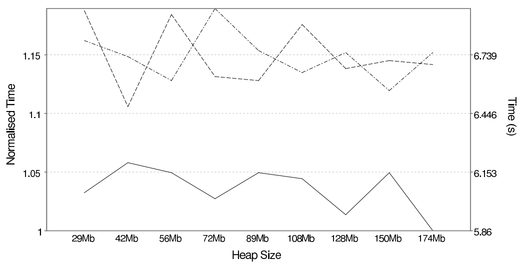
_205_raytrace



_209_db

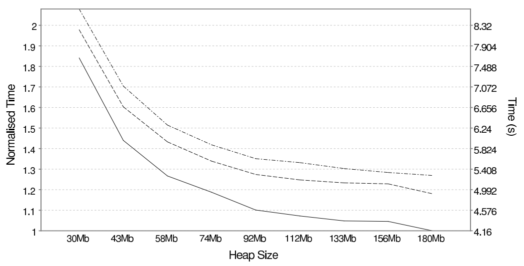


_213_javac



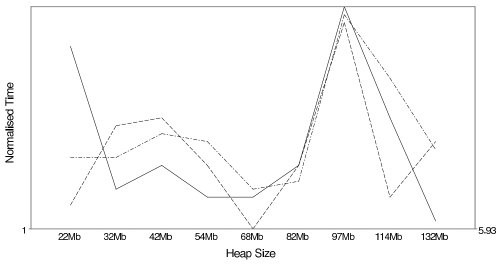
_222_mpegaudio

_227_mtrt

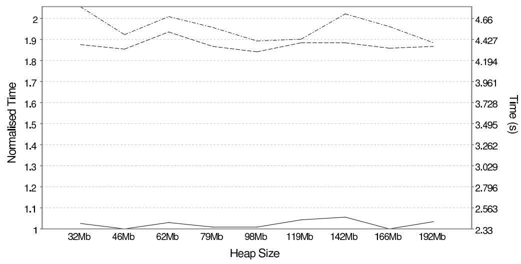


_228_jack

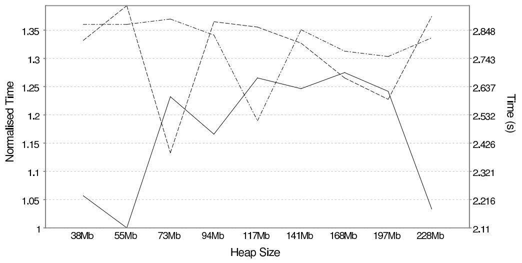
Figure B.7: i686 write barrier test - total time.



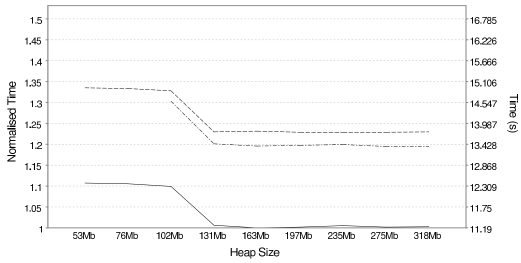
_201_compress



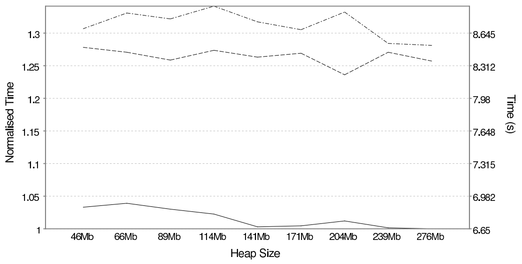
_202_jess



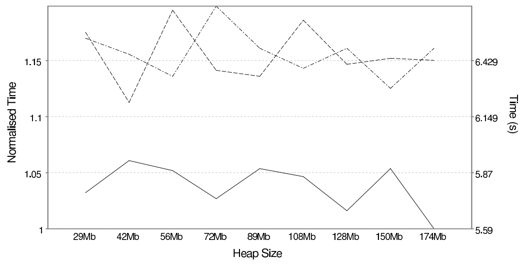
_205_raytrace



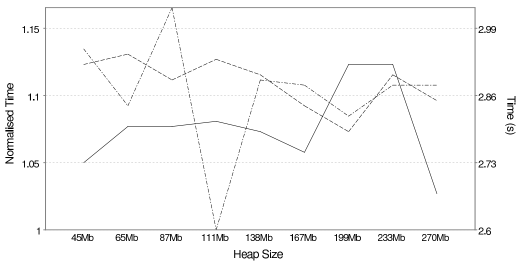
_209_db



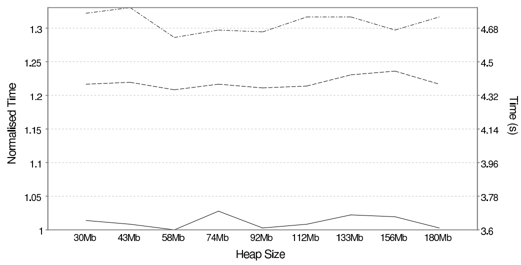
_213_javac



_222_mpegaudio



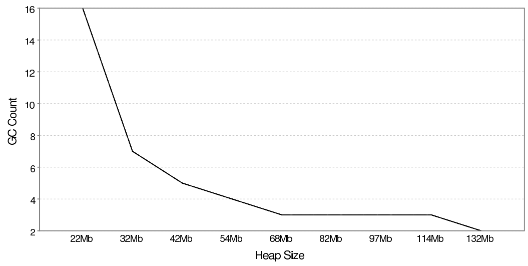
_227_mtrt



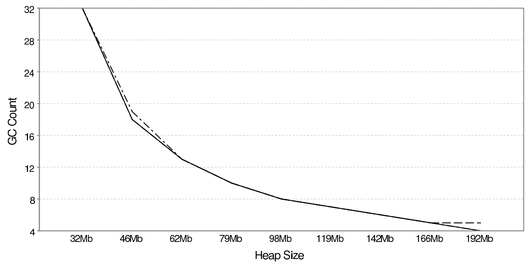
_228_jack

Figure B.8: i686 write barrier test - mutator time.

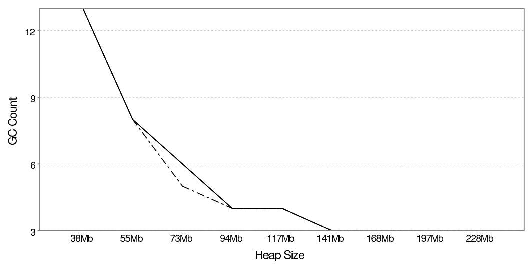
SemiSpace (SS) SS-WriteBarrier-Inline SS-WriteBarrier-OutOfLine



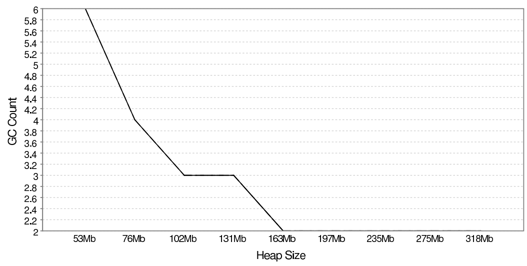
_201_compress



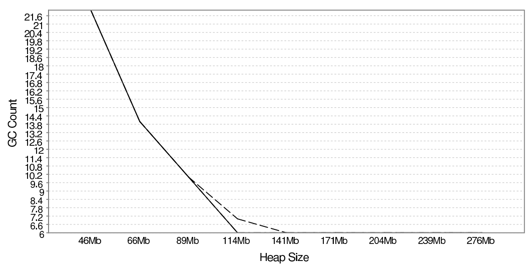
_202_jess



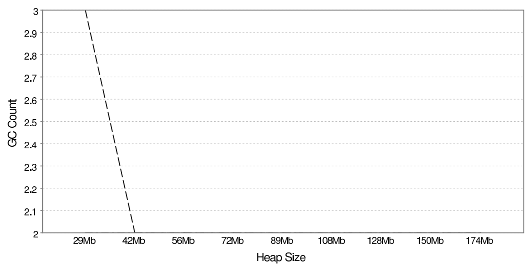
_205_raytrace



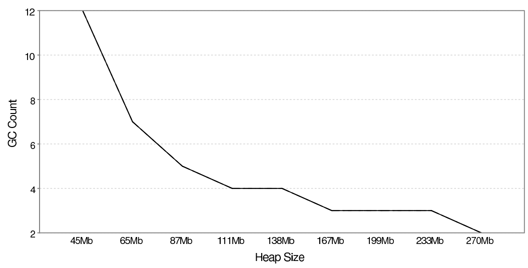
_209_db



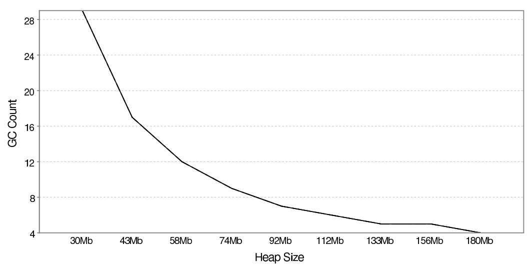
_213_javac



_222_mpegaudio



_227_mtrt

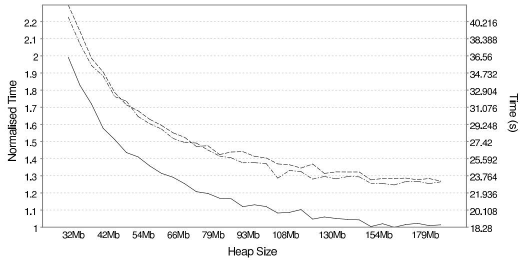


_228_jack

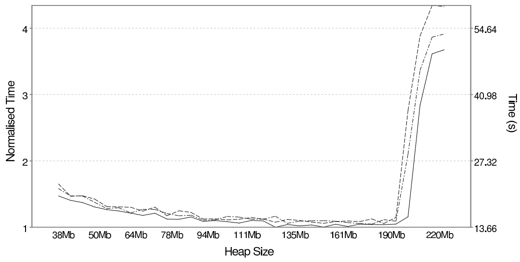
Figure B.9: i686 write barrier test - collection count.

B.3.2 Motorala PPC Architecture

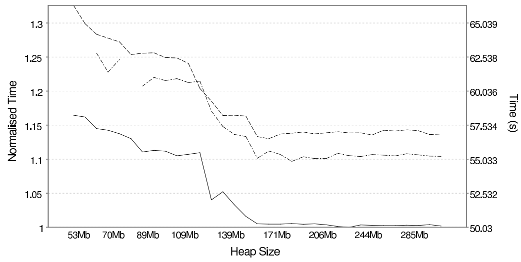
_201_compress



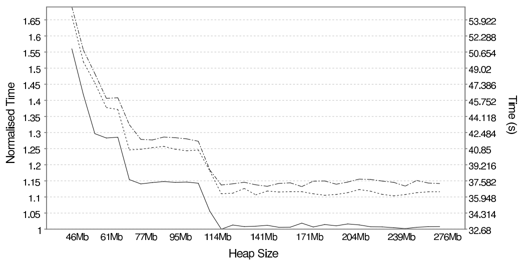
_202_jess



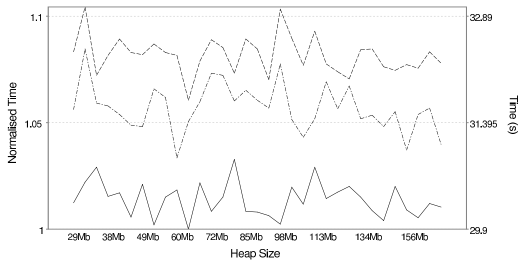
_205_raytrace



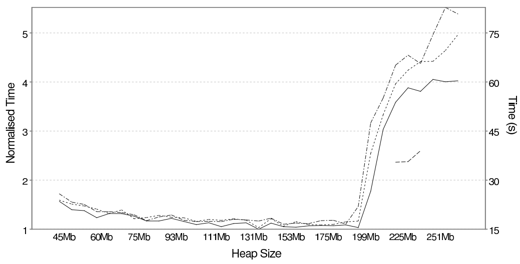
_209_db



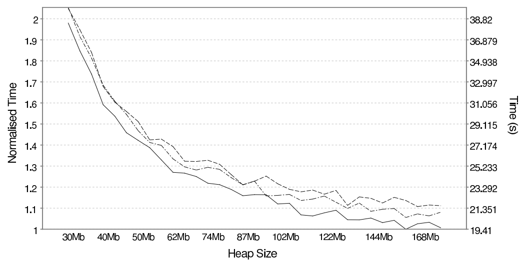
_213_javac



_222_mpegaudio



_227_mtrt



_228_jack

Figure B.10: PPC write barrier test - total time.

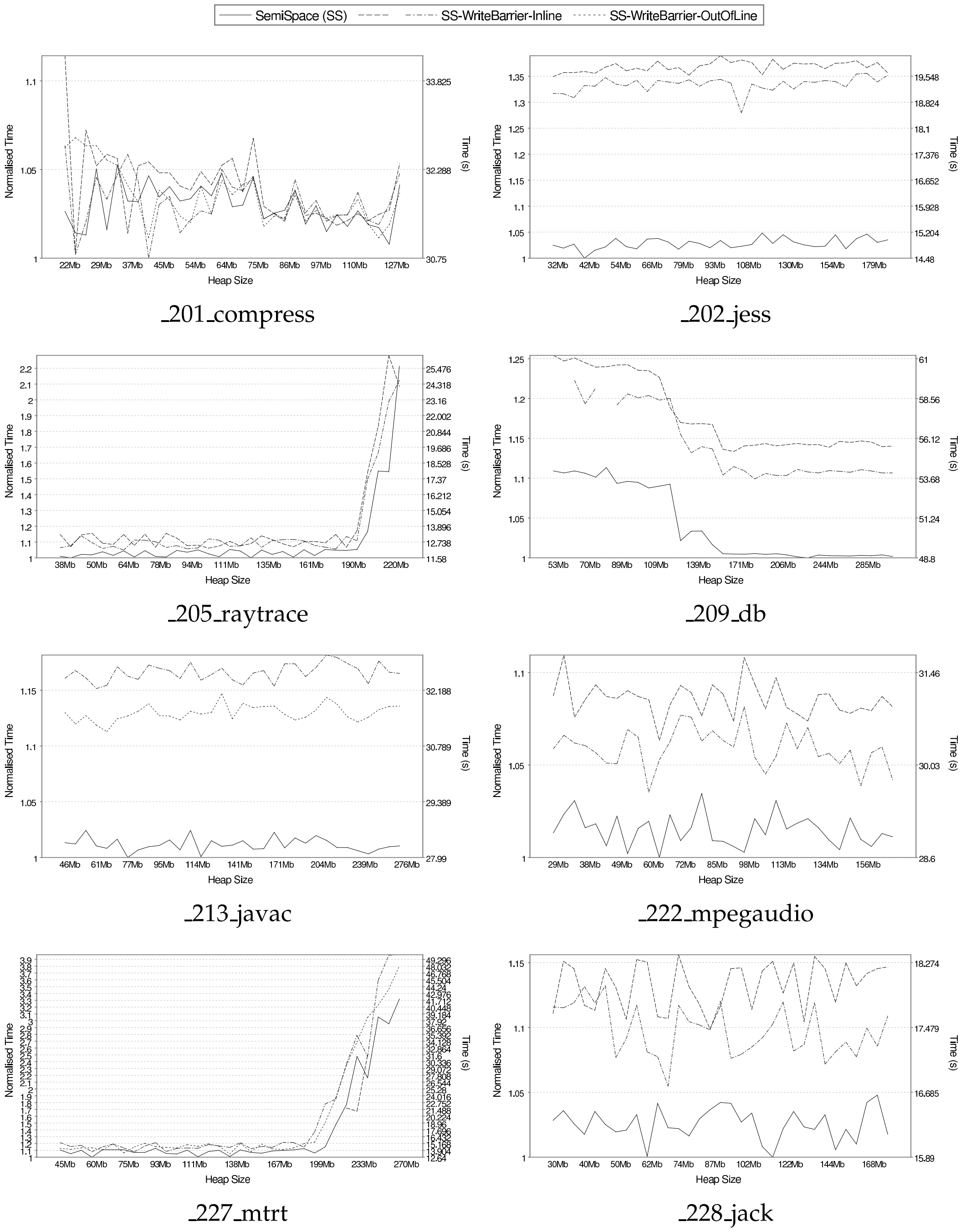
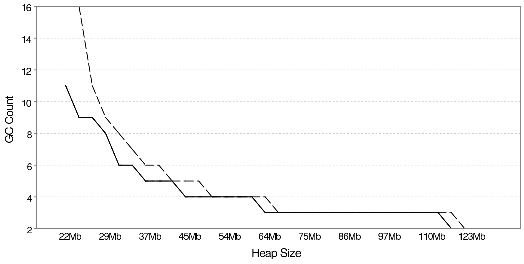
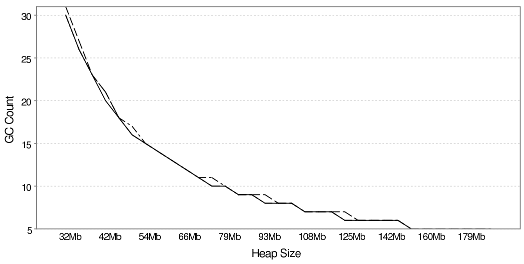


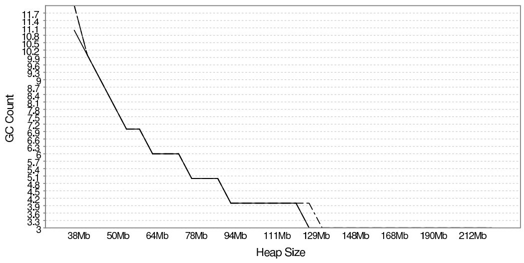
Figure B.11: PPC write barrier test - mutator time.



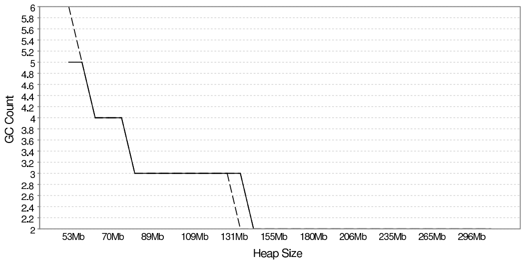
_201_compress



_202_jess

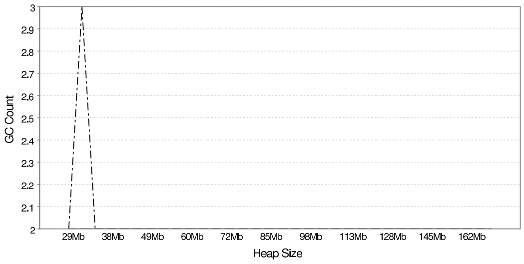


_205_raytrace

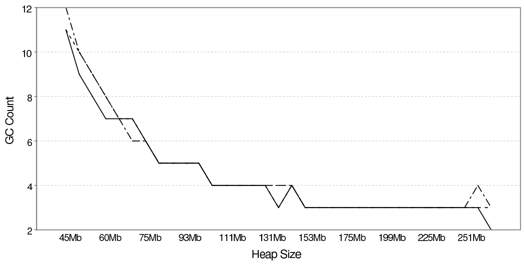


_209_db

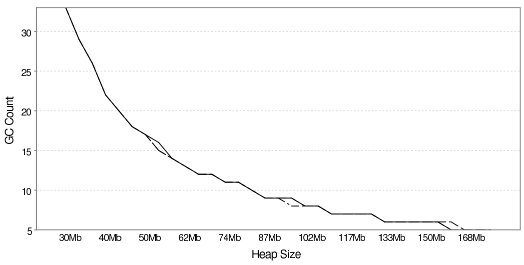
_213_javac



_222_mpegaudio



_227_mtrt

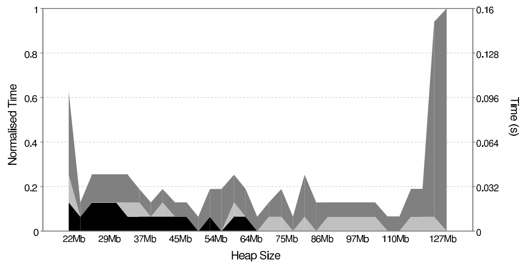


_228_jack

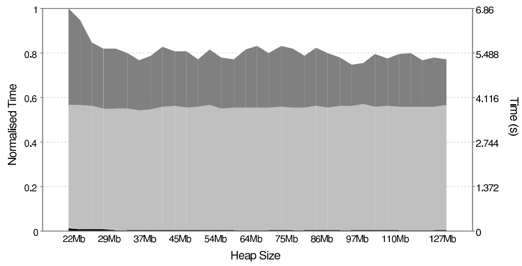
Figure B.12: PPC write barrier test - collection count.

B.4 Roots, Decrements and Cycle Detection

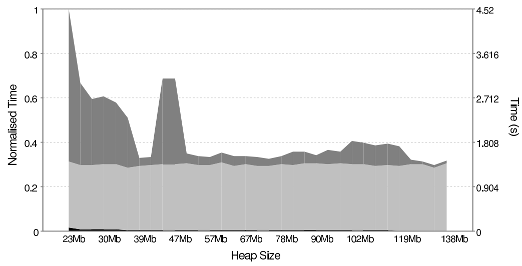
Roots Decrements Cycle Detection



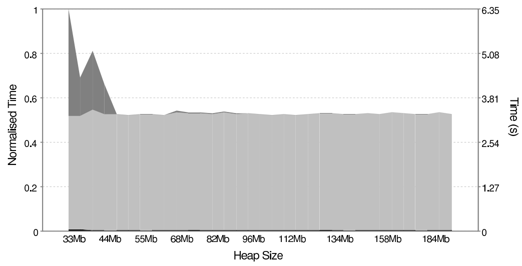
_201_compress



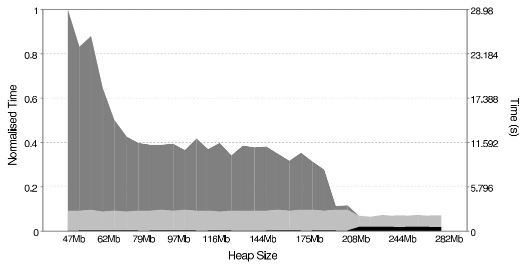
_202_jess



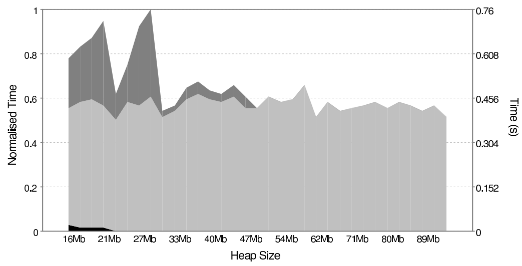
_205_raytrace



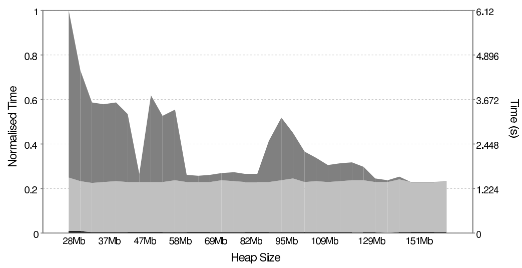
_209_db



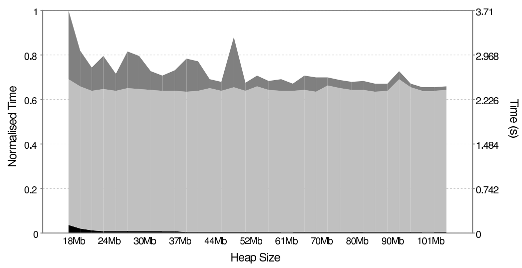
_213_javac



_222_mpegaudio



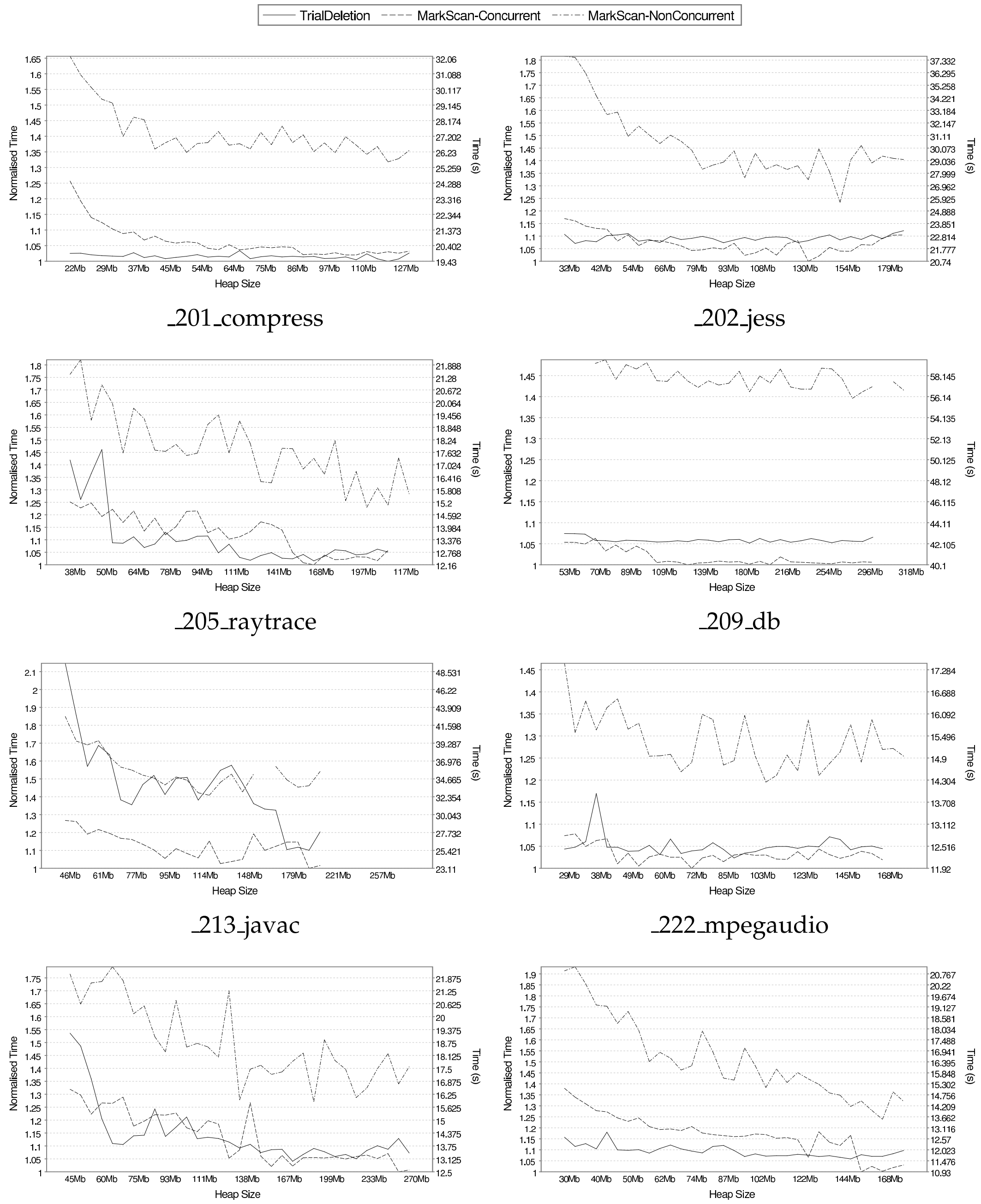
_227_mtrt



_228_jack

Figure B.13: Breakdown of reference counter time costs.

B.5 Cycle Detection



_201_compress

_202_jess

_205_raytrace

_209_db

_213_javac

_222_mpegaudio

_227_mtrt

_228_jack

Figure B.14: Cycle detection - total time.

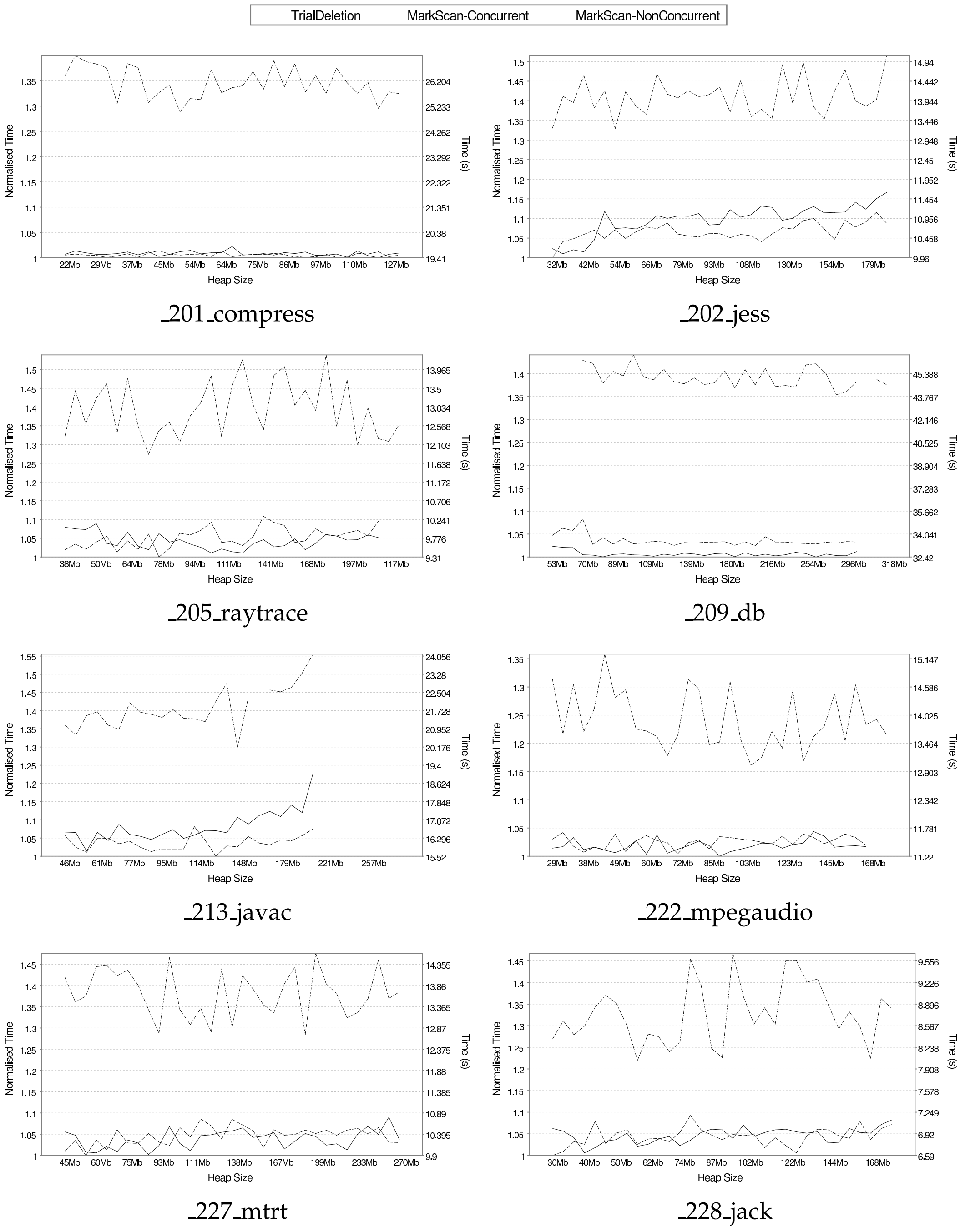


Figure B.15: Cycle detection - mutator time.

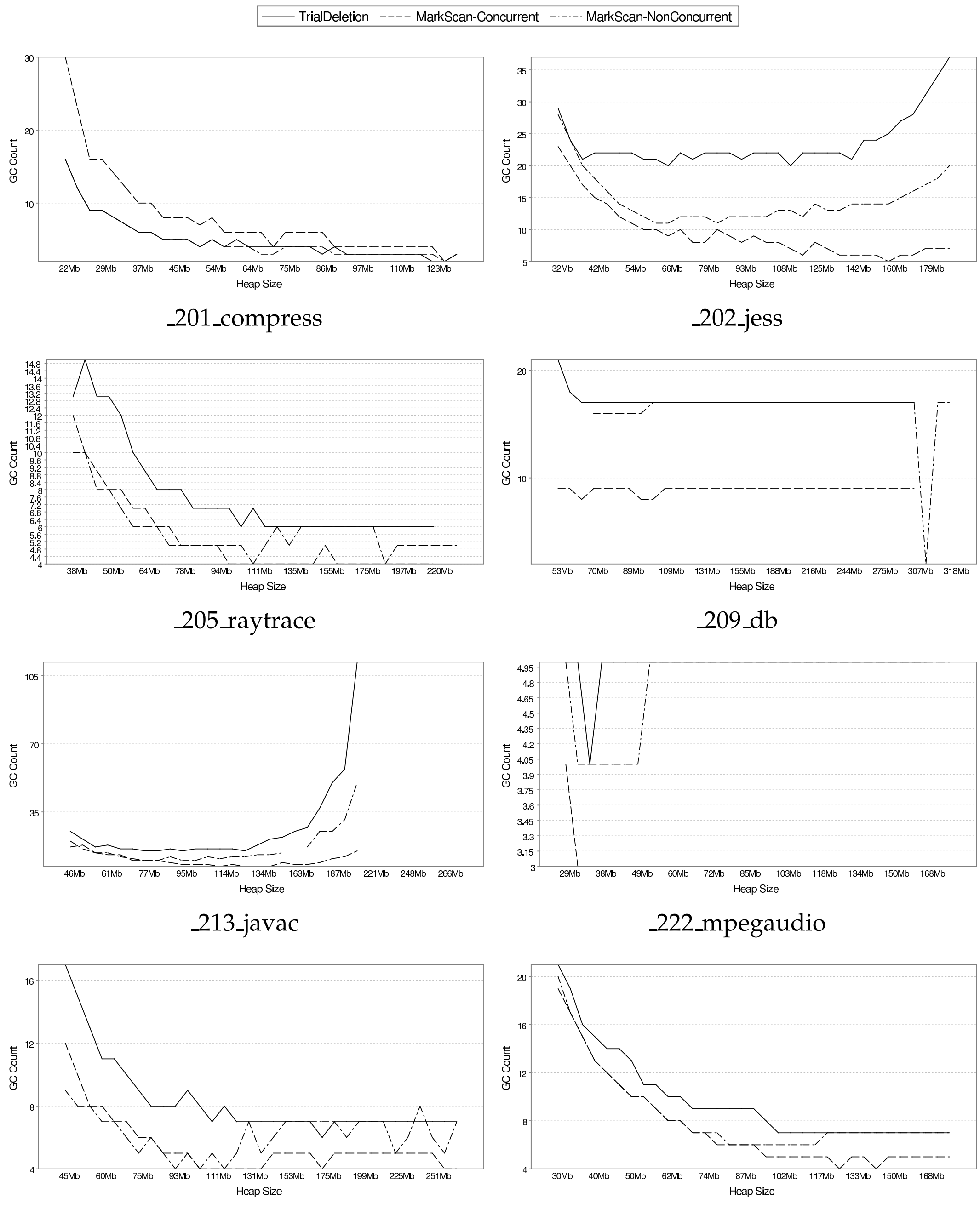
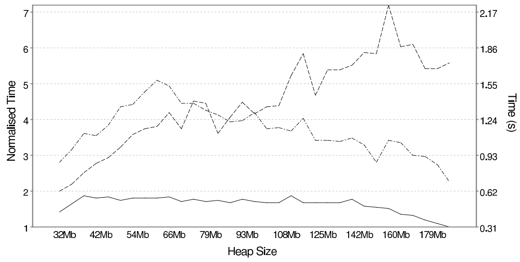


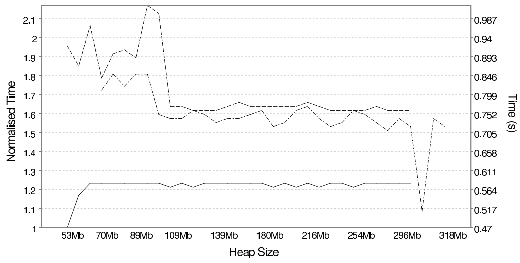
Figure B.16: Cycle detection - collection count.

_201_compress

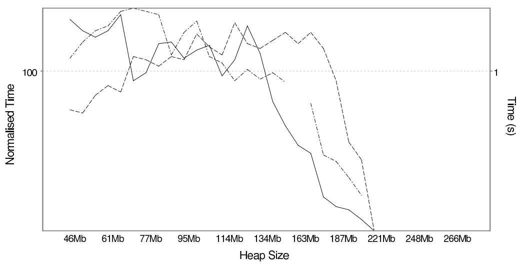


_202_jess

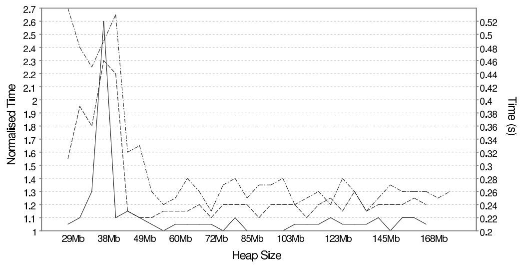
_205_raytrace



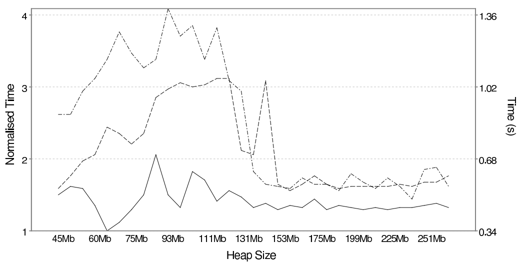
_209_db



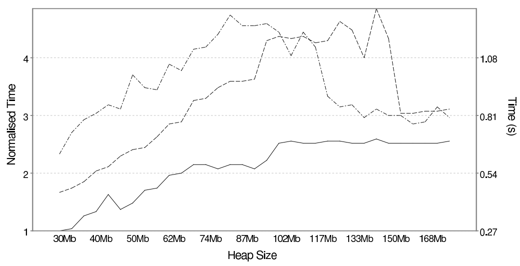
_213_javac



_222_mpegaudio



_227_mtrt



_228_jack

Figure B.17: Cycle detection - average pause times.

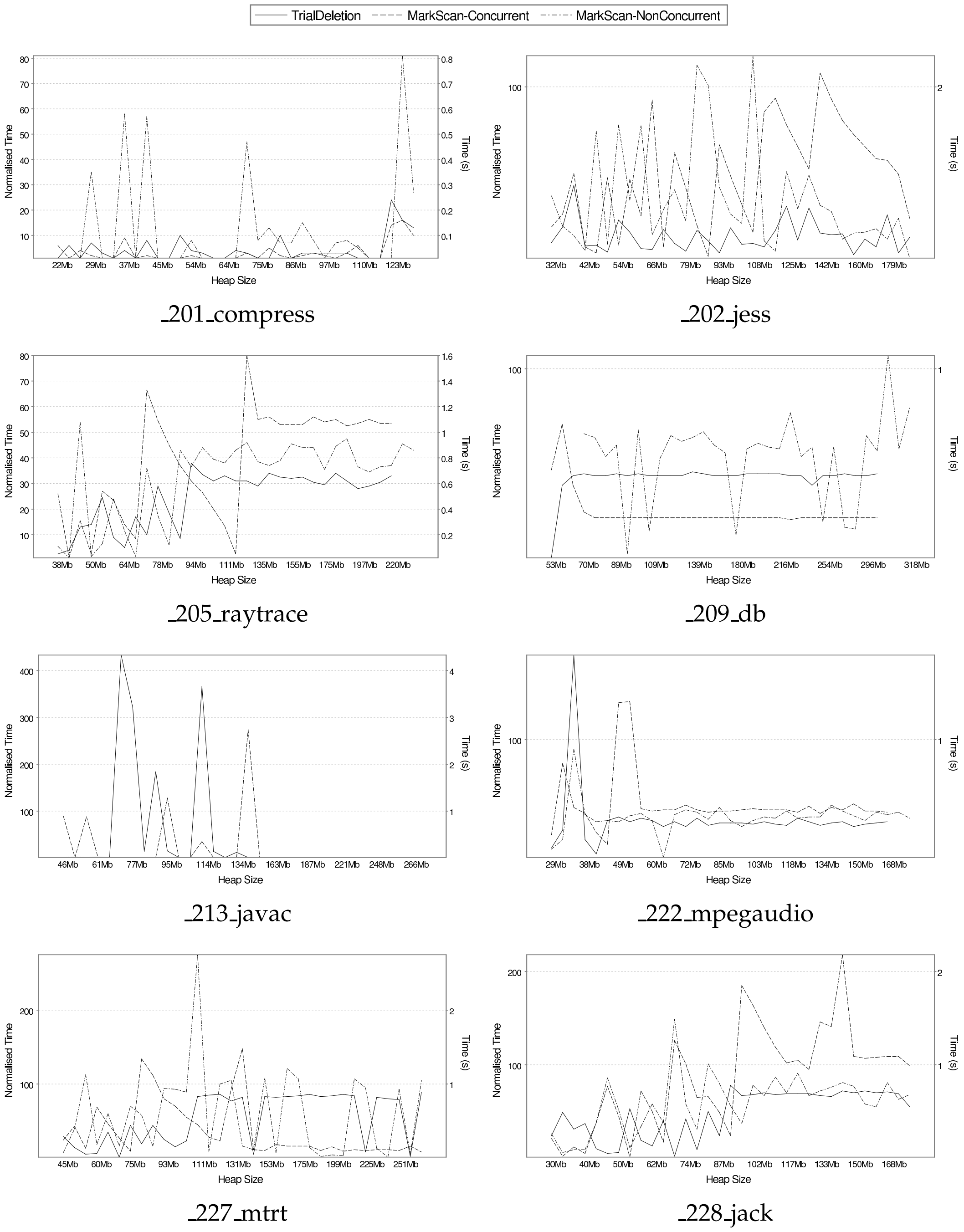


Figure B.18: Cycle detection - maximum pause times.

Glossary

Allocation - The process of allocating memory to the program.

Allocation Slow Path - The allocation code that aquires memory at the page level and requires synchronisation.

Collection - The process of reclaiming unused memory from the program.

Garbage - An object that is no longer needed by the system. It has no references from live objects but may reference live objects.

Locality - How much distance is between related objects.

Meta Data - Data used by the collector to perform the garbage collection.

Mutator - Any thread running in the program.

Nursery - The area that "young" objects reside in.

Object Cycle - Two or more objects that reference each other.

Reference Count - The number of references to an object.

Roots - Entry points for the graph of live objects.

Stuck Reference Count - A reference count that has over-flowed and requires special checking to reset.

Write Barrier - A piece of code that is executed at every non-deferred pointer mutation.

Bibliography

ALPERN, B., ATTANASIO, C. R., BARTON, J. J., BURKE, M. G., CHENG, P., CHOI, J.-D., COCCHI, A., FINK, S. J., GROVE, D., HIND, M., HUMMEL, S. F., LIEBER, D., LITVINOV, V., MERGEN, M., NGO, T., RUSSELL, J. R., SARKAR, V., SERRANO, M. J., SHEPHERD, J., SMITH, S., SREEDHAR, V. C., SRINIVASAN, H., AND WHALEY, J. 2000. The Jalapeño virtual machine. *IBM System Journal 39*, 1 (Feb.). (pp. 14, 15)

ALPERN, B., ATTANASIO, C. R., COCCHI, A., LIEBER, D., SMITH, S., NGO, T., BARTON, J. J., HUMMEL, S. F., SHEPERD, J. C., AND MERGEN, M. 1999a. Implementing Jalapeño in Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, Volume 34(10) of *ACM SIGPLAN Notices* (Denver, CO, Oct. 1999), pp. 314–324. ACM Press. (pp. 13, 14)

ALPERN, B., COCCHI, A., LIEBER, D., MERGEN, M., AND SARKAR, V. 1999b. Jalapeño – a Compiler–Supported Java Virtual Machine for Servers. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Compiler Support for System Software (WCSSS'99)* (May 1999).

ARNOLD, M., FLINK, S., GROVE, D., HIND, M., AND SWEENEY, P. F. 2000. Adaptive Optimization in the Jalapeño JVM. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, Volume 35(10) (Boston, MA, 2000), pp. 47–65. (p. 15)

BACON, D. F., ATTANASIO, C. R., LEE, H. B., RAJAN, V. T., AND SMITH, S. 2001. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices (Snowbird, Utah, June 2001). ACM Press. (pp. 1, 10, 18, 19)

BLACKBURN, S. M., CHENG, P., AND MCKINLEY, K. S. 2003. A Garbage Collection Design and Bakeoff in JMTk: An Efficient Extensible Java Memory Management Toolkit. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices (Anaheim, CA, Oct. 2003). ACM Press. (pp. 2, 15, 16)

BLACKBURN, S. M., JONES, R., MCKINLEY, K. S., AND MOSS, J. E. B. 2002. Beltway: Getting around garbage collection gridlock. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices (Berlin, June 2002), pp. 153–164. ACM Press. (p. 5)

BLACKBURN, S. M. AND MCKINLEY, K. S. 2002. In or out? putting write barriers in their place. In D. DETLEFS Ed., *ISMM'02 Proceedings of the Third International*

Symposium on Memory Management, ACM SIGPLAN Notices (Berlin, June 2002), pp. 175–184. ACM Press. (p. 24)

BLACKBURN, S. M. AND MCKINLEY, K. S. 2003. Ulterior Reference Counting: Fast Garbage Collection without a Long Wait. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices (Anaheim, CA, Oct. 2003). ACM Press. (pp. 1, 4, 7)

BURKE, M. G., CHOI, J.-D., FLINK, S., GROVE, D., HIND, M., SARKAR, V., SERRANO, M. J., SREEDHAR, V. C., SRINIVASAN, H., AND WHALEY, J. 1999. The Jalapeño Dynamic Optimizing Compiler for Java. In *ACM 1999 Java Grande Conference* (June 1999), pp. 259–269. (p. 15)

COLLINS, G. E. 1960. A method for overlapping and erasure of lists. *Communications of the ACM 3*, 12 (Dec.), 655–657. (p. 6)

COMMITTEE, J. 1997. Draft american national standard for information systems - programming languages - smalltalk. (p. 4)

DETREVILLE, J. 1990. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64 (Aug.), DEC Systems Research Center, Palo Alto, CA. (pp. 3, 12)

DEUTSCH, L. P. AND BOBROW, D. G. 1976. An efficient incremental automatic garbage collector. *Communications of the ACM 19*, 9 (Sept.), 522–526. (pp. 9, 10)

DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., SCHOLTEN, C. S., AND STEFFENS, E. F. M. 1978. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM 21*, 11 (Nov.), 965–975. (p. 34)

ECMA. 2002. *ECMA-334: The C# Language Specification* (Second ed.). ECMA. (p. 4)

HUGHES, R. J. M. 1982. A semi-incremental garbage collection algorithm. *Software Practice and Experience 12*, 11 (Nov.), 1081–1084. (p. 8)

IBM RESEARCH. 2003. The Jikes Research Virtual Machine User's Guide Post 2.3.0.1(CVS Head). http://www-124.ibm.com/developerworks/oss/jikesrvm/userguide/HTML/userguide.html. (p. 14)

INGALLS, D., KAEHLER, T., MALONEY, J., WALLACE, S., AND KAY, A. 1997. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *OOPSLA'97 ACM Conference on Object-Oriented Systems, Languages and Applications — Twelth Annual Conference*, Volume 32(10) of *ACM SIGPLAN Notices* (Atlanta, GA, Nov. 1997), pp. 318–326. ACM Press. (p. 13)

JONES, R. E. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley. With a chapter on Distributed Garbage Collection by R. Lins. (pp. 3, 4, 6, 8, 9, 12)

JOY, B., STEELE, G., GOSLING, J., AND BRACHA, G. 2000. *The Java Language Specification* (Second Edition ed.). Addison-Wesley. (p. 4)

LEVANONI, Y. AND PETRANK, E. 2001. An on-the-fly reference counting garbage collector for Java. In *OOPSLA'01 ACM Conference on Object-Oriented Systems, Languages and Applications*, Volume 36(10) of *ACM SIGPLAN Notices* (Tampa, FL, Oct. 2001). ACM Press. (pp. 1, 6, 9, 50)

LIEBERMAN, H. AND HEWITT, C. E. 1983. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM 26(6)*, 419–429. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981. (p. 6)

LINS, R. D. 1992. Cyclic reference counting with lazy mark-scan. *Information Processing Letters 44*, 4, 215–220. Also Computing Laboratory Technical Report 75, University of Kent, July 1990. (p. 10)

MCBETH, J. H. 1963. On the reference counter method. *Communications of the ACM 6*, 9 (Sept.), 575. (p. 10)

MCCARTHY, J. 1960. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM 3*, 184–195. (p. 6)

MEYER, B. 1988. *Object-oriented Software Construction*. Prentice-Hall. (p. 4)

SCHORR, H. AND WAITE, W. 1967. An efficient machine independent procedure for garbage collection in various list structures. *Communications of the ACM 10*, 8 (Aug.), 501–506. (pp. 6, 8)

STANDARD PERFORMANCE EVALUATION CORPORATION. 2003. SPEC JVM Client98 Help. http://www.spec.org/jvm98/jvm98/doc/benchmarks/index.html. (p. 57)

TAIVALSAARI, A. 1998. Implementing a java virtual machine in the java programming language.

TANENBAUM, A. S. 1999. *Structured Computer Organization* (Forth ed.). Prentice-Hall. (p. 7)

UNGAR, D. M. 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices 19*, 5 (April), 157–167. 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. (p. 9)

WILSON, P. R. 1992. Uniprocessor garbage collection techniques. In Y. BEKKERS AND J. COHEN Eds., *Proceedings of International Workshop on Memory Management*, Volume 637 of *Lecture Notes in Computer Science* (University of Texas, USA, 16–18 Sept. 1992). Springer-Verlag. (pp. 1, 3, 6, 24)

ZORN, B. 1990. Barrier methods for garbage collection. Technical Report CU-CS-494-90 (Nov.), University of Colorado, Boulder. (p. 24)

ZORN, B. G. 1989. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley. Technical Report UCB/CSD 89/544. (p. 8)