An Investigation into Automatic Dynamic Memory Management Strategies using Compacting Collection

Daniel John Frampton

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

November 2003

© Daniel John Frampton

Typeset in Palatino by TeX and LaTeX 2ε.

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

Daniel John Frampton
14 November 2003

To my parents.

Acknowledgements

I would like to take this opportunity to thank some of the people that helped me through this process. In doing this I am sure to forget a few people, but I will do my best.

Firstly I would like to thank Steve, for providing me with the opportunity to do this project. With a special ability to make things *sound* easy, he was able to trick me into diving head first into some sticky situations. These experiences, although at times frustrating, were also the most interesting and rewarding.

I would also like to thank Luke, who also endured a similar project concurrent to my own. Being able to talk about many of the issues with someone in the same situation made the path a lot clearer. Assistance in setting up and supporting *toki*, the server that was purchased for this project was appreciated. The *whiteboard-action* and *red-pen-action* sessions were invaluable tools.

This project would not have been possible without the good work by the Jikes RVM team. Special thanks go to Perry and Dave for helpful feedback during the project.

I would also like to thank Richard Jones, not only for the excellent reference [Jones 1996] that he has gifted the garbage collection world, but also for the bibliography [Jones] he keeps that made the process of keeping a consistent bibliography so much easier.

I can not thank my parents enough, and I credit them with teaching me *how* to think, a skill that has been invaluable to me throughout my life.

I would also like to thank Jo, for helping to keep many of the other aspects of my life moving during the last couple of months.

Lastly, I would like to thank the people that took the time to read and provide feedback on my work, who were (in no particular order), my parents, Jo, Luke, Ruth, and Steve.

Abstract

Modern object oriented languages such as Java and C# have been gaining widespread industry support in recent times. Such languages rely on a runtime infrastructure that provides automatic dynamic memory management services. The performance of such services is a crucial component of overall system performance.

This thesis discusses work undertaken in relation to automatic memory management using the Java Memory Management Toolkit (JMTk) running on the Jikes Research Virtual Machine (Jikes RVM). The primary goal of this work was to develop an automatic memory management strategy employing a *compacting collector* to run on this platform.

Compacting collectors are an important class of collectors used in several production runtimes, including Microsoft's Common Language Runtime and IBM's Java Runtime Environment. The development of a strategy using compaction makes an important contribution to JMTk, and provides a platform where side-by-side comparisons between compacting collectors and other important classes of collector can be made.

A compacting collector differs from the collectors that currently exist in JMTk in several important ways. Prior to this work, JMTk and Jikes RVM did not have an implementation of a compacting collector, nor the structure to fully support one.

This work has achieved its primary goal in providing an implementation of a compacting collector. It describes how both JMTk and Jikes RVM were modified to support such collectors. Although substantial, this project should be considered but a first step into the investigation of this class of collectors. It is anticipated that through broadening the set of operations supported by JMTk and Jikes RVM that this work will also allow new classes of collectors to be implemented and compared.

The cost of performing a compacting collection was shown to be very significant given the current implementation. The use of compaction in a generational collector demonstrated increased performance, bringing it in-line with other generational collectors in JMTk.

This work shows that there are benefits in reducing memory fragmentation through the use of compacting collectors. When discounting the cost of the collection, the implemented compacting collectors come close to matching or outperforming other collection strategies. The difficulty now lies in attempting to reduce the cost of compacting collection.

Contents

Acknowledgements **vii**

Abstract **ix**

1 Introduction **1**
1.1 Contribution 2
1.2 Structure 2

2 Automatic Dynamic Memory Management **3**
2.1 Object Representations 5
2.2 Wasted Memory and Memory Fragmentation 5
2.3 Locality 6
2.4 Allocation Techniques 7
2.4.1 Bump Pointers 7
2.4.2 Free Lists 7
2.5 Garbage Collection Techniques 8
2.5.1 Barriers 9
2.5.2 Reference Counting 9
2.5.3 Tracing Collection 11
2.5.4 Copying Collection 12
2.5.5 Compacting Collection 14
2.5.6 Tradeoffs 15
2.5.7 Generational Collection 16
2.6 Summary 17

3 Research Platform **19**
3.1 Jikes RVM 19
3.1.1 Jikes RVM Compilation 19
3.1.2 Object Representation 20
3.1.2.1 Address Based Hashing 21
3.2 Java Memory Management Toolkit (JMTk) 21
3.2.1 The Plan 22
3.2.2 Allocation and Collection Policies 22
3.2.3 Utilities 23
3.2.4 VM – MM Interface 23
3.2.5 Memory Allocation 23
3.2.5.1 Bump Pointer 24

3.2.5.2 Segregated Free List 24
3.2.5.3 Treadmill 25
3.2.6 Example Memory Management Strategies 25
3.2.6.1 Mark Sweep 26
3.2.6.2 Semi Space 27
3.3 Summary 27

4 A Sliding Compacting Collector 29
4.1 The Concept 29
4.2 The Basic Algorithm 29
4.3 Implementation Issues 30
4.3.1 Object Representation 30
4.3.2 Forwarding Pointers 31
4.3.3 Immortal Type Information 31
4.3.4 Dynamic Linking 31
4.3.5 Address Based Hashing 32
4.3.6 Segregation of Scalars and Arrays 32
4.3.7 Accessing Objects During a Collection 33
4.3.8 Two-Phase Collection 34
4.3.9 Finalizable Objects 34
4.3.10 Reference Types 35
4.3.11 Multiple Allocators For a Space and Kernel Thread 35
4.4 The Algorithm 35
4.4.1 Object Header 36
4.4.2 Allocation Policy 36
4.4.2.1 Chunks 36
4.4.2.2 Regions 37
4.4.2.3 Extending and Creating Regions 38
4.4.3 Collection Policy 38
4.4.3.1 Pre Copy GC Instances 40
4.4.3.2 Mark Live Objects 41
4.4.3.3 Calculate Forwarding Pointers 42
4.4.3.4 Update References 42
4.4.3.5 Move Objects 42
4.4.3.6 Restore Status Words 44
4.4.4 Complete Memory Management Strategy 44
4.5 Summary 45

5 A Free List Compacting Collector 47
5.1 Motivation 47
5.2 The Algorithm 47
5.2.1 Sweep vs. Compact 48
5.2.2 Heap Iteration 48
5.2.3 Rebuilding Free Lists 48

5.2.4 Sweeping Free Lists 50
5.3 Summary 50

6 A Generational Compacting Collector 53
6.1 Motivation 53
6.2 Two-Phase Collection 53
6.3 The Algorithm 53
6.4 Complete Memory Management Strategy 54
6.5 Summary 54

7 Performance Evaluation 57
7.1 Benchmarking Methodology 57
7.1.1 Experimental Platform 57
7.1.2 Benchmarks 58
7.1.3 Configurations 58
7.1.4 Heap Sizes 60
7.1.5 Collection of Metrics 61
7.1.6 Missing Results 62
7.2 Non-Zero Status Words 62
7.3 Phase Timings 64
7.4 Flipped Object Model 65
7.5 Constant Size Header 66
7.6 Free List: Compact vs Sweep 67
7.7 Limited Physical Memory 68
7.8 Performance Bakeoff 69
7.9 Generational 72
7.10 Summary 72

8 Conclusion 73
8.1 Summary 73
8.2 Further Work 74
8.2.1 Performance Tuning 74
8.2.2 *N* Generational Collector 75
8.2.3 Alternate Compacting Algorithms 75
8.2.4 Generational Sliding Compaction 75
8.2.5 Compact vs. Sweep Heuristic 75
8.2.6 Limited Memory Testing 75
8.3 Conclusion 76

A Complete Results 77
A.1 Non-Zero Status Words 78
A.2 Phase Timings 80
A.3 Flipped Object Model 82
A.4 Constant Size Header 84
A.5 Free List: Compact vs Sweep 89

A.6 Limited Physical Memory 96
A.7 Performance Bakeoff 98
A.8 Generational 105

Glossary **109**

Bibliography **111**

List of Figures

2.1	A dangling pointer is created as a result of freeing a live object.	4
2.2	Several modules referencing a single data structure.	4
2.3	Internal and external memory fragmentation within a free list.	6
2.4	A Bump Pointer	7
2.5	A set of reference counted objects.	10
2.6	Cyclic Garbage.	10
2.7	A set of objects after a mark phase.	12
2.8	A set of objects before a copying collection.	12
2.9	The same objects during the collection.	13
2.10	A set of objects after a copying collection.	14
2.11	A set of object collected by sliding compaction.	15
3.1	Jikes RVM Object Model	20
3.2	Address Based Hashing.	21
3.3	Interface between the virtual machine and memory manager	24
3.4	An empty full block within a segregated free list.	25
3.5	A block within a segregated free list.	25
3.6	JMTk Mark Sweep Plan	26
3.7	JMTk Semi Space Plan	27
4.1	Object header layout for the compacting collector.	36
4.2	A region.	38
4.3	A chain of regions.	38
4.4	JMTk Sliding Mark Compact Plan	44
5.1	Final block for a size class after compaction.	50
5.2	Final block for a size class with rebuilt free list.	50
6.1	JMTk Generational Mark Compact Plan	55
7.1	Percentage of live objects with non-zero status words for _213_javac.	63
7.2	Phase timings for compacting collectors running _202_jess.	64
7.3	Total time for the original vs. flipped object models.	65
7.4	Total time when segregating arrays and scalars vs. wasting header space.	66
7.5	Mutator time for compacting every *n* GCs (104MB heap).	67
7.6	GC time for compacting every *n* GCs (104MB heap).	68
7.7	Total time for full-heap collectors across heap sizes.	69

7.8 GC counts for full-heap collectors across heap sizes for _202_jess. 69
7.9 Total time for full-heap collectors (41MB heap). 70
7.10 Mutator time for full-heap collectors (41MB heap). 71
7.11 Total time for full-heap collectors (104MB heap). 71
7.12 GC time for generational collectors across heap sizes for _202_jess. 72

A.1 Average percentage: Percentage of live objects that had non-zero status. 78
A.2 Worst case percentage: Percentage of live objects that had non-zero status. 79
A.3 Phase timings: Sliding Mark Compact 80
A.4 Phase timings: Mark Compact Free List 81
A.5 Total time summary: Original Object Model vs. Flipped Object Model. . 82
A.6 Total time: Original Object Model vs. Flipped Object Model. 83
A.7 Total time: Segregation vs. increasing scalar header size. 84
A.8 Average GC time: Segregation vs. increasing scalar header size. 85
A.9 Mutator time: Segregation vs. increasing scalar header size. 86
A.10 GC time: Segregation vs. increasing scalar header size. 87
A.11 GC count: Segregation vs. increasing scalar header size. 88
A.12 Summary: Mark Compact Free List compacting every *n* GCs (41MB). . . 89
A.13 Summary: Mark Compact Free List compacting every *n* GCs (104MB). . 90
A.14 Total time: Mark Compact Free List compacting every *n* GCs. 91
A.15 Average GC time: Mark Compact Free List compacting every *n* GCs. . . 92
A.16 Mutator time: Mark Compact Free List compacting every *n* GCs. 93
A.17 GC time: Mark Compact Free List compacting every *n* GCs. 94
A.18 GC count: Mark Compact Free List compacting every *n* GCs. 95
A.19 Limited physical memory tests with 96MB physical memory. 96
A.20 Limited physical memory tests with 1GB physical memory. 97
A.21 Summary: Bakeoff of all non-generational collectors (41MB). 98
A.22 Summary: Bakeoff of all non-generational collectors (104MB). 99
A.23 Total time: Bakeoff of all non-generational collectors. 100
A.24 Average GC time: Bakeoff of all non-generational collectors. 101
A.25 Mutator time: Bakeoff of all non-generational collectors. 102
A.26 GC time: Bakeoff of all non-generational collectors. 103
A.27 GC count: Bakeoff of all non-generational collectors. 104
A.28 Total Time: Generational Mark Compact vs. Mark Sweep. 105
A.29 Average GC Time: Generational Mark Compact vs. Mark Sweep. 106
A.30 Mutator Time: Generational Mark Compact vs. Mark Sweep. 107
A.31 GC Time: Generational Mark Compact vs. Mark Sweep. 108

List of Tables

7.1 SPECjvm98 Benchmarks. 58

List of Algorithms

4.1 Allocate an object. 37
4.2 Slow allocation path acquiring a new chunk. 39
4.3 Extend the current region. 40
4.4 Create a new region. 40
4.5 Steps in a sliding compacting collection. 41
4.6 Pre-Copying a GC Instance. 41
4.7 Initial trace for marking live objects. 42
4.8 Iterating through the heap. 43
4.9 Second trace to update references. 43

5.1 Steps in a free list compacting collection. 48
5.2 Iterating through the heap. 49
5.3 Sweep processing for each cell. 51

Chapter 1

Introduction

Modern object oriented languages such as C# [ECMA 2002a] and Java [Joy et al. 2000] are becoming increasingly important in industry. With many of the major players getting right behind such languages, including Sun and IBM with Java and Microsoft with .Net, it is very clear that they will continue to be important for some time to come.

These modern languages require runtime systems, such as Microsoft's Common Language Infrastructure [ECMA 2002b] and Sun's Hotspot Java Virtual Machine [Microsystems 2001], that take the responsibility of *dynamic memory management* away from the programmer.

Although there are many factors that influence overall performance, the cost of garbage collection has been found to be very significant [Blackburn et al. 2003].

Compacting collection is a technique utilised by several production virtual machines, including Microsoft's Common Language Infrastructure [Richter 2000a; Richter 2000b] and IBM's Java Garbage Collector [Borman 2002]. Although a preferred technique in industry, compacting collectors have been notably absent from the popular memory management research platform, JMTk.

The purpose of the project is to investigate this important class of collectors, including both implementing one or more compacting collectors in the target platform, and analysing collector performance under various situations.

Blackburn, Cheng, and McKinley [2003] includes a detailed performance comparison of many full heap collectors including *mark-sweep* and *copying* collectors. A performance comparison within a single virtual machine where all things are equal, including many reusable components, provides a solid basis for practical performance comparisons of various techniques. Understanding both the costs and benefits of compacting collection is also a key motivation.

The addition of compacting collection techniques to this research platform will allow detailed side-by-side comparison of these techniques against others. Additionally, the work required to allow such collectors to function within JMTk and Jikes RVM has the potential to allow entirely different classes of collectors to be introduced to the platform by allowing new fundamental operations.

1.1 Contribution

Several compacting algorithms are described in detail, including compacting collectors using both *bump pointer* and *free list* allocation techniques. A *generational* compacting collector is also described. Difficulties encountered during the implementation of these collectors are listed, along with detailed explanations of how they were resolved.

An analysis of the costs and benefits of compacting collection, including comparisons with other full-heap collectors is given. An optimal compaction frequency within a hybrid mark-sweep/compact collector is also investigated.

1.2 Structure

Chapter 2 introduces the concept of *automatic dynamic memory management*, and serves to introduce the reader to many of the techniques described in the literature.

The research platform is then described (chapter 3), and some of the aspects important to this work are described in detail.

The following three chapters (4, 5 and 6) describe three compacting algorithms that were implemented as part of this project.

Chapter 7 contains an evaluation of several aspects of the performance of the collectors. Complete listings of results is included as appendix A.

Chapter 8 sums up the work completed and details further work that could build on the findings of this project.

A glossary is provided at the end of the document.

Chapter 2

Automatic Dynamic Memory Management

This chapter introduces dynamic memory management and then explains how and why the process of managing dynamically allocated memory can be automated.

All programs require memory. Memory can be statically allocated, such as global data structures whose size is known at compile time. Stack allocation, where the lifetime of data is related to where it is positioned in the call stack is also common. Memory can also be *dynamically* allocated by a program. When this occurs the lifetime of the object is not clear.

The region of memory that memory is dynamically allocated from is referred to as the *heap*. Traditionally, the process of allocating and deallocating memory from the heap was performed explicitly as a part of the executing program. The most common example of this is the pair of standard C functions: `malloc` and `free`.

The fact that the programmer has to keep track of all of the dynamically allocated memory it is using is a source of major error in software systems. Two major errors arise from incorrectly managed memory:

Memory Leaks, caused when memory is dynamically allocated but the programmer forgets to deallocate the memory once it is no longer required. If repeated in a long running program the amount of memory wasted or leaked memory can become very significant.

Dangling Pointers, created when memory is deallocated while the program is still using the data that used to be stored at that location. An example of what causes a dangling pointer can be seen in figure 2. Bugs involving dangling pointers can be difficult to resolve as:

- There may be new data allocated in the place of the old data.
- The program may not actively check that the data is valid when it reads from that location, causing errors in completely different sections of the program.

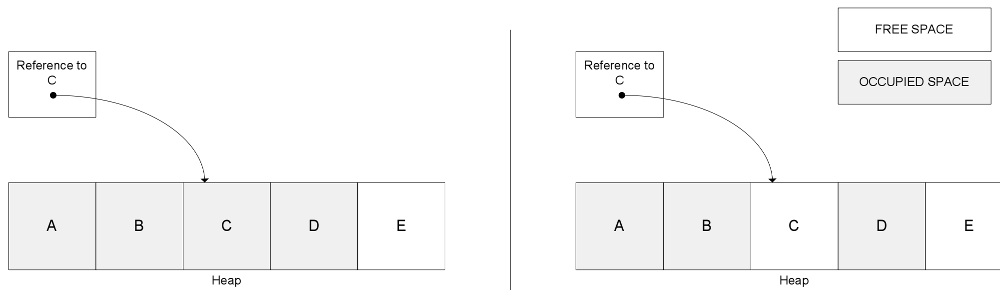


Figure 2.1: A dangling pointer is created as a result of freeing a live object.

The manual management of dynamically allocated memory resources also hinders the ability for software systems to be developed in modular manner [Jones 1996; Wilson 1994]. Consider figure 2, which shows a shared data structure with several modules accessing it. It is not possible for the modules themselves to know if the data structure they are using is being used by other modules without some bookkeeping at a global level.

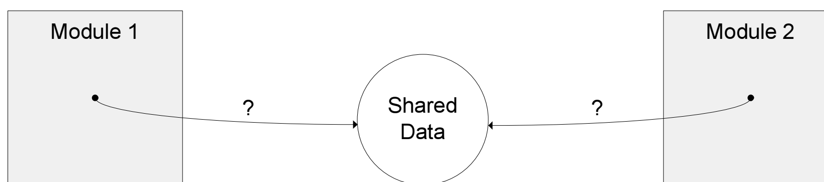


Figure 2.2: Several modules referencing a single data structure.

Jones [1996] also highlights that it can be very difficult for the programmer to determine the liveness of complex data structures. This can lead to inefficient memory usage as the programmer may have no choice but to leave the object alive longer than strictly necessary to ensure that no dangling pointers are made.

It is clear that requiring the programmer to manually manage this memory is undesirable. As the nature of the problem is common to a very large group of programs, it would be beneficial to have this process performed automatically as a service to the application. This automated process of firstly identifying unused dynamically allocated memory as *garbage*, and then making this space available again is known as *garbage collection*.

Although the concept of garbage collection is not new (papers date from 1960), it is becoming of increasing importance in industry and continues to be an actively researched area in computer science. Modern programming languages such as Java and C#, which rely on automatic memory management to execute, are increasing in popu-

larity. The garbage collection mechanisms employed by a runtime system can have a significant impact on overall runtime performance [Blackburn et al. 2003]. That said, it is also clear from such research that no garbage collector is ideal for all situations.

Although many of the principles discussed in this chapter are relevant to garbage collection in general, this section serves only to introduce the key garbage collection concepts that are required in order to more fully understand the rest of the material. There are specific papers and books dedicated to the area such as Wilson [1994] and Jones [1996] that provide a more complete picture of garbage collection, and I would recommend these to an interested reader.

2.1 Object Representations

The object representation, or the way an object is laid out in memory can have a significant impact on both what garbage collection techniques can be employed, and how efficient these techniques can be. For example different garbage collection techniques rely on some of the following:

Pointer field enumeration Given an object reference, is it possible to determine where the pointer fields of that object are stored. These fields may point to other objects and this relationship is important to nearly all collectors.

Storing data for objects A garbage collection technique may require the ability to store information for each object. This type of information can include counters, several flags, or even pointer fields.

Moving to the next object A garbage collection technique may require the ability to move from one object in the heap to an adjacent object.

Finding the first/last object A garbage collection technique may require the ability to discover the first object in an area of memory. This capability, in combination with the previous point, makes it possible to scan through the heap moving from one object to the next.

Typically an object's representation includes a header, which provides both what the garbage collector requires, and whatever else the runtime needs for its own purposes. The rest of the object representation stores the data associated with that object.

2.2 Wasted Memory and Memory Fragmentation

Memory fragmentation occurs when the arrangement of used and free memory within the heap makes it difficult or impossible to allocate objects, even though enough memory would be available given a different arrangement. We take the example of a simple free list in figure 2.3 that has had some data allocated and deallocated. It is clear that the arrangement of the active and inactive data is suboptimal. This figure serves to assist in describing internal and external fragmentation.

Internal Fragmentation Occurs when more memory than is actually required is used to hold an object. This can be seen in cell A in the figure. Although there are only two bytes of data in the cell, the cell size is four bytes, and all of the memory is used to store the two bytes.

External Fragmentation Occurs when the layout of the cells makes it more difficult to service allocation requests, not because there is not enough memory, but because no single contiguous block of memory is large enough to satisfy the request. For example, although there are a total of eight bytes of free memory (excluding the internal fragmentation), it would not even be possible to allocate an object that is five bytes in size.

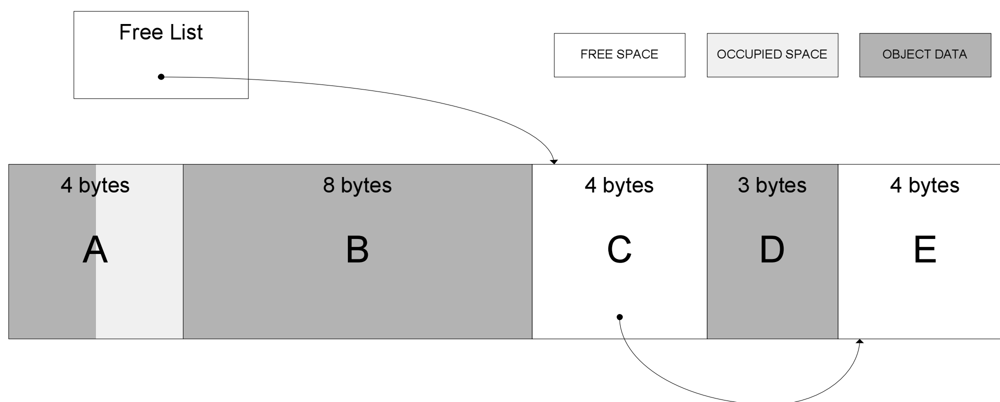


Figure 2.3: Internal and external memory fragmentation within a free list.

In addition to internal and external fragmentation, it is possible for memory to be wasted in other ways. Consider the case where a memory management strategy requires each object to have an additional 16 bytes in its header. This additional memory is neither internal nor external fragmentation. If we compare this to a memory management strategy that requires no additional data in each object's header but results in higher fragmentation, it is not clear which technique actually uses memory more efficiently as a whole.

In order to completely take into account this and other memory overheads and costs involved for a memory management strategy, one simple measure that can be used is to discover the minimum heap size that a set work load can run in for each of the strategies.

2.3 Locality

The way objects are laid out in memory can have a significant affect on the overall performance of a program [Zorn 1989]. There are several important ways that locality

can affect programs, and intelligent allocation mechanisms designed to take advantage of good locality. If an application is actively processing a group of objects and they are all very close together in memory, the application is likely to perform better than if these objects are spread out across the heap. Good locality exhibits high cache hit ratios and low numbers of page faults.

Many factors affect locality, including both program behaviour and the memory management strategy, and it is often the case that good locality is at least partially due to good luck. However, there are several important observations that can assist in optimising the memory management strategy to improve locality in the majority of cases. These include: allocating objects of a similar size or age together, allocating large objects together, and allocating objects that reference each other together.

2.4 Allocation Techniques

The following sections discuss the important classes of allocator. Specific allocators included in JMTk, and additional allocators implemented as part of this project, are discussed later in this document. This section serves only to introduce the reader to basic allocation techniques.

2.4.1 Bump Pointers

The simplest form of allocator is a bump pointer. A bump pointer is essentially a cursor that moves through memory to satisfy each allocation request. While allocation using a bump pointer is very fast, this allocation technique relies on certain types of collectors to be able to reclaim the space occupied by garbage. An example of a bump pointer can be seen in figure 2.4.

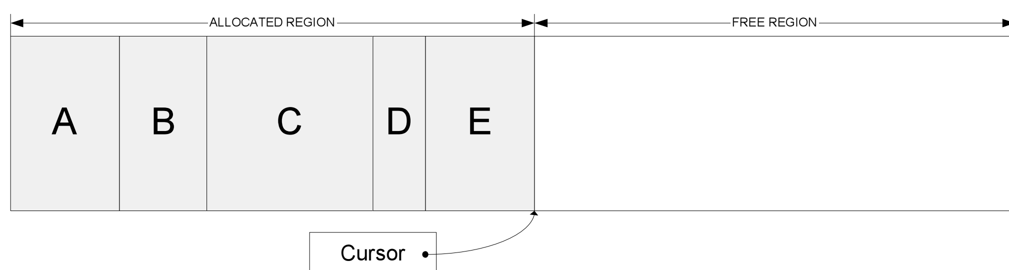


Figure 2.4: A Bump Pointer

2.4.2 Free Lists

A free list is essentially a chain of free cells that are available for objects to be allocated into. Referring back to figure 2.3 one can see that the free list in that case is *C*,

E. When a cell is allocated it is removed from the free list, and when it is deallocated it is returned to the free list. Although an intuitively simple concept, there are many variants on this basic idea, and large papers dedicated to discussing them, most notably that by Wilson et al. [1995]. Good allocation policies have been shown to result in near zero fragmentation under various real world program loads [Johnstone and Wilson 1997].

There are many possible policies for free list allocators. The way cells are split and merged can have a significant impact on memory fragmentation. While splitting cells clearly reduces internal fragmentation, having many small cells all over the place directly increases external fragmentation. Some allocators do not split and coalesce blocks at all, while others place various restrictions on how this can be done. One such approach is a *buddy system* [Wilson et al. 1995], which requires blocks are split and coalesced according to grouping rules. See Wilson, Johnstone, Neely, and Boles [1995] for a more complete discussion of splitting and coalescing techniques.

The way a cell is chosen is also important. For example, when trying to allocate space for an object, do you return the first cell the object fits into – *first-fit*, or do you search through the entire free list until you find the smallest possible cell that the object can fit into – *best-fit*?

It is possible to manage multiple free lists for objects of different sizes or size classes. Such an allocator is known as a *segregated free list*. If a free list is segregated by size it becomes possible to always use a first fit policy, which can speed up allocation. Comfort [1964] describes such an allocator.

When using a segregated free list, fragmentation can become more of a problem [Jones 1996; Wilson et al. 1995]. If a program allocates and then deallocates a large number of objects for one size class, and then from that point on no longer allocates many objects of that size class, the empty free list for that size class will be wasted memory. If the segregated free lists are built on top of a second level allocator of basic blocks, when a block used by one size class becomes empty it can be returned for use by all other size classes. This type of allocator is referred to as a two level allocator [Jones 1996; Boehm and Weiser 1988].

Selection of size classes also has a significant impact on fragmentation. Very few size classes results in poor fits and increased internal fragmentation. Too many size classes results in increased external fragmentation, as each size class may have a partially filled block.

2.5 Garbage Collection Techniques

We have seen how objects are allocated into the heap. This section discusses various techniques for recovering memory after objects become garbage. Garbage collection literature dates from around 1960 and the area is still actively researched.

Although many new and interesting memory management strategies are suggested and implemented, the methods used for comparison between these techniques is often lacking. The use of 'soft' targets, or a comparison involving very few al-

gorithms, heap sizes, or benchmarks are common. Comparisons between memory management strategies running within different runtimes are also flawed, as the differences between other aspects of the runtime such as differences in compilers may be a more important factor in overall performance. Some comparisons are also purely theoretical, which can ignore the potentially significant effects of locality.

Zorn [1989] and Blackburn et al. [2003] are notable exceptions and provide some insight into how best to compare memory management strategies.

Any garbage collection mechanism is required to satisfy two basic principles closely related to the two basic types of dynamic memory management mistakes discussed at the start of this chapter. These basic principles are:

Safety No object will be deallocated until the executing program no longer holds a reference to it.

Completeness All objects that the executing program no longer holds references to will *eventually* be deallocated.

2.5.1 Barriers

Barriers are simply sections of code that are executed implicitly every time some operation occurs. For example a *field write barrier* is executed every time the executing program attempts to write to a field protected by it. The ability to include write barriers on reference fields is an essential component of many memory management strategies. For example a write barrier makes it possible to maintain an accurate reference count for each object. A write barrier also makes it possible for the memory manager to keep track of all references into a particular region of memory, allowing more incremental collection techniques. Zorn [1990] describes this subject in detail.

Write barriers come at a cost in both increased program size and execution time. Blackburn and McKinley [2002] describes how the intelligent management of fast and slow paths within write barriers, combined with a high level of control of how the barrier is compiled, can assist in reducing the cost to negligible levels on some architectures.[1]

2.5.2 Reference Counting

Reference counting collectors[2] work by counting how many references are pointing to each object. An example of a set of objects with counted references is given in Figure 2.5. When there are zero references pointing to an object it can be safely deallocated. The first paper to discuss reference counting is due to Collins [1960].

Reference counting is often used in real-time systems, as the deallocation of unreferenced objects happens during execution of the program, making it unnecessary

[1]Performance is found to be as low as zero on PowerPC architectures but up to 10% on IA32 architectures.

[2]Some chose not to call simple reference counters *collectors* as there is no special phase for a garbage collection, but instead this job is distributed throughout execution.

for the runtime system to pause user code to calculate which objects should be deallocated.

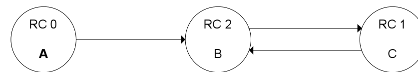


Figure 2.5: A set of reference counted objects.

One major problem with reference counting is that it is fails to collect *cyclic garbage* [McBeth 1963]. Figure 2.6 shows a simple situation wherein which two objects hold references to each other. Although these objects are garbage, they will never have a reference count of zero and so will never be collected. This problem often solved through the use of an additional collector that is called occasionally to collect cycles [Bacon et al. 2001; Levanoni and Petrank 2001].

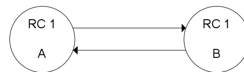


Figure 2.6: Cyclic Garbage.

The runtime overhead of maintaining counts of all references is significant. Deutsch and Bobrow [1976] introduced the idea of *deferred reference counting*, where mutations to references on the stack and hardware registers are ignored except for at discrete collections. This has the implication that objects with a reference count of zero can only be collected at these discrete points in time.[3] This can be achieved by either remembering objects with a reference count of zero (a *zero count table*) or by queueing all decrements during runtime to be processed at these collections. As the vast majority of reference mutations occur on the stack these approaches improve performance considerably.

Levanoni and Petrank [2001] took this idea further and created a new deferred reference counting algorithm. This algorithm is based on the observation that if a reference field P that first pointed to object A, changes to B and finally C, then all the collector has to do is decrement A's reference count and increment C's, ignoring the intermediate reference to B. This concept of *coalescing* multiple pointer mutations into a single mutation can dramatically reduce the work done by the collector.

[3] An uncounted reference on the stack might be referring to an object that has a reference count of zero. Such an object must not be considered garbage.

2.5.3 Tracing Collection

In a runtime environment there is a set of objects that are explicitly live. Such objects include those referenced from the stack and hardware registers, objects stored in static fields, and objects that are referenced directly from within the runtime. This set of objects is referred to as the set of *roots*. This set is essential for a very large group of algorithms that define liveness as the property of being reachable from the roots.

A tracing garbage collection works by following the graph of references between objects from the roots. Any object that is traced is considered live, and any object that is not traced is considered garbage. Note that in order to allow tracing of cyclic structures each live object must have its pointer fields examined *exactly* once; when an already marked object is encountered pointer fields should not be examined. The processing of the trace can be accomplished in several ways:

Recursion The simplest way of implementing a trace is to recursively call the same routine for each pointer field encountered. This method is not suitable for large sets of objects as the call stack can become very deep and causing a stack overflow.

Queuing Have a routine processing a marking queue; for each object we process we push its pointers onto the queue to be processed instead of recursing. This technique requires additional space for storing the queue.

Pointer Reversal There is a space penalty involved with queuing objects for processing as there needs to be a place in memory to store the queues. It is possible to use the pointer fields in the graph to store the information required to trace [Schorr and Waite 1967; Knuth 1973]. This is an expensive operation and temporarily corrupts the data stored in the heap.

An example of a tracing collector is the mark-sweep or mark-scan collector [Jones 1996; McCarthy 1960]. A mark-sweep collector works by first tracing through the graph of objects and marking objects it encounters as live – *the mark phase*. It then deallocates all objects that were not encountered during the trace – *the sweep phase*.

When marking objects it is necessary to store the information about which objects are marked. The simplest approach is to include a mark bit in each object's header. A set of objects traced using a mark bit within each object is shown in figure 2.7. There have been alternate techniques to marking objects to overcome some of the following shortcomings in the use of a mark bit within each object:

- It is not always possible to include a mark bit within the object header as some runtimes do not have bits available for such a purpose, and using an additional word for such a bit can be especially wasteful for small objects.
- Marking objects through the entire heap will dirty many pages of memory, which can incur a significant performance penalty when physical memory is scarce.
- Sweeping can be complicated as it can be difficult to locate unmarked objects.

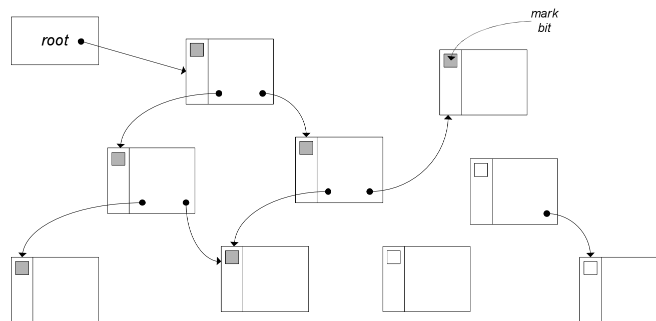


Figure 2.7: A set of objects after a mark phase.

These problems can be alleviated through the use of a *mark bitmap* [Jones 1996]. A mark bitmap is essentially a separate area of memory where mark bits are stored for multiple objects. A bitmap can be used for the entire heap, or alternatively, several smaller bitmaps can be used for different regions of memory. Zorn [1989] places the additional cost of writing to a bitmap at around 12 instructions, as opposed to a single instruction for writing to the header at a fixed offset.

Some collectors do not actively sweep the heap at the end of a collection, but instead make the mutator sweep during allocation time, a technique known as *lazy sweeping* [Jones 1996].

2.5.4 Copying Collection

Copying collection [Jones 1996] is a form of tracing collection in which live objects are copied to a different region of memory during the trace. In general these different regions of memory are labeled the *from space* – the area that contains the objects being collected, and the *to space* – the area that objects are copied into. This process is illustrated in figures 2.8, 2.9, and 2.10. A *forwarding pointer* needs to be stored in each object so that other references to it are updated correctly and only a single copy of each object is made. A simple copying collector is often referred to as a semi-space collector [Cheney 1970].

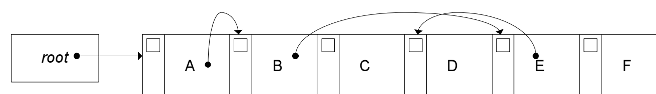
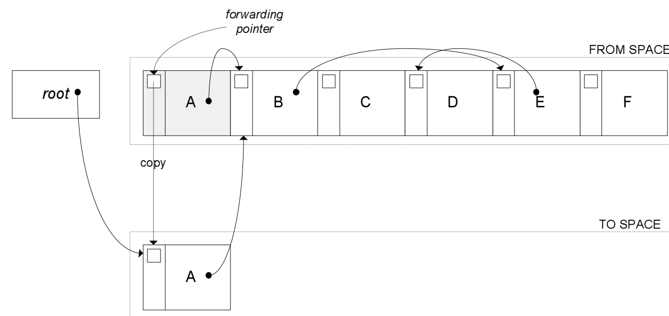


Figure 2.8: A set of objects before a copying collection.

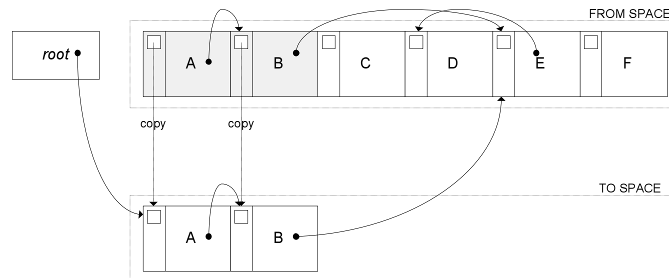
When performing a copying collection the order in which objects are visited becomes more significant due to locality effects [Jones 1996]. If a trace is conducted using a depth-first search then objects that reference each other are placed close to

each other, while if a breadth-first search is conducted then objects that reference each other can be spread broadly across the heap.

One of the most significant properties of a copying collection is that the cost of the collection is relative to the set of live objects, not the size of the heap. This is due to the fact that only live objects are processed, with garbage objects ignored. This property is at the heart of many generational collection mechanisms which are discussed in section 2.5.7.



(a) After object *A* has been traced and copied.



(b) After objects *A* and *B* have been traced and copied.

Figure 2.9: The same objects during the collection.

Copying collection also allows the use of simpler bump pointer allocation mechanisms, which can improve allocation performance, and essentially eliminate fragmentation at each collection. However, copying collectors do not generally make efficient use of memory as they require additional space be reserved for copying objects to – *a copy reserve*. In the worst case, where all objects in the from space are alive, this copy reserve is half the size of the heap. More incremental copying collectors such as Beltway [Blackburn et al. 2002] get around this problem by only collecting sections of the heap at a time. Such methods do however require the use of write barriers.

Copying collectors also have a tendency to use pages of memory in an unusual way [Jones 1996]. While the mechanisms within most operating systems *discard* pages from physical memory to disk on an oldest-first basis, the copying collector *requests* pages on an oldest-first basis. This makes it likely that when running with limited physical memory that each time the allocator moves to a new page it will not be in

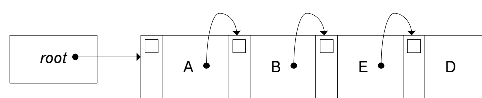


Figure 2.10: A set of objects after a copying collection.

physical memory.

In addition to these requirements, copying collectors are also more demanding of the host runtime [Jones 1996]. In mark-sweep and similar non-moving tracing collectors, all that is required is that live objects are identified. However, in a copying collection all references must be found and updated to the object's new location correctly.

The cost of using a semi-space collector with large, long lived objects can be significant as the objects are repeatedly copied at each collection. Baker [1992] describes the *treadmill*, a collector similar in concept to a copying collector that performs the copying virtually. Instead of managing a to and from space, it manages a pair of doubly linked lists – *from* and *to*. Newly allocated objects are linked into the *from* doubly linked list. During a collection live objects are unlinked from the *from* list and linked into the *to* list. After a collection the *from* and *to* lists are flipped. Due to the additional space wasted to store the linked list this technique is suited best to larger objects.

2.5.5 Compacting Collection

Compacting collectors aim to divide the heap into two contiguous areas: one area containing only live object data, and the other containing only free space. Note that this process does not involve compressing the data in any way.[4] As there is a significant cost involved in moving objects, it is beneficial to attempt to move objects in such a way as to improve locality as much as possible. Generally speaking, compacting algorithms either compact objects in an arbitrary order, in order of age, or by locality of reference (such as in a copying collector).

There are three general approaches to compacting a set of objects of various sizes [Jones 1996]:

Forwarding address algorithms These algorithms use space within each object to store a forwarding pointer that indicates where the object is to be copied to.

Table-based methods These methods use additional data structures, stored either in a separate region of memory or within space taken up by garbage, to store information about where objects are to be relocated.

Threaded methods A chain of object references that point to an object is created, and

[4]Some prefer to use the term *compactification* to avoid any misunderstanding.

when the new location of the object is known this chain is followed and references are updated. Jonkers [1979] describes such a collector.

Sliding compaction algorithms [Richter 2000a; Richter 2000b; Jones 1996] work by essentially removing garbage objects and sliding live objects down in memory. One such algorithm is the Lisp 2 algorithm [Jones 1996] which requires several passes over the heap:

Mark Objects are marked through the use of a transitive closure as in mark-sweep collection.

Calculate Forwarding Pointers Forwarding pointers are calculated for each object by moving through the heap from low to high memory.

Update References Another iteration through the heap looks for all pointer fields and updates them to the new locations stored in object forwarding pointers.

Move Objects A final iteration through the heap copies all objects to target locations.

The effect of a sliding compacting collection is shown in figure 2.11. Objects C and F are identified as garbage, so objects D and E are slid down in memory to form a single contiguous block: A, B, D, E.

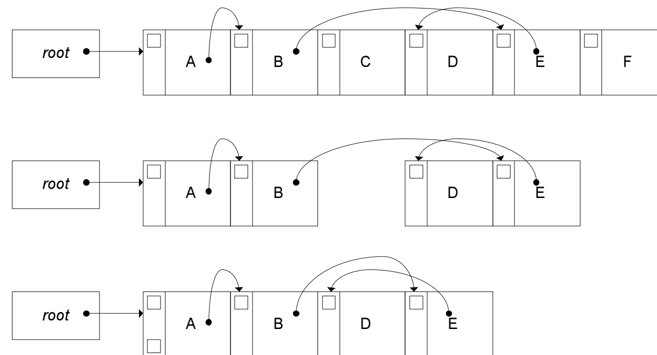


Figure 2.11: A set of object collected by sliding compaction.

2.5.6 Tradeoffs

It is clear from the literature that there is no single best memory management strategy [Blackburn et al. 2003]. Two important characteristics of garbage collection mechanisms are:

Throughput How much garbage can it collect? This factor will have the greatest impact on the overall runtime of the program. In situations where responsiveness is not essential memory management systems are often tuned to maximise throughput.

Responsiveness In hard real-time applications or programs that require heavy interaction with the user it is not acceptable for execution to pause for long periods while garbage collection takes place. In such situations throughput is often sacrificed to place guarantees on maximum pause times.

There is exciting new research [Blackburn and McKinley 2003] to try and develop a collection algorithm with both high throughput and responsiveness, although this algorithm can still be beaten in some situations.

Another classic tradeoff in memory management systems is a time-space tradeoff. The most extreme example is a *null* collector that never tries to reclaim any space. Such a collector, when given a large enough heap, can only be beaten in performance through its ultimately poor locality properties. More reasonably, methods for marking such as pointer reversal can be avoided improving execution speed at the cost of additional space for a marking queue.

2.5.7 Generational Collection

Generational collectors [Ungar 1984; Lieberman and Hewitt 1983] take advantage of several trends identified in the lifetimes of and relationships between objects. Important characteristics of object life cycles [Lieberman and Hewitt 1983; Blackburn et al. 2003] that can assist in optimising memory management strategies are:

- Most objects die young, and when objects do not die young they are likely to live a long time.
- Most references are from new objects to old objects, not vice-versa.
- Large objects are likely to be long lived.

Generational collectors allocate new objects into a *nursery*. Then through use of a write barrier (section 2.5.1) they watch for reference updates within old objects to point to an object in the nursery. This set of objects becomes a pseudo root set to allow a copying collection of the nursery into a *mature* space. As many objects die young the cost of the copying collection is often quite low.

Some generational collectors include many generations [Blackburn et al. 2002; Richter 2000a; Richter 2000b; Appel 1989], while others are limited to two (a nursery and a mature space). All modern high performance virtual machines utilise generational collection in some form.

In contrast with small objects, many large objects that are allocated do not die young. Certain other groups of objects such as objects of a particular class, or objects allocated from a particular point in the program can also have abnormally long lifetimes. In a generational collector such objects are often allocated directly into the mature space rather than the nursery. This technique is referred to as *object pretenuring*.

2.6 Summary

The two basic allocation techniques were introduced. It is clear that there is a trade-off between the speed at which an object is allocated and the ability for the memory manager to reuse the allocated region.

We saw that a bump pointer is the fastest allocator, but it is not possible to reclaim any memory without a moving collection[5]. Many types of free lists exist and trade off speed and memory fragmentation behaviour.

Basic garbage collection techniques, including mark-sweep and copying collection, were introduced. Compacting collectors, that aim to locate all live data together, were also introduced. Finally, generational collectors, that take advantage of characteristics in object lifetimes, were discussed.

It is clear that there are many possible variations on these techniques, but in general most collectors can be described in this basic terminology.

[5]Although it is possible to build a free list from the dead objects within the region, it is generally more efficient to use a high performance free list as there is more control of future allocation.

Chapter 3

Research Platform

This section introduces the research platform that was used for this project. Detailed discussion of aspects of the research platform that were important to the project is included where appropriate.

3.1 Jikes RVM

Jikes RVM, formerly known as Jalapeño [Alpern et al. 1999; Alpern et al. 2000], is a high-performance Java Virtual Machine. One of its major strengths is an aggressive optimisation system, discussed in section 3.1.1. Jikes RVM is an open source project led by a research team at the IBM T.J. Watson Research Centre. Although originally only targeting the PowerPC architecture, the project has more recently [Alpern et al. 2002] added support for the Intel/IA32 architecture.

Jikes RVM is written predominantly in Java. It requires a host Java virtual machine to build a boot image, which is then bootstrapped by a small C program. Jikes RVM is self hosting – in fact it is the *only* self hosting VM written in Java. Most VMs are written in C/C++, and any others that are written in Java require a host runtime to execute.

Jikes RVM includes its own collectors and allocators, but they have more recently been replaced by default with JMTk collectors.

3.1.1 Jikes RVM Compilation

The strength of a virtual machine's compilation system is a significant factor in determining overall runtime performance. Jikes RVM has multiple levels of compilation, from the baseline compiler to a highly complex optimising compiler. While the optimising compiler can produce code that runs much faster, there is a significant cost required to compile the code. In situations where compiled code is run frequently the benefit from using optimised code can outweigh the cost, while in situations where compiled code is run very few times it generally will not.

An adaptive optimisation system recognises this fact and attempts to, at runtime, only spend time using the optimising compiler for selected parts of the program. Arnold et al. [2000] and Burke et al. [1999] describe in more detail the Jikes RVM compilation system.

It is important to note that the adaptive optimisation system makes the behaviour of the runtime non-deterministic. This has implications for debugging memory management and can make this process much more complex. It can also mean that benchmarks can show different performance between runs.

3.1.2 Object Representation

The Jikes RVM object model has been aggressively optimised [Bacon et al. 2002]. Figure 3.1 shows the layout of both scalars and arrays in memory.

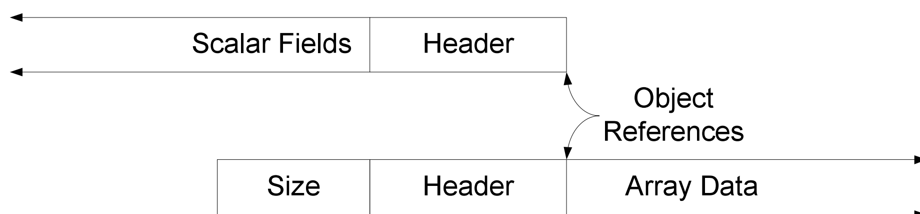


Figure 3.1: Jikes RVM Object Model

This object model has the following properties:

Simple Null Pointer Checks All access to scalars will be at small negative offsets from the object reference. All access to arrays will first need to do a bounds check and read the size of the array, which is also always at a small negative offset. When the object reference is a null pointer, subtracting small negative values from the reference will result in an address in the highest pages of memory. It is possible to protect these pages using hardware traps.[1] So null pointer checks do not need to be performed explicitly.

Fast Array Access The address of any element in an array is simply the element index times the size of each element plus the object reference. There is no need to add an additional value to compensate for any offset from the object header.

Header Offsets Unchanged Between Scalars and Arrays Header information stored in an object is required and accessed independent of whether an object is an array or a scalar (for example a mark bit). All header offsets are the same for both arrays and scalars in this model.

The object header includes the following information:

TIB Pointer A type information block pointer is a reference to the data structure that provides access to essential information regarding the type of the object. This is important for calling methods, finding fields, calculating the size of the objects, etc.

[1]On Linux it is also possible to protect the lowest few pages in memory, which makes it possible to have this benefit without laying scalars backward.

Status Word The contents of the status word depends on the variant of the object model being used. It always includes locking bits to implement thin locks [Bacon et al. 1998] to provide synchronisation for Java objects. It can also contain the object's hash code when address based hashing (section 3.1.2.1) is disabled. There are also several (2-8) *available bits* that are not used and made available to the memory manager to use as, for example, marking bits.

3.1.2.1 Address Based Hashing

According to the Java Language Specification [Joy et al. 2000], it shall be possible get a hash code for any object. The requirements for this hash code are that for any given object the hash code remains the same throughout the life of the application. It is desirable for the hash code values to be distributed such that as few objects as possible share the same hash code as this can increase performance.

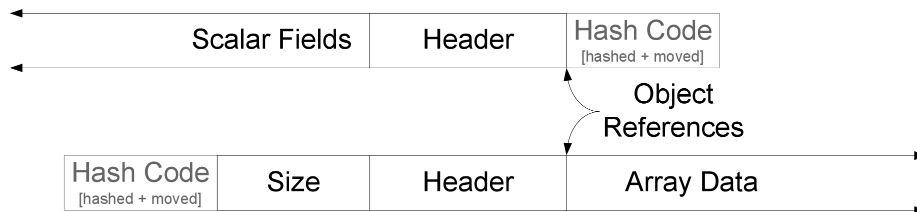


Figure 3.2: Address Based Hashing.

When the Jikes RVM runs using Address Based Hashing it simply uses the objects address as the basis for the hash code. Naturally this solution does not work simply with moving collectors, and was solved through the use of the following states:

Unhashed The object has not yet been hashed.

Hashed The object has had its hash code read. The hash code is returned based on the address of the object.

Hashed and Moved When a *Hashed* object is moved the old hash code is stored as an additional word adjacent to the object. This requires an additional four bytes per hashed and moved object, as shown in figure 3.2.

3.2 Java Memory Management Toolkit (JMTk)

Although there are a wide of automatic memory management strategies that a managed runtime can employ, there remain many similarities at various levels. Similarities exist from the way objects references are traced through the heap, to the implementation of various queues and allocation policies, and the basic management of mapped/unmapped virtual memory resources. The Java Memory Management Toolkit (JMTk) aims to implement these building blocks and provide a toolkit with which various memory management strategies can be implemented.

Along with the toolkit comes a basic abstraction around which the different strategies can be implemented. Naturally this abstraction makes some assumptions about the memory management strategies, but within these bounds the toolkit aims to deliver a large degree of flexibility with as little complexity as possible. With mark-sweep, copying, and reference counting collectors, and free-list, treadmill and bump pointer object allocators, JMTk supports a large percentage of memory management strategies.

3.2.1 The Plan

A plan describes a complete strategy for memory management. The plan manages:

- object allocation requests;
- GC requests;
- queries at GC safe points to determine and trigger a GC if required; and
- the execution of write barriers (optional).

All plans divide the available memory into several *spaces*. The plan then selects allocation and collection policies for these different spaces. This allows enormous flexibility in the implementation of memory management strategies.

For each plan there is an accompanying *header*, defining the requirements of the plan in terms of object header information such as mark bits and forwarding pointers.

There is a large degree of similarity between different plans, and for this reason there is an inheritance hierarchy. Currently JMTk only supports stop-the-world garbage collection, meaning that while a collection is taking place, the execution of other code is paused. This group is an important and large group of collectors. By default, spaces are defined for boot space, immortal object space, and meta data space.

All stop-the-world collectors also share a main collection processing loop, and the logic responsible for the computation of the root set.

JMTk also includes a stop-the-world plan designed to be the base plan for all generational collectors. It defines the regions for a nursery and a mature space, and provides the basic mechanisms to support generational collection. This includes allocating objects into the nursery, copying them into the mature space, and determining appropriate times to trigger both nursery and full-heap collections.

3.2.2 Allocation and Collection Policies

Policies describe the allocation and collection mechanisms for different styles of memory management. Policies are divided into two parts:

Space Policy This level of policy has a one-to-one relationship with each space defined in the plan. The space level policy describes how collections are performed across objects in that region of memory.

Local Policy Local policy is concerned primarily with allocation. For each kernel thread to allocate into a space a local allocator object is required. In some situations (e.g. semi space) a single allocator can work against multiple spaces, although only one at any given time.

JMTk includes implementations of mark-sweep, copying, reference counting, and treadmill allocation and collection policies.

3.2.3 Utilities

JMTk includes many utility or helper classes to perform basic operations. There are two motivations for these classes. Firstly, it reduces the time required to implement new plans and policies in JMTk. Secondly, as two strategies share as much code as possible, the results gained in a comparison of different strategies are more likely to change based on the actual differences between the strategies, rather than implementation differences.

The primary components included are:

Deques An efficient double ended queue implementation is provided. All queues are only synchronised at the page level, to ensure that the majority of queue operations are unsynchronised and fast. Default implementations of single, double and triple value queues are provided, but the base functionality can be easily extended.

Memory Management Basic functionality to manage the use of memory at the page level. This includes ensuring memory is mapped and that the total amount of memory used is accounted for.

Object Allocators Basic object allocators are implemented, including a segregated free list and a simple bump pointer allocator.

Space Allocators Simple allocators are implemented to allow fast allocation of coarse grain chunks of data for use by the object allocators and queues.

3.2.4 VM – MM Interface

The final component of JMTk is the interface with the virtual machine. The implementation of this interface is for a specific virtual machine, and at the time of writing only one implementation exists.

The interface can be split into two components, one for the interface that the memory manager provides to the host virtual machine, and one for the interface that the virtual machine must provide to JMTk. This information is shown in Figure 3.3.

3.2.5 Memory Allocation

One of the greatest strengths of JMTk is the way it tracks the usage of memory. JMTk includes classes that are responsible for acquiring new regions of virtual memory

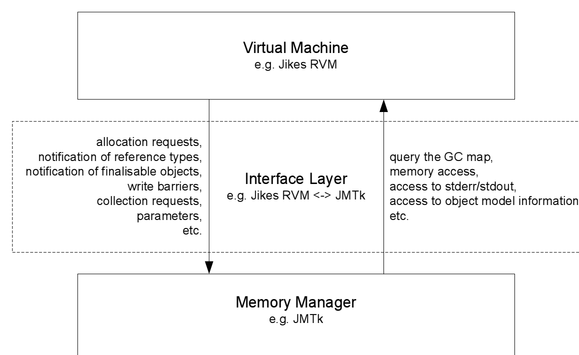


Figure 3.3: Interface between the virtual machine and memory manager

space: VMResources. JMTk also includes classes designed to count the amount of memory currently in use within each memory space. These counters are essential for determining when GCs should be triggered.

JMTk includes implementations of classes designed to provide synchronised access to virtual memory address spaces for both bump pointer allocators (MonotoneVMResource) and free lists (FreeListVMResource). All memory allocation at this level is done in *pages*.

3.2.5.1 Bump Pointer

JMTk includes an implementation of a bump pointer. The bump pointer acquires memory within the memory space in *chunks* 32KB in size. Objects are allocated into the chunk and when it is exhausted a new chunk is requested. If the new chunk is contiguous[2] then allocation continues and is allowed to flow into the new chunk, otherwise the cursor is updated to the start of the new chunk.

This bump pointer is used to allocate objects under copying collection, into immortal spaces, and nurseries within generation collectors.

3.2.5.2 Segregated Free List

JMTk includes an implementation of a segregated free list. See section 2.4.2 for a more complete description of free lists in general. The segregated free list implementation includes 40 size classes,[3] optimised for close fits for objects of the most common sizes, with a worst case internal fragmentation of $\frac{1}{8}$.[4]

Synchronised allocation of memory occurs at the *block* level. Blocks are large grain units of memory that come in seven size classes on a logarithmic scale from 512 bytes

[2]Before this work this check was not made and allocation was always recommenced at the start of the next chunk. A simple fix was suggested and is now part of JMTk.

[3]The segregated free list also supports a second mode using only 28 size classes. As this is not used by any of the collectors studied this mode of operation is not discussed.

[4]These measurements were taken from comments within the code.

to 32KB. Each size class is linked to a single block size, with blocks split and coalesced according to a buddy system. Blocks are doubly linked, and free lists are stored within each block. The block header, as that shown in figure 3.4, includes *next* block, *previous* block, and *free list* pointers. The block header also stores a counter of the number of cells in use, and can be extended to store additional information for a specific allocator, such as a mark bitmap.

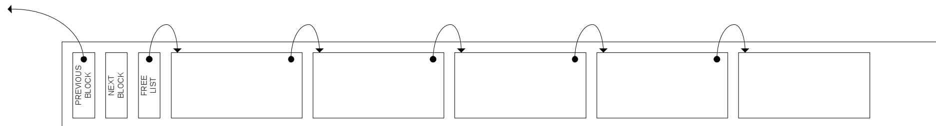


Figure 3.4: An empty full block within a segregated free list.

A block showing a free list after some allocation and collection has taken place is given in figure 3.5. Once all cells within a block are freed, the block can be returned and can be used by other size classes. The ability to reuse blocks within other size classes dramatically reduces total fragmentation [Jones 1996; Wilson et al. 1995].

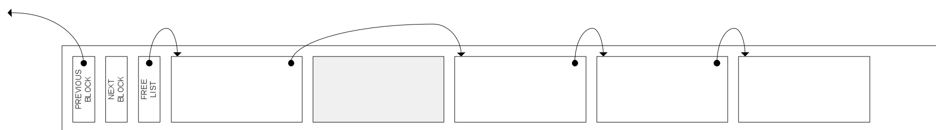


Figure 3.5: A block within a segregated free list.

3.2.5.3 Treadmill

JMTk includes an implementation of Baker's treadmill [Baker 1992] as a *large object allocator*. All allocation requests are rounded to page granularity. A more complete description of a treadmill can be found in section 2.5.4. The implementation within JMTk uses three words for each object to maintain the doubly linked lists – an object can only be in one list at a given time.

3.2.6 Example Memory Management Strategies

How these components all fit together is best demonstrated through some examples. The following sections describe two fundamental memory management strategies that together utilise most of the features of JMTk. The amount of code specific to a single strategy is minimal, in essence it is the glue that connects the different components of JMTk together in a useful fashion.

The rapid experimentation that is then possible makes it simple to try new ideas and implement new algorithms [Blackburn et al. 2003]. The two strategies that have been selected for discussion are Mark Sweep and Semi Space. Together these two collectors utilise many of the key components of JMTk.

All plans define a common base set of spaces:

Boot Space During the process of building the Jikes RVM boot image all objects are allocated into this space. This space is never collected and is essentially immortal. Objects are allocated into the boot space using a bump pointer.

Immortal Space Objects that are known to never become garbage, including some objects allocated by the memory manager and virtual machine are allocated into the immortal space. The immortal space is never collected and placing objects within it reduces the amount of work performed by the collector. Objects are allocated into the immortal space using a bump pointer.

Meta Space Meta data that is created and managed by the memory manager is allocated into the meta data space. This space is where things such as queues are allocated to. Pages are allocated directly to consumers of meta data.

3.2.6.1 Mark Sweep

The mark-sweep plan (shown in figure 3.6) adds a mark sweep and a large object space to the existing spaces. The large object space is required as the segregated free list that is used to allocate objects into the mark sweep space has a maximum size class smaller than the actual limit of object sizes.

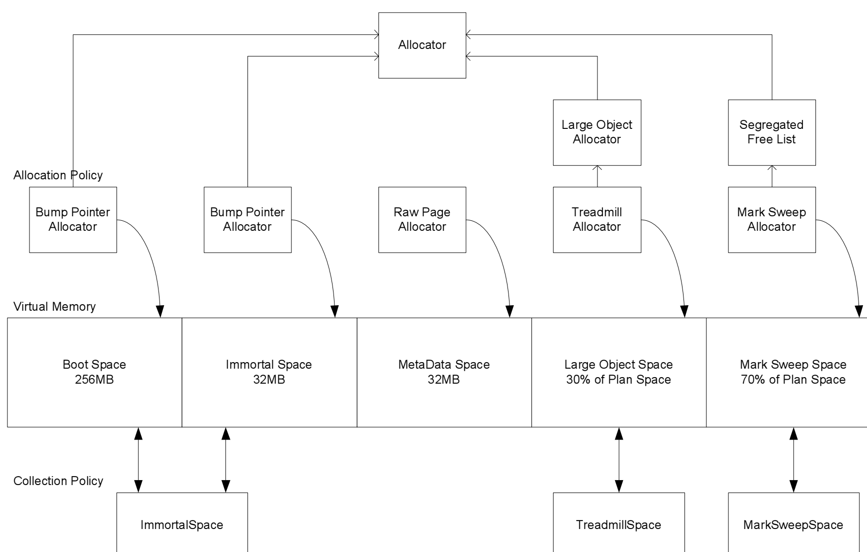


Figure 3.6: JMTk Mark Sweep Plan

3.2.6.2 Semi Space

The semi space plan (shown in figure 3.7) adds a low semi space, a high semi space and a large object space to the existing spaces. Although not required the large object space is included to avoid repeatedly copying large objects between the low and high semi spaces.

At each collection the low and high semi spaces alternate between being the *to* and *from* spaces for under a copying collection. New objects are allocated into the *from* space and live objects are copied into the *to* space during a collection. A single bump pointer allocator is used to allocate objects into the current *from* space.

Collections are triggered when the heap is nearly half full as the worst case where all objects are live needs to be considered.

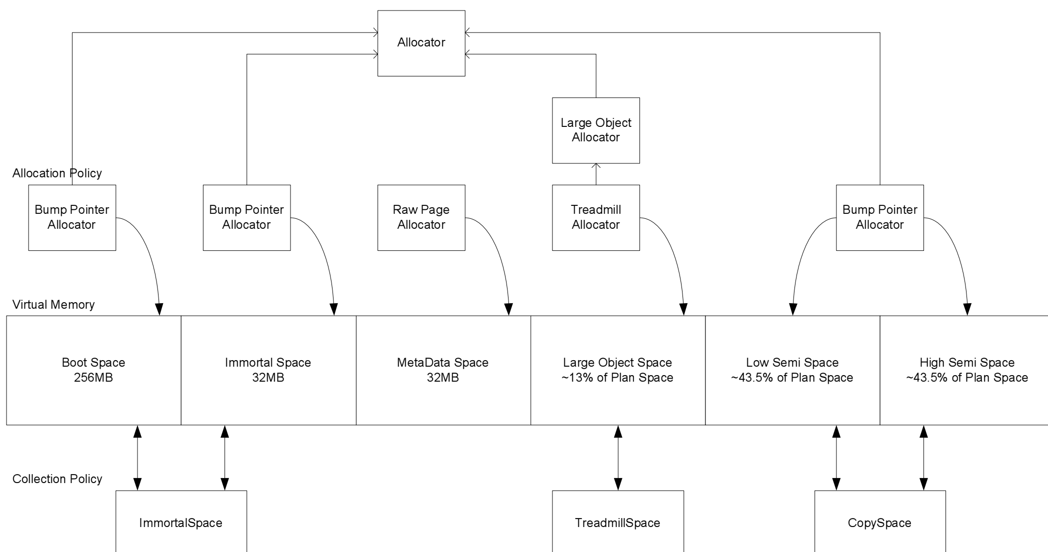


Figure 3.7: JMTk Semi Space Plan

3.3 Summary

This chapter introduced the implementation and experimentation platform. This included JMTk, a platform designed to allow rapid experimentation with memory management strategies while still providing high performance. This platform includes implementations of many reusable components. It is however lacking the important class of compacting collectors.

Chapter 4

A Sliding Compacting Collector

This chapter describes the compacting collector that was implemented during the project. I start by describing the collector at a high level, and then go on to discuss the various issues that arose during the development of this collector. A detailed description of the implementation including discussion of allocation policy, collection policy, and a complete strategy using these policies.

4.1 The Concept

The concept of a sliding compacting collector is very simple. Essentially the idea is to remove all dead objects from the heap, and then slide all objects together so the heap is divided into two areas. One area with a contiguous group of live objects, and the other area containing only free memory.

The algorithm itself is not complex, but it is deceptively difficult to implement in comparison to other collectors. Jones [1996] agrees that there is an additional level of support required by the runtime and compiler in order to support moving collectors over non-moving collectors. In this situation the collector not only has to identify garbage correctly, but it must update all pointers to live objects correctly.

4.2 The Basic Algorithm

Now that it is clear what the objective of the collector is, we move on to an algorithm that achieves this objective. While there are many compacting algorithms the following were the criteria used to select the algorithm:

- Objects could be of variable sizes. *Some compacting algorithms only work with objects of a single size, and although they can be extended by segregating different sized objects, it was decided that such an arrangement would be too different to currently implemented algorithms to make a very useful comparison.*
- As little space as possible should be required in the object's header for GC information. *If it is possible to use the same size object header for comparison between the collector and currently implemented collectors the results would be more valuable.*

- The emphasis for this project is on achieving a good result after a collection, not on making the collection perform well. *The project does not allow sufficient time to both implement and performance tune such a collector. As it is possible to compare execution ignoring the cost of collection, if we focus on the end result rather than the collector then we are likely to have some more valuable comparisons.*

The algorithm that best achieved these goals was a variant of the Lisp 2 Collector [Jones 1996]. Although this algorithm requires a forwarding pointer and this appears against the selection criteria, the reason this cost can be avoided is discussed later in section 4.3.2. The collector has four phases:

1. Mark. *A trace from the roots is made and all live objects are marked.*

2. Calculate forwarding pointers. *Objects are processed one by one in heap order. The algorithm maintains two pointers, one pointing to the current object, and one pointing to the current location in memory the next live object will be copied to.*

3. Update forwarding pointers. *A second trace from the roots is made and references are updated to where the objects will be after being relocated.*[1]

4. Relocate. *Objects are processed one by one in the order they are in the heap and copied to the target address stored in the forwarding pointer.*

4.3 Implementation Issues

4.3.1 Object Representation

A discussed in section 3.1.2, Jikes RVM lays out scalars from high to low memory. As this is the opposite way to arrays, it is impossible to use one simple algorithm to walk the heap for both arrays and scalars. Even if arrays and scalars are allocated to different sections of memory, it would be necessary to write one routine to iterate through arrays from low to high memory, and one routine to iterate through scalars from high to low memory.

This fact, coupled with the fact that the reason scalars were laid out backwards was a historical one,[2] justified making the necessary changes to the object representation to lay scalars out forward in memory.

This change involved significant debugging effort, as much of the code in seemingly unrelated areas made assumptions about the object model. In order to make the runtime work with objects laid forward in memory it was necessary to update several parts of the runtime, including some Java code that was used to generate C header files.

[1] The Lisp 2 Algorithm actually updated the references as it scanned the heap. As JMTk supports multiple spaces some of which do not use this algorithm this was selected as a simple way to ensure any reference into the mark compact space was updated correctly.

[2] The original implementation was for PowerPC AIX, but when running on Linux/Intel it is possible to lay scalars out either forwards or backwards and still get null pointer checks from hardware traps.

4.3.2 Forwarding Pointers

The algorithm requires that for each live object a forwarding pointer is stored, indicating where the object is going to be relocated. It was considered desirable to avoid including an additional word in each object's header in order to store the forwarding pointer. Given an average object size of 32 bytes [Bacon et al. 2002], this would equate to a $\frac{4}{32}$ or $\frac{1}{8}$ space overhead.

Copying collectors in JMTk use the status word to store the forwarding pointer. This is possible as the status word has already been safely copied to the new object. This is not the case in a compacting collector, as the status word can not be copied until the final phase of the collection after pointers have been calculated.

The observation that the status word is usually zero was made (see section 7.2). This made it much cheaper to use the status word but save non-zero values elsewhere to be restored after updating forwarding pointers.

4.3.3 Immortal Type Information

The runtime holds information about types that allow it to (among other things):

- Find where in an object references to other object's are stored.
- Find out the total size of the object instance, including all fields and header information.

The ability to perform these operations is essential during a garbage collection. In Jikes RVM this type information is allocated into the managed heap, even though there are always references retained to these object meaning they will never be collected.

The fact that these objects are stored in the managed heap does not cause problems for copying collectors as the old instance of the object is always available until the end of the collection. It does cause a problem for a compacting collector though as it will temporarily corrupt these data structures by pointing to where the type information *will be*, rather than where it *is*.

As type information is essentially immortal anyway, a decision was made to alter the code to allocate type information into an uncollected region of memory. This required not only making the types involved immortal, but creating new methods to allocate immortal reference offset arrays[3].

4.3.4 Dynamic Linking

The Java language supports complex inheritance and interface relationships between classes. For example, given a parent and a child class, it is possible to change and

[3] Such arrays indicate for an object where reference fields are located in relation to the object reference. Such an array is stored in Jikes RVM as an array of integers, and it was not desirable to have all integer arrays allocated into immortal space.

recompile the parent class without touching the child class. The reason for this is because all of these relationships are *dynamically* determined at runtime.

An important part of information that is dynamically determined at runtime are field offsets. For performance reasons it is desirable to store a field of any object (and all child classes) at the same offset for the header. This means that any access to that field from any compiled code that accessed that field does not need to access any type information data structures.

Rather than dynamically linking all of the information of all of the types that are loaded, Jikes RVM only dynamically links fields as they are accessed. A table filled with these offsets is stored and initialised to a value indicating that the field or method needs to be dynamically linked. The first time such a value is encountered the field offset is resolved and stored.

This value needs to be neither a valid field or method[4] offset; with the original Jikes RVM object model the chosen value was zero. Once scalars were laid out forward in memory, zero became a valid field offset. For this reason a new offset was chosen: minus one.

Because of the fact all memory is initialised to zero automatically in Java, the constant was being ignored in several places. The locations in the code that this was occurring were discovered and a patch was made and is now part of Jikes RVM.

4.3.5 Address Based Hashing

The implementation of address based hashing in Jikes RVM is described in section 3.1.2.1. The problem with address based hashing is that it makes the object header size variable. This complicates the process of iterating through the heap. It is therefore not possible to calculate the reference to object *B* given a reference to object *A* as there is no way to know if *B* is hashed and moved or not until a reference is obtained.

This problem can be resolved through the use of a *dynamic hash offset*. As with address based hashing this means that when a hashed object is moved an additional word is allocated to store the hash code, but it is allocated at the *end* of the object after all of the object's fields – this offset is different for objects of different sizes. This makes the operation of reading the hash code more expensive but allows a fixed size header.

After refinement of the algorithms was made it was found unnecessary to use the dynamic hash offset code as there were enough remaining bits to store a 10 bit hash code in the object header. This is given the two word object model, which is required anyway for the use of the status word as a forwarding pointer.

4.3.6 Segregation of Scalars and Arrays

Arrays and scalars have different sized headers. This stems from the fact that an array header has to store the length of the array in addition to the other information.

[4]The resolution of methods follows a similar and related procedure, but was not affected by the changes in the object model and so is not discussed here.

Although it would have be possible to make the array's length field at a variable location as was done with dynamic hash offsets, the following observations were made leading to the decision not to do this:

- The number of arrays is much more significant than the number of hashed objects,
- Each and every access to an array requires accessing the length as all operations on arrays are bounds checked.

The problem was solved by allocating arrays and scalars separately. Two different approaches were tried:

1. Allocating scalars and arrays into different regions of virtual memory (different JMTk spaces).
2. Allocating scalars and arrays into the same region of virtual memory using different allocators (different JMTk locals). This type of operation is already supported in JMTk as part of the requirement to support parallel operation.

Initially it might appear that there is a performance penalty in allocating arrays and scalars separately. This is however not the case. The reason for this is that each object type has its allocation routine compiled separately. As a given object type or array is always either a scalar or an array, the optimising compiler can optimise out the branching logic. A comparison between extending the header size in scalars to match arrays versus segregating arrays and scalars is given later in this document in section 7.5.

4.3.7 Accessing Objects During a Collection

Some objects need to be accessible to the runtime during a garbage collection. As Jikes RVM runs itself, mutations to internal data structures can occur during a collection. This can be thought of as *partially concurrent* collection as both mutation and collection operations are occurring concurrently – any objects in use by the runtime that are not immortal will reside in the managed heap. Some of these objects, specifically those tied up with the scheduling mechanism need to be accessible during a collection or the runtime may crash.

The problem of updating references to these objects was always something that a copying collector was required to do. However, in a copying collector even though the *new* version of the object had been created, the *old* instance remains accessible at the original reference until the very last step of the collection when the memory is cleared. In a compacting collector this is not the case, as the collector corrupts sections of the heap as it relocates objects.

The types of the objects that come under this category are not all special types, but normal types referred to from a special location. For this reason it is not possible to identify these types at allocation time and allocate them in a different area of memory that is not compacted.

The solution to this problem was to copy such objects into a *safe* area of memory that is not part of the compacting space during the mark phase. For these sensitive objects the initial phase essentially collects as a copying collector, and ensures that these objects will be accessible throughout the collection.

4.3.8 Two-Phase Collection

The algorithm requires multiple phases, one for the initial marking phase, and a second to update references to point to the new locations of objects. Prior to this work, no collectors in JMTk required more than a single phase. All collectors could be abstracted to perform work in the following steps:

Prepare Prepare for the GC at both the global level (per space or virtual memory address region), and at the local level (per processor).

Trace Trace through the objects from the roots, passing each reference in turn to the collector, and then continue to process queues that the collector is pushing references onto.

Release Finish up after a GC both at the local and global level.

While this processing loop supports a very wide variety of collectors, it does not assist a collector to perform a second trace from the roots. A major part of the contribution to JMTk from this project was to develop a neat abstraction for a two-phase collector. This abstraction allows initial prepare-trace-release processing, followed by a second (optional) prepare-trace-release. This allows a collector to collect in a single trace if the algorithm supports it (such a collector is described later in this document in chapter 5).

4.3.9 Finalizable Objects

A finalizable object is one that has some code that it expects the runtime to execute when it has been identified as garbage. JMTk and Jikes RVM include support for finalizable objects.

The problem with this implementation is that there is no place finalization can be executed with the desired effect. If finalization is run after the mark phase then the references in the finalization queues will not be updated to point to the new object locations. Alternately it can not be run after forwarding pointers have been calculated as it would be impossible for the finalization routines to resurrect any objects as required.

The solution was to run the current finalization processing after the mark phase to resurrect objects are required, and to then update the references to the finalization queues after forwarding pointers have been calculated but before any objects are relocated.

4.3.10 Reference Types

Java includes has support for *weak*, *strong* and *phantom* references, that each have specified rules governing when objects referred to by them are considered live. JMTk and Jikes RVM include support all three reference types. Reference processing suffered from the same problems as finalizable objects in that the processing needs to be split into two parts: one executed after the live set has been identified to resurrect objects as appropriate, and one after forwarding pointers had been calculated to update references on the relevant queues.

Due to time constraints, the poor state of the reference processing code, and the limited use of reference types by the benchmarks that were to be used for evaluating the collector, it was decided that reference types would not be supported under the initial implementation of the compacting collector.

The removal of reference types involved altering the code to make all reference types *strong* references (i.e. any reference type would keep the referent alive so long as the reference type was also alive). These changes were applied to all collectors to ensure as fair a comparison as possible.

4.3.11 Multiple Allocators For a Space and Kernel Thread

There is a difference between the header size of scalar objects and arrays because of an arrays size header field. To allow scanning the heap it is necessary to either segregate the allocation of scalars and arrays or waste header space on scalars to make all headers the same size.

The cost of wasting space in scalars was expected to be large, so it was considered beneficial to segregate the allocation of scalar and array objects. As each kernel thread can allocate both arrays and scalars, this led to the situation where a single kernel thread had multiple allocators for a single space.

This caused some problems in JMTk where assumptions are made about having a single allocator for any kernel thread/space combination. To resolve this issue the concept of an *allocator chooser* was introduced. Essentially the allocator chooser is a small allocator that simply looks at whether the request is for a scalar or an array and redirects the request to the appropriate allocator. The sections of the code that relied on this assumption were updated to use this allocator instead.

Results on the benefit of segregating arrays and scalars instead of wasting this space are given in section 7.5.

4.4 The Algorithm

The previous sections gave an overview of the algorithm and some of the key implementation issues that were encountered. This section begins by describing allocation and collection policies for the sliding mark compact collector. It then continues to describe how such a policy can be integrated into a complete memory management strategy within JMTk.

4.4.1 Object Header

This collector uses the standard two-word Jikes RVM object header. Note that due to the issues discussed in section 4.3.5, this collector does not currently work with address based hashing on. The layout of the object header is shown in figure 4.1. The status word is overwritten with a forwarding pointer as required.

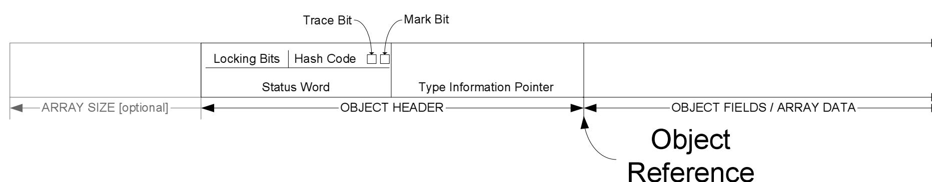


Figure 4.1: Object header layout for the compacting collector.

A class HEADER provides access to this information. It includes methods for querying the status of the trace and mark bits – IS-MARKED and IS-TRACED), in addition to methods for updating them – SET-MARK-BIT, CLEAR-MARK-BIT, SET-TRACE-BIT and CLEAR-TRACE-BIT. There are also methods GET-FORWARDING-POINTER and SET-FORWARDING-POINTER to manipulate the objects forwarding pointer.

4.4.2 Allocation Policy

Because this is a compacting collector, it is possible to always allocate objects using a bump pointer allocator. As this is the shortest and fastest allocation sequence it was selected. It is not possible to use a generic bump pointer as the allocator needs to put in some additional data to assist with scanning the heap.

As with the standard bump pointer this allocator needs to remember *Cursor* and *Limit* pointers. In addition the allocator stores a *Start* pointer, which points to the first region used by the allocator, and a *Region* pointer that points to the region that is currently being used. When there is sufficient space in the current region to satisfy an allocation request, the allocation *fast path* (Algorithm 4.1) is called.

4.4.2.1 Chunks

JMTk is designed to support parallel execution, so it is possible that multiple kernel threads will be running at any given time. As a single shared memory address range is in use by the space, access to new sections of memory need to be synchronised. Rather than synchronising on every allocation, which would be very expensive, memory is allocated to a kernel thread in larger *chunks* of memory from which it can then allocate objects. When one chunk is used up another is requested.

This keeps synchronisation to a minimum while allowing the kernel threads to share a single address space. The chunk size used was 32Kb, which is the same as for

```
ALLOCATOR.ALLOCATE(bytes)
 1  // Where the cursor is before and will be after allocation
 2  oldCursor ← cursor
 3  newCursor ← cursor + bytes
 4
 5  if newCursor > limit
 6    then
 7        // There is not enough space in the region
 8        return ALLOCATE-SLOW(bytes)
 9
10  // Update the allocator's cursor
11  cursor ← newCursor
12  return oldCursor
```

Algorithm 4.1: Allocate an object.

the JMTk bump pointer. The mechanism for the synchronised allocation of chunks used is included as a part of JMTk.

4.4.2.2 Regions

Because of the way chunks may be allocated to different processors, the objects for a single processor may not lie in a single contiguous region of memory. Because of this some bookkeeping information needs to be stored to make it possible for the collector to scan the heap. This information is stored by grouping the chunks allocated by each kernel thread into *regions*.

A *region* is a contiguous region of memory consisting of one or more chunks. Within each region three special pointer fields are reserved to store information for the memory manager. The amount of space reserved and not available for object allocation is at most 0.03%[5] of the total region size. Figure 4.2 shows a region.

The reserved pointers are:

Region Limit This points to the limit of the region. This is used to allow recycling of regions as the limit for the bump pointer can be read from it.

Last Object There is likely to be some amount of free space at the end of each region that is not used. The last object pointer allows the collector to know where the last object is and stop there. This pointer need not be a valid object reference, but no valid object reference may exist within the region after this address.

Next Region This allows the memory manager to find the next region.

[5] 12 bytes out of a minimum region size of one chunk (32Kb).

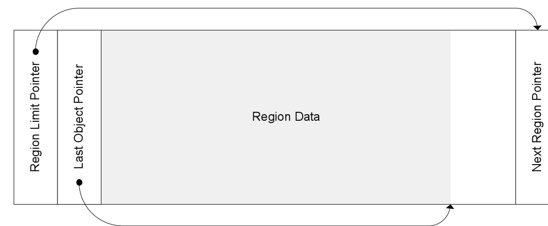


Figure 4.2: A region.

When multiple regions exist for a single allocator, they are connected as a singly linked chain as shown in Figure 4.3.

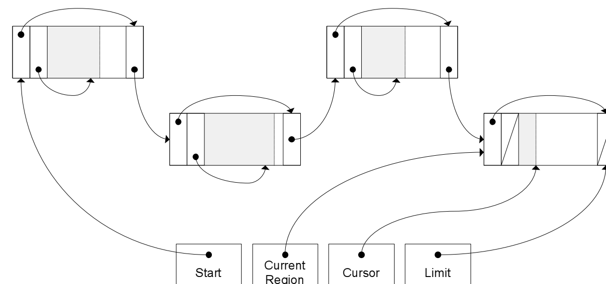


Figure 4.3: A chain of regions.

4.4.2.3 Extending and Creating Regions

When there is insufficient memory available within the current region to service an object allocation request, the ALLOCATE-SLOW() method is called (Algorithm 4.2). This method will first check to see if the current region is the last on the region chain, and if not it will simply recycle the next region in the chain and return.

Once all existing regions have been exhausted, the method will acquire a new chunk of memory. If this is the first chunk of memory, or it is not contiguous with the allocator's current region, a new region will be created (Algorithm 4.4). If the newly acquired chunk forms a contiguous area of memory with the allocator's current region it will extend the region to include the new block (Algorithm 4.3).

4.4.3 Collection Policy

This section describes in detail how the information stored in *regions* and the runtime's type information data structures can be used to perform a sliding compacting collection.

```
ALLOCATE-SLOW(bytes)
 1  if region.next = NIL
 2     then // Allocate a new Chunk
 3           C ← ALLOCATE-CHUNK()
 4
 5           if C = NIL
 6              then // We need to free up some more space.
 7                    TRIGGER-COLLECTION()
 8                    C ← ALLOCATE-CHUNK()
 9                    if C = NIL
10                       then ERROR(" OutOfMemory ")
11
12           if (limit + addressSize) = C
13              then // If this is contiguous with the current region extend it
14                    EXTEND-REGION(C)
15
16              else // Otherwise create a new region
17                    CREATE-NEW-REGION(C)
18
19     else // Recycle an old region
20           region ← region.next
21           limit ← R.limit
22           cursor ← C + 2 * addressSize
23
24  // Allocate the object
25  return ALLOCATE(bytes)
```

Algorithm 4.2: Slow allocation path acquiring a new chunk.

The garbage collection proceeds as described in Algorithm 4.5. Note that although these phases are called in the order listed, the order is only guaranteed by a correctly written *plan*. Furthermore, all steps except for CALCULATE-FORWARDING-POINTERS and MOVE-OBJECTS involve processing within other spaces which is interleaved with processing within the mark compact space. This section describes in detail how the processing specific to the space is carried out. More information on how this collection policy can be integrated into a memory management strategy is described later.

It is important to note that some of the collection phases are processed based on the space or virtual memory address range a object is in, while others are processed based on the set of objects allocated by a single allocator. In all algorithms listed those related to the space and allocator have been explicitly named as such.

```
ALLOCATOR.EXTEND-REGION(C)
1  // Extend the limit of the region
2  region.limit ← C + chunkSize − addressSize
3
4  // Allow the allocator to continue allocating into this region
5  limit ← region.limit
```

Algorithm 4.3: Extend the current region.

```
ALLOCATOR.CREATE-NEW-REGION(C)
 1  if first = NIL
 2    then  // Set the allocators starting region
 3          first ← C
 4    else  // Update the old region
 5          next ← C
 6          lastObject ← cursor
 7
 8  // Set the limit of the new region
 9  region.limit ← C + chunkSize − addressSize
10
11  // Update the allocator to work within the new region
12  limit ← region.limit
13  cursor ← C + 2 ∗ addressSize
```

Algorithm 4.4: Create a new region.

4.4.3.1 Pre Copy GC Instances

Object instances used by the collector itself need to be available during the collection to allow processing to continue. These objects are related to threads and the Jikes RVM scheduler, although in general any runtime may place this requirement on its memory management system. If these instances are moved at unsafe times during a collection, it is possible and likely that the runtime will crash or behave unexpectedly. The processing of GC instances is handled by the plan, and it will call PRE-COPY-GC-INSTANCE (Algorithm 4.6) for each GC instance it discovers within the Mark Compact space. This processing occurs at the *space* level.

The collector assumes that the plan provides a *safe space*, into which the GC instances can be copied. The mark compact collector is the first collector within JMTk to require such an area.

```
MARK-COMPACT()
 1  Pre − Copy − GC − Instances()
 2  MARK()
 3  CALCULATE-FORWARDING-POINTERS()
 4  UPDATE-REFERENCES()
 5  MOVE-OBJECTS()
 6  RESTORE-STATUS()
```

Algorithm 4.5: Steps in a sliding compacting collection.

```
SPACE.PRE-COPY-GC-INSTANCE(obj)
 1  if HEADER.SET-MARK-BIT(obj)
 2    then // We have marked the object
 3           HEADER.SET-TRACE-BIT(obj)
 4           new ← COPY-OBJECT(obj, PLAN.SAFE-SPACE)
 5           HEADER.SET-FORWARDING-POINTER(obj, new)
 6
 7    else  // The object has already been copied
 8           new ← HEADER.GET-FORWARDING-POINTER(obj)
 9
10  return new
```

Algorithm 4.6: Pre-Copying a GC Instance.

4.4.3.2 Mark Live Objects

Once all objects required to be fixed for the collection have been copied out of the mark compact space, all live objects are marked. As with other collection policies in JMTk, live objects are identified by finding the transitive closure of the graph of objects from the set of roots. As each object is first encountered, it is placed onto a processing queue. As objects are taken off the processing queue, their pointer fields are inspected and traced. Algorithm 4.7 describes the processing for each object traced in the mark compact space. This processing occurs at the *space* level.

The basic tracing mechanism used, including the management of the processing queues, is part of JMTk. Specific to the mark compact collector is the code responsible for tracing objects within the mark compact space, including the code to ensure each object is processed exactly once. The initial trace needs to consider the possibility of objects moved due to pre-copied GC instances.

```
SPACE.TRACE-OBJECT-MARK(obj)
 1  if HEADER.SET-MARK-BIT(obj)
 2    then // We have marked the object
 3          PLAN.ENQUEUE(obj)
 4
 5    else if HEADER.IS-TRACED
 6            then (obj)
 7                  // The object has been copied (a pre GC instance)
 8                  new ← HEADER.GET-FORWARDING-POINTER(obj)
 9
10            else
11                  new ← obj
12  return new
```

Algorithm 4.7: Initial trace for marking live objects.

4.4.3.3 Calculate Forwarding Pointers

At this point all live objects in the heap have been marked, and during this phase the collector determines where objects need to be moved in order to leave the heap in a compacted state. The algorithm does this by iterating through the heap and then essentially *reallocating* objects into the heap. The process of iterating through the heap is shown in Algorithm 4.8. To do this *reallocation*, a reference to the current target region and a target cursor are kept. The logic is essentially a subset of the allocation sequence, although it is never possible that a new chunk needs to be acquired – the compacted set of objects can be at most the same size as the current set of objects. Note that the forwarding pointers are stored within each object's status word. When the status word is non-zero it needs to be saved and restored at the end of the collection.

4.4.3.4 Update References

Once all forwarding pointers have been calculated, it is necessary to update all references to objects within the mark compact space. This process occurs by tracing the live set of the heap and updating references during the trace. The mechanisms to do this are again part of JMTk. Note that after this phase *all* references to objects within the space have to be considered invalid until objects have been moved. The tracing logic is shown in Algorithm 4.9.

4.4.3.5 Move Objects

Once all references have been updated it is the role of the collector to again iterate through the heap (Algorithm 4.8) and copy objects to their forwarded location. This

```
ALLOCATOR.ITERATE-HEAP()
 1  region ← start
 2
 3  // Loop through all regions
 4  while region ≠ NIL
 5  do current ← region + 2 ∗ addressSize + headerSize
 6     currentLimit ← region.last
 7
 8     while current < currentLimit
 9     do // For every object
10        objectSize ← VM.GET-OBJECT-SIZE(current)
11
12        PROCESSING FOR EACH OBJECT HERE
13
14        current ← current + objectSize
15
16     region ← region.next
```

Algorithm 4.8: Iterating through the heap.

```
SPACE.TRACE-OBJECT(obj)
1  if HEADER.CLEAR-MARK-BIT(obj)
2     then // We are the first to trace to object
3          PLAN.ENQUEUE(obj)
4
5  new ← HEADER.GET-FORWARDING-POINTER(obj)
6
7  if new ≠ NIL
8     then return new
9     else return obj
```

Algorithm 4.9: Second trace to update references.

phase also must ensure that the last object pointer fields for the regions are updated to reflect the new locations of objects within the heap.

As objects are moved to their new locations header information is reset, including clearing mark and trace bits and zeroing the status word.

4.4.3.6 Restore Status Words

When forwarding pointers were set for objects that had non-zero status words the status was saved. All of these saved status words are now restored to ensure that the state of all objects is consistent after the collection.

4.4.4 Complete Memory Management Strategy

The memory management strategies build upon the basic spaces (boot space, immortal space and meta space) described in section 3.2.6. It provides three additional spaces as shown in figure 4.4:

Mark Compact Space This is the default space into which new objects are allocated. Sliding mark compact allocation and collection policies are used.

Large Object Space To avoid compacting large objects, a large object space using treadmill allocation and collection policies is used for objects larger than 16KB.

Safe Space As is a requirement for the sliding mark compact collector, a small mark-sweep collected safe space is provided to copy GC instances to during a collection. This space uses the standard mark-sweep allocation and collection policies.

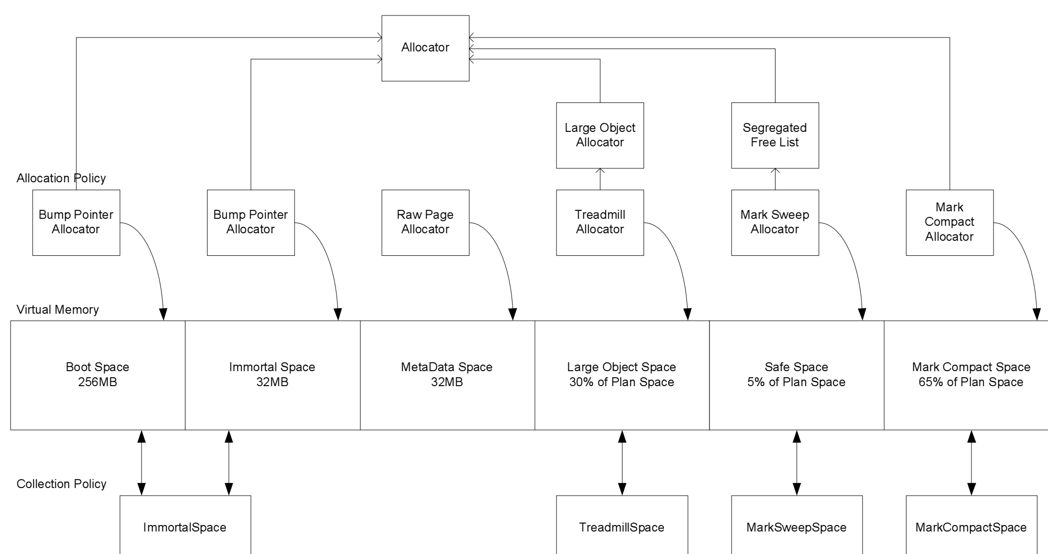


Figure 4.4: JMTk Sliding Mark Compact Plan

4.5 Summary

A simple concept – sliding live objects down in memory to fill gaps once containing garbage objects – was translated into detailed object allocation and collection policies. Several implementation issues faced in the development of these policies were identified and discussed. Finally, a method by which these policies could be used within a complete memory management strategy was provided.

Chapter 5

A Free List Compacting Collector

This chapter describes a modified version of the sliding compacting collector that uses a free list to allocate objects. The previous chapter described in detail the design and implementation of a sliding compacting collector. This chapter aims only to identify where this algorithm differs from the previous.

5.1 Motivation

After the initial implementation of the sliding compacting collector (chapter 4) it was discovered that the cost of a compaction was very high. Because of this it was considered desirable to be able to perform a less expensive collection most of the time and compact only every n collections. This was not possible with the sliding compacting collector as it used a bump pointer for allocation.

The cost of iterating through the heap is also expected to be reduced when working within a free list. This is because it is not necessary to compute object sizes, but instead the constant cell size for the size class can be used.

5.2 The Algorithm

The aim of this collector is to take each size class of a segregated free list and compact it so that fragmentation is reduced as low as possible given the current numbers of live objects for each size class.

There are a few important differences between this algorithm and the sliding mark compact algorithm:

- Allocation is performed using the standard JMTk segregated free list instead of a customised bump pointer.
- Iteration through the heap and the calculation of forwarding pointers is different.
- A more complicated procedure to reset the allocator – each free list needs to be rebuilt.

- Some logic to determine when to perform a sweep, and a sweeping routine needs to be written to allow compacting to happen only every n collections.

5.2.1 Sweep vs. Compact

Because this collector is designed to either compact or sweep objects, the steps for the collection change to those shown in algorithm 5.1. The decision of whether to compact can be made based on various criteria, but in this project the only criteria was a compaction frequency, n. This is a configurable parameter that allows compaction to happen never ($n = 0$), always ($n = 1$), or any other specified frequency.

```
MARK-COMPACT-FREE-LIST(compact)
 1  PRE-COPY-GC-INSTANCES()
 2  MARK()
 3  if compact
 4    then  // We are compacting this time
 5          CALCULATE-FORWARDING-POINTERS()
 6          UPDATE-REFERENCES()
 7          MOVE-OBJECTS()
 8          RESTORE-STATUS()
 9
10    else  // We are sweeping this time
11          SWEEP()
```

Algorithm 5.1: Steps in a free list compacting collection.

5.2.2 Heap Iteration

The new method for iterating through the heap is shown in Algorithm 5.2. The process is quite simple. For each size class the doubly linked list of blocks is iterated and each cell within each block is processed. This method of iterating through the heap does not require any calculation of object size information, and because of this should be less expensive. This expectation is supported by the results given in section 7.3.

5.2.3 Rebuilding Free Lists

The segregated free list implementation in JMTk stores free lists at the block level. After compacting the cells within these blocks it is essential that the free lists are updated. Clearly, as this is a compacting collection, only the last block for each size class can have any free cells. This means that the block level free list can be cleared for all blocks except the last.

There are then two possibilities for the final block. The first is that the final block is completely full; in this case the free list for this block can also be cleared. The second

```
ALLOCATOR.ITERATE-HEAP()
 1  s ← 0
 2  while s < sizeClasses
 3  do block ← firstBlock[s]
 4     cellSize ← cellSize[s]
 5     blockSize ← blockSize[s]
 6
 7     while block ≠ NIL
 8     do
 9        current ← block + headerSize + blockHeaderSize[s]
10        limit ← block + blockSize
11
12        while current < limit
13        do  // For every object
14
15           PROCESSING FOR EACH OBJECT HERE
16
17           current ← current + cellSize
18
19        // Loop through all blocks
20        block ← block.next
21
22     // Loop through all size classes
23     s ← s + 1
```

Algorithm 5.2: Iterating through the heap.

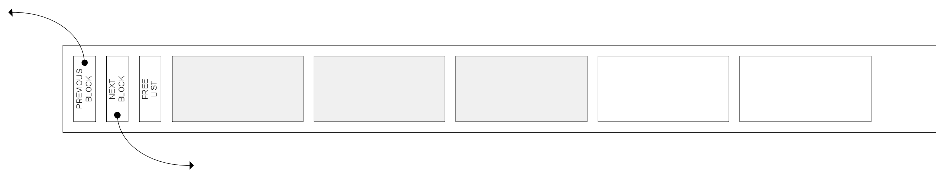


Figure 5.1: Final block for a size class after compaction.

possibility (shown in figure 5.1) is that there are several empty cells at the end of the last block. In this case the memory for these cells needs to be cleared and the free list rebuilt. The desired result is given in figure 5.2.

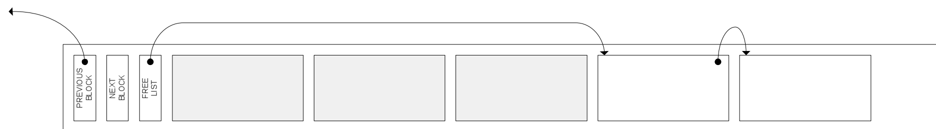


Figure 5.2: Final block for a size class with rebuilt free list.

Once all block free lists have been updated the segregated free list needs to be updated to point to the newly constructed free list within the current block, or cleared in that case that the last block was completely full.

5.2.4 Sweeping Free Lists

In order to allow periodic compactions, a non-compacting collection over the free list needed to be implemented. Due to the time available a very simple sweep was chosen. The heap is iterated as in algorithm 5.2, and for each object algorithm 5.3 is run. The sweep walks through all cells after the mark phase and frees any that are unmarked. The sweep also resets the header information on all objects to allow the next collection to proceed correctly.

5.3 Summary

The chapter described a simple method of extending the sliding compacting collector to work with a free list allocator. The use of this allocator allowed for periodic compaction, rather than forcing a compaction every collection.

```
ALLOCATOR.SWEEP(obj)
1  if HEADER.IS-MARKED(obj)
2    then
3         FREE-CELL(obj)
4
5    else
6         HEADER.RESET-BITS(obj)
```

Algorithm 5.3: Sweep processing for each cell.

Chapter 6

A Generational Compacting Collector

This chapter describes a generational compacting collector. As JMTk provides generational collection out-of-the-box, this process was relatively simple.

6.1 Motivation

High performance Java and C# runtimes almost invariably utilise generational collectors. Blackburn, Cheng, and McKinley [2003] show the performance improvement that can be gained from combining collection policies with a nursery within Jikes RVM and JMTk. Investigating how a generation compacting collector performs relative to other generational collectors, and comparing this to how the compacting collector performs against other full heap collectors might provide some insight into the characteristics of both collectors.

6.2 Two-Phase Collection

The compaction algorithm requires several phases, while other existing JMTk collectors required only a single phase. This was already solved for full-heap collectors, but not for generational collectors. As JMTk is structured, generational collectors are a specialisation of stop-the-world collectors. In order to implement two phase generational collectors, a copy of the existing JMTk generational collection code was created and set to inherit from the two-phase collector, instead of the stop-the-world collector. Because of JMTk's abstraction, this resulted in creating two-phase generational collectors essentially for free, with hardly any of the two-phase generational code needing to be changed.

6.3 The Algorithm

Due to time constraints and the observed similarities between the algorithms, only one of the free list and sliding compacting collectors was selected to be used in a gen-

erational collector. Preliminary performance comparisons showed that the free list compacting collector outperformed the sliding compacting collector. This, coupled with the fact that the free list compacting collector offered greater flexibility by providing the option to sweep, led to the decision to use it as the algorithm to test in within a generational collector.

6.4 Complete Memory Management Strategy

All generational strategies build upon a basic generational plan that adds a mature space and a nursery to the basic spaces (boot space, immortal space and meta space) described in section 3.2.6. The spaces provided by the generational mark compact collector beyond these three basic spaces are shown in figure 6.1:

Nursery This is the space into which most new objects are allocated. Large objects are allocated directly into the large object space.

Large Object Space To avoid compacting large objects, a large object space using treadmill allocation and collection policies is used for objects larger than 8KB.

Mature Space Any objects that survive a nursery collection are allocated into the mature space. This mature space uses free list mark compact allocation and collection policies.

Safe Space As is a requirement for the sliding mark compact collector, a small mark-sweep collected safe space is provided to copy GC instances to during a collection. This space uses the standard mark-sweep allocation and collection policies. Note that to save space on the figure the allocation and collection policy for the safe space have been omitted.

6.5 Summary

Because of JMTk, implementing a generational version of an existing collector is trivial. In this case there was additional work combing the two-phase and generational concepts.

The complete memory management strategy for the generational compacting collector was described.

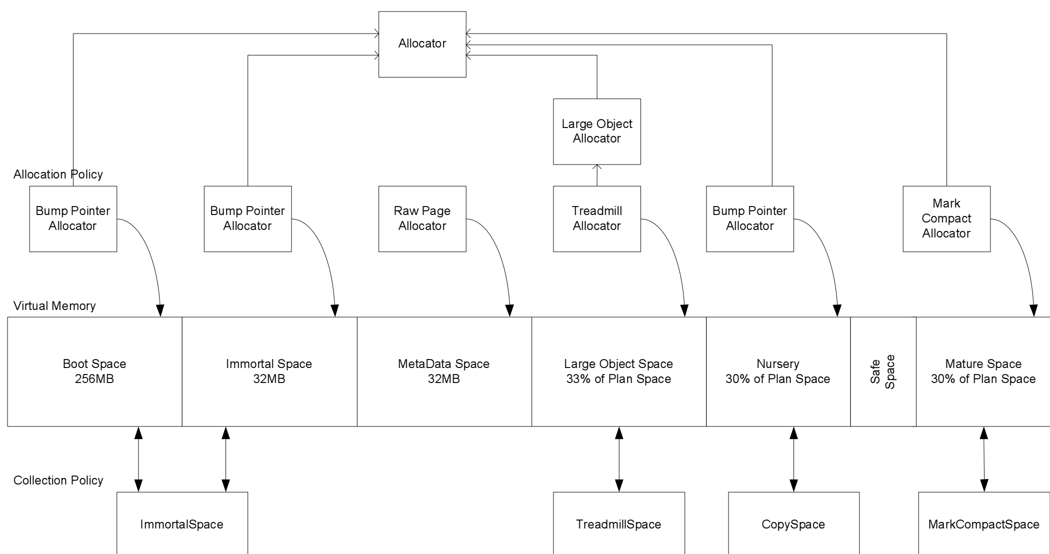


Figure 6.1: JMTk Generational Mark Compact Plan

Chapter 7

Performance Evaluation

This chapter describes the methods used to analyse the performance of memory management strategies utilising the compacting collectors. It also provides an overview of the results for each experiment that was conducted. A more detailed discussion of these results is given in the following chapter. It is important to note that not all results are given in this chapter; the complete results are given in appendix A.

7.1 Benchmarking Methodology

This section describes the techniques used to benchmark the virtual machine running various memory management strategies. The platform, the benchmarks, and the memory management strategies that were compared are described.

7.1.1 Experimental Platform

All experiments were performed on a single machine. The specifications of this machine are:

- Intel(R) Pentium(R) 4 2.6 GHz with hyper-threading, an 800MHz FSB, a 64 byte L1 and L2 cache line size, an 8KB 4-way set associative L1 data cache, a 12K L1 instruction trace cache, and a 512KB unified 8-way set associative L2 on-chip cache.
- ASUS(R) P4C800 motherboard (Intel(R) i875P chipset).
- 1GB of memory (Matched pair of 512MB modules)
- Red Hat Linux 9, with SMP Kernel version 2.4.20-19.

Some experiments required that the size of physical memory was limited. This was achieved by passing appropriate parameters to the Linux kernel at boot time. Full details of what was done are given in the relevant results section.

7.1.2 Benchmarks

Eight benchmarks from the SPECjvm98 suite [Standard Performance Evaluation Corporation 2003] were used. The amount of pressure these benchmarks place on the memory manager varies. Generally speaking, these benchmarks are not an ideal set of benchmarks to fully test a memory management strategy [Blackburn et al. 2003]. However, these are real world applications that do place some pressure on the memory manager.

The SPECjvm98 benchmarks used are listed in table 7.1 with information regarding minimum heap sizes and allocation loads as specified on the SPEC web-site [Standard Performance Evaluation Corporation 2003].

Name	Description	Min. Heap	Alloc.
_201_compress	LZW compression on real file data.	20MB	334MB
_202_jess	Java Expert Shell System (logic puzzles).	2MB	748MB
_205_raytrace	Single threaded ray-tracer.	*not specified*	
_209_db	Memory resident database.	16MB	224MB
_213_javac	JDK 1.0.2 Java compiler.	12MB	518MB
_222_mpegaudio	MP3 decompressor.	*insignificant GC*	
_227_mtrt	Multi-threaded version of _205_raytrace.	16MB	355MB
_228_jack	Parser generator	2MB	481MB

Table 7.1: SPECjvm98 Benchmarks.

As can be seen by the minimum heap sizes and allocation workloads the most GC intensive benchmarks are _202_jess, _228_jack and _213_javac. Actual minimum heap sizes and allocation loads will change between configurations, but the figures above serve as a guide.

Each benchmark is executed twice within a single instantiation of the runtime, and only the second execution is counted. This approach aims to factor out the majority of the cost of compilation, most of which will occur during the first execution. The virtual machine was always executed in single processor mode.

For any given experiment the benchmark was run five times and the fastest time was used. There is not a large amount of variation between each run and this fastest run is considered the least disturbed by other system factors [Blackburn et al. 2003].

7.1.3 Configurations

Several different configurations were developed and tested during this project, in addition to existing configurations within Jikes RVM and JMTk. This section aims to provide the reader with the necessary information to relate benchmark results back to the theory discussed in the previous chapters.

All configurations were compiled in FastAdaptive mode. This forces ahead of time optimised compilation of the runtime itself and required system classes. It also enables the adaptive optimisation system for the most efficient combination of compiled

code at runtime. All assertion checking is also switched off to improve speed. This is the recommended configuration for benchmarking purposes.

Because of different choices that were made during development several branches of the code were made. Jikes RVM and JMTk were branched independently and then combined to make the various configurations. The branches for Jikes RVM were:

Original Object Model This is the original Jikes RVM source code modified to make all reference types strong references (see section 4.3.10), type information immortal (see section 4.3.3), and address based hashing disabled (see section 4.3.5). This object model *does not* support the compacting collectors. It also includes the fixes to dynamic linking that makes the flipped object model (see section 4.3.4).

Flipped Object Model This is the original object model with scalar objects laid out forward in memory (see section 4.3.1).

Constant Header Object Model This is the flipped object model with the single change that scalars and arrays have the same size header. This was achieved by wasting space for scalar objects as described in section 4.3.6.

And the branches for JMTk were:

Base This is the original JMTk source with all required additions and modifications for the compacting collectors. This can be used with either the original or flipped object models; however, the compacting collectors will not work with the original object model.

This branch includes the required changes to allow immortal type information (see section 4.3.3), two phase collection (see section 4.3.8), multiple allocators for a single space and kernel thread (see section 4.3.11), and collectors that place more strict requirements on object accessibility during collections (see section 4.3.7).

It also includes implementations of the sliding compacting collector (see chapter 4), the free list compacting collector (see chapter 5), and the generational compacting collector (see chapter 6).

Constant Header This is the same as the base JMTk branch, with the exception of the compacting collectors being modified to have arrays and scalars allocated together (see section 4.3.6). This could have been included in the *base* branch, but was not due to time constraints.

These were combined to give the following set of configurations. All results are recorded against one or more configurations from this set.

MarkSweep This is the standard JMTk mark-sweep configuration, using the *original object model* and the *base* JMTk code.

MarkSweep-FOM This is the standard JMTk mark-sweep configuration, using the *flipped object model* and the *base* JMTk code.

SemiSpace This is the standard JMTk semi-space configuration, using the *original object model* and the *base* JMTk code.

SemiSpace-FOM This is the standard JMTk semi-space configuration, using the *flipped object model* and the *base* JMTk code.

GenMS This is the standard JMTk generational mark-sweep configuration, using the *original object model* and the *base* JMTk code. A variable sized nursery is used.

GenMS-FOM This is the standard JMTk generational mark-sweep configuration, using the *flipped object model* and the *base* JMTk code. A variable sized nursery is used.

MarkCompact This is the sliding compacting collector (see chapter 4), using the *flipped object model* and the *base* JMTk code.

MarkCompact-CH This is the sliding compacting collector (see chapter 4) with arrays and scalars allocated together. This uses the *constant header object model* and the *constant header* JMTk code.

MarkCompact-Inst This is the sliding compacting collector (see chapter 4), using the *flipped object model* and the *base* JMTk code. Additional instrumentation code has been added to gather statistics of object counts.

MarkCompactFL-*n* This is the free list compacting collector (see chapter 5), using the *flipped object model* and the *base* JMTk code. The n is the frequency at which compactions occur. 0 – never compact, 1 – always compact, $n > 1$ – compact every n collections.

MarkCompactFL-CH This is the free list compacting collector (see chapter 5) with arrays and scalars allocated together. This uses the *constant header object model* and the *constant header* JMTk code. It is based on the MarkCompactFL-1 configuration and as such compacts every GC.

MarkCompactFL-Inst This is the free list compacting collector (see chapter 5), using the *flipped object model* and the *base* JMTk code. Additional instrumentation code has been added to gather statistics of object counts. It is based on the MarkCompactFL-1 configuration and as such compacts every GC.

GenMC This is the generational compacting collector (see chapter 6), using the *flipped object model* and the *base* JMTk code. A variable sized nursery is used.

7.1.4 Heap Sizes

In order to more accurately determine the properties of a memory management strategy it is essential to test the strategy at various heap sizes. Some strategies will make efficient usage of space and run well in a small heap, while others will use additional space to allow faster execution in a larger heap. For all benchmarks a minimum heap

size was determined that allowed execution of the benchmark under most configurations. Then the benchmark was run at various heap sizes ranging from the minimum to six times the minimum.

7.1.5 Collection of Metrics

When executing a benchmark using JMTk, it is possible to pass in command line options that make the runtime print out statistics during execution. The information that is printed out includes information about:

- The finish time: END.
- The start and finish times of each collection (GC^i_{start} and GC^i_{end}), counted from the when the runtime was started.
- Timing information for each of the phases of each collection.
- The number of live objects with non-zero status words found during each collection for the compacting collectors (GC^i_{saved}).
- The total number of live objects found during each collection for the compacting collectors (GC^i_{live}).

The output is then processed by a series of scripts to produce charts of the following statistics. Note that this includes factoring out the first run from the statistics. The total number of collections performed during the benchmark run, GC_{count} is also calculated.

Total run time. This is the total amount of time the benchmark took to run. It is given simply by:

$$TOTAL_{time} = END - RUN_{start} \tag{7.1}$$

GC time. This is the total amount of time spent performing collection activity during execution. It is given by:

$$GC_{time} = \sum_{i=1}^{GC_{count}} (GC^i_{end} - GC^i_{start}) \tag{7.2}$$

Mutator time. This is the amount of time that the benchmark is running. It is given simply by:

$$MUT_{time} = TOTAL_{time} - GC_{time} \tag{7.3}$$

Pause time. The pause time is an indicator of how responsive the collection strategy is to the benchmark. High average pause times indicate that the collection strategy may less suitable for real time or highly interactive applications.

$$PAUSE^{i}_{time} = \frac{GC^{i}_{time}}{GC^{i}_{count}} \tag{7.4}$$

Percentage of objects with non-zero status words. By using the information from each collection regarding the number of live objects and the number of these objects with non-zero status, we can calculate a percentage.

$$PCT_{saved} = \frac{\sum_{i=1}^{GC_{count}} GC^{i}_{saved}}{\sum_{i=1}^{GC_{count}} GC^{i}_{live}} \tag{7.5}$$

It is possible to derive other statistics from this information or captured through adding information to what is output by the runtime. Such information includes maximum pause times and the benchmark's reported runtime.

7.1.6 Missing Results

There are several situations that arise that mean some results could not be obtained:

Small Heap Sizes Different memory management strategies require different amounts of memory to perform the same amount of work. For this reason in some situations there will only be results for a subset of the configurations for the smallest heap sizes.

Research Virtual Machine Jikes RVM is a *research* virtual machine, and as such is not expected to be as stable as production virtual machines. Specific benchmark runs at specific heap sizes can cause the runtime to crash. Sometimes this can be avoided by simply re-running benchmarks, as the errors are sometimes intermittent – most likely due to the effects of the adaptive optimisation system. Any such errors were investigated by me but where problems occurred within unmodified code, time constraints meant that not all issues could be resolved.

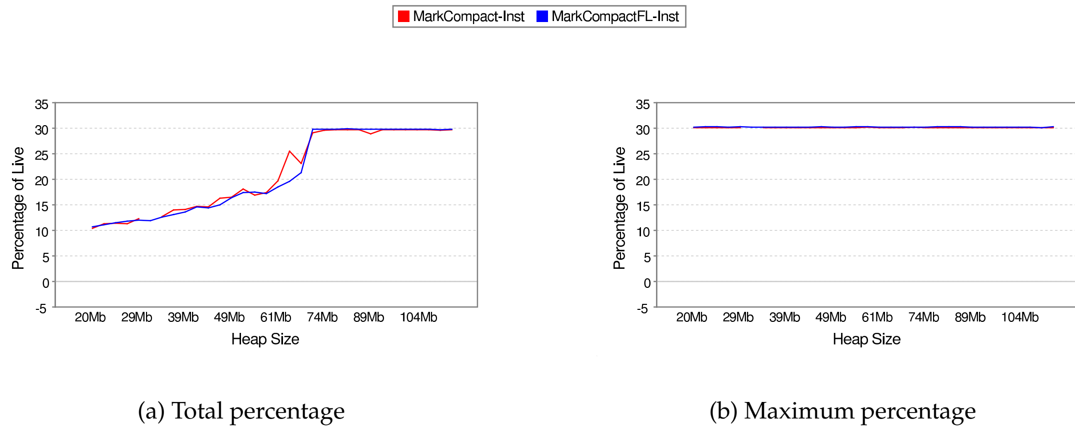
In particular this has affected the stability of the generational configurations, with many errors arising in the optimising compilers on smaller heap sizes.

Any results missing for reasons different to those listed above will be discussed separately within the relevant results sections.

7.2 Non-Zero Status Words

During a compacting collection, the status word of each live object is used to store the object's forwarding pointer. In the case that the status word is non-zero, it must be saved and restored after the collection. The alternative to this approach is to allocate

additional header space to store the forwarding pointer. This issue is discussed in detail in section 4.3.2.



(a) Total percentage

(b) Maximum percentage

Figure 7.1: Percentage of live objects with non-zero status words for _213_javac.

The percentage of objects with non-zero status words was calculated for all benchmarks for both full-heap compacting collectors with the necessary instrumentation (MarkCompact-Inst and MarkCompactFL-Inst). With the exception of _213_javac (see figure 7.1), all benchmarks recorded a total percentage less than 2%, and a peak percentage less than 9%. A peak percentage of close to 30% is seen in _213_javac on all heap sizes; this is most likely due to systematic usage of hash tables or other data structures that require calculation of object hash codes. There is no significant difference between MarkCompact-Inst and MarkCompactFL-Inst.

Two different factors lead to non-zero status words (see section 4.4.1 for information on the object header): synchronisation and hashing. On synchronised objects the locking bits in the header switch on and off as threads access the object. This type of behaviour explains the noise in the charts for most benchmarks, particularly in the worst case scenario. Hash codes behave quite differently, and once a hash code is set the status word for that object must be saved at every subsequent collection it remains live.

It is interesting to note the lower average percentages for _213_javac in small heap sizes. This is caused by early GCs where not all objects have yet been hashed, thereby reducing the average. With larger heaps, fewer or no GCs are triggered before all such objects have been hashed.

By pushing status words onto a queue instead of storing the forwarding pointer in the header, we are saving one word in every object at a cost of two words for each object that requires its status to be saved. If we assume a worst case scenario where $\frac{1}{3}$ of all objects have a non-zero status word, there is a space saving of $\frac{1}{3}$. This is because an extra word per object is a cost of 1, while an extra cost of two words for only one third of live objects is a total cost of $\frac{2}{3}$. This is in *addition* to the fact that no space is wasted for garbage objects with non-zero status words. Note that this is only a comparison of *space* costs, although it is thought that there would be minimal

time cost differences between these approaches. In the average case where only 2% of status words need to be saved, the use of an additional forwarding pointer would require 25 times more space.

Complete results are given in the appendix in section A.1.

7.3 Phase Timings

The time taken to perform the collection is divided into the following phases:

Roots The time taken to determine the root set for both traces.

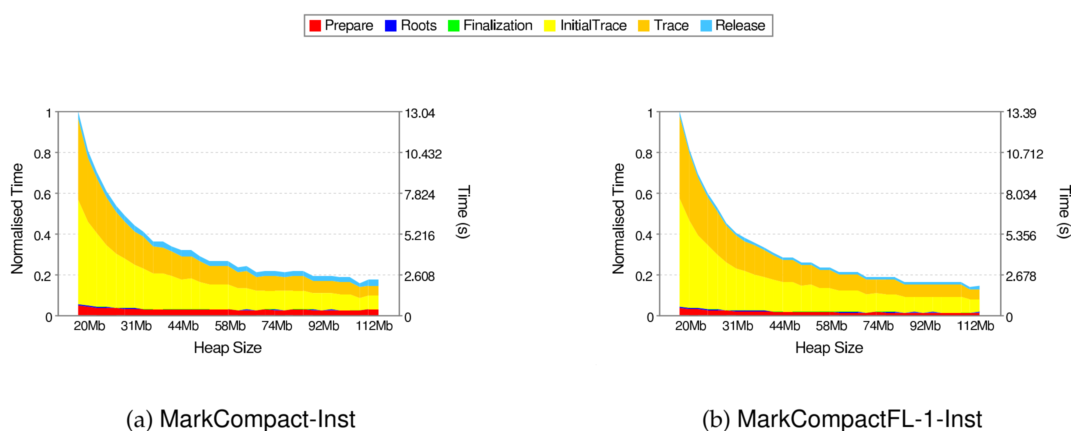
Prepare The time taken to prepare for the compacting collection; the largest cost of this is the calculation of forwarding pointers to prepare for the second trace.

Finalization The time spent processing finalizable objects. This includes both looking through finalizable objects to see if they are garbage and need to be resurrected for finalization, and updating the finalization structures after forwarding pointers have been calculated.

Initial Trace The time spent tracing objects during the mark phase of the collection.

Second Trace The time spent tracing objects while updating pointers into the compacted space.

Release The time spent finishing up from the collection; the largest cost of this is copying objects to their new locations.



(a) MarkCompact-Inst

(b) MarkCompactFL-1-Inst

Figure 7.2: Phase timings for compacting collectors running _202_jess.

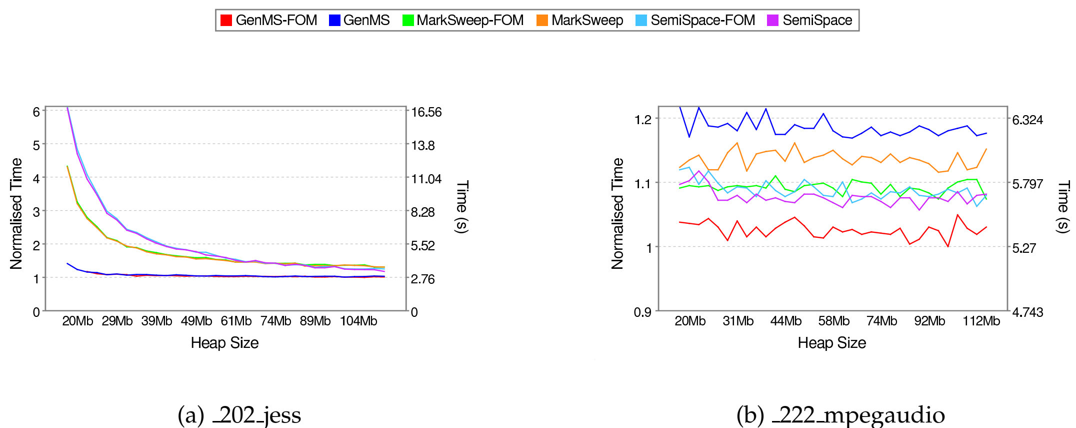
The total time spent in each phase was calculated, and then was normalised against the slowest run. The resulting graph gives an indication of actual time and the percentage of total collection time spent in each phase. Figure 7.2 shows the result for each configuration running _202_jess. Several general observations were made:

- The cost of the two tracing phases far outweighs the cost of all other phases. This makes them primary candidates for initial optimisation work. It also makes investigation into implementing a compacting collector that updates forwarding pointers without requiring an additional trace attractive.
- The cost of the prepare and release phases is always less for the free list compacting collector. This was an expected result as the cost of iterating the heap should be less when not calculating each objects size.
- The initial trace is always more expensive that the second trace. Both the amount of work performed and the number of nodes reached for each of these phases is similar. Further investigation into why there is such a clear difference is justified by this result.

Other benchmark runs showed similar results, although the percentage of time in the release phase is greater for those benchmarks that place a higher GC load on the virtual machine. Complete results are given in the appendix in section A.2.

7.4 Flipped Object Model

After making the change to lay scalars out forwards in memory, some basic tests were carried out to ensure that there was no associated cost. The configurations GenMS, MarkSweep, and SemiSpace were compared with their counterparts using the flipped object model: GenMS-FOM, MarkSweep-FOM, and SemiSpace-FOM.



(a) _202_jess

(b) _222_mpegaudio

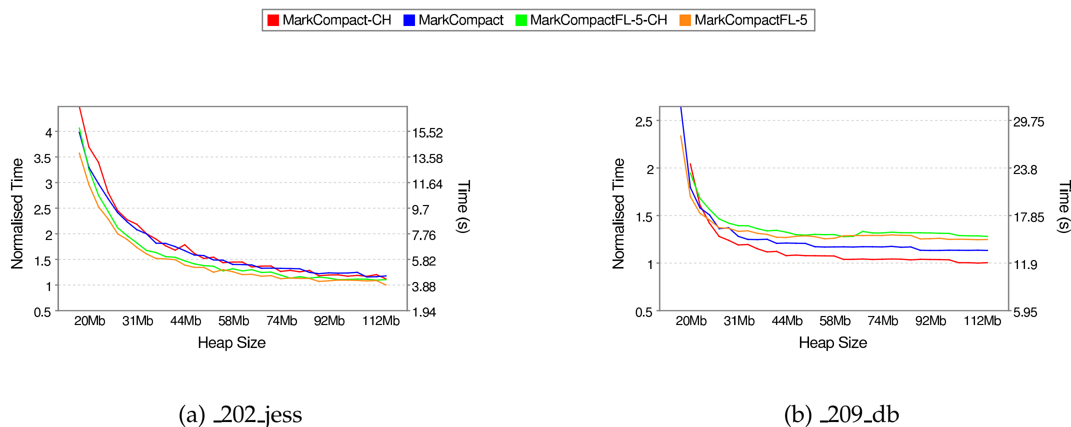
Figure 7.3: Total time for the original vs. flipped object models.

Under more GC stressful benchmarks such as _202_jess (figure 7.3a), there was no perceivable difference between the performance of memory management strategies in the original object model compared to the flipped object model. In _222_mpegaudio (figure 7.3b) there was a noticeable difference between GenMS and GenMS-FOM. The amount of garbage collection work in this benchmark is considered insignificant and the time constraints did not allow for any further investigation.

These results indicate that, as expected, there is no reason to not lay scalar objects forward in memory on the Linux/IA32 platform. Complete results can be found in the appendix in section A.3.

7.5 Constant Size Header

The possibility of using an additional word within the header of each scalar to make object header sizes the same for both array and scalar objects was considered. This allows the use of a single allocator for arrays and scalars, as it allows heap iteration. MarkCompact and MarkCompact-CH were compared alongside MarkCompactFL-5 and MarkCompactFL-5-CH (see section 7.6 for details why FL-5 was used). For this additional cost to be considered, it must result in faster execution times. This was not the case in most benchmarks.



(a) _202_jess

(b) _209_db

Figure 7.4: Total time when segregating arrays and scalars vs. wasting header space.

Figure 7.4b shows the only significant win for a constant header size configuration with MarkCompact-CH outperforming MarkCompact by between 10% and 15% in _209_db. All other benchmarks showed either a slight advantage for non CH builds, or very closely matched performance figures such as with _202_jess (figure 7.4a).

There were problems with MarkCompact-CH failing when running _205_raytrace and _227_mtrt. As the MarkCompactFL-CH did execute these correctly, and the differences between constant header configurations and their counterparts were consistent between the free list and sliding compacting collectors, this result was not looked into further.

These results showed that there was no measurable performance gain from using the constant sized header, so the approach was discarded for the rest of the project.

Complete results can be found in the appendix in section A.4.

7.6 Free List: Compact vs Sweep

One of the motives for the development of the free list compacting collector was the ability to compact the heap periodically, instead of at each collection. Once this is possible, the question of how to determine when to trigger a sweep versus a compacting collection arises. Initially, the only heuristic that was considered was to compact every n collections.[1] This test involved comparing all of the MarkCompactFL-n configurations.

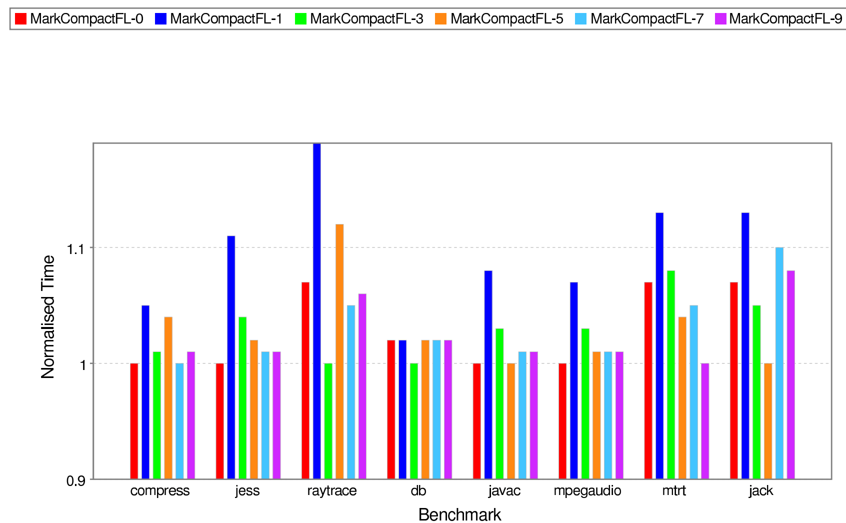


Figure 7.5: Mutator time for compacting every n GCs (104MB heap).

As would be expected the results showed a trade-off between increased collection times (more compacting GCs) and improved mutator performance. Figure 7.5 shows the mutator time, where the most aggressive compacting configuration, MarkCompactFL-1 performs very well in the GC intensive benchmarks _202_jess and _213_javac. However, the increase in GC time illustrated in figure 7.6 is much more significant than this benefit, making MarkCompactFL-1 perform poorly overall.

These results supported the selection of MarkCompactFL-5 as the optimal selection when using compaction frequency as the heuristic. MarkCompactFL-5 was within a close margin of the best performer in terms of GC time, mutator time, and total time for all benchmark and heap size combinations.

Complete results for each of the tests run are in the appendix in section A.5.

[1] Any other sensible heuristic would require gathering additional information from within the segregated free list and was not considered in this project due to time constraints.

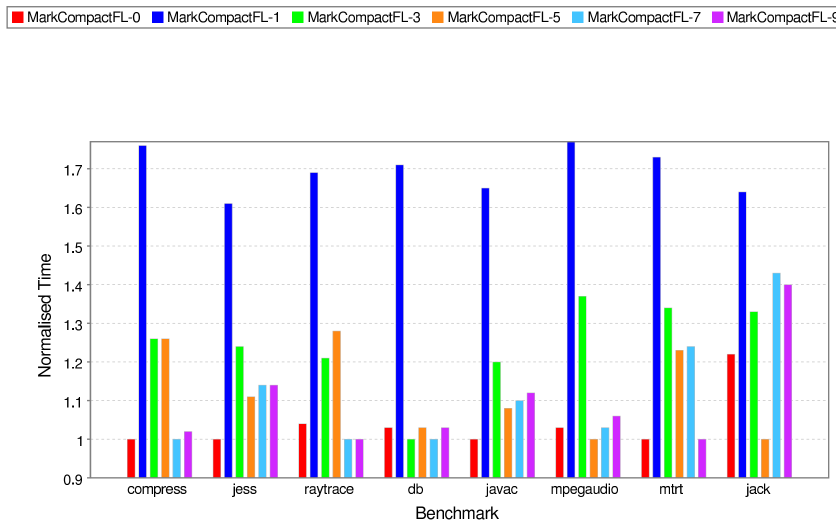


Figure 7.6: GC time for compacting every *n* GCs (104MB heap).

7.7 Limited Physical Memory

Compacting collectors aim to eliminate memory fragmentation after a collection, thereby minimising the number of pages in use by the virtual machine. Because of this it would be expected that compacting collectors exhibit good locality. Limiting physical memory should allow the exploration of locality issues at the page level, as opposed to tests with ample physical memory that predominantly test cache locality effects. For this experiment MarkCompact, MarkCompactFL-1, MarkCompactFL-5, MarkSweep-FOM and SemiSpace-FOM were used.

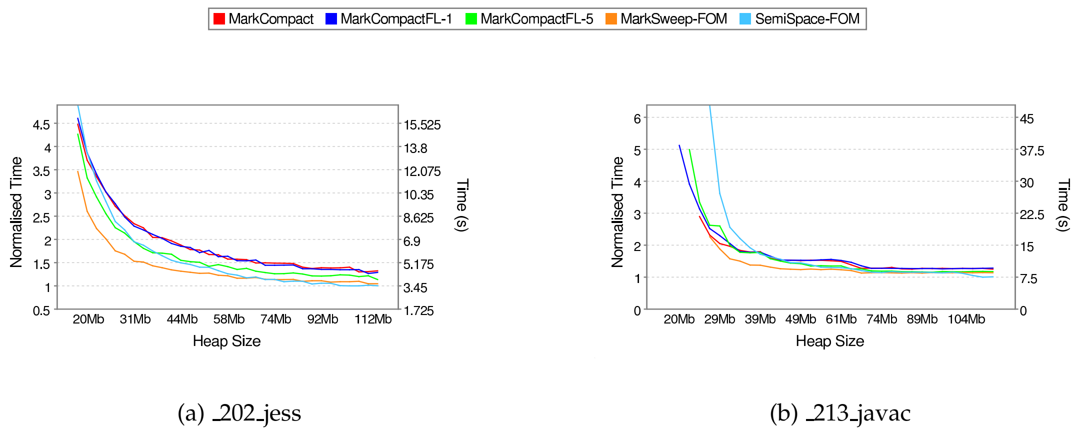
The set of configurations were run at a single heap size of 41MB in two situations: one where ample physical memory was available, and one where only limited memory was available. Although it would have been ideal to perform tests with a hard limit on the amount of physical memory available to the runtime, this was not possible within this project.[2] Instead, the total amount of physical memory available to the operating system was reduced from 1GB to 96MB.

Unfortunately, this method led to very noisy results. In many cases there were orders of magnitude differences between the performance of successive runs of the same benchmark/configuration pair. Additionally, the execution of these benchmarks took an extremely long amount of time – up to 60 minutes for a single execution of the virtual machine. No insight into the locality performance of the collectors could be obtained from the results, although they have been included in the appendix in section A.6.

[2] In order to test using such a method a different version of the Linux kernel supporting such restrictions would be required.

7.8 Performance Bakeoff

This section aims to make a performance comparison between several full-heap collectors, including the original JMTk MarkSweep and SemiSpace collectors, and the new compacting collectors: MarkCompact, MarkCompactFL-5, and MarkCompactFL-1. It is important to note that the compacting collectors have not been given the same performance tuning as the other collectors; the observation of general trends in performance is more valuable than simply looking for the fastest or slowest configuration.



(a) _202_jess (b) _213_javac

Figure 7.7: Total time for full-heap collectors across heap sizes.

Figure 7.7 shows the total time required for _202_jess and _213_javac for the full-heap collectors when running across a variety of heap sizes. MarkSweep performs very well across all heap sizes. As was expected from the literature the performance of SemiSpace was excellent with low heap residency, but degraded as heap residency increased. This is due to the cost of keeping a copy reserve, and is best illustrated through the GC counts shown in figure 7.8. As the heap size decreases the gap between SemiSpace and the other collectors increases.

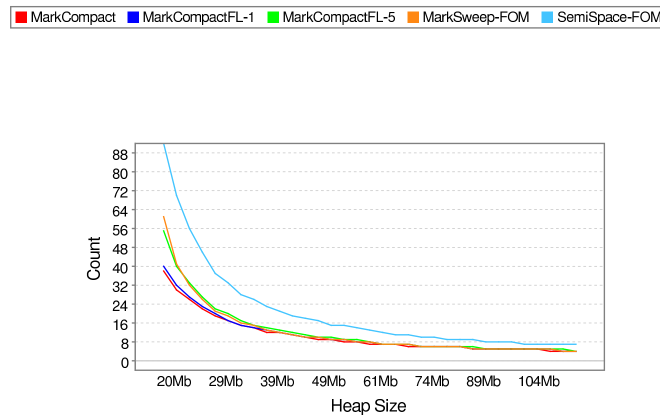


Figure 7.8: GC counts for full-heap collectors across heap sizes for _202_jess.

Although the performance of the sliding compacting collector, MarkCompact, is quite poor, it is interesting to note that at the very smallest heap sizes the performance is very close to that of MarkSweep. This supports the hypothesis that a compacting collector is very efficient when memory is limited; further performance optimisations to the compacting collector are required to further test this hypothesis.

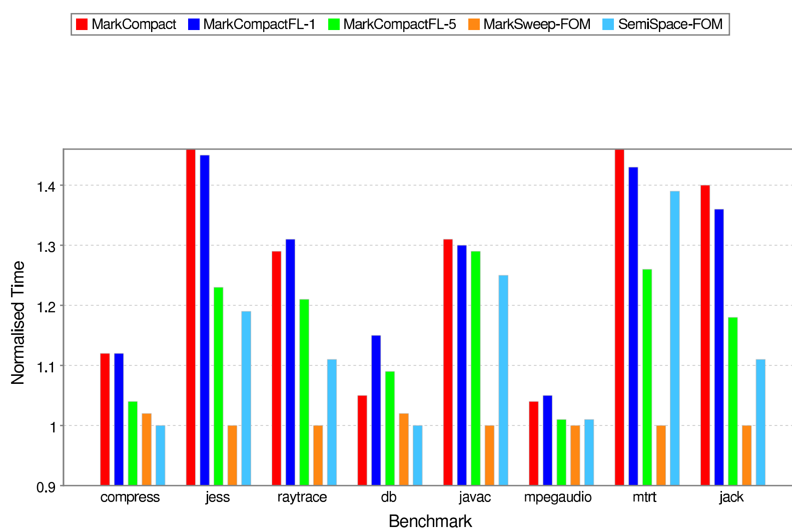


Figure 7.9: Total time for full-heap collectors (41MB heap).

Figure 7.9 shows the performance of the full-heap collectors in a small heap of 41MB. At this heap size MarkSweep performance is difficult to match. The compacting collectors MarkCompactFL-1 and MarkCompact perform very poorly on the GC intensive benchmarks _202_jess, _213_javac and _228_jack. Although MarkSweep has the fastest total times, mutator time performance in figure 7.10 shows that MarkCompact often outperforms it.

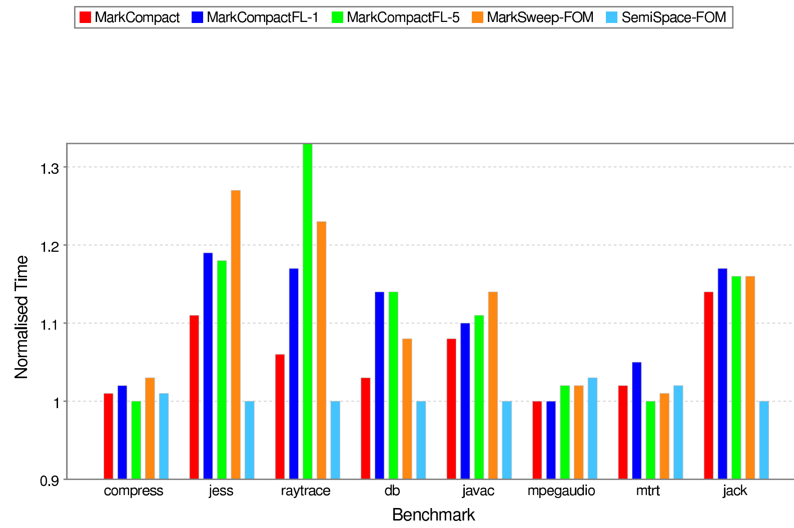


Figure 7.10: Mutator time for full-heap collectors (41MB heap).

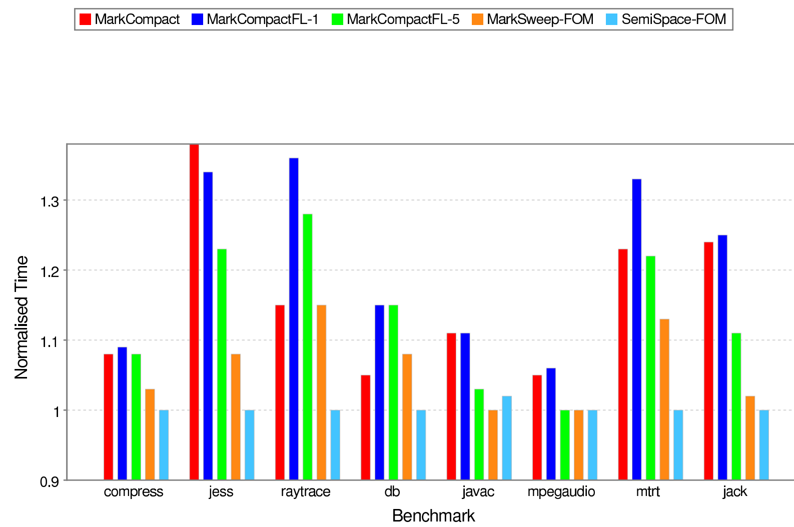


Figure 7.11: Total time for full-heap collectors (104MB heap).

On larger heap sizes (figure 7.11), SemiSpace slightly outperforms MarkSweep on GC intensive benchmarks. Again MarkCompact performs very poorly due to massive collection times. Complete results for each of the tests run are in the appendix in section A.7.

7.9 Generational

A compacting collector should be able to provide improved performance at the cost of a more expensive collection. This is expected to be very inefficient when considering large numbers of short lived objects. A generational collector aims to avoid this cost through the use of a copy-collected nursery. This experiment aims to compare GenMC with the existing GenMS-FOM.

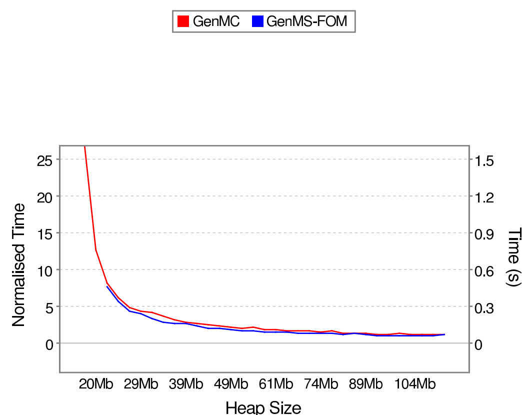


Figure 7.12: GC time for generational collectors across heap sizes for _202_jess.

It was expected that there will be less of a difference between these two strategies and their full-heap counterparts. This is due to the fact that the majority of collections are nursery collections and this collection is almost identical for both collectors.

Stability problems with the generational configurations made it impossible to generate a full set of results. Sufficient data points to perform a comparison was obtained. It was found that there was no significant difference between the two generational collectors, as illustrated in figure 7.12.

Complete results for each of the tests run are in the appendix in section A.8.

7.10 Summary

A methodology for benchmarking virtual machines was discussed. Several experiments were then devised and carried out to assist in analysing the performance of the compacting collectors. Results for each of these experiments were given, and where possible, each experiment presented and justified a recommendation.

Chapter 8

Conclusion

8.1 Summary

Modern object oriented languages such as Java and C# have been gaining widespread industry support in recent times. Such languages rely on a runtime infrastructure that provides automatic dynamic memory management services. The performance of such services is a crucial component of overall system performance.

Compacting collectors are used in several important production runtimes, including Microsoft's Common Language Runtime and IBM's Java Runtime Environment. Through the addition of a similar type of collector to JMTk, the ability to effectively compare the performance of these algorithms against others has been provided.

Such collectors have been shown to be very demanding of the host runtime, with the implementation of these algorithms leading to several significant modifications to the underlying infrastructure of both JMTk and Jikes RVM. It is anticipated that this work in broadening the set of operations supported by JMTk and Jikes RVM will also allow new classes of collectors to be implemented and compared.

Chapter 2 explored the rich literature surrounding this area of research. There are a multitude of different automatic memory management strategies that a runtime infrastructure can employ, but there is no single best solution for all situations.

Chapter 3 described the target platform, a high performance Java virtual machine, with cutting edge compiler and memory management technologies.

Chapters 4, 5 and 6 provided descriptions of several compacting algorithms. These utilised both bump pointer and free list allocation policies, and the final algorithm was a generational collector.

Chapter 7 provided a detailed performance evaluation of the compacting collectors, and led to the validation of many of the decisions taken through the design process.

A method for reusing the status word to store a forwarding pointer was described and found to be effective. The average percentage in most situations was only 2%, a situation that would have the use of an additional header word using 25 times more space.

An analysis of the performance of each phase of the compacting collectors gave some insight into what areas should be targeted for optimisation. This work also validated the hypothesis that iterating the heap would be more efficient over a free list.

The new object model in which scalar objects are laid out forward in memory was compared to the current model. No reason to continue the use of the original object model was discovered for Jikes RVM running on Linux/IA32.

An optimal compaction frequency was found for the free list compacting collector through a detailed performance comparison over multiple heap sizes.

A performance comparison between several full-heap collectors, including both standard JMTk collectors and the new compacting collectors showed that compaction can provide real performance benefits. While the cost of performing the compaction makes the collectors perform poorly, they are yet to be optimised.

Finally, it was found that a generational compacting collector provided equivalent performance to other high performance generational collectors in JMTk.

These results are however, far from a conclusive investigation into compacting collection techniques, and raises many questions for future research.

8.2 Further Work

The results of this work are not conclusive, but they do make compacting collection algorithms appear promising as a method of reducing fragmentation and improving performance in some situations. This research identifies two key challenges faced with compacting collection. Firstly, there is the challenge to reduce the cost of a compacting collection as much as possible. Secondly, there is the challenge to develop new and innovative ways to combine compacting collection with other collection strategies; through this, it is possible to provide benefits of reduced fragmentation when required, without paying the high price of compaction at each and every collection.

8.2.1 Performance Tuning

This project did not allow sufficient time to performance tune the collectors. The clearest example of this is the difference in performance between MarkCompact-FL0 and MarkSweep. The costs of the different phases of collection were identified, and this would be a good starting point for such performance tuning.

This phase cost information shows that the cost of tracing from the roots is very significant. Using another technique to update forwarding pointers, such as remembering the set of pointers into the compacted space during the mark phase should be investigated.

The method used for sweeping within the free list collector should also be looked into. A solution similar to the mark bitmap used with JMTk's mark-sweep collector could improve performance considerably.

After a compacting collection the way that blocks are considered for allocation should also be investigated. Currently, all blocks are reconsidered for allocation, even

though it is a trivial observation that only the last block could possible have any free cells.

8.2.2 *N* Generational Collector

JMTk's generational collectors include only two generations: the nursery and the mature space. Other implementations of generational collection have more generations, and the size of these are sometimes dynamically determined [Richter 2000a; Richter 2000b]. While there is a cost associated with this as the write barrier needs to perform more work, it would be worthwhile implementing such a collector in JMTk to determine if there is any benefit associated with this cost.

When many generations are in use, a more incremental approach can be taken to collection, which is an important consideration for real-time or interactive applications.

8.2.3 Alternate Compacting Algorithms

Only one simple compacting algorithm was implemented during this project. It would be interesting to compare sliding compaction to other compacting collectors, such as table-based or threaded algorithms.

8.2.4 Generational Sliding Compaction

In this project only the free list compacting collector was placed within a generational collection strategy. It would be beneficial to compare the two compacting algorithms in a generational setting, particularly after any performance enhancements have been made.

8.2.5 Compact vs. Sweep Heuristic

Only a simple heuristic for triggering compacting collections was implemented. With more detailed information regarding the fragmentation levels within the heap, it should be possible to develop a heuristic that triggers a compacting collection more intelligently.

Sansom [1991] describes a method of implementing a garbage collector that combines a compacting collector and a copying collector, and triggers the compacting collector based on heap residency. Other hybrid compacting collectors such as this should be investigated.

8.2.6 Limited Memory Testing

The amount of limited memory testing that was conducted as part of this project was not able to provide conclusive results. Further work on the development and application of a framework for testing virtual machines under limited physical memory situations should assist in showing the benefits to locality of compaction.

8.3 Conclusion

This thesis has contributed a fully functional set of compacting collectors to JMTk. The ability of compaction to improve runtime performance was demonstrated.

Hybrid collectors, that could occasionally take advantage of a compacting collection, while still being able to perform cheaper mark-sweep collections, were very promising.

A new and effective technique to store forwarding pointers in way that made more efficient use of memory was described and evaluated.

There has been little work done on performance optimisation of the collectors, and one remains optimistic that such optimisations will reduce, or eliminate, the performance gap between them and other full-heap collectors.

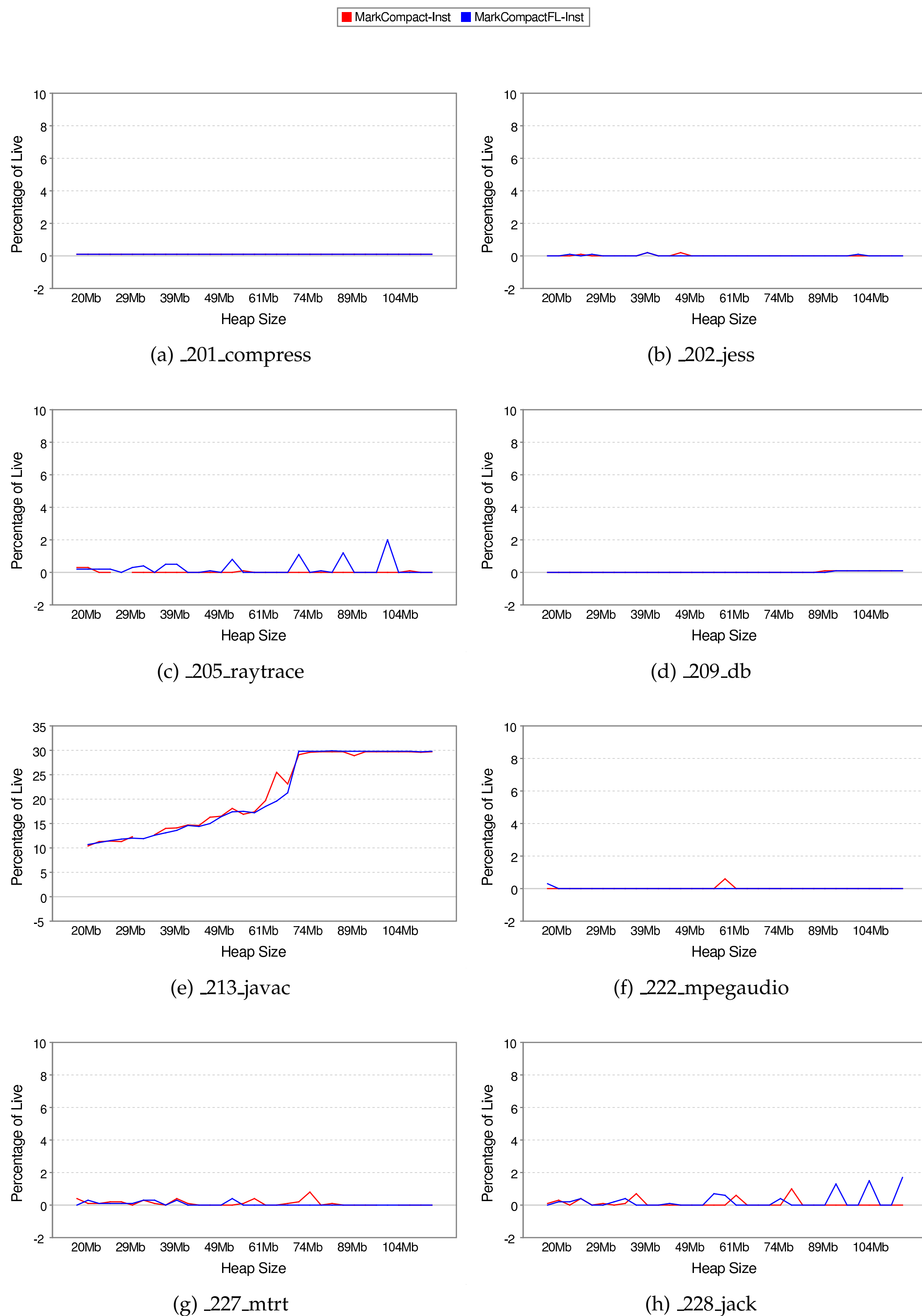
Appendix A

Complete Results

This appendix shows complete results for all experiments conducted as part of the project. Representative and summary information is used within individual results sections in chapter 7 to communicate major findings, but those more involved in the field may be interested in more detailed results.

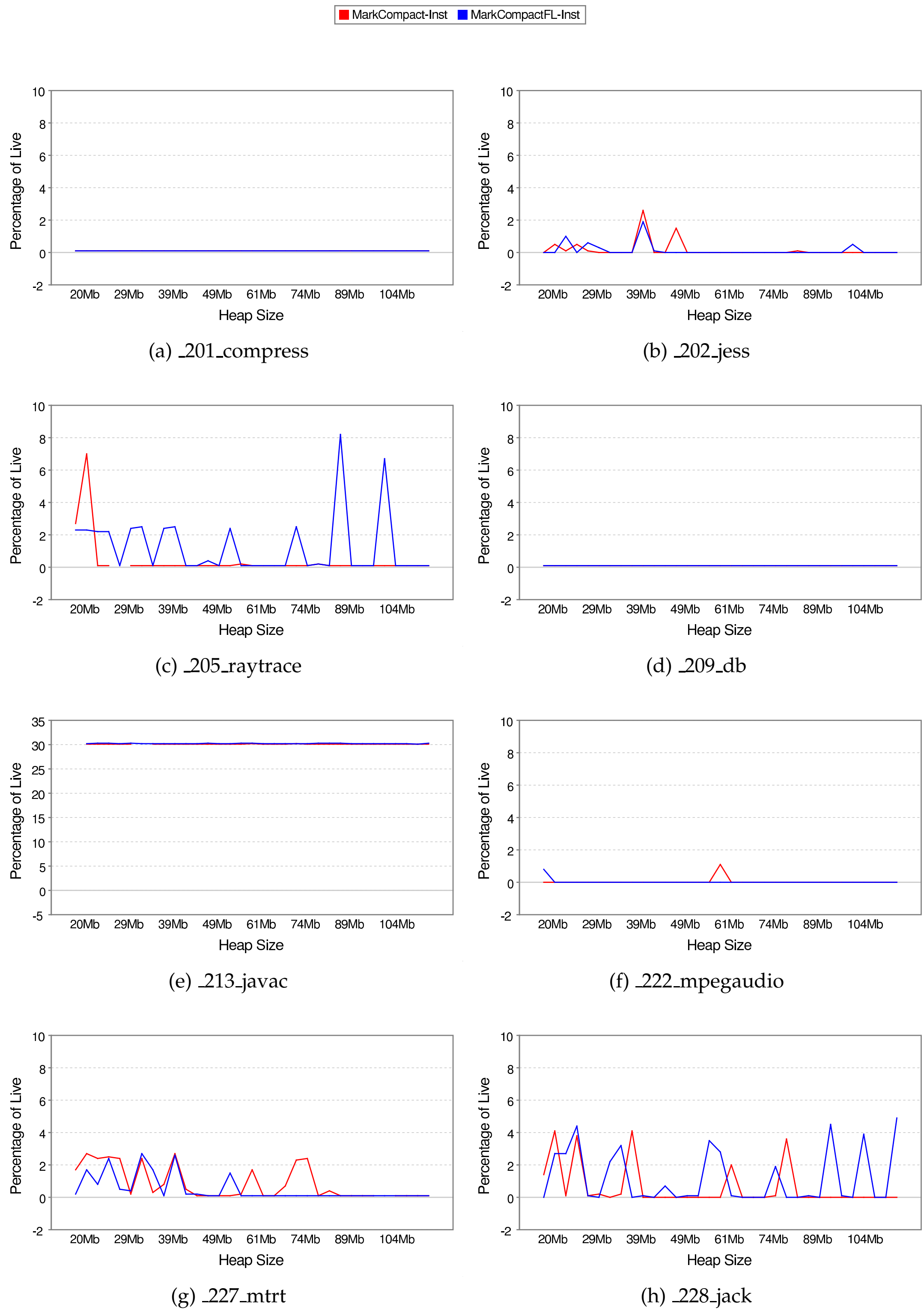
For each section in the performance evaluation section, a matching section is provided within this appendix listing complete results.

A.1 Non-Zero Status Words



(a) _201_compress
(b) _202_jess
(c) _205_raytrace
(d) _209_db
(e) _213_javac
(f) _222_mpegaudio
(g) _227_mtrt
(h) _228_jack

Figure A.1: Average percentage: Percentage of live objects that had non-zero status.



(a) _201_compress

(b) _202_jess

(c) _205_raytrace

(d) _209_db

(e) _213_javac

(f) _222_mpegaudio

(g) _227_mtrt

(h) _228_jack

Figure A.2: Worst case percentage: Percentage of live objects that had non-zero status.

A.2 Phase Timings

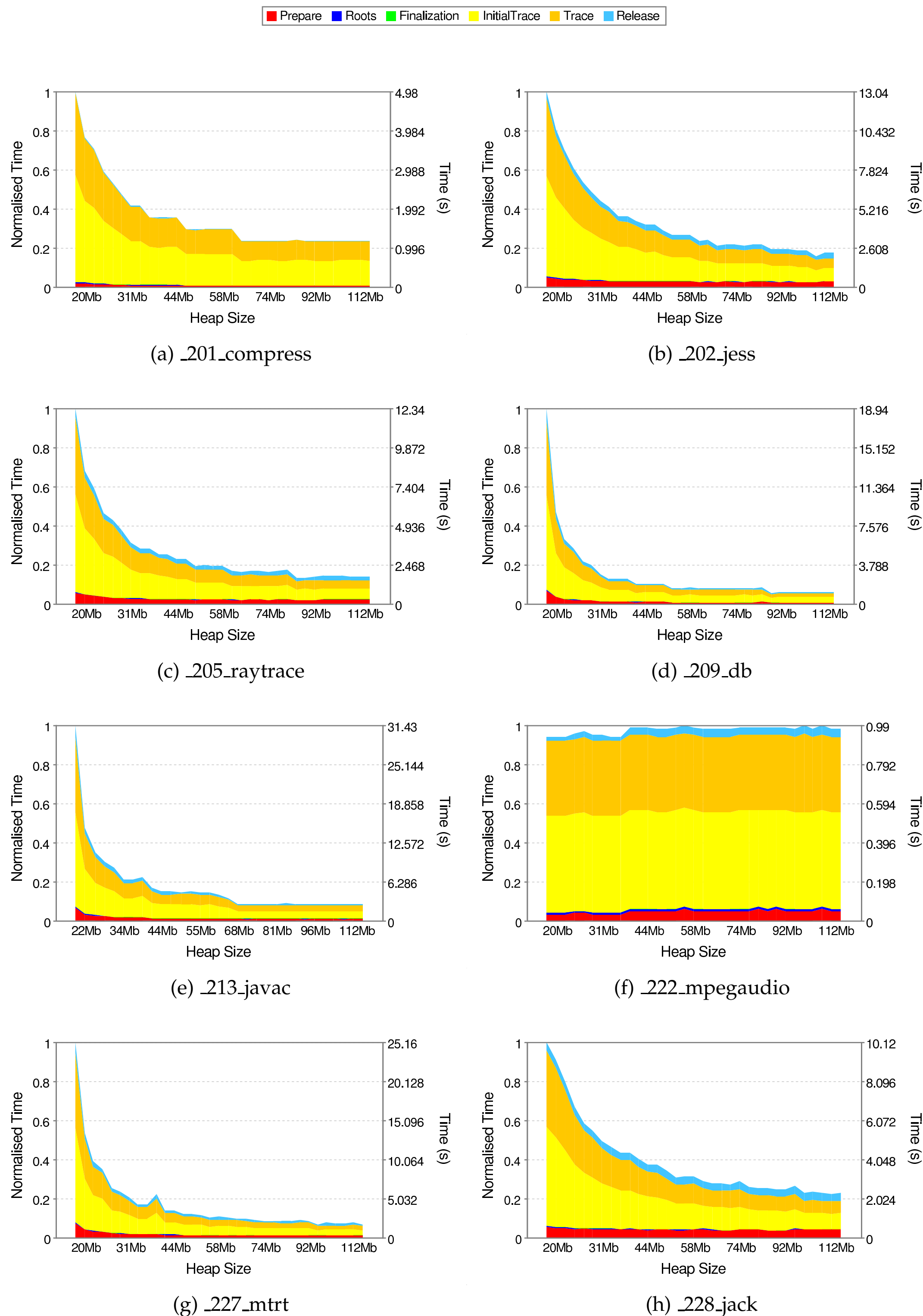
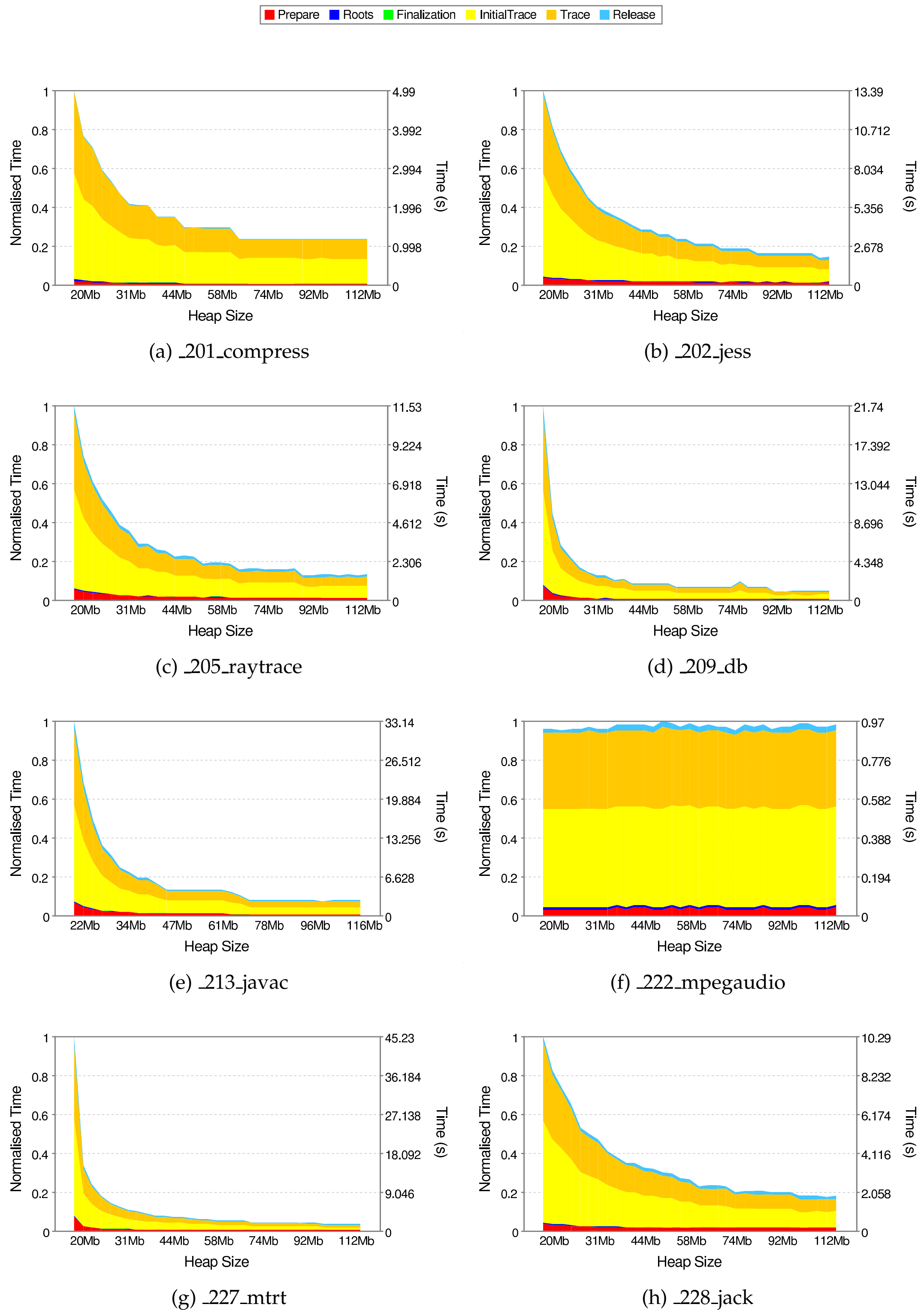


Figure A.3: Phase timings: Sliding Mark Compact



(a) _201_compress (b) _202_jess

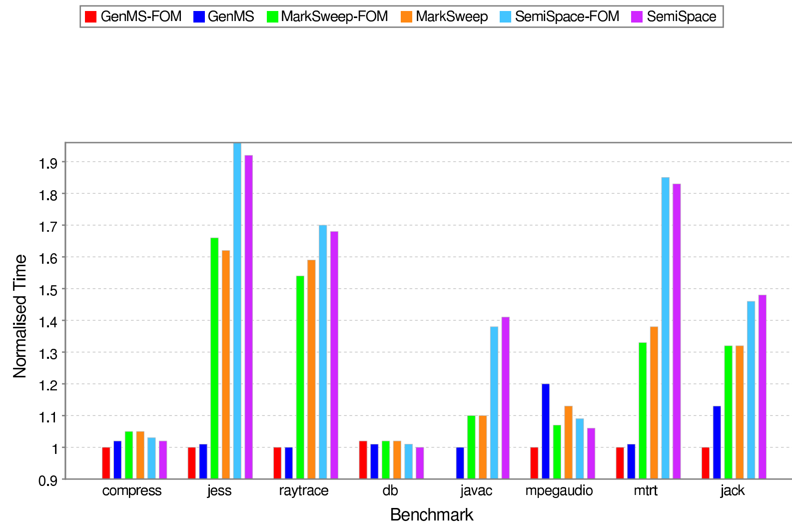
(c) _205_raytrace (d) _209_db

(e) _213_javac (f) _222_mpegaudio

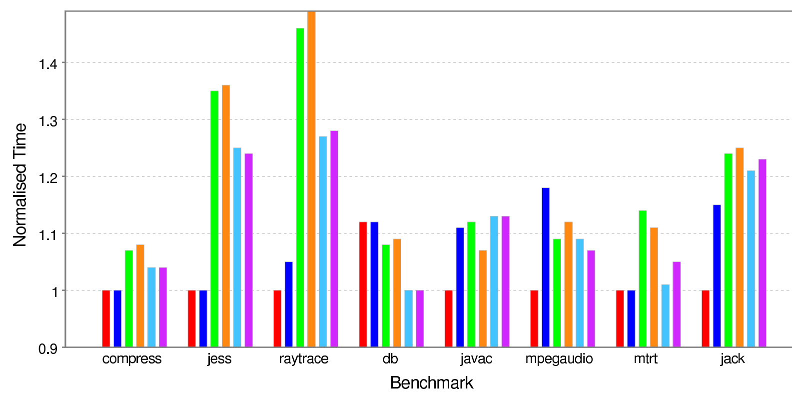
(g) _227_mtrt (h) _228_jack

Figure A.4: Phase timings: Mark Compact Free List

A.3 Flipped Object Model



(a) 41MB heap



(b) 104MB heap

Figure A.5: Total time summary: Original Object Model vs. Flipped Object Model.

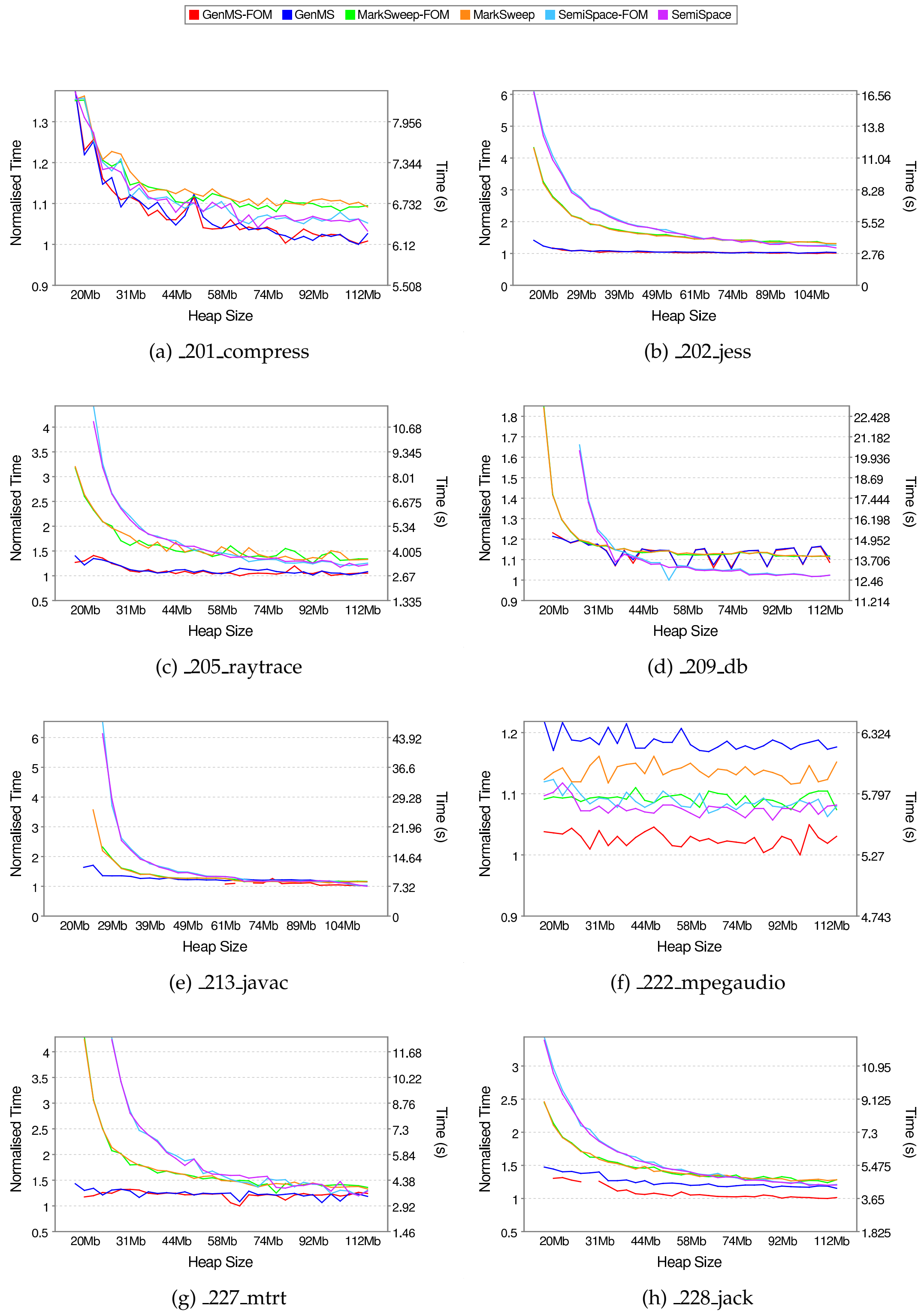
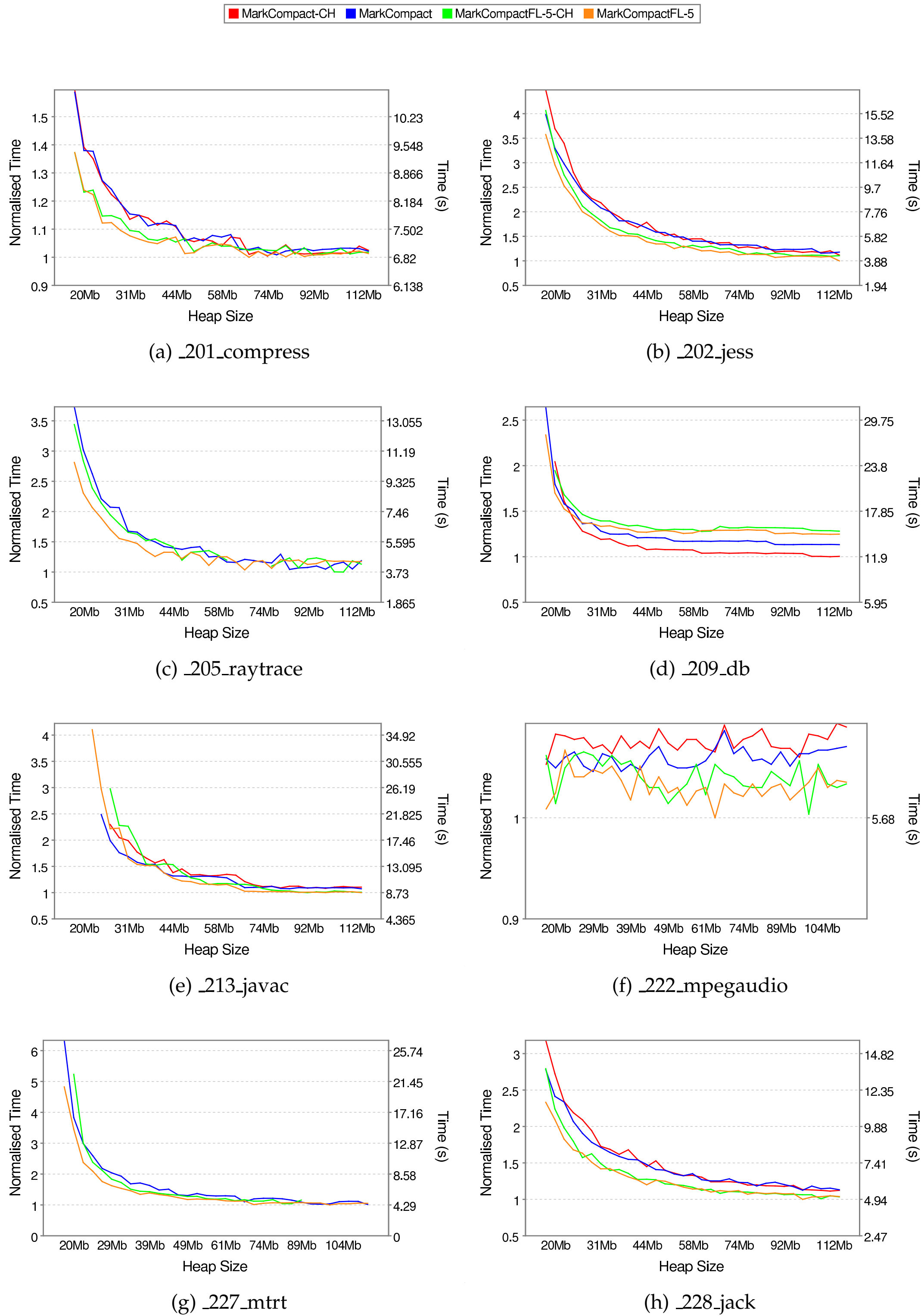


Figure A.6: Total time: Original Object Model vs. Flipped Object Model.

A.4 Constant Size Header



(a) _201_compress

(b) _202_jess

(c) _205_raytrace

(d) _209_db

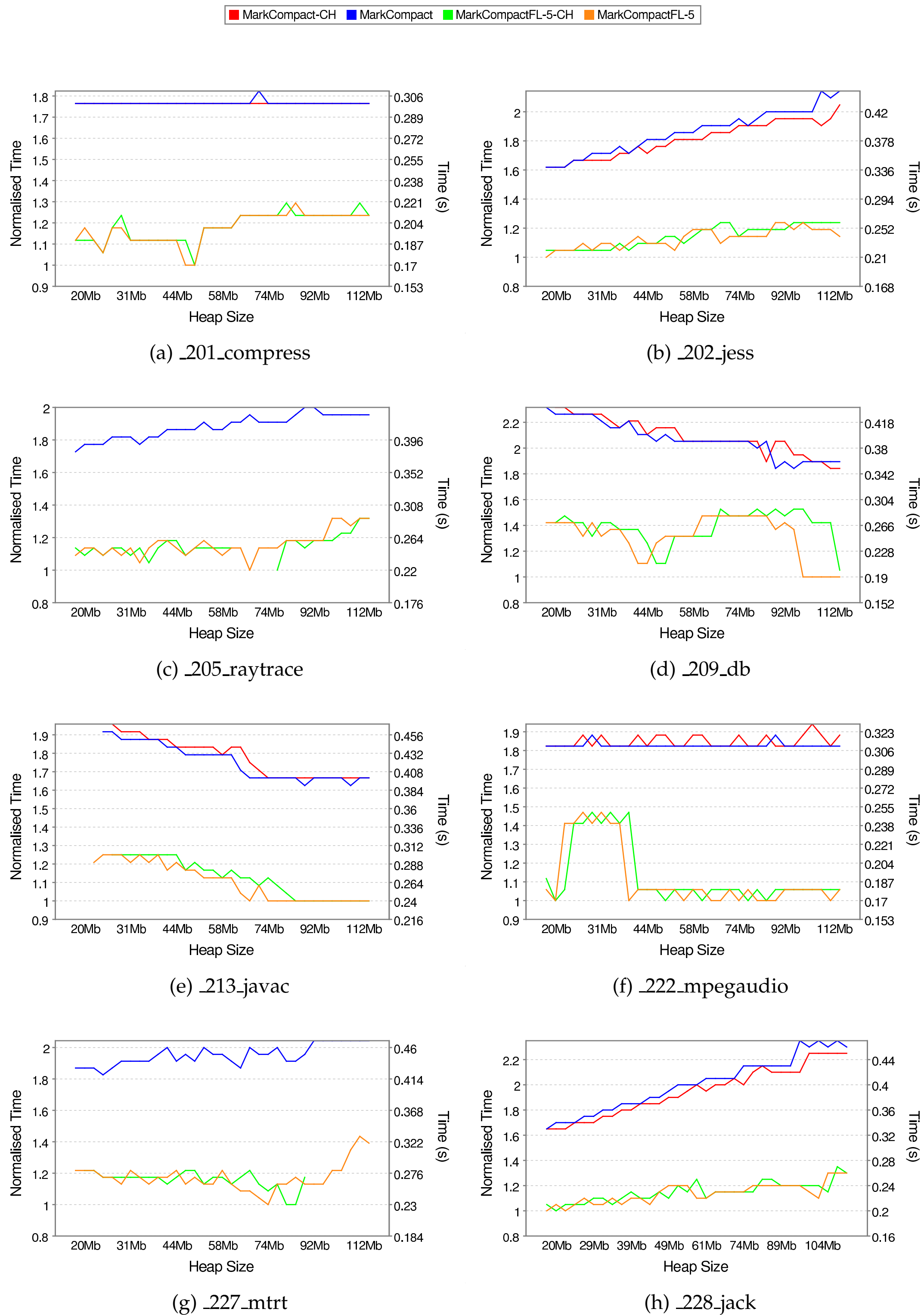
(e) _213_javac

(f) _222_mpegaudio

(g) _227_mtrt

(h) _228_jack

Figure A.7: Total time: Segregation vs. increasing scalar header size.



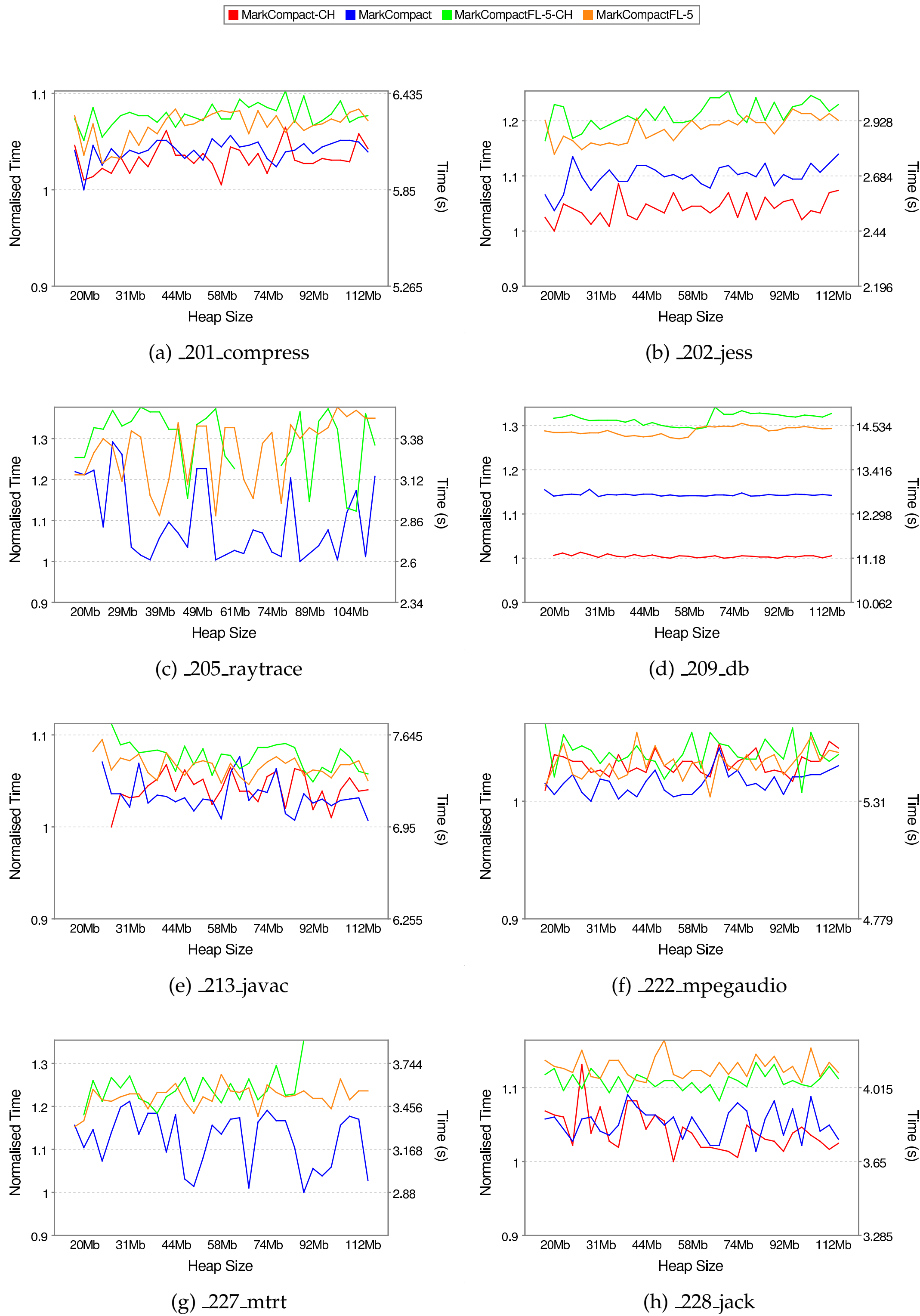
(a) _201_compress (b) _202_jess

(c) _205_raytrace (d) _209_db

(e) _213_javac (f) _222_mpegaudio

(g) _227_mtrt (h) _228_jack

Figure A.8: Average GC time: Segregation vs. increasing scalar header size.



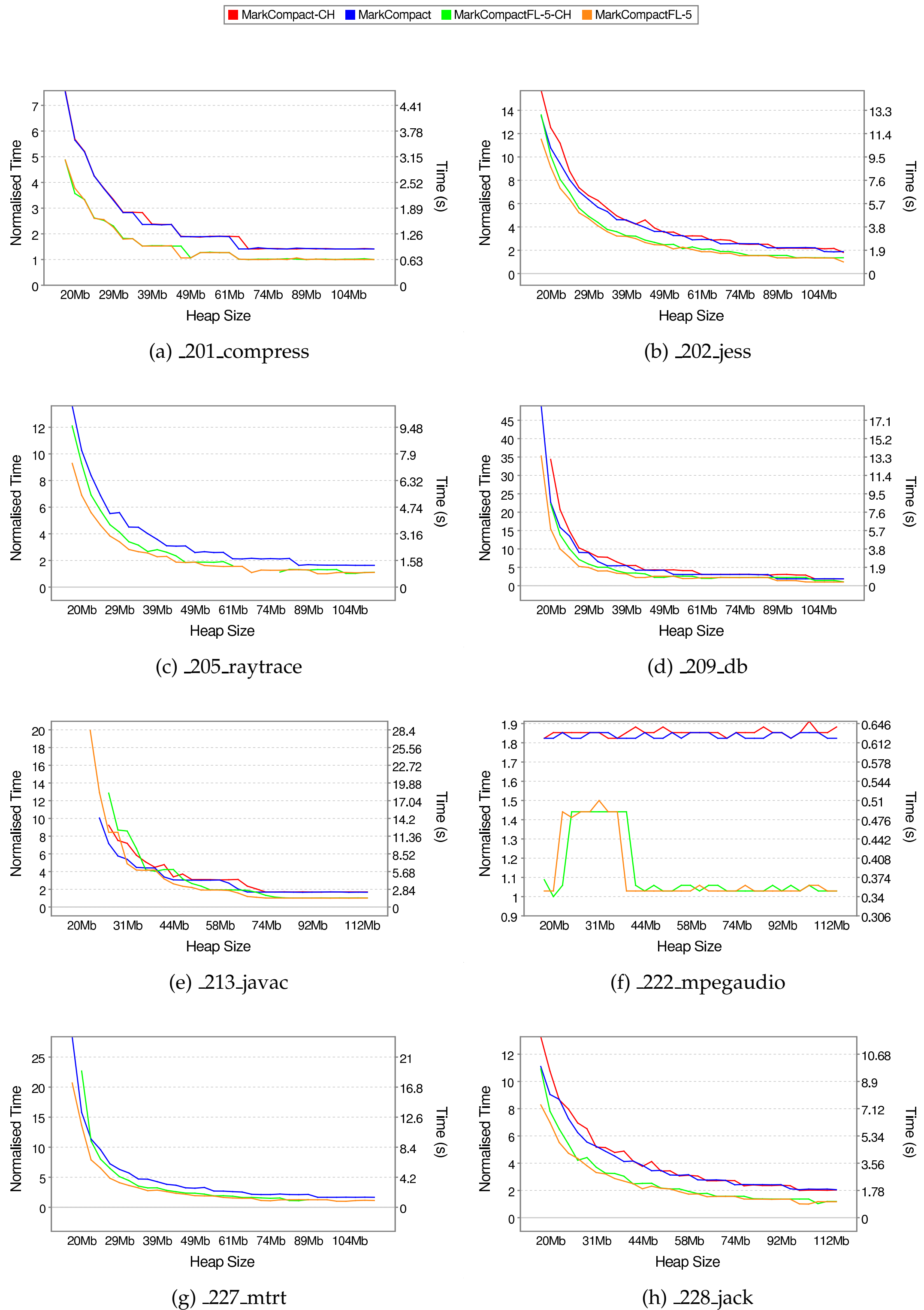
(a) _201_compress (b) _202_jess

(c) _205_raytrace (d) _209_db

(e) _213_javac (f) _222_mpegaudio

(g) _227_mtrt (h) _228_jack

Figure A.9: Mutator time: Segregation vs. increasing scalar header size.



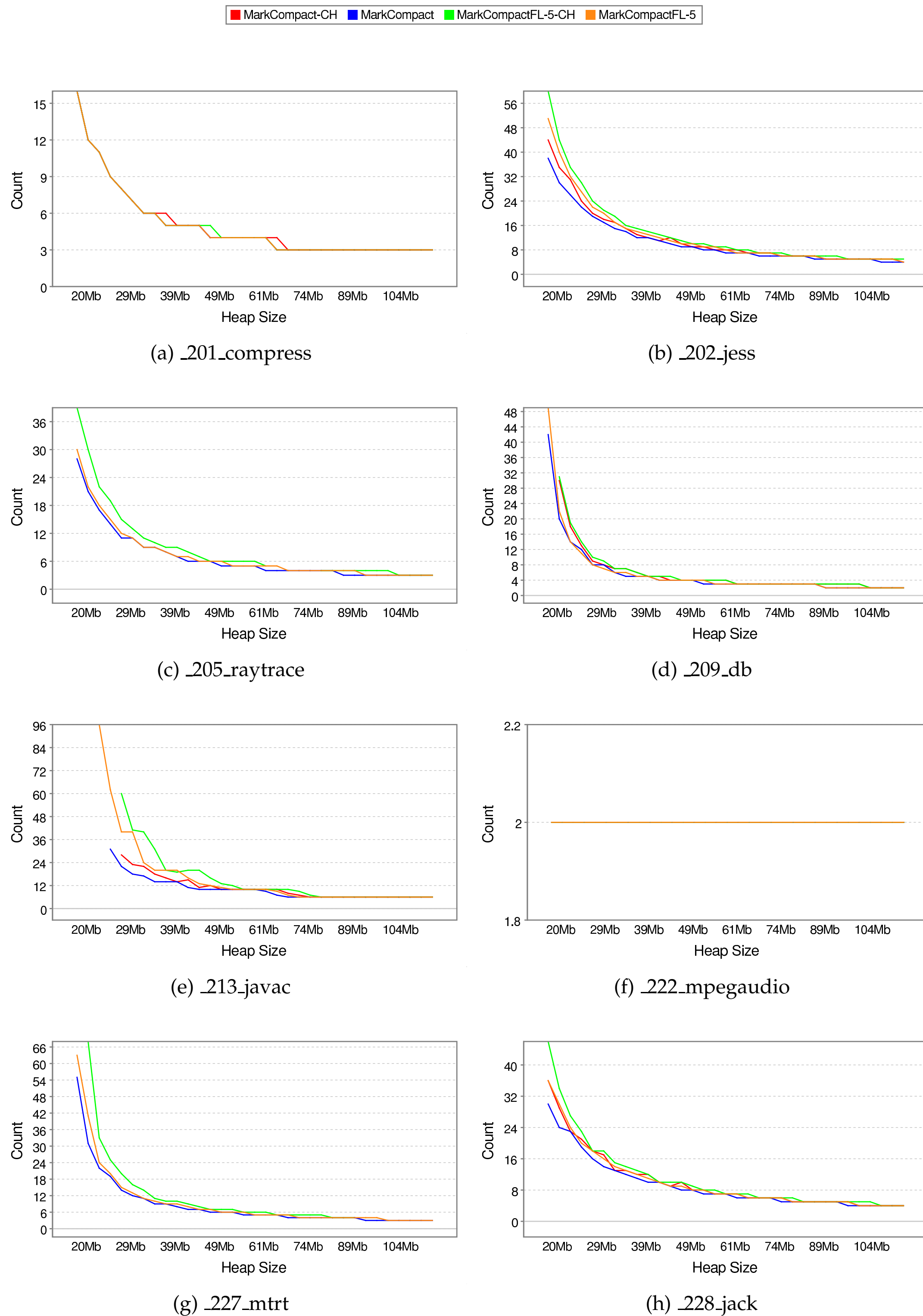
(a) _201_compress (b) _202_jess

(c) _205_raytrace (d) _209_db

(e) _213_javac (f) _222_mpegaudio

(g) _227_mtrt (h) _228_jack

Figure A.10: GC time: Segregation vs. increasing scalar header size.



(a) _201_compress

(b) _202_jess

(c) _205_raytrace

(d) _209_db

(e) _213_javac

(f) _222_mpegaudio

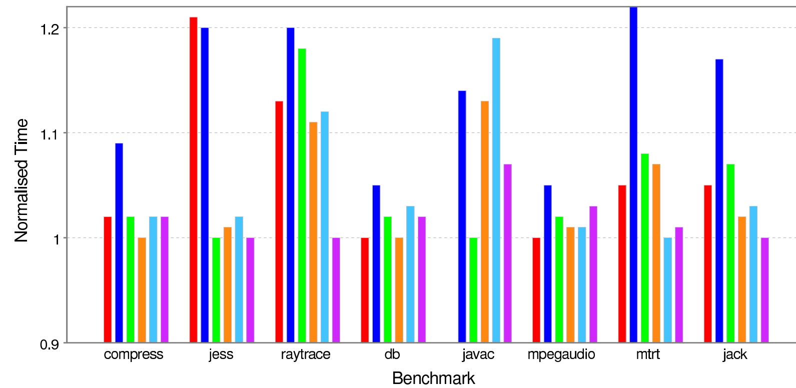
(g) _227_mtrt

(h) _228_jack

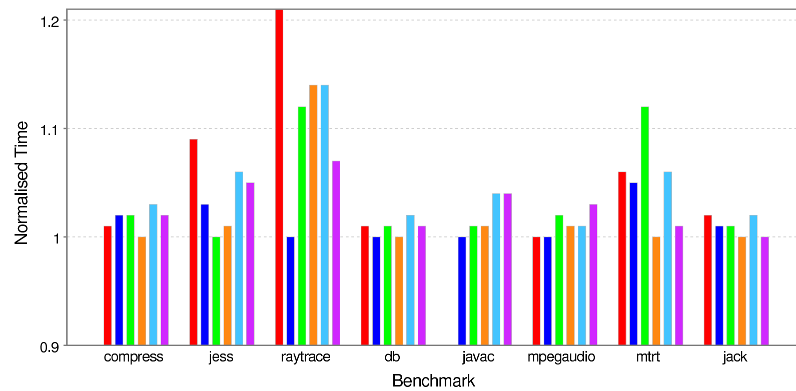
Figure A.11: GC count: Segregation vs. increasing scalar header size.

A.5 Free List: Compact vs Sweep

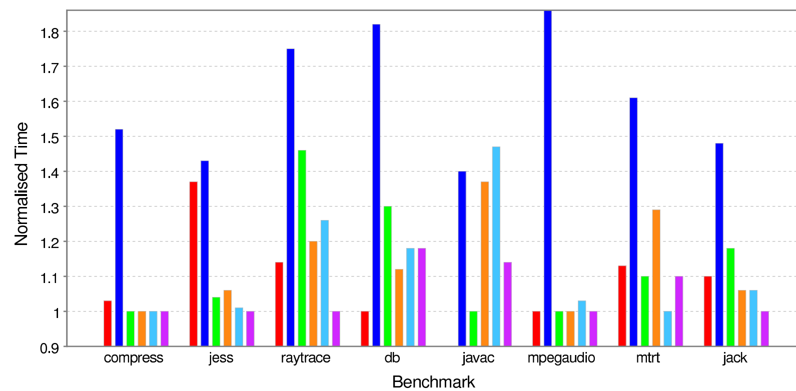
MarkCompactFL-0 MarkCompactFL-1 MarkCompactFL-3 MarkCompactFL-5 MarkCompactFL-7 MarkCompactFL-9



(a) Total time

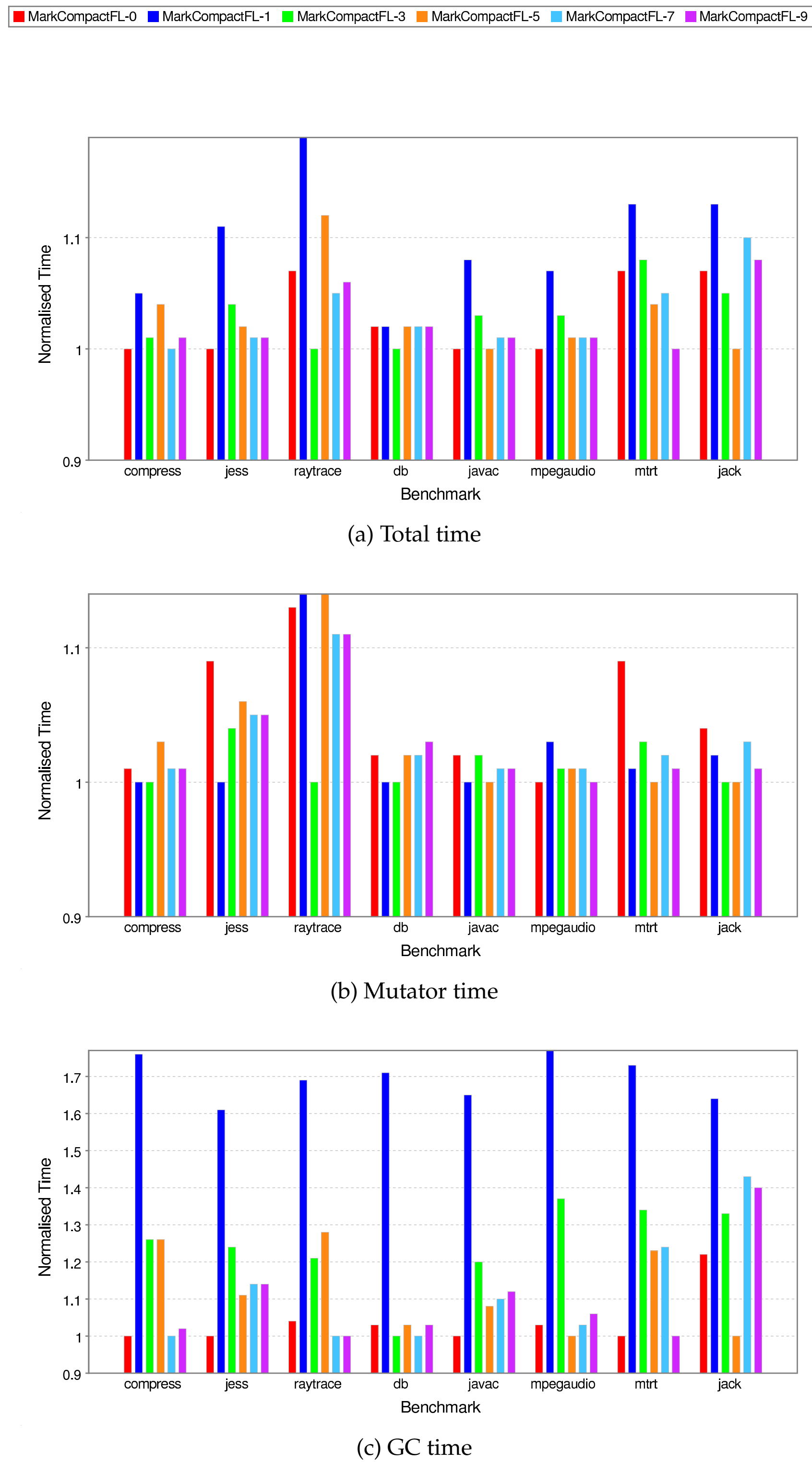


(b) Mutator time



(c) GC time

Figure A.12: Summary: Mark Compact Free List compacting every *n* GCs (41MB).



(a) Total time

(b) Mutator time

(c) GC time

Figure A.13: Summary: Mark Compact Free List compacting every *n* GCs (104MB).

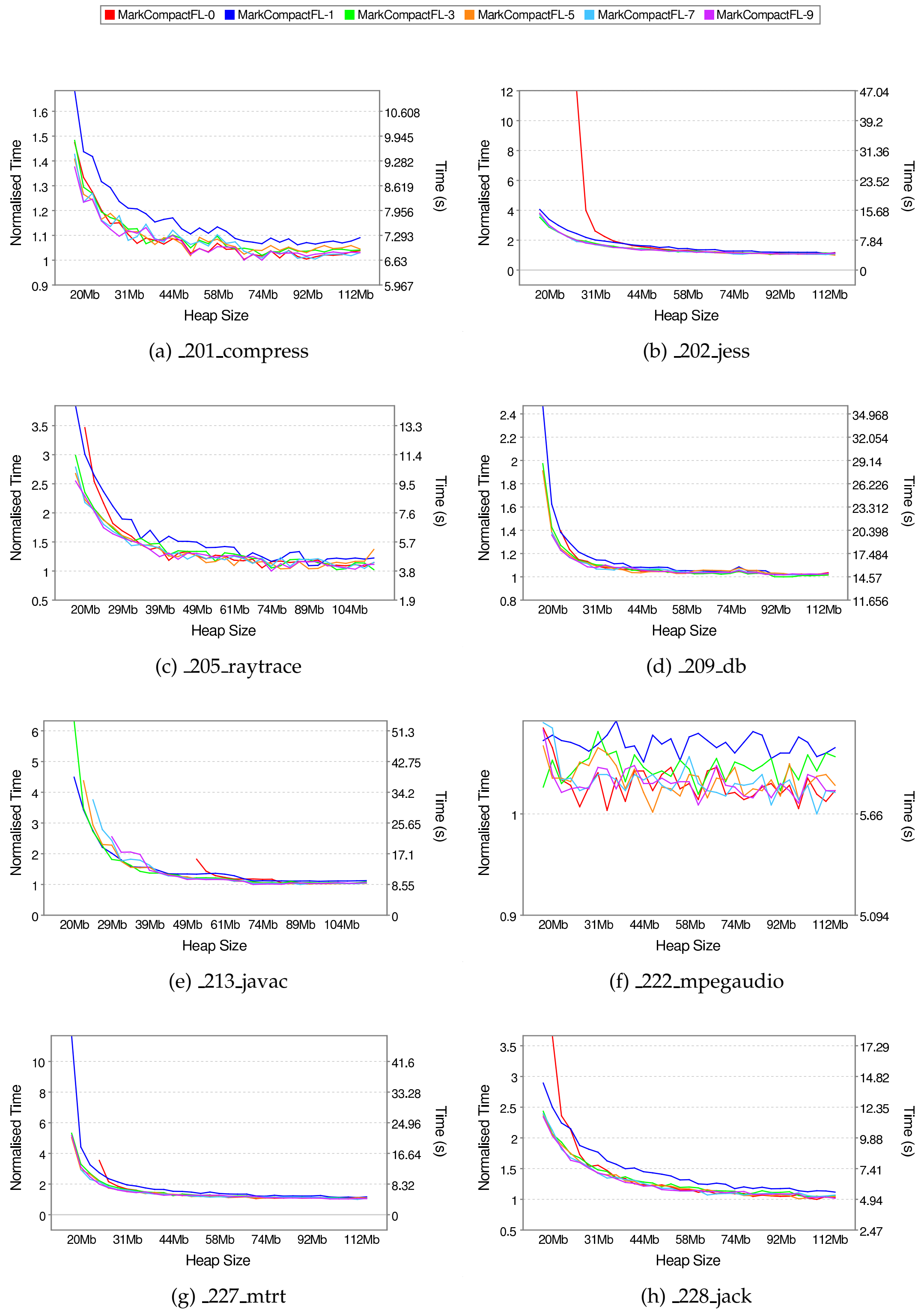
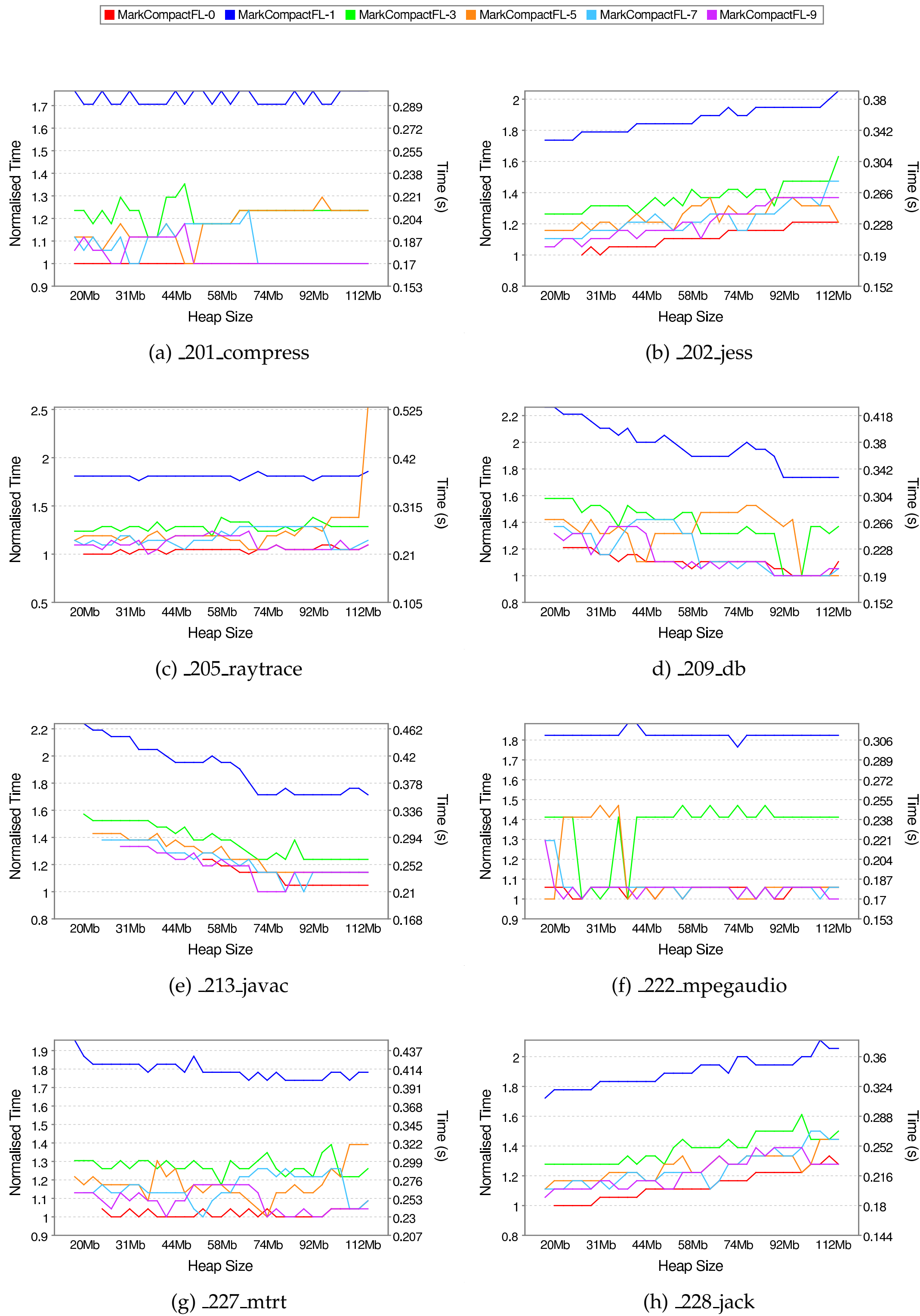


Figure A.14: Total time: Mark Compact Free List compacting every *n* GCs.



(a) _201_compress

(b) _202_jess

(c) _205_raytrace

d) _209_db

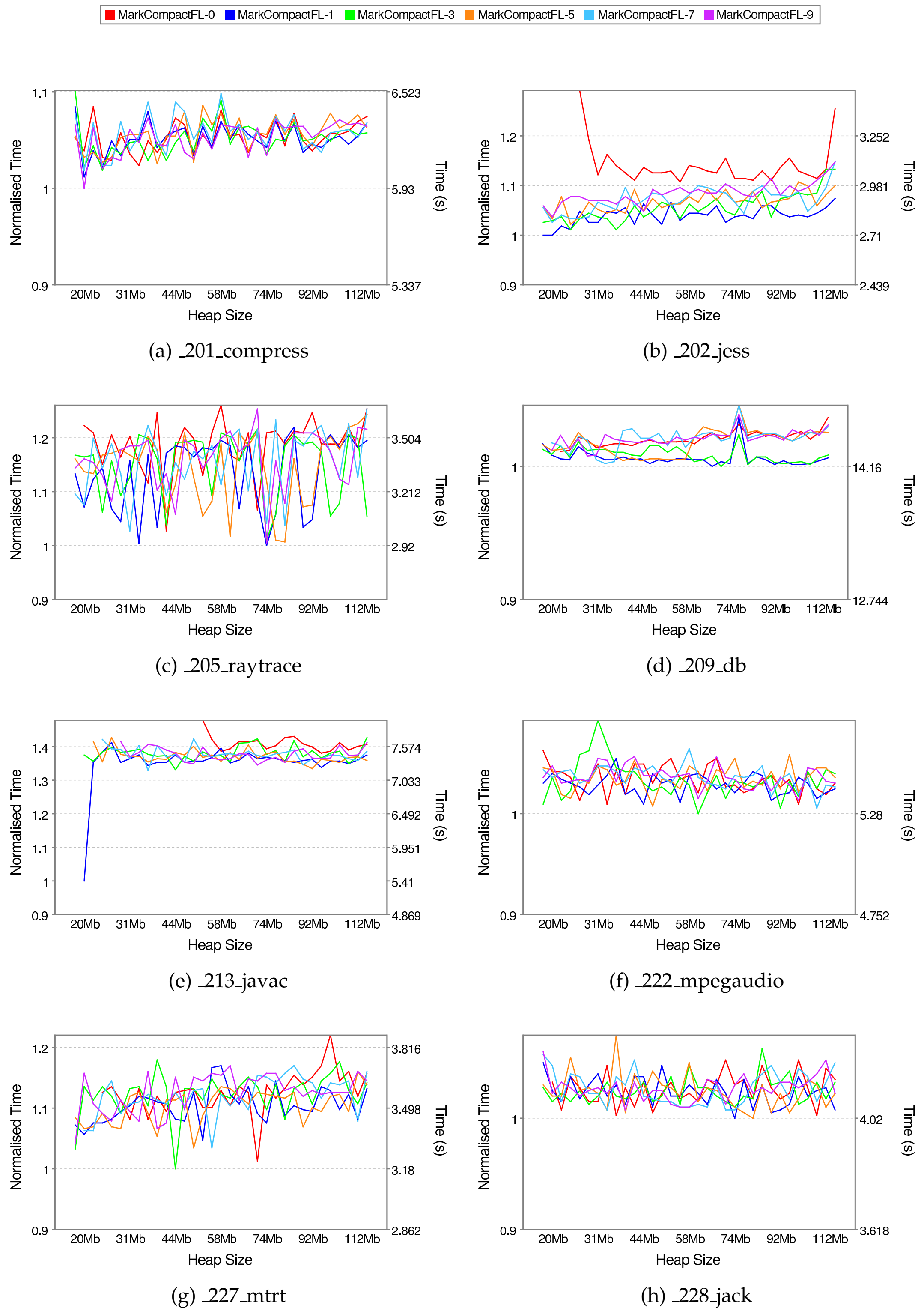
(e) _213_javac

(f) _222_mpegaudio

(g) _227_mtrt

(h) _228_jack

Figure A.15: Average GC time: Mark Compact Free List compacting every n GCs.



(a) _201_compress (b) _202_jess

(c) _205_raytrace (d) _209_db

(e) _213_javac (f) _222_mpegaudio

(g) _227_mtrt (h) _228_jack

Figure A.16: Mutator time: Mark Compact Free List compacting every n GCs.

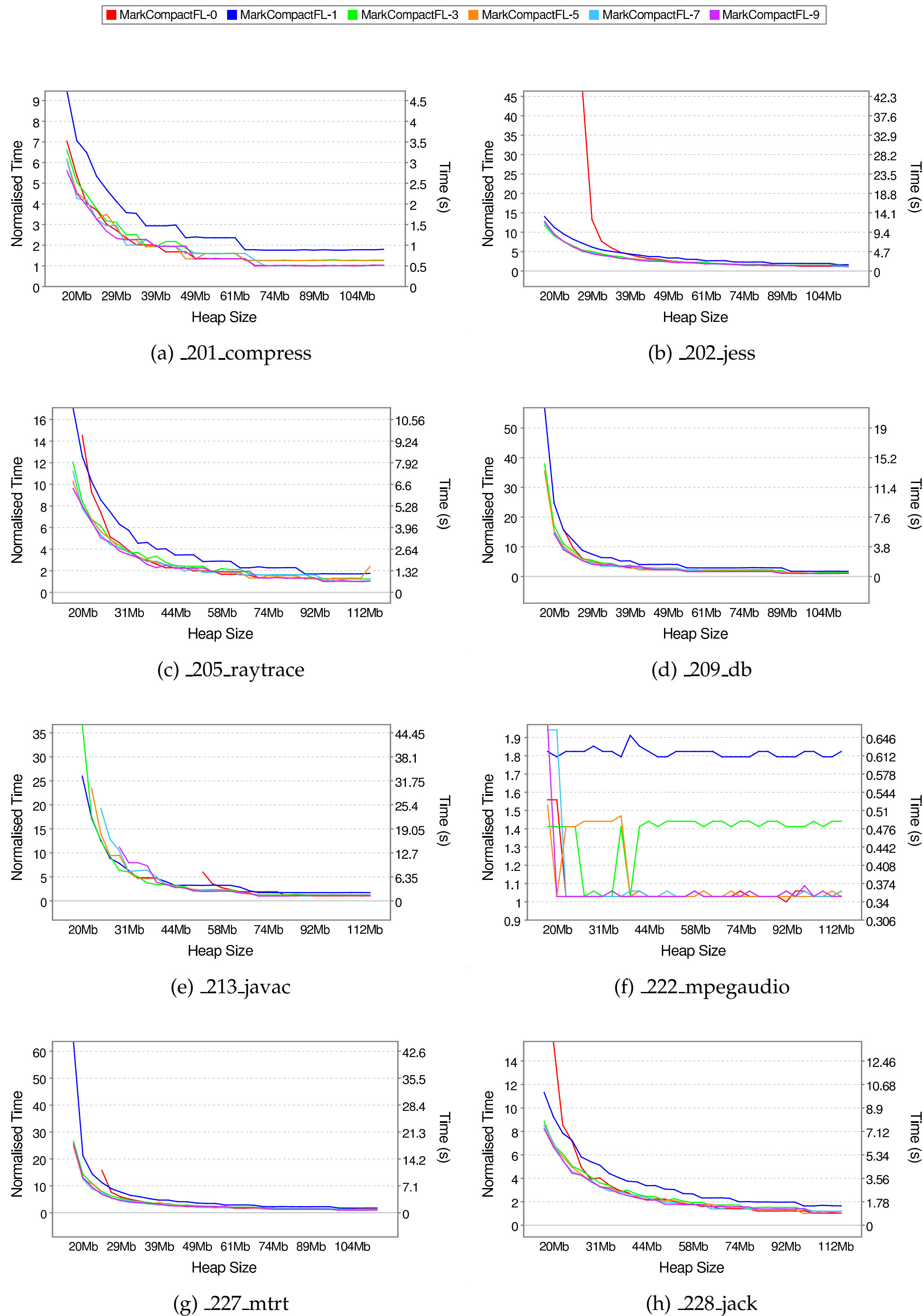
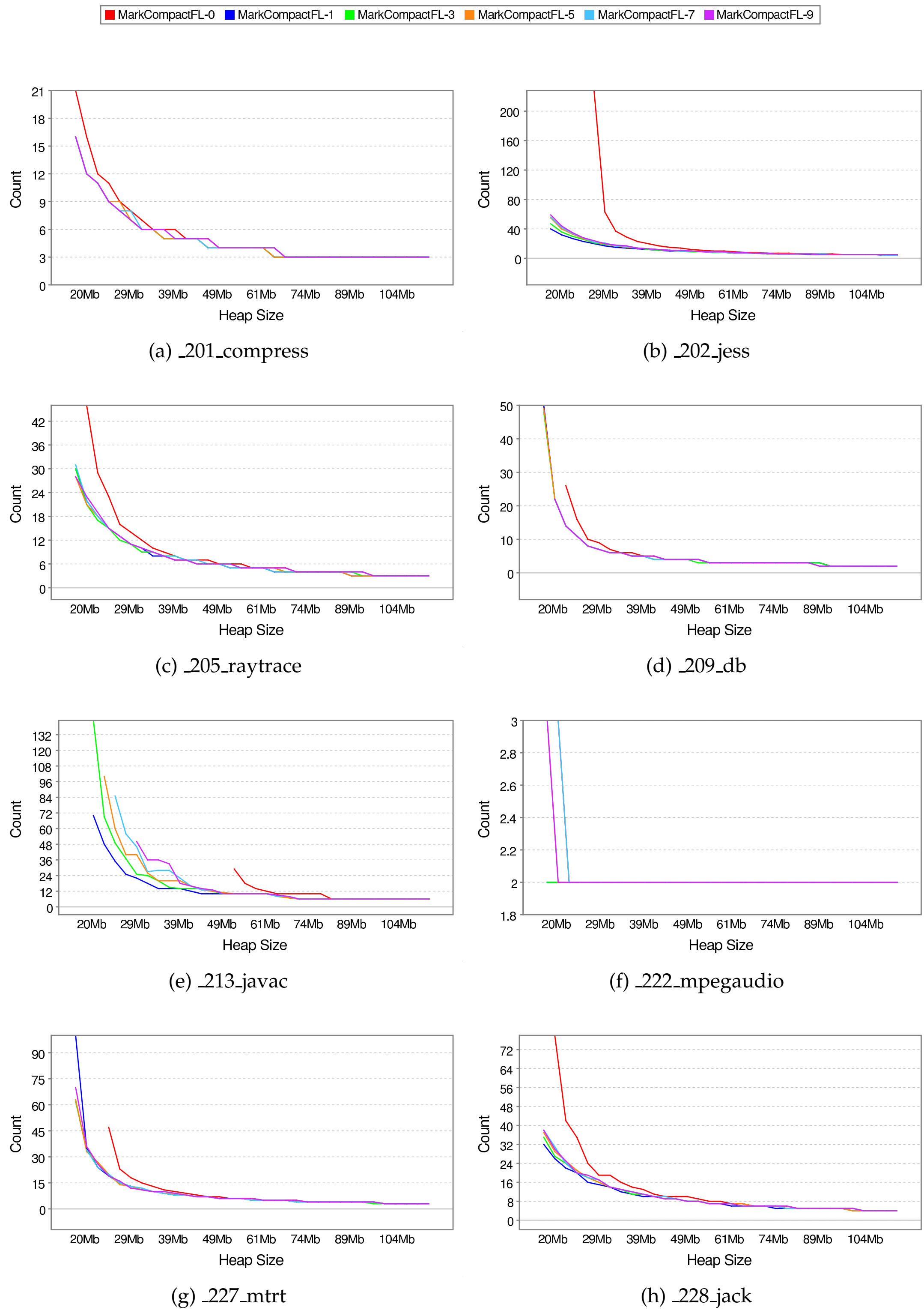


Figure A.17: GC time: Mark Compact Free List compacting every n GCs.



(a) _201_compress

(b) _202_jess

(c) _205_raytrace

(d) _209_db

(e) _213_javac

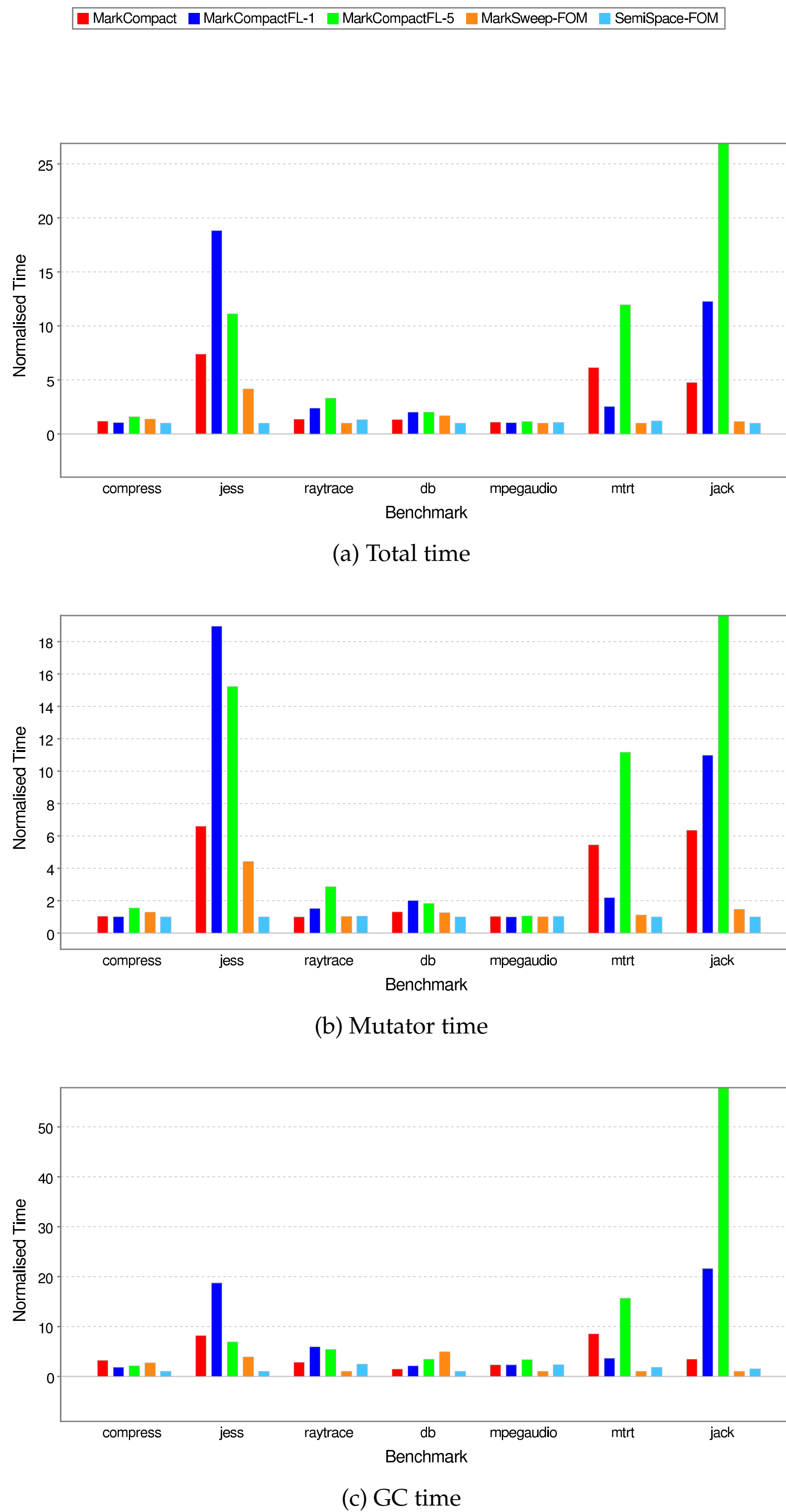
(f) _222_mpegaudio

(g) _227_mtrt

(h) _228_jack

Figure A.18: GC count: Mark Compact Free List compacting every n GCs.

A.6 Limited Physical Memory

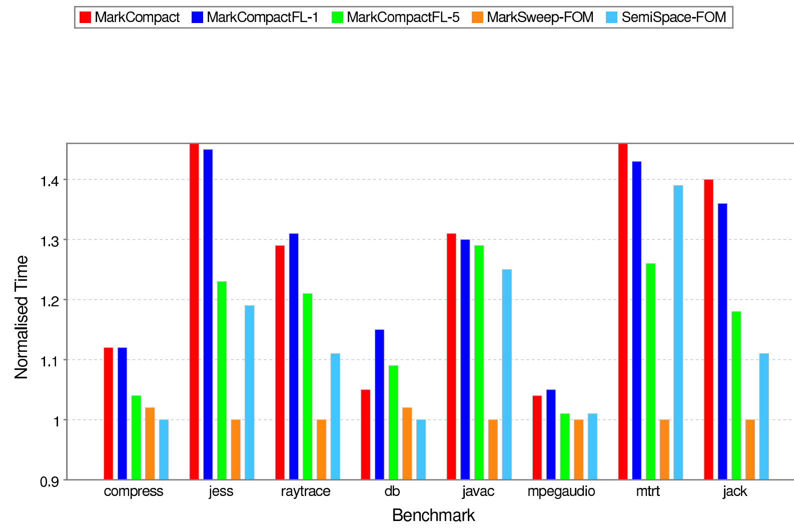


(a) Total time

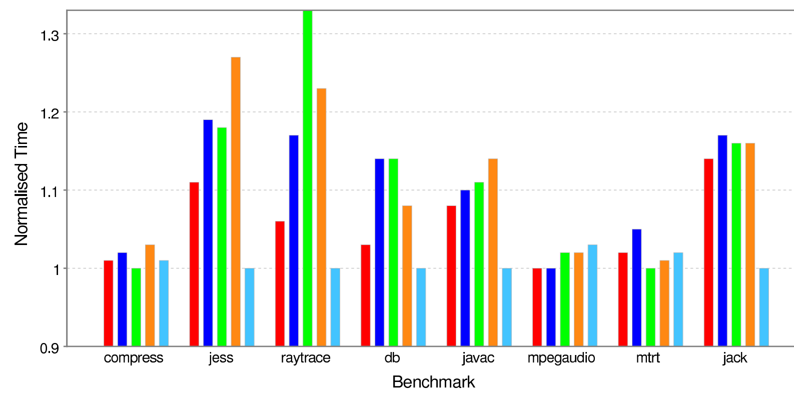
(b) Mutator time

(c) GC time

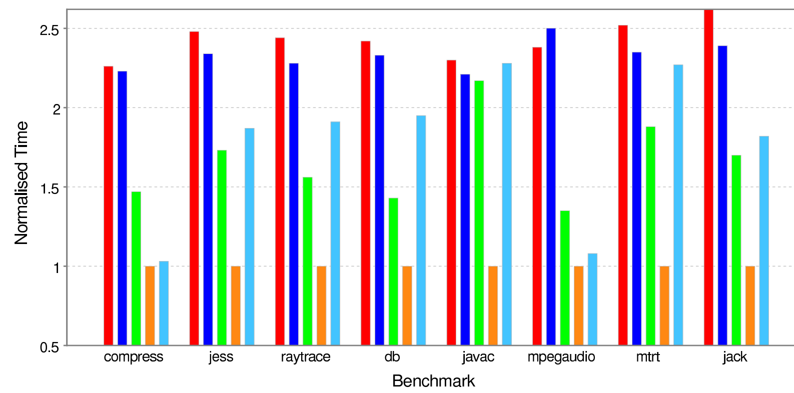
Figure A.19: Limited physical memory tests with 96MB physical memory.



(a) Total time



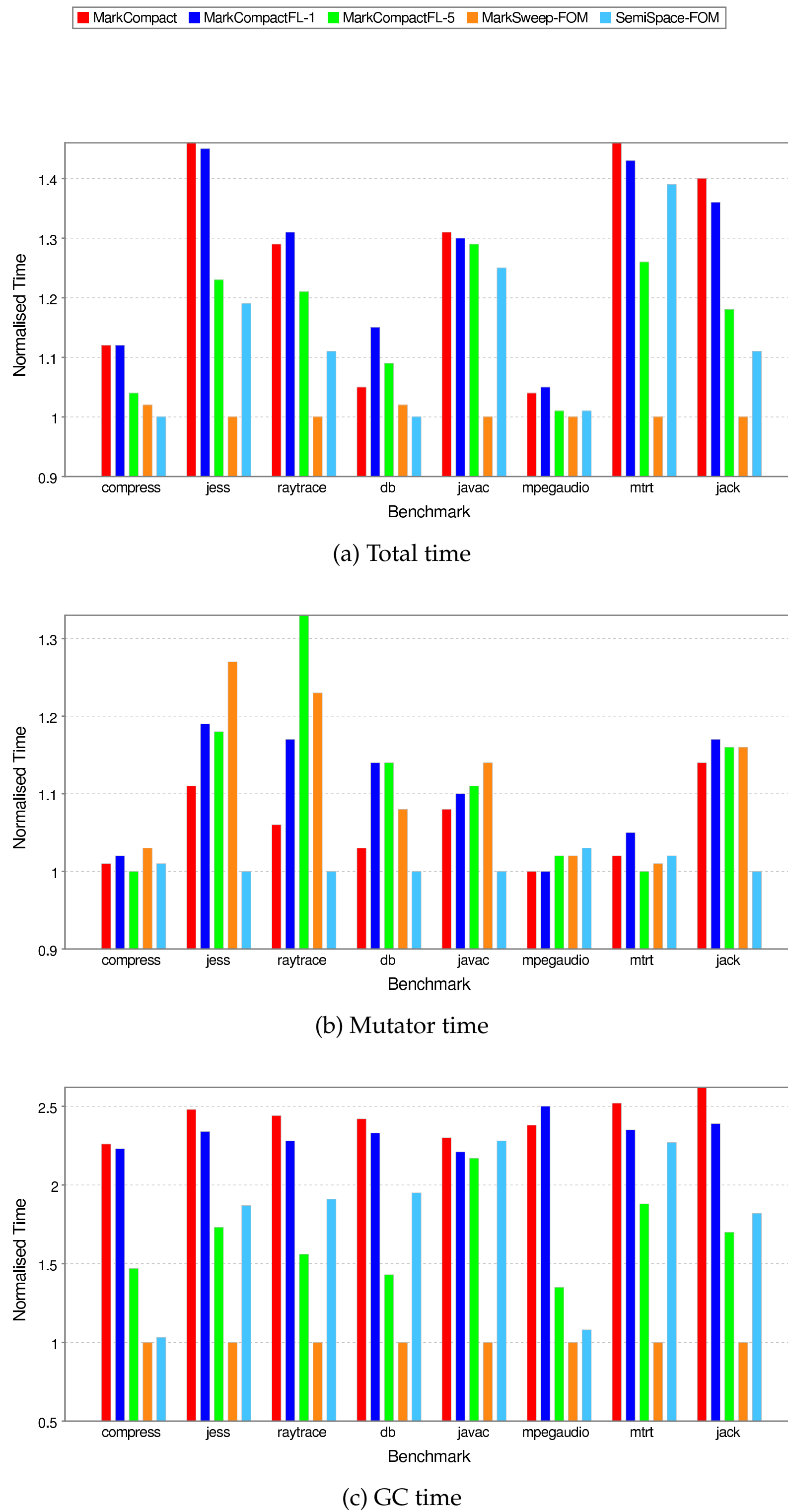
(b) Mutator time



(c) GC time

Figure A.20: Limited physical memory tests with 1GB physical memory.

A.7 Performance Bakeoff

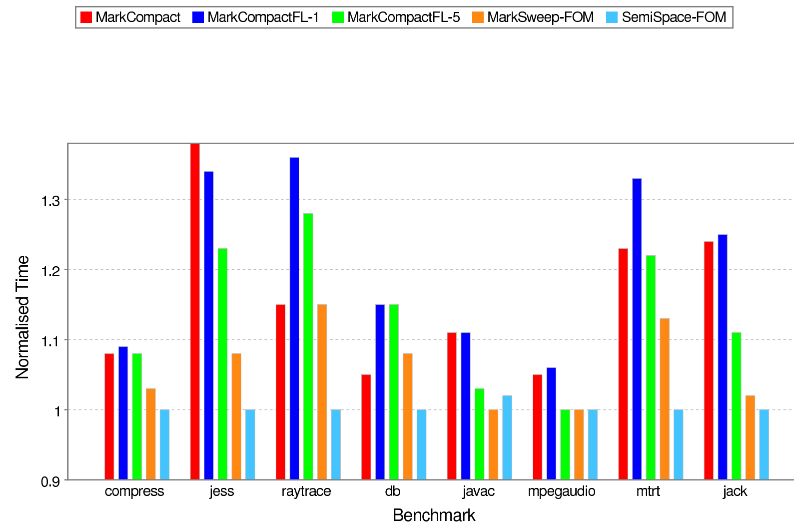


(a) Total time

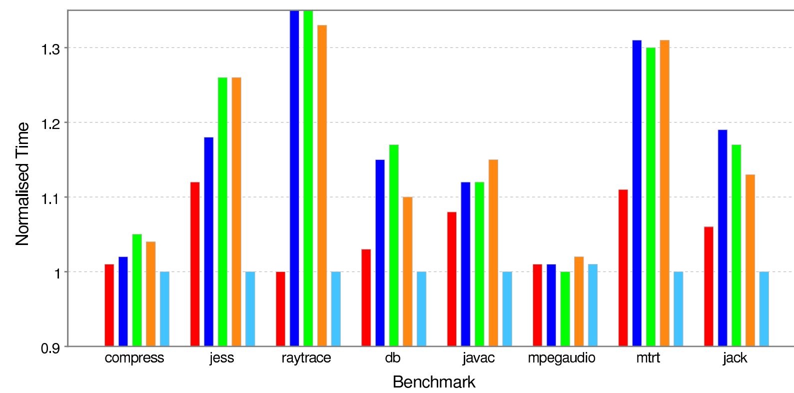
(b) Mutator time

(c) GC time

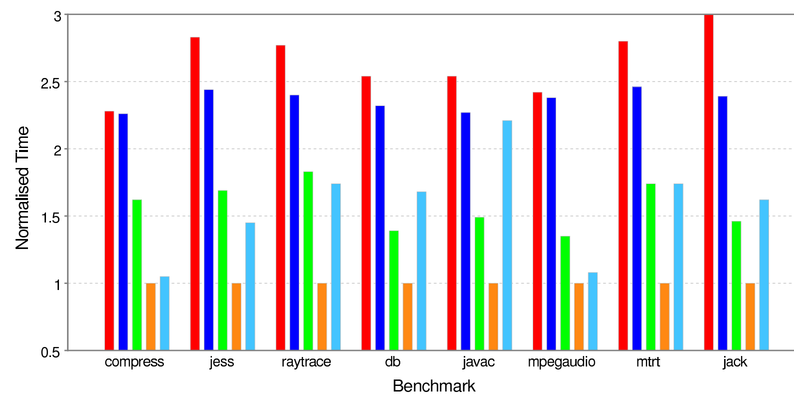
Figure A.21: Summary: Bakeoff of all non-generational collectors (41MB).



(a) Total time

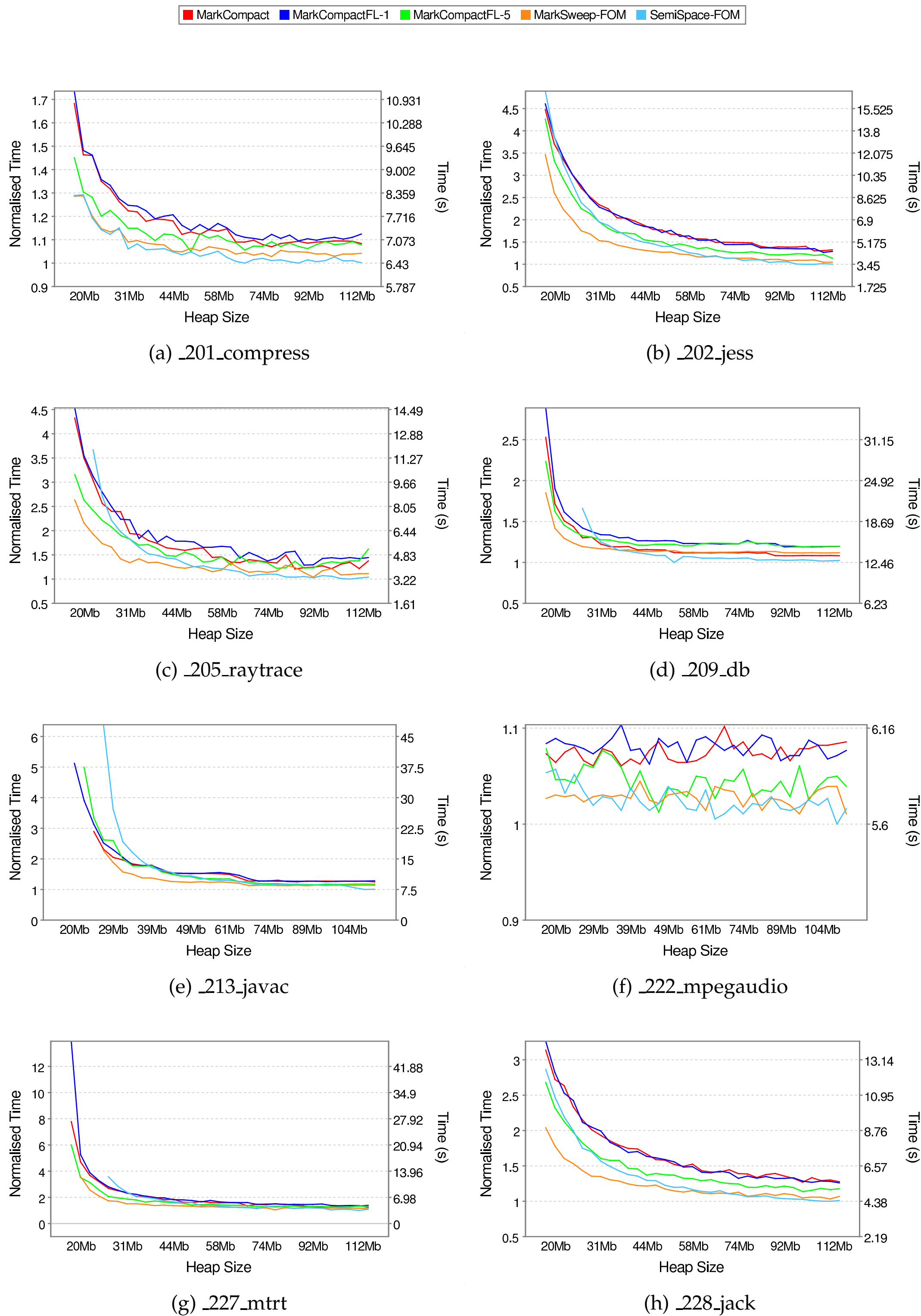


(b) Mutator time



(c) GC time

Figure A.22: Summary: Bakeoff of all non-generational collectors (104MB).



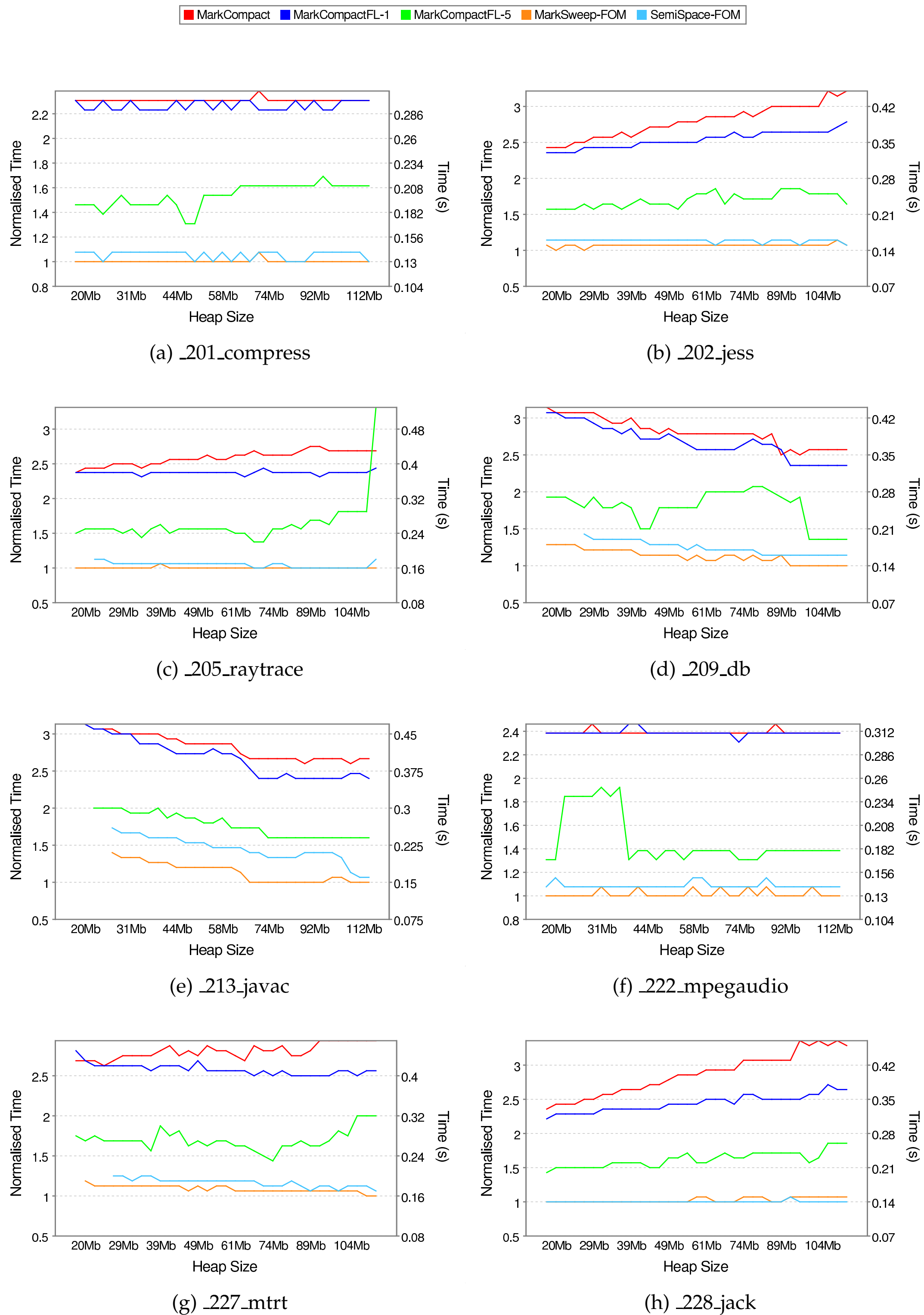
(a) _201_compress (b) _202_jess

(c) _205_raytrace (d) _209_db

(e) _213_javac (f) _222_mpegaudio

(g) _227_mtrt (h) _228_jack

Figure A.23: Total time: Bakeoff of all non-generational collectors.



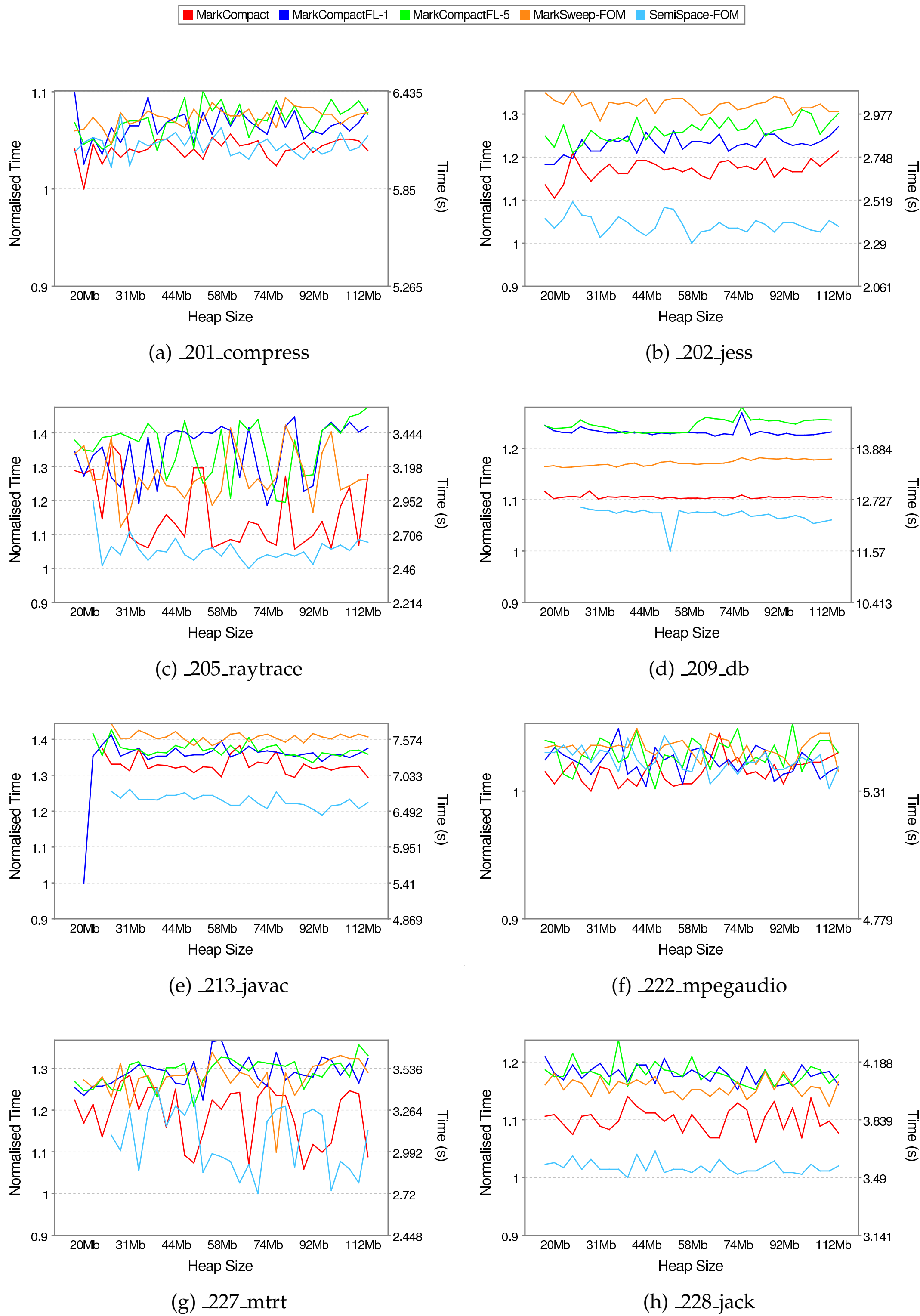
(a) _201_compress (b) _202_jess

(c) _205_raytrace (d) _209_db

(e) _213_javac (f) _222_mpegaudio

(g) _227_mtrt (h) _228_jack

Figure A.24: Average GC time: Bakeoff of all non-generational collectors.



(a) _201_compress (b) _202_jess

(c) _205_raytrace (d) _209_db

(e) _213_javac (f) _222_mpegaudio

(g) _227_mtrt (h) _228_jack

Figure A.25: Mutator time: Bakeoff of all non-generational collectors.

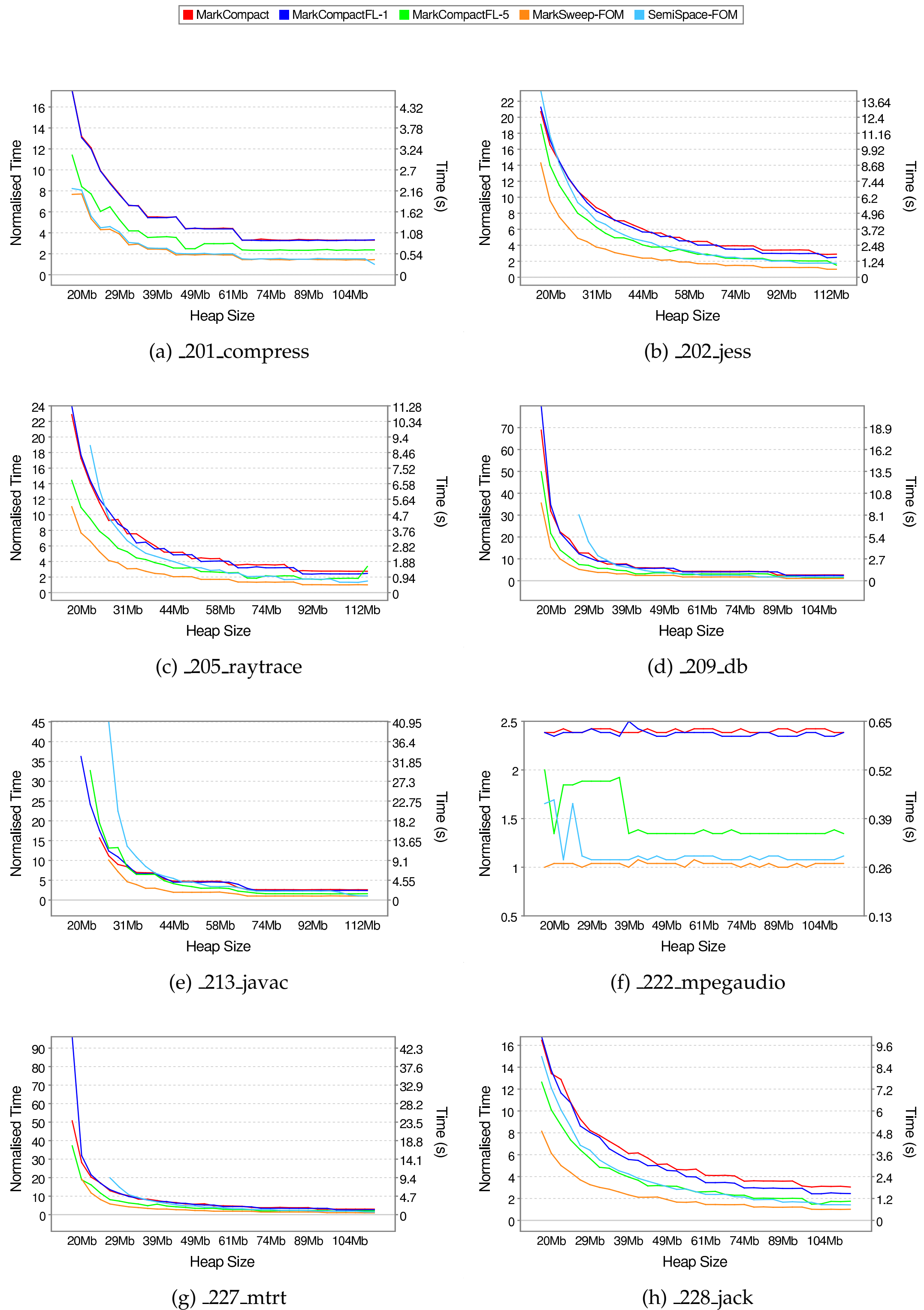
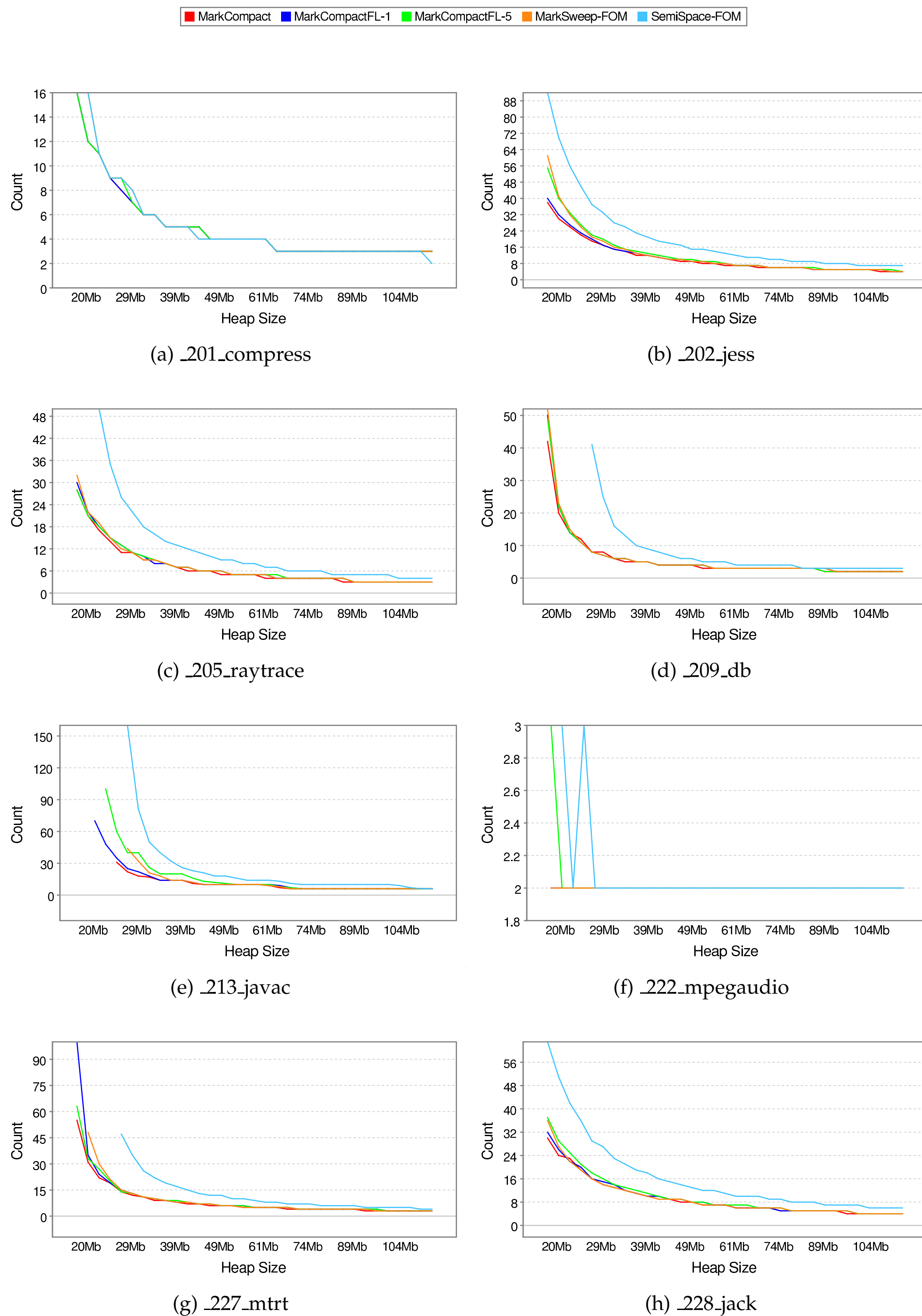


Figure A.26: GC time: Bakeoff of all non-generational collectors.



(a) _201_compress (b) _202_jess

(c) _205_raytrace (d) _209_db

(e) _213_javac (f) _222_mpegaudio

(g) _227_mtrt (h) _228_jack

Figure A.27: GC count: Bakeoff of all non-generational collectors.

A.8 Generational

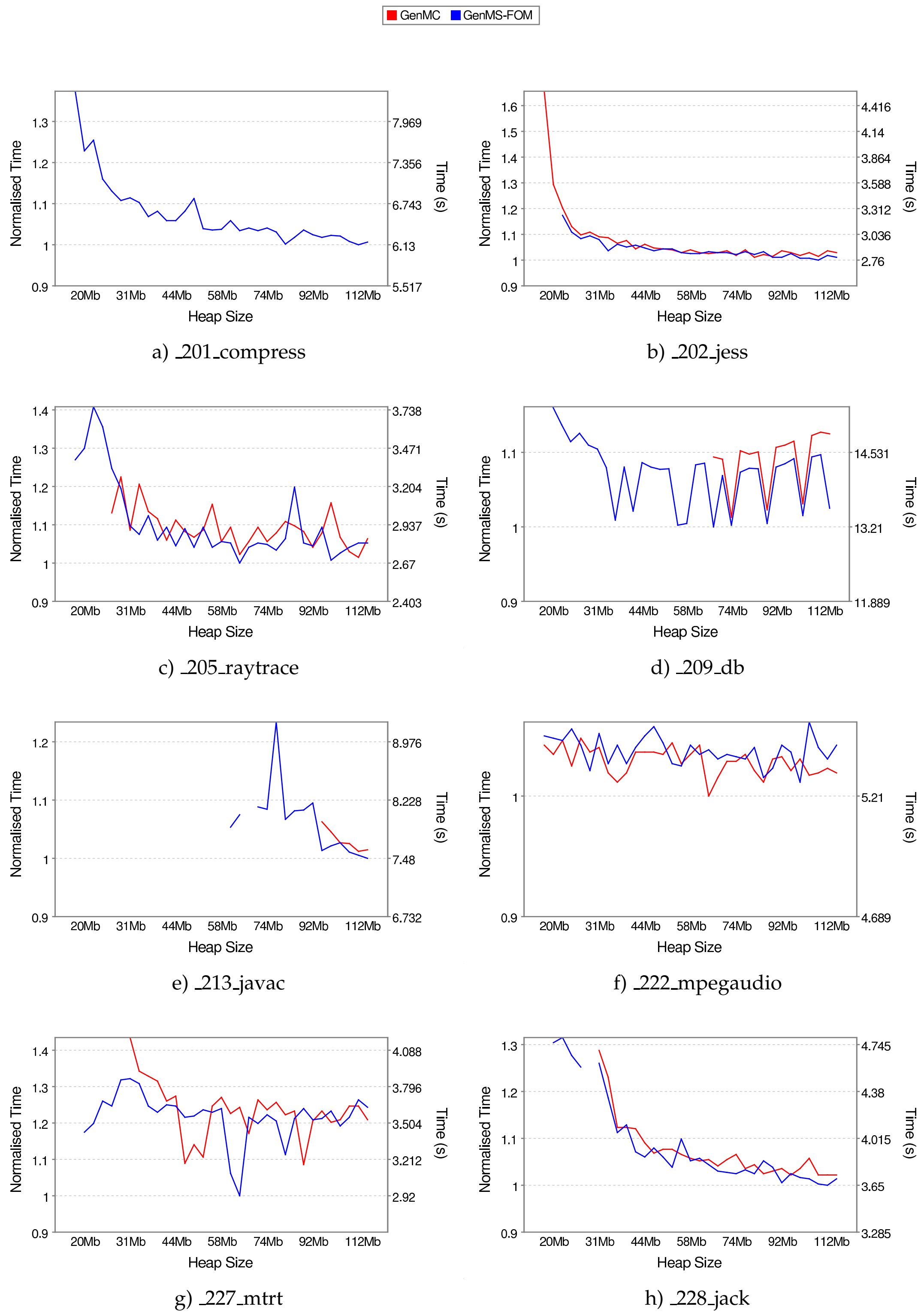


Figure A.28: Total Time: Generational Mark Compact vs. Mark Sweep.

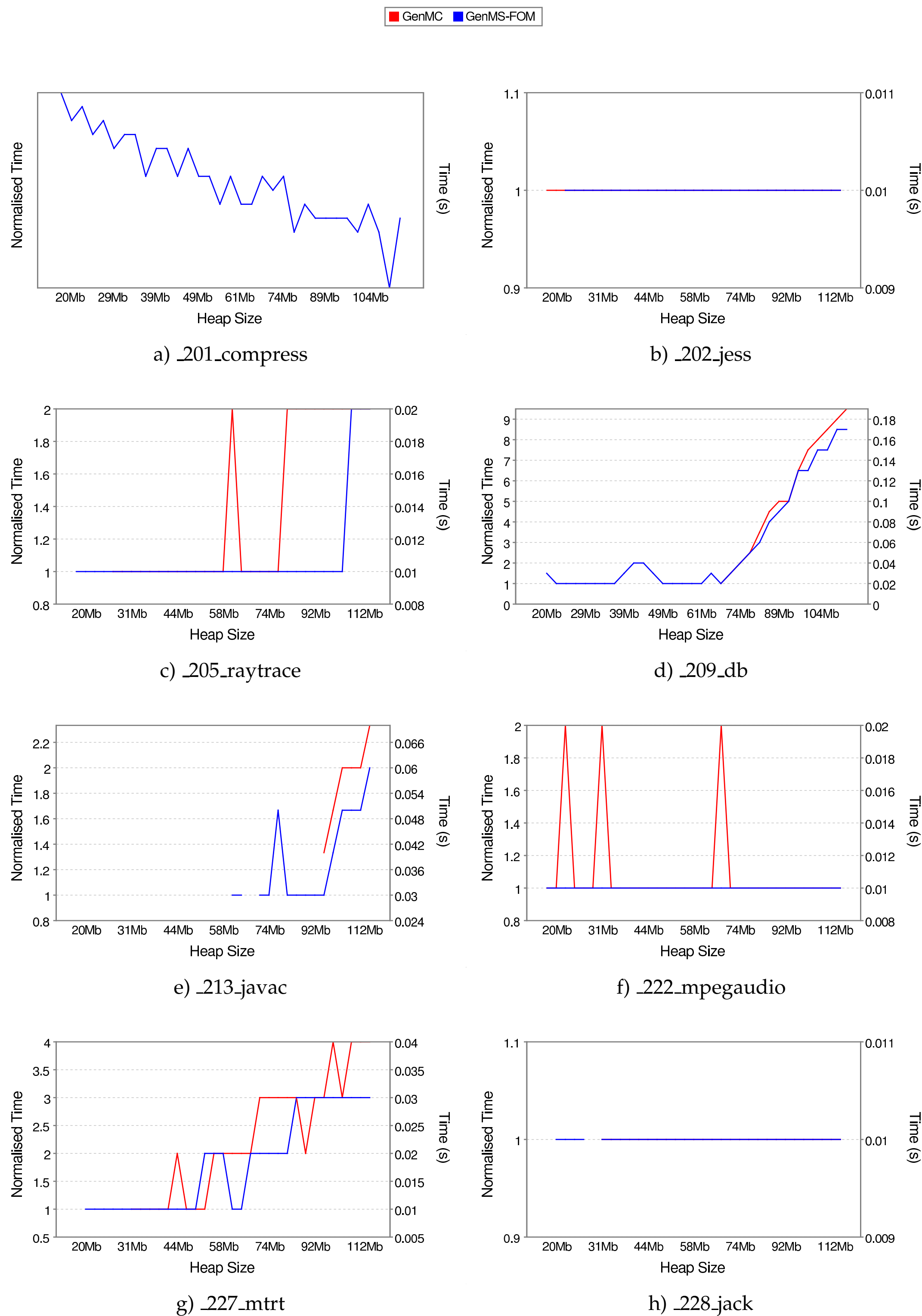


Figure A.29: Average GC Time: Generational Mark Compact vs. Mark Sweep.

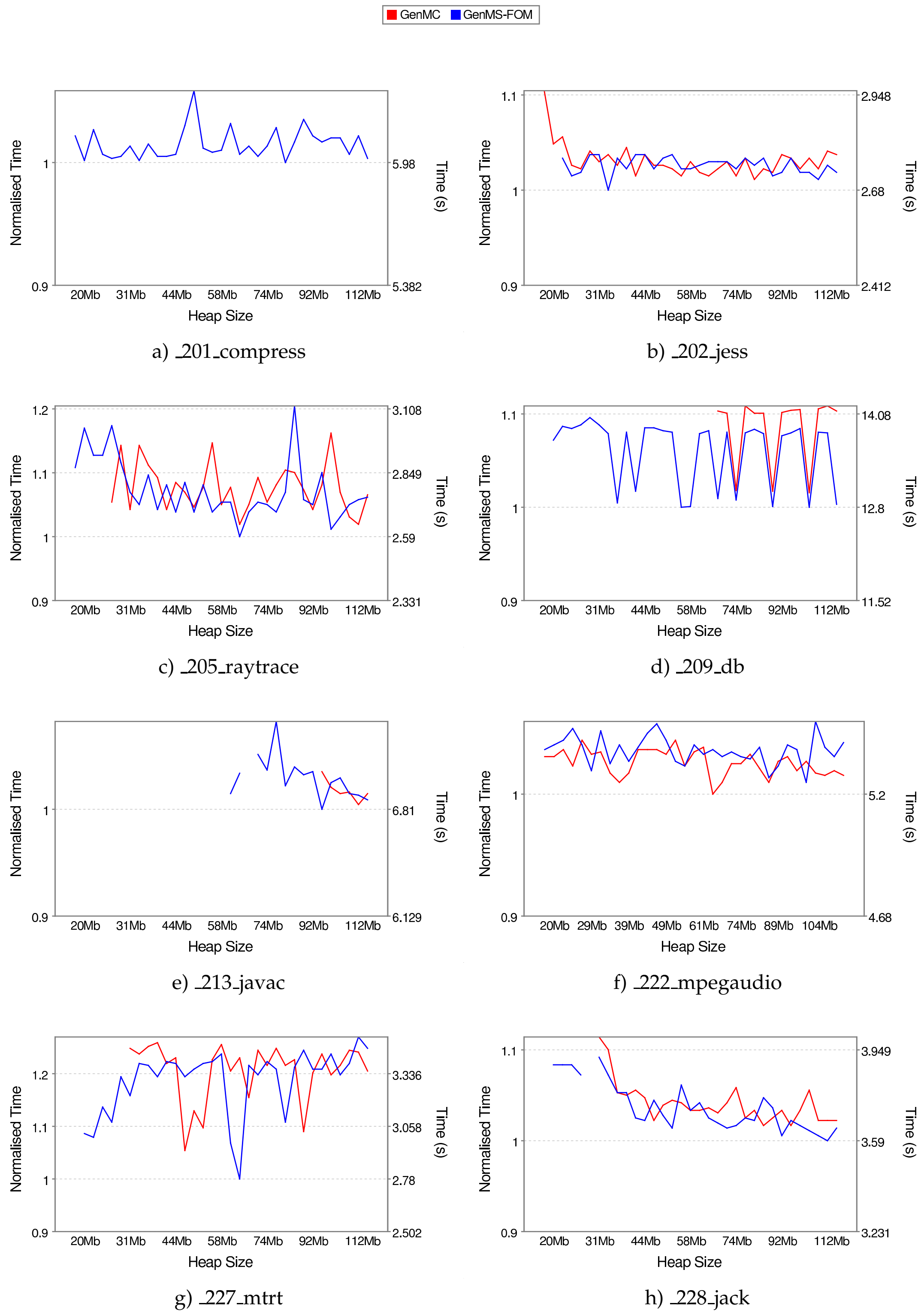
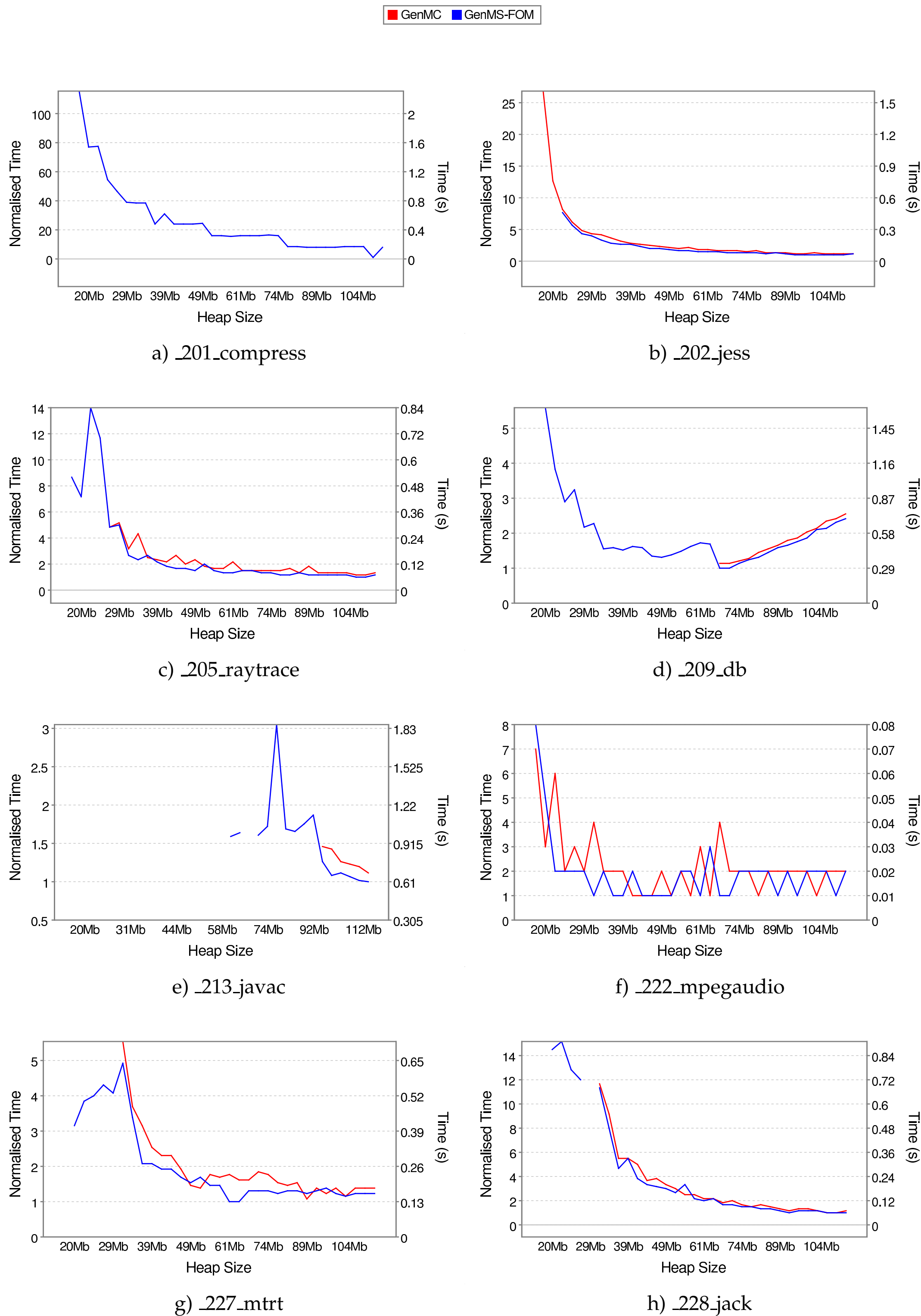


Figure A.30: Mutator Time: Generational Mark Compact vs. Mark Sweep.



a) _201_compress

b) _202_jess

c) _205_raytrace

d) _209_db

e) _213_javac

f) _222_mpegaudio

g) _227_mtrt

h) _228_jack

Figure A.31: GC Time: Generational Mark Compact vs. Mark Sweep.

Glossary

barrier, read: Code that is executed whenever an object is read from.

barrier: See *write barrier* and *read barrier*.

barrier, write: Code that is executed whenever an object is written to.

bitmap: An area in memory storing an array of bits. Most commonly used for marking bits.

collection, concurrent: A *collection* that runs concurrently with mutators.

collection, incremental: A *collection* that can run in several smaller steps or increments.

collection, parallel: A *collection* where multiple kernel threads are involved

collection: The processes of reclaiming space taken by garbage.

garbage: Any object that is not reachable by the executing program but still taking space in the heap.

garbage collection: See *collection*.

GC: See *collection*.

generation: A *space* containing objects related based on age.

heap residency: The percentage of the total size of the heap that is in use.

heap: The area of memory in which dynamically allocated objects reside.

kernel thread: An operating system thread. Multiple kernel threads (such as when multiple processors are involved) are truly concurrent while threads within the runtime are managed by the runtime's scheduler.

mature space: Any *generation* in a generational collector other than the *nursery*. This term is normally used when there are only two generations.

mutator: A thread or process that runs the user program. This is called the mutator as it is the way it changes the object graph that is important for automatic memory management.

nursery: The first *generation* in a generational collector. Generally speaking, objects are first allocated into the nursery.

page fault: Occurs when a request is made for a *page* that is not in the *resident set* of pages.

page: The unit at which virtual memory systems manage memory. For the architecture used the page size is 4096 bytes.

policy: A set of rules or algorithm that governs how objects are allocated or collected.

protected memory: Memory that when accessed can be used to trigger operations such as a *barrier*, or direct program behaviour such as in the case of handling invalid or null references.

resident set: The set of *pages* that is currently in physical memory.

space: An area of *virtual memory* that have allocation and collection *policies* attached to it.

toki: A man that was turned into an ape in the game of the same name by Ocean in 1990. See figure 1.

Figure 1: Toki

virtual memory: In a system with virtual memory, applications deal with virtual addresses that are mapped to physical memory or a backing store such as disk. This, among other things, allows *protected memory*, and ensures processes can not access each others memory.

Bibliography

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. 1999. 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. (p. 19)

ALPERN, B., ATTANASIO, D., 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.).

ALPERN, B., BUTRICO, M., COCCHI, A., DOLBY, J., FINK, S., GROVE, D., AND NGO, T. 2002. Experiences porting the Jikes RVM to Linux/IA32. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '02)* (San Francisco, CA, Aug. 2002). (p. 19)

APPEL, A. W. 1989. Simple generational garbage collection and fast allocation. *Software Practice and Experience 19*, 2, 171–183.

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. 19)

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. (p. 10)

BACON, D. F., FINK, S. J., AND GROVE, D. 2002. Space- and Time-Efficient implementation of the Java object model. In *European Conference on Object-Oriented Programming* (Malaga, Spain, 2002). (pp. 20, 31)

BACON, D. F., KONURU, R. B., MURTHY, C., AND SERRANO, M. J. 1998. Thin locks: Featherweight synchronization for java. In *SIGPLAN Conference on Programming Language Design and Implementation* (1998), pp. 258–268. (p. 21)

BAKER, H. G. 1992. The Treadmill, real-time garbage collection without motion sickness. *ACM SIGPLAN Notices 27*, 3 (March), 66–70. (pp. 14, 25)

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* (Anaheim, CA, Oct. 2003). (pp. 1, 5, 9, 15, 26, 53, 58)

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. (pp. 13, 16)

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. 9)

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* (Anaheim, CA, Oct. 2003). (p. 16)

BOEHM, H.-J. AND WEISER, M. 1988. Garbage collection in an uncooperative environment. *Software Practice and Experience 18*, 9, 807–820.

BORMAN, S. 2002. Sensible sanitation – Understanding the IBM Java Garbage Collector, parts 1, 2 and 3: Garbage collection. (p. 1)

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. 19)

CHENEY, C. J. 1970. A non-recursive list compacting algorithm. *Communications of the ACM 13*, 11 (Nov.), 677–8. (p. 12)

COLLINS, G. E. 1960. A method for overlapping and erasure of lists. *Communications of the ACM 3*, 12 (Dec.), 655–657. (p. 9)

COMFORT, W. T. 1964. Multiword list items. *Communications of the ACM 7*, 6 (June). (p. 8)

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

ECMA. 2002a. *ECMA-334: The C# Language Specification* (Second ed.). ECMA. (p. 1)

ECMA. 2002b. *ECMA-335: Common Language Infrastructure* (Second ed.). ECMA. (p. 1)

JOHNSTONE, M. S. AND WILSON, P. R. 1997. The memory fragmentation problem: Solved? In P. DICKMAN AND P. R. WILSON Eds., *OOPSLA '97 Workshop on Garbage Collection and Memory Management* (Oct. 1997). (p. 8)

JONES, R. The garbage collection bibliography. (p. vii)

JONES, R. E. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley. With a chapter on Distributed Garbage Collection by R. Lins. (pp. vii, 4, 5, 8, 11, 12, 13, 14, 15, 25, 29, 30)

JONKERS, H. B. M. 1979. A fast garbage compaction algorithm. *Information Processing Letters 9*, 1 (July), 25–30. (p. 15)

JOY, B., STEELE, G., GOSLING, J., AND BRACHA, G. 2000. *The Java Language Specification* (Second Edition ed.). Addison-Wesley. (pp. 1, 21)

KNUTH, D. E. 1973. *The Art of Computer Programming* (Second ed.), Volume I: Fundamental Algorithms, Chapter 2. Addison-Wesley.

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. (p. 10)

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. 16)

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.

MICROSYSTEMS, S. 2001. The Java HotSpot Virtual Machine. Technical White Paper. (p. 1)

RICHTER, J. 2000a. Garbage collection: Automatic Memory Management in the Microsoft .Net Framework. *MSDN Magazine*. (pp. 1, 15, 75)

RICHTER, J. 2000b. Garbage collection part 2: Automatic Memory Management in the Microsoft .Net Framework. *MSDN Magazine*.

SANSOM, P. M. 1991. Dual-mode garbage collection. Technical Report CSTR 91–07 (June), Department of Electronics and Computer Science, University of Southampton. *Proceedings of Third International Workshop on Implementation of Functional Languages on Parallel Architectures*. (p. 75)

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. (p. 11)

STANDARD PERFORMANCE EVALUATION CORPORATION. 2003. SPEC JVM Client98 Help. http://www.spec.org/jvm98/jvm98/doc/benchmarks/index.html. (p. 58)

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. 16)

WILSON, P. R. 1994. Uniprocessor garbage collection techniques. Technical report (Jan.), University of Texas. Expanded version of the IWMM92 paper. (p. 5)

WILSON, P. R., JOHNSTONE, M. S., NEELY, M., AND BOLES, D. 1995. Dynamic storage allocation: A survey and critical review. In H. BAKER Ed., *Proceedings of International Workshop on Memory Management*, Volume 986 of *Lecture Notes in Computer Science* (Kinross, Scotland, Sept. 1995). Springer-Verlag. (p. 8)

ZORN, B. 1990. Barrier methods for garbage collection. Technical Report CU-CS-494-90 (Nov.), University of Colorado, Boulder. (p. 9)

ZORN, B. G. 1989. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley. Technical Report UCB/CSD 89/544. (pp. 6, 9, 12)