NOTE TO USERS

HIGH-PERFORMANCE COPYING GARBAGE COLLECTION WITH LOW SPACE OVERHEAD

A Dissertation Presented

by

NARENDRAN SACHINDRAN

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2005

Department of Computer Science

HIGH-PERFORMANCE COPYING GARBAGE COLLECTION WITH LOW SPACE OVERHEAD

A Dissertation Presented

by

NARENDRAN SACHINDRAN

Approved as to style and content by:

J. Eliot B. Moss, Chair

Emery D. Berger, Member

Kathryn S. McKinley, Member

Csaba Andras Moritz, Member

W. Bruce Croft, Department Chair
Department of Computer Science

To Namitha, Shreya, and all my family.

ACKNOWLEDGEMENTS

I have been fortunate to have Prof. Eliot Moss as my advisor during my six years at U.Mass. This thesis would not have been possible without his constant guidance, patience, encouragement, and always insightful advice. Eliot has taught me everything I now know about performing research. Prof. Emery Berger has been advising me for the past two years. He helped give direction to my thesis by making me focus my work on memory-constrained devices. His guidance has been invaluable.

Prof. Kathryn McKinley has been advising me since I joined UMass, and continued to help me constantly even after moving to Texas. She organized a study group at the start of my Ph.D. where I was introduced to garbage collection, and the knowledge I gained at the time has been very helpful in my research. Prof. Csaba Andras Moritz was the external member on my thesis committee and his suggestions during my proposal and thesis defense were very insightful and helped in shaping my work.

I am extremely grateful to Rick Hudson for his initial idea that sparked the work on this thesis. Thanks to IBM Research for the Jikes RVM virtual machine and to Steve Blackburn, Perry Cheng, and Kathryn McKinley for designing and implementing MMTk. The work in this thesis would not have been possible without the Jikes RVM and MMTk infrastructure. I would also like to thank all members of the ALI lab, especially Chris Hoffmann and Asjad Khan, for their constant help.

Many professors have advised and guided me through my years in undergraduate and graduate school, and without their encouragement I would not have chosen a career in research. Thanks to Prof. A. Ravichandran, my project advisor at R.E.C. Trichy, Prof. K.V. Iyer at R.E.C., and Prof. Clement Yu, Prof. Ugo Buy, and Prof. Patrick Troy at the University of Illinois-Chicago.

My father, mother, sister Shubha, and brother-in-law Murali encouraged me to do a Ph.D. I cannot thank them enough for their support and help through all these years. My parents-in-law have also been constantly encouraging. I was inspired to do a Ph.D. by my mother, who got a doctorate after dedicating many years at home to bringing us up.

My wife Namitha has been my greatest source of support. I would certainly not have completed this thesis without her unstinting support, love, and encouragement. Our daughter Shreya has been a source of constant joy. I dedicate this thesis to my wife, daughter and family.

ABSTRACT

HIGH-PERFORMANCE COPYING GARBAGE COLLECTION WITH LOW SPACE OVERHEAD

FEBRUARY 2005

NARENDRAN SACHINDRAN

B.E., REGIONAL ENGINEERING COLLEGE, TRICHY, INDIA

M.S., UNIVERSITY OF ILLINOIS AT CHICAGO

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor J. Eliot B. Moss

Managed run-time environments such as Java and .NET are now used for a wide variety of applications. They provide a number of advantages to application developers, including reduced application development time, safer code, and portability. A major component of the run-time systems for these environments is the garbage collector, which relieves application developers from having to manage memory explicitly. While garbage collectors should provide good throughput and short response times with low space overheads, most modern garbage collectors that meet the first two requirements tend to impose a significant space overhead.

Copying garbage collection, one of many garbage collection techniques, provides a number of advantages over other collection methods including cheap allocation and prevention of fragmentation. However, the space overhead of previous copying techniques have been particularly high. In this thesis, we first perform a detailed study of previous

copying collection techniques, including generational copying collection and the Train collector. We identify limitations of the Train collector that cause it to perform poorly especially in the presence of large, cyclic data structures. We describe a new copying garbage collection technique, Mark-Copy, that extends generational copying collection, and overcomes the factor-of-two space overheads of a regular copying collector. Additionally, it overcomes some of the limitations of the Train collector. We show that Mark-Copy can provide high throughput while running in limited amounts of memory. We then present MC^2, a collector that extends Mark-Copy and provides high throughput and short pause times while running with low space overhead, thus making it suitable for applications running on memory-constrained devices. These qualities also make MC^2 attractive for other environments, including desktops and servers.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Java and .NET are becoming increasingly popular platforms because of the advantages that they provide, including safety and portability. Garbage collection, which is an important feature of the runtime systems for these platforms, relieves programmers from the burden of explicitly freeing dynamically allocated memory, thus improving programmer productivity and making applications more reliable.

Garbage collectors must typically meet three requirements. They must provide high throughput, yield short response times, and run with low space overheads. Most modern collectors do not meet all three of these requirements and require significant space overheads in order to provide good throughput and response times. Copying collectors have especially high space overheads, typically a factor of three or more.

Low space overheads are particularly important for handheld devices such as cellular phones and PDAs. These devices are now widely used, and their market is growing rapidly. They tend to have small amounts of memory (typically a few 100 kilobytes to a few megabytes) and they have a limited power budget. Since the memory subsystem consumes a large fraction of the power, it is important to use memory efficiently.

Handheld devices also run a wide variety of applications that have diverse requirements. Smart cellular phones are voice-centric devices, but they also typically run applications such as video games, media players, e-mail clients, and internet browsers. These phones also include digital cameras that can be used to capture and transmit still images and video. Data-centric handheld devices such as PDAs run scaled-down versions of many desktop

applications. More recently, applications such as GPS navigation are becoming popular on these handheld devices.

Most of these applications handle large amounts of data and they require good throughput and response times from the underlying system while running in constrained memory. To use Java on these devices, it is important to design garbage collectors that can provide good throughput and response times while running with limited amounts of memory.

Additionally, any type of system, including desktops and servers, is ultimately memory constrained. The amount of memory on a system is restricted either by hardware constraints (e.g., 32-bit hardware), or by budget constraints. Modern applications, however, require increasing amounts of memory. Multimedia, scientific modeling, and database applications are examples of applications that work with large amounts of data. Web servers need to scale to thousands of users and server memory can be very expensive. These trends necessitate a renewed focus on developing garbage collection algorithms that can perform efficiently in constrained memory.

1.1 Limitations of Modern Garbage Collectors

Most modern garbage collection techniques cannot provide good throughput and short response times while running in small memories, and thus do not meet the requirements of applications that run on memory-constrained devices. We describe here the limitations of popular garbage collection techniques: mark-sweep, mark-compact, copying, generational, and reference counting collection.

Mark-Sweep: A number of collection techniques that can provide good response times use mark-sweep collection. Examples of collectors in this category are the collector by Dijkstra et al. [30] and Yuasa's collector [69]. A problem with mark-sweep collectors is that they suffer from fragmentation. Johnstone and Wilson [36] show that fragmentation is not a problem for carefully-designed allocators, for a range of C and C++ benchmarks. However, they do not demonstrate their results for long-running programs, and our expe-

rience indicates that fragmentation can be a problem for Java programs. Susceptibility to fragmentation makes purely mark-sweep collectors unsuitable for devices with constrained memory.

Researchers and implementors have also proposed mark-sweep collectors that use compaction techniques to combat fragmentation. Ben-Yitzhak et al. [11] describe a scheme that incrementally compacts small regions of the heap via copying. However, their technique requires additional space during compaction to store metadata, and does not address the problem of large metadata. Further, for a heap containing n regions, the technique requires n marking passes over the heap in order to compact it completely. These passes can lead to poor performance when the heap is highly fragmented.

The only mark-sweep collector we are aware of that meets all the requirements we lay out for handheld devices is the Metronome collector implemented by Bacon et al. [7]. Metronome uses mark-sweep collection and compacts the heap using an incremental copying technique that relies on a read barrier. It can provide good throughput and CPU utilization in constrained memory. However, in order to make the read barrier efficient, it requires advanced compiler optimization techniques.

Mark-Compact: Mark-(sweep)-compact (MSC) collectors [26, 33, 38, 42] use bump pointer allocation, and compact data during every collection. These collectors prevent fragmentation and typically preserve the order of allocation of objects, often yielding good locality.

Compaction typically requires two or more passes over the heap. However, since these heap traversals exhibit good locality of reference, they are efficient. MSC collectors can provide good throughput and their space utilization is excellent. They run efficiently in very small heaps. However, they tend to have long pauses.

Copying Collection: Purely copying collection techniques also have versions that can provide good response times. The most well-known of these are Baker's collector [9], Brooks's collector [19], and the Train collector [34]. Baker's and Brooks's techniques use

semi-space copying and hence have a minimum space requirement equal to twice the live data size of a program. Also, they use a read barrier, which is not very efficient.

The Train collector can run with very low space overheads. It can suffer from large remembered sets, though there are proposals on limiting that space overhead [48]. However, our experiments with the Train collector show that it tends to copy large amounts of data in small heaps, especially when programs have large, cyclic structures.

Generational Collection: Generational collectors divide the heap into multiple regions called *generations*. Generations segregate objects in the heap by age. A two-generation collector divides the heap into two regions, an allocation region called the *nursery*, and a promotion region called the *old generation*. Generational collectors trigger a collection every time the nursery fills up. During a nursery collection, they copy reachable nursery objects into the old generation. When the space in the old generation fills up, they perform a *full collection* and collect objects in the old generation.

Generational collectors can provide good throughput and short average pause times. However, the limitations of the collection technique used in the old generation (namely, fragmentation, minimum space overhead being twice the maximum live size, and large maximum pause time) determine the overall space requirements and pause time characteristics of the collector. The drawbacks of MS, copying, and MSC therefore carry over to generational collection.

Reference Counting: Reference counting collectors can provide good throughput and short pause times. However, like mark-sweep collectors, they can suffer from fragmentation, and thus require additional compaction schemes to run in small heaps. Blackburn and McKinley [15] present a generational reference counting collector that can provide excellent throughput and CPU utilization. However, in order to provide short pauses, the algorithm can have high space overheads. This overhead is required because the collector performs a fixed amount of collection at every invocation, thus placing a hard bound on the pause time. Doing collection in this manner requires occasionally expanding the heap for

the collector to keep up with the allocation rate, thus making the collector unsuitable for memory constrained devices.

Summary: Mark-sweep collectors and reference counting suffer from fragmentation. Most compaction techniques that combat fragmentation either do not limit metadata overheads or are not efficient when fragmentation is high. An exception is the Metronome collector. However, Metronome requires advanced compiler optimizations in order to perform well.

Mark-compact collectors can provide good throughput with little space overhead, but they can suffer from long pauses. Copying collectors typically have a minimum factor-of-two space overhead and require additional space in order to provide good throughput and pause times. Generational collectors provide good throughput and short average pauses, but their overall performance depends on the collection technique used in the old generation. Thus a generational mark-sweep collector suffers from fragmentation, a generational mark-compact collector suffers from occasional long pauses, and a generational copying collector suffers from a high minimum space overhead.

1.2 Thesis Contributions

The main contribution of the thesis is a new copying garbage collection technique that is suitable for applications running on memory-constrained devices. The collector overcomes the factor-of-two overhead of standard copying collectors and can provide good throughput and response times while running in limited amounts of memory.

The thesis first evaluates in detail the Train collector, the only previous purely copying collector we are aware of that does not require a factor-of-two minimum space overhead. We describe an implementation of the Train collector in the Jikes RVM virtual machine and present a performance evaluation. We find that the collector suffers from some performance issues such as high metadata and copying overheads. We show that even though

the collector has low space overheads, its high copying overheads make its performance unsuitable for memory constrained environments.

We then propose and evaluate a new garbage collection technique, called *mark-copy* (MC) [50]. Like MOS, MC divides the heap into a number of small regions, and copies one or more of these regions successively, thus overcoming the factor-of-two overhead associated with copying collection. However, it uses a separate marking phase that makes the collection efficient and ensures that the collection is *complete* (all garbage on the heap is eventually collected). MC improves over the Train collector by reducing the amount of copying work significantly. We describe the MC algorithm and an implementation in Jikes RVM. We also present a performance evaluation that compares the performance of MC with state-of-the-art collection techniques. We show that MC can provide high throughput in small heaps. However, it can suffer from occasional long pauses and high metadata overhead.

To overcome the limitations of MC, we present MC^2 (Memory-Constrained Copying) [51], a collector that extends MC and makes it suitable for memory constrained devices. MC^2 has low, bounded space overheads and short pause times, and can provide soft real-time performance. We present details of the collection algorithm, an implementation of MC^2 in Jikes RVM and a comparison of its performance with efficient mark-sweep and mark-compact collectors.

1.3 Outline

Chapter 2 presents some background in garbage collection. We describe all major collection techniques and explain their limitations. Chapter 3 describes the experimental framework and methodology we use for our implementation and performance evaluation. Chapter 4 contains a description of the Train collection technique, details of our implementation of the collector, and a performance evaluation. Chapter 5 describes the MC collector and presents an implementation and evaluation of the collector. Chapter 6 presents MC^2

and describes how it overcomes the limitations of MC. It presents implementation details of MC^2 and a comparison of its performance with state-of-the-art high performance collectors. Finally, Chapter 7 concludes and proposes directions for future work.

CHAPTER 2

BACKGROUND

We describe here current garbage collection techniques along with limitations of each of the techniques. We consider only *accurate* garbage collection, i.e., techniques that can always determine unambiguously whether a value is a pointer or not. We do not deal with conservative garbage collection, which can function without perfect type knowledge. While conservative collectors are useful for languages like C++ that do not support garbage collection, we focus on languages that require garbage collection, specifically Java. Jones and Lins' garbage collection text [37] and Paul Wilson's GC survey [67] contain more detailed descriptions of the collectors we discuss here.

Garbage collectors fall into four categories:

- *Non-incremental collectors* stop program execution for the entire garbage collection period.
- *Incremental collectors* interleave collection effort with program execution, yielding shorter pauses than non-incremental collectors.
- *Parallel collectors* are either non-incremental or incremental collectors adapted to run on multiprocessor systems. They stop program threads running on all processors before starting collection, and then divide the collection work among multiple collector threads (typically one collector thread per processor).
- *Concurrent collectors* run on multiprocessor systems with one or more collector threads running concurrently with the program's threads.

We do not consider parallel and concurrent collection techniques here, and so limit our discussion to non-incremental and incremental techniques. Before we describe these collection techniques in detail, we define their terms.

2.1 Reachability

An ideal garbage collector reclaims any space on the heap that will never be used by the running program in the future. Most garbage collectors, however, use a simpler, more conservative *reachability* criterion to determine whether space on the heap needs to be retained or reclaimed.

Garbage collectors define a *root set* that includes statically-allocated pointer variables, local pointer variables that are allocated in activation records, and pointers stored in registers. When triggered, a garbage collector considers objects on the heap directly referenced from the root set as *live*. Additionally, it considers any object that is reachable from a live object as live. Thus, garbage collection retains space occupied by any object that is reachable along some pointer path starting from the root set, and reclaims all other space.

2.2 Tracing vs. Reference Counting

Garbage collectors can be broadly classified as either *tracing* or *reference counting* collectors. Bacon et al. [8] show tracing and reference counting to be duals of one another, and that all garbage collectors are various types of hybrids of tracing and reference counting.

Tracing collectors [43, 31] perform garbage collection by starting from the root set and traversing the object graph to find reachable objects. During the traversal, a tracing collector processes each object it reaches. At the end of the traversal, tracing collection reclaims space occupied by unprocessed objects.

Reference counting collectors [27] maintain a count of the number of references to every object on the heap. Every time a program creates a reference to an object, the collector increments the reference count for the target object. When the program deletes a refer-

ence to an object, the collector decrements the object reference count. Reference counting collection reclaims space occupied by any object whose reference count is zero.

2.3 Tricolor Marking

When studying tracing collectors, it is helpful to use the abstraction of tricolor marking [30], originally developed for concurrent collection. At the start of a collection, all objects in the heap are conceptually colored white. As the collector traverses the object graph, every object that is reached (but not yet processed) is colored gray. After an object is completely processed (i.e., all objects that it references have been either colored gray or black), its color is changed to black. At the end of the collection, all reachable objects are colored black and the unreachable objects remain white.

2.4 Non-Incremental Tracing Collectors

Non-incremental collectors are best suited to uniprocessor systems. They collect the entire heap all at once, usually when no free space exists in the heap, with program execution suspended for the duration of the collection. Non-incremental collection normally causes long pauses and may not be suitable for interactive applications. We consider only tracing collectors in this category since reference counting collectors are typically incremental with short pause times.

We consider in turn mark-sweep, mark-compact, and copying collectors.

2.4.1 Mark-Sweep

Mark-Sweep (MS) collectors [43] perform collection in two phases. In the mark phase, MS traverses and marks objects reachable from roots, and in the sweep phase, MS reclaims space occupied by unreachable (unmarked) objects.

The Boehm-Demers-Weiser collector [18] is a widely used MS collector. It uses a two-level allocator and segregated size classes. A low-level allocator acquires fixed-size

blocks from the OS. Each block stores objects of a single size in order to minimize external fragmentation, and a high-level allocator assigns objects to the blocks. The allocator stores each block in a list corresponding to its size class. To allocate an object, the allocator finds the smallest size class that can accommodate the object, and it then places the object in the first available slot in the free list for the size class. The mark phase of the collector marks reachable objects, and the sweep phase (performed lazily) reclaims unreachable space. The collector returns all empty blocks to a global pool from which they can be assigned to any size class.

2.4.1.1 Space Overhead

Space overheads for MS include a single mark bit per object and a mark stack to hold the addresses of marked but unprocessed objects.

Mark bits can be stored directly in objects (if there is space available in the object header) or in a separate bitmap table. A bitmap table has space overhead approximately equal to 3% (1/32) of the total heap space for word aligned objects on a 32- bit architecture. The advantages of a bitmap table are improved marking locality and better sweeping efficiency (since the collector does not need to touch objects). However, setting a bit in the bitmap may be a costlier operation.

The mark stack is usually small but can grow to be as large as the heap in the worst case. The Boehm-Weiser collector [18] minimizes stack depth by storing start and end addresses of large objects on the stack and pushing at most 128 words of an object onto the stack at any time. The collector handles stack overflow by dropping new entries when the stack is full, replacing the stack with a new one twice the original size, and restarting marking from unmarked children of marked objects.

The Deutsch-Schorr-Waite collector [53] uses a technique called *pointer reversal* that avoids using a mark stack. Pointer reversal stores the address of the parent node in one of the pointer fields of the node currently being processed. For binary trees, this technique

requires only one additional bit per object. However, for marking variable size nodes [63] this technique requires $log(n)$ bits where n is the maximum number of pointers an object can contain.

2.4.1.2 Collector Properties

MS collection has a number of advantages. It is relatively simple to implement. It does not require any special action for reclaiming cyclic data structures. MS does not place any overhead on pointer operations. Also, it does not relocate data, which reduces collection costs.

However, MS has some limitations. Like most tracing garbage collectors, it thrashes (spends most of the time in garbage collection) when heap residency is high. Also, the sweep phase of the collector needs to scan the entire heap and not just the live objects. However, using a bitmap and *lazy sweeping* [18, 70] makes this cost very small.

The biggest disadvantage of MS is that it tends to fragment the heap. Fragmentation has two effects. First, it adds to the space overhead, preventing objects from being allocated even when there is enough free space in the heap. Second, fragmentation distributes objects over more pages than necessary, and possibly intersperses old and new objects, thus degrading program locality.

The only solution to fragmentation is to move and compact objects regularly. Mark-compact and copying collectors (described below) perform compaction as part of the collection process. Mark-sweep collectors typically need to be used in conjunction with a compaction scheme in order to overcome fragmentation.

2.4.2 Mark-Compact

Mark-(Sweep)-Compact (MSC) collectors perform multiple (2–3) passes over the heap during garbage collection. Compaction algorithms can be classified by the relative position of objects after compaction. *Arbitrary compaction* moves objects without regard to their original order. Although arbitrary compaction is simple to implement and fast, it yields

poor spatial locality. *Sliding compaction* moves objects to one end of the heap, thus maintaining allocation order. This usually results in good locality of reference.

Four types of compacting collectors exist: *two-finger, forwarding address based* (Lisp-2), *table based*, and *threaded*. We describe each of these collection styles below. Cohen and Nicolau [26] also describe these collectors and present a theoretical evaluation.

2.4.2.1 Two Finger

Edwards' two finger compaction [52] performs two passes over the heap after the mark phase. The mark phase marks live objects on the heap and counts them. The first compaction pass relocates objects from the higher end of the heap to holes in the lower end, overwriting the first field of copied objects with a forwarding address. The second pass scans the compacted region and updates pointers.

The collector has linear complexity and does not require additional memory. It may disperse objects after collection but it maintains their relative order. This compaction usually does not improve spatial locality. Edwards' collector is suitable for fixed-size objects. However, it can be extended to handle variable size objects if objects of different size are allocated in different regions of the heap.

2.4.2.2 Lisp-2

Lisp-2 compaction makes three passes over the heap after the mark phase. It requires an additional word per object in order to perform compaction. The first pass calculates a forwarding address for each object and stores it in the object header. The second pass updates all pointers with the forwarding address. The third pass copies objects to their new locations.

Lisp-2 preserves allocation order and can handle objects of varying sizes. However, it requires an additional word per object, which affects space utilization and program locality.

2.4.2.3 Table-Based Compaction

Table-based compaction [33] makes two passes over the heap after marking. In the first pass, it compacts data and builds a *break table* of relocation information in the free space. Break table entries store the start address and size of each free chunk. As the collector compacts chunks of objects, it slides the table down the heap, which may cause entries to become unsorted. After the first pass completes, the collector sorts the break table. In the second pass, the collector updates pointers. To find the new address for a pointer p, it searches the break table for adjacent entries (a,s) and (a',s') such that $a \leq p < a'$. The new pointer value will be $p - s$. While the search is an $n \log(n)$ operation, a hash table (if space is available to construct it) can improve search time.

Table-based compaction preserves object allocation order and does not require additional space. The pointer update phase has good locality since it references only the break table and does not need to touch objects in order to find relocation information.

2.4.2.4 Threaded Compaction

Threaded compaction [32] arranges the heap so that every reference to an object is chained to the object using a linked list. Jonkers' threaded collector [38] performs two compaction passes after marking. During the first pass, the collector threads forward pointers. It also maintains a count of the total amount of live data. When the scan reaches an object, it calculates its new address (using the current live data count) and updates all threaded forward pointers. During this pass, the collector also threads backward pointers. In the second pass, the collector actually moves objects and updates all backward pointers with the new object address. An improvement over Jonkers' technique described by Martin [42] eliminates one compaction pass by combining it with the mark phase.

Threaded compaction is suitable when objects have a pointer-sized header field that contains non-pointer (or immutable) information. It requires no additional space when there are no internal pointers and it also preserves allocation order.

2.4.2.5 Space Overhead

Mark-compact collectors have low space overheads. Most compacting collectors can function without requiring any additional space. Some, like Lisp-2 (additional header word) and threaded schemes (internal pointers), have some bounded space overhead. An additional mark bitmap, while not required, can significantly improve performance.

2.4.2.6 Collector Properties

Mark-compact collectors use bump pointer allocation, which makes allocation cheap. Since free space on the heap is contiguous, the collectors perform allocation by incrementing a pointer across the free portion of the heap.

Compaction prevents fragmentation. Most compaction schemes also preserve object allocation order and thus can improve program locality. Additionally, once data is compacted, it often does not need to be copied again.

However, mark-compact can be expensive since it needs to make several passes over the heap after the mark phase. Also, some mark-compact collectors cannot be made incremental.

2.4.3 Copying

We now consider copying collection techniques. We first describe semi-space collection, the most basic copying collection technique. We then describe generational copying collection and modern copying collectors such as Older First and Beltway.

2.4.3.1 Semi-Space

A Semi-space (SS) collector [31] divides the heap into two semi-spaces, called *from* space and *to* space. It allocates into from-space, and collects when from-space fills up. SS finds objects on the heap that are directly reachable by tracing the roots, and it first copies these objects into to-space. It then performs a copying traversal over these objects (e.g., a Cheney scan [23]) and copies all other reachable objects into to-space. After copying,

it frees from-space and reverses the role of the two spaces. Although SS is simple to implement, it is inefficient. First, it can use only half the total available space. Second, it scans the entire heap every time it performs a collection.

2.4.3.2 Generational Copying Collection

Generational collectors [40, 64] are the most widely used copying collectors. Generational copying collectors divide the heap into multiple regions called *generations*. Generations segregate objects in the heap by age. A two-generation copying collector (2G) uses two regions, an allocation region called a *nursery*, and a promotion region consisting of two semi-spaces called the *old generation*. There are two types of 2G collectors. A *fixed-size nursery collector* (FG) maintains a constant size nursery, while a *variable-size nursery collector* (VG), e.g., in the manner of Appel [3], allows the nursery to consume up to half the available space in the heap. VG collectors usually perform better than FG collectors, but at the expense of longer average pauses (periods during which application program execution is suspended in order to perform garbage collection). We also define a *bounded-size nursery collector* (BG), an extension of the FG collector that uses a variable size nursery with an upper bound on the nursery size. The BG collector tends to perform better than an FG collector, while providing a bound on nursery collection pauses.

Generational collection is based on the hypothesis that most objects live a very short time, while a small fraction live much longer. A generational collector thus filters out short-lived objects by performing frequent nursery collections and reclaims space occupied by dead long-lived objects by performing less frequent older generation collections.

Since a generational copying collector requires an entire semi-space as a region into which to copy survivors during old generation collections (full collections), it suffers from one of the problems of an SS collector: heap occupancy can never exceed half the heap space. It thus requires space equal to at least twice the maximum live size of a program to

be able to operate. Generational copying collectors usually achieve good performance only at heap sizes that are at least 2.5–3 times the maximum live size.

Similarly, generational collectors that manage the old generation using other techniques, such as MS and MSC, suffer from the same problems that these collectors face. A generational MS collector needs to handle old generation fragmentation and a generational MSC collector needs to handle occasional long pauses.

2.4.3.3 Older First

The Older First (OF) collector [59, 60, 61] exploits the fact that generational copying collectors prematurely copy the very youngest objects in the nursery. It lays out objects in the heap in order of decreasing age and slides a fixed-size window across the heap starting from the oldest objects. At each collection, it copies reachable objects that are in the window, and then slides the window towards younger objects. Thus, it usually avoids copying the youngest objects. When the window bumps into the allocation point (i.e., copies the youngest reachable objects), it is reset to the oldest end of the heap. OF generally reduces the amount of copying and outperforms generational copying collectors. Also, it requires additional space equal to the size of one collection window, not an entire semi-space. However, the OF technique is not *complete*: it will never reclaim cycles of garbage that are larger than the collection window.

2.4.3.4 Beltway

The Beltway collectors [13] include two configurations that perform significantly better than a generational copying collector. The Beltway.X.X collector adds incrementality to the generational copying collector by dividing the generations (called *belts*) into fixed size increments. It collects only one increment at a time, hence the additional free space required at any time is equal to the increment size. However, for any increment value below 100%, the collector is not complete: it suffers from the same problem as OF. The Beltway.X.X.100 collector solves the completeness problem by adding a third belt with a single increment,

performing a full collection when the third belt grows as large as half the heap space. Since the Beltway collectors determine the amount of space reserved for copying dynamically, the Beltway.X.X.100 collector does use more than half the heap space. However, since the third belt cannot grow to be larger than half the heap space, the Beltway.X.X.100 collector has the same problem as the generational copying collector: in general, it cannot run in a heap smaller than twice the maximum live data size for a program.

2.4.3.5 Space Overhead

In order to provide completeness, most copying collectors require a minimum factor-of-two space overhead. An exception is the Train collector which we describe in Section 2.5.2.3. This space overhead is usually higher than the overhead of a mark-sweep collector (10–20% on average). The worst-case overhead for a semi-space copying collector is also a factor of two. Other copying collectors like Older-First and Beltway use remembered sets that could grow to be as large the heap, although in practice this case never happens.

2.4.3.6 Properties

Copying collectors compact live data, thus reducing the working set. The cost of collection is proportional to the amount of live data in the heap. Unlike compacting collectors that copy data in allocation order, the traversal order determines the data layout for copying collectors. Like mark-compact collectors, copying collectors use cheap bump-pointer allocation.

Copying collectors tend to do more work than mark-sweep collectors, since they regularly copy data. Since repeated copying of large objects and permanent objects can be expensive, copying (and mark-compact) collectors treat these objects specially. Copying collectors typically place large objects in a separate large object area that is managed using mark-sweep [21, 65]. The area may need to be compacted occasionally to combat fragmentation.

2.5 Incremental Collectors

Incremental garbage collectors focus on improving pause time behavior, at the expense of some loss in overall throughput. Short pause times are important for interactive and real time applications. Reference counting collectors are naturally incremental since they interleave collection work with program activity. In this section, we discuss reference counting, consider issues related to incremental tracing collection, and describe existing incremental tracing collectors.

2.5.1 Reference Counting

Reference counting collectors [27] maintain a count of the number of references to every object in the heap. The collector increments and decrements object reference counts as the program creates and destroys pointers. It reclaims space occupied by objects whose reference count drops to zero.

Reference counting collectors are suitable for interactive applications. They distribute collection overheads throughout program execution and hence tend to have short pause times. In addition, since the basic reference counting algorithm identifies garbage objects as soon as they die, the space can be reused immediately. Unlike tracing collectors, reference counting collectors do not require significant additional space in the heap to avoid thrashing.

However, the basic algorithm has a number of drawbacks. First, it does not reclaim cyclic garbage, since the reference count of objects in a cycle never drops to zero. Second, the collector requires an additional word per object to maintain reference counts, and manipulating the counts can be expensive. The counts need to be manipulated every time a pointer is pushed or popped from a stack, and even list traversal requires incrementing and decrementing counts. Third, the cost of removing the last pointer to an object can be unbounded. Finally, like mark-sweep collectors, reference counting collectors also suffer

from fragmentation. Hence, they need to use some form of memory compaction in order overcome the negative effects of fragmentation.

A number of solutions have been proposed and implemented to overcome these drawbacks. Lazy deletion [66] smooths the cost of deletion of the last pointer to an object. Instead of performing recursive deletion, it pushes objects with a zero reference count onto a stack. When the collector reallocates an object O on the stack, it pushes all objects referenced by O onto the stack. While this technique usually smooths deletion costs, large arrays still pose a problem. In addition, this technique loses the benefit of immediacy of collection.

Deferred reference counting (DRC) [29] takes advantage of the fact that a majority of pointer stores are made to local variables. Stores to non-locals can be as low as 1% of all stores. DRC does not perform reference count manipulations on modifications to local variables. When the reference count of an object drops to zero, the collector adds the object to a zero count table (ZCT). The collector removes an object from the ZCT when the program creates a reference to it. To reclaim garbage, the collector performs a mark from the stack and reclaims unmarked ZCT objects. DRC requires additional space overheads for the ZCT and it also loses the immediacy advantage of reference counting, but it still bounds pauses.

Several reference counting techniques use sticky reference counts or even single bit reference counts [68, 62] to reduce space overheads. They take advantage of the fact the number of references to an object is usually low. These techniques typically use a tracing collector to restore overflow counts.

Cyclic garbage can also be reclaimed using occasional tracing collection. Several cycle detection techniques [16, 25, 41] have also been proposed, but they tend to be quite expensive.

More recently, Blackburn and McKinley [15] describe a generational reference counting collector that uses copying collection in the nursery and reference counting in the old

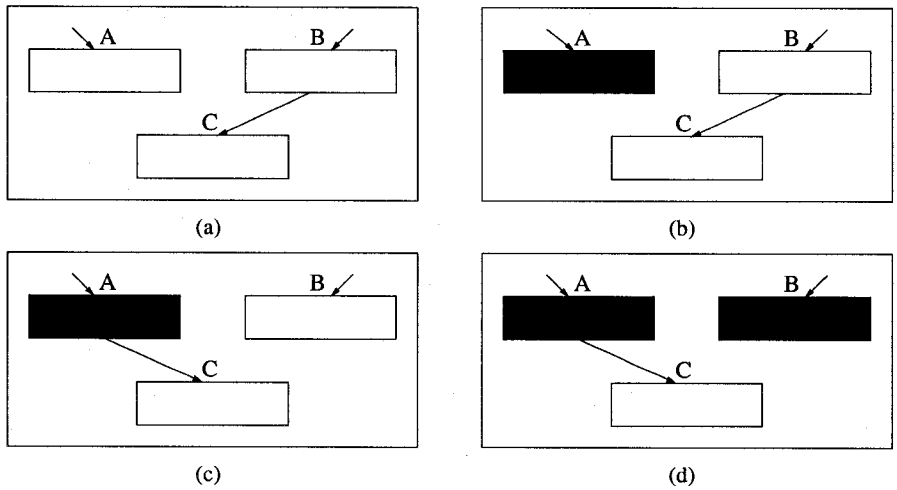


Figure 2.1. Incremental collection error.

generation. They demonstrate that the collector can provide high throughput and short pause times.

2.5.2 Incremental Tracing Collectors

Since incremental tracing collectors perform small portions of tracing in between periods of program activity, a problem that they face is that the mutator might alter the object graph as they perform tracing.

Figure 2.1 shows an example of an error that could arise. Figure 2.1(a) shows a heap with three objects *A*, *B*, and *C* which have not been marked by the collector. Objects *A* and *B* are reachable from roots. The collector marks *A* (Figure 2.1(b)), colors it black since it is completely processed, and then transfers control to the program. The program makes *A* point to *C* and nulls out the reference in *B* (Figure 2.1(c)). When the collector resumes marking, it marks *B* and colors it black. The collector will not mark *C*, since it incorrectly concludes that all reachable objects have been marked, and it will reclaim space occupied by *C*.

This problem arises from the combined effect of two program actions:

- the program creates a reference from a black object to a white object,
- the program destroys the original reference to the white object

Incremental tracing collectors handle this coordination problem between the mutator and collector in two ways – using either a *read barrier* or a *write barrier.*

2.5.2.1 Read Barriers

Read barriers prevent the mutator from seeing white objects. Any time the mutator attempts to read a white object, the read barrier colors the object gray, and adds it to the list of objects to be processed. This prevents the mutator from installing a pointer to a white object into a black object (since the white object will be grayed first).

Baker's real time incremental copying collector [9] uses a read barrier to coordinate with the mutator. Any time the mutator attempts to read a white object, the object is grayed and copied to to-space, thus preventing the mutator from seeing white objects. All objects allocated during a collection cycle are assumed to be live, and are colored black. Thus data allocated during a collection cycle cannot be reclaimed until the next cycle.

The read barrier, when implemented in software, adds to each pointer read a check, and a conditional call to a copy routine. The barrier turns out to be quite expensive on stock hardware, since read operations are very common. A variant of this scheme, proposed by Brooks, adds an indirection field to every object [19]. All objects are referred to via this field. If an object has not been copied, its indirection field refers to itself. If an object has been copied, the field refers to the new copy. This scheme helps remove the conditional check from Baker's read barrier. However, this scheme still has an overhead of tens of percent [64].

The Appel-Ellis-Li collector [4] is based on Baker's copying collector but does not require any compiler modifications. The collector uses a page-level read barrier that is supported by memory protection hardware. The read barrier is more conservative than Baker's:

it allows the mutator to see only black objects. When collection starts, the collector copies all objects reachable from roots and turns on memory protection for the new (gray) pages. If the mutator accesses an object on a gray page, a page access trap is triggered. The collector then removes protection from the page, scans objects on the page, copies reachable from-space objects, and turns on protection for the new pages. It then restarts the mutator. The collector also processes gray pages between handling page faults (interleaved with allocation).

Bacon, Cheng, and Rajan [7] describe an incremental mark-sweep collector that performs defragmentation dynamically, using read barrier based copying. They use special compiler techniques to reduce the read barrier overheads of the collector. Their collector can provide short, predictable pause times and consistent mutator utilization (percentage of CPU time devoted to running the application). However, defragmentation is achieved by copying objects in highly fragmented pages to a new page. Although this technique would improve locality, it could cause unrelated objects to be compacted close together. Thus, the cache locality could potentially be worse than that of a copying collector.

2.5.2.2 Write Barriers

Write barrier based incremental collectors operate by preventing either one of the two conditions listed in Section 2.5.2. Wilson [67] classifies these collectors as either *snapshot-at-the-beginning* or *incremental-update*. A snapshot-at-the-beginning collector prevents the second condition from happening. Any time the mutator overwrites a pointer to a white object, the write barrier records the original reference (by coloring the white object gray), ensuring that the white object will be reached and processed. An incremental-update collector prevents the first condition from happening. When the mutator stores a pointer to a white object into a black object, one of the objects is grayed, thus causing the collector either to rescan the previously black object, or to process the previously white object.

Yuasa's snapshot collector [69] uses a write barrier that stores overwritten values in a marking stack, thus ensuring that all objects will be reached by the garbage collector. However, snapshot collectors tend to be conservative [67]: they do not reclaim any objects that become unreachable during a collection.

Dijkstra's incremental update collector [30] uses a write barrier that traps pointer writes to white objects (regardless of the color of the source object), and colors the white objects gray. The barrier forces the collector to scan the white object and ensures that it sees every reachable object. Since the write barrier does not record overwritten pointer values, objects that become unreachable before the collector reaches them will be reclaimed in the same collection cycle. Steele's collector [57] traps pointer writes from black to white objects and colors the black object gray, forcing the collector to rescan the black object. Its write barrier is more expensive than Dijkstra's, but it is less conservative than Dijkstra's collector.

On some systems, the overhead of a software write barrier can be removed with virtual memory assistance. The Boehm-Demers-Shenker collector [17] performs incremental update marking but depends on OS dirty bits for synchronization. The collector attempts to terminate every time the mark stack becomes empty. To terminate the marking phase, the collector suspends all mutator threads and examines the VM dirty bits. If dirty pages exist, marking recommences from marked objects on the dirty pages, and the dirty bits are cleared. When no dirty pages exist and the stack is empty, collection terminates. The collector does not require compiler modifications to implement a write barrier. A disadvantage of the scheme is that the granularity of dirty information is coarse.

2.5.2.3 Train Collection

The Mature Object Space (*Train, MOS*) collector [34, 54] is a coarse-grained incremental copying collector that can provide short pauses. MOS uses a generational heap layout and divides the old generation into equal size windows called *cars*, which bound the amount of copying performed in each collection. It groups cars into logical units called

trains, and performs copying in a manner that moves large cycles of garbage into trains that it can reclaim entirely. Although the MOS collector can provide bounded pause times (by bounding the size of a car), it tends to copy a large amount of unreachable data before it moves the data to a separate train and reclaims the space it occupies. This copying cost has a significant impact on the overall performance of the collector, as we will show

2.5.2.4 Incremental Incrementally Compacting Collection

Lang and Dupont [39] describe a collector that performs incremental compaction in a heap managed by a mark-sweep collector. Ben-Yitzhak et al. [11] propose a collector that is similar to Lang and Dupont's, but is also parallel. Lang and Dupont divide the heap into equal size segments. At each collection, the collector does marking for the entire heap, and then compacts one segment. Thus the collector is primarily mark-sweep, but the compaction helps to overcome fragmentation. When combined with an incremental mark phase, it can be made a fully incremental collector.

Both of the above schemes require n marking passes in order to compact a heap consisting of n regions. The large number of marking passes can lead to poor performance when the heap is highly fragmented. In addition, Ben-Yitzhak's collector requires additional space during collection to store remembered set entries. The collector does not address the problem of large remembered sets.

CHAPTER 3

METHODOLOGY

We describe here the experimental framework that we use to implement garbage collectors, the benchmarks we use to compare the performance of the collectors, and implementation details of standard garbage collectors we use in the performance evaluation.

3.1 Experimental Framework

We first describe the virtual machine that we use for our experiments and the memory management toolkit that we use to build the garbage collectors.

3.1.1 Jikes RVM

Jikes RVM [1, 2] is an open source Java virtual machine that we use to implement and evaluate all our garbage collectors. It is written almost entirely in Java, making it portable and allowing it to use the same memory space for virtual machine and application objects.

Jikes RVM does not have an interpreter: it compiles all bytecode to native code before execution. Jikes RVM has two compilers, a baseline compiler that essentially macro-expands each bytecode into non-optimized machine code, and an optimizing compiler. It also has an adaptive run-time system that first baseline compiles all methods and later optimizes methods that execute frequently. Methods can be optimized at three different levels depending on the execution frequency. However, the adaptive system does not produce reproducible results, since it uses timers and may optimize different methods in different runs.

We use a *pseudo-adaptive* configuration to run our experiments with reproducible results. We first run each benchmark 7 times with the adaptive run-time system, logging the

names of methods that are optimized and their optimization levels. We then determine the methods that are optimized in a majority of the runs, and the highest level to which each of these methods are optimized in a majority of runs. We run our experiments with only these methods always optimized (to that optimization level) and all other methods always baseline compiled. The resulting system behavior is repeatable, and does very nearly the same total allocation as a typical adaptive system run; it differs from adaptive system behavior in that it tends to invoke the optimizing compiler before the application has built up its live data set, whereas adaptive runs tend to invoke the (memory-hungry) optimizing compiler in the thick of the application. Thus, adaptive system maximum live size tends to be bigger (and unpredictable). However, the pseudo-adaptive system tends to run closer to its peak since its average and peak live sizes are closer to one another. Thus, when scaled by live size, pseudo-adaptive is consistently closer to its peak memory usage more of the time.

Another issue is that in the adaptive system, the base compiler gathers an edge profile by inserting instrumentation. The later optimization phases use the information for code placement [47]. Pseudo-adaptive does not have an edge profile available to it. Thus, we expect some degradation of code quality.

Jikes RVM system classes can be compiled either at run time or at system build time. We compile all the system classes at build time to avoid any non-application compilation at run time. The system classes are stored in a region called the *boot image* that is separate from the program heap.

3.1.2 MMTk

We implemented all garbage collectors using the Memory Management toolkit (MMTk) [12] that comes packaged with Jikes RVM. MMTk provides most of the generic functionality required by a copying collector, making implementation of a new collector relatively easy.

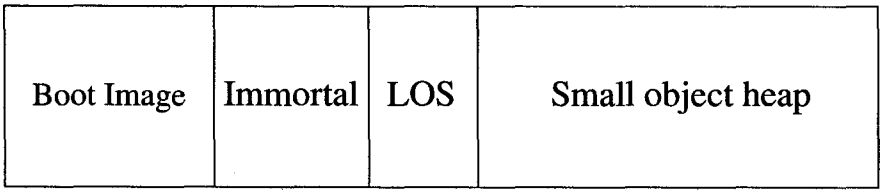


Figure 3.1. Heap layout in an MMTk collector.

MMTk divides the available virtual address space into a number of regions. Figure 3.1 shows the virtual memory layout for any collector implemented in MMTk. The *boot image* is situated at the low end of available virtual address space. It stores all system objects.

Adjacent to the boot image space is the *immortal space*. This space is used to store objects that are treated as always reachable. The type information block (TIB) objects, which include the virtual method dispatch vectors, etc., and which are pointed to from the header of each object of their type, are allocated by MMTk into immortal space. Additionally, we allocate all type objects and reference offset arrays for class objects into immortal space, since some collectors require that these objects not move during collection.

Next to immortal space is Large Object Space (LOS). MMTk uses LOS to manage objects that are larger than 8KB. MMTk rounds up the size of large objects to whole pages (4 KB), and allocates and frees them in page-grained units. The remainder of the address space (small object heap) can be used by collectors to manage regular objects allocated in the heap.

3.2 Benchmarks

We use eight benchmarks to evaluate the collectors, six from the SPECjvm98 suite [55] plus pseudojbb and health. pseudojbb is a modified version of the SPEC JBB2000 benchmark [56]. pseudojbb executes a fixed number of transactions (70000), which allows better comparison of the performance of the different collectors. health is an object oriented Java version of the Olden C benchmark [49, 20]. We run all SPEC benchmarks

Benchmark	Description
_202_jess	a Java expert system shell
_209_db	a small data management program
_213_javac	a Java compiler
_222_mpegaudio	an MPEG audio decoder
_227_mtrt	a dual-threaded ray tracer
_228_jack	a parser generator
pseudojbb	SPECjbb2000 with a fixed number of transactions
health	simulation of a health care system

Table 3.1. Description of the benchmarks used in the experiments.

using the default (size 100) parameters, and ignoring explicit GC requests. We are aware that this in not an ideal set of benchmarks for evaluating memory-constrained collectors. However, these are the best available benchmarks for the Jikes RVM infrastructure, and they represent a wide range of program behaviors. They are regularly used to evaluate the performance of garbage collection techniques. _209_db performs database operations and _222_mpegaudio is an audio decoder. Both are similar to applications that might be used on memory-constrained devices. Chen et al. [22] use a set of embedded benchmarks for their evaluation of garbage collectors in constrained memories. However, the benchmarks run only on Sun's KVM and we could not run them on Jikes RVM due to issues with special classes implemented natively in KVM.

Table 3.1 describes each of the benchmarks we use. Since we use multiple versions of JikesRVM for our experiments, and the live size and total allocation of the benchmarks (including JikesRVM objects allocated on the heap) varies with the version, we list the benchmark live sizes in the experimental results section of each chapter. MMTk uses a resource table that occupies 4MB in immortal space. We move it to the boot image and do not include it in live size measurements, since it skews the live size value considerably for small benchmarks.

3.3 Hardware

We run our experiments on two different hardware configurations. The first is a Macintosh PowerPC with two 533 MHz G4 7410 processors (though the system uses only one of them), 32 KB on-chip L1 data and instruction caches, 1 MB unified L2 cache and 640 MB of memory, running PPC Linux 2.4.10. We used this for early experiments in the thesis. We evaluate the Mark-Copy collector on this platform.

The second is a system with a Pentium P4 1.7 GHz processor, 8KB on-chip L1 cache, 12KB on-chip ETC (instruction cache), 256KB on-chip unified L2 cache, and 512 MB of memory, running RedHat Linux 2.4.20-31.9 (with the perfctr patch applied). We evaluate the Train collector and MC^2 on this platform.

We run all experiments with the machines in single user mode to maximize repeatability.

3.4 Garbage Collectors

We now describe the implementations of standard collectors that we use for performance evaluation of our new collectors.

3.4.1 Mark-Sweep

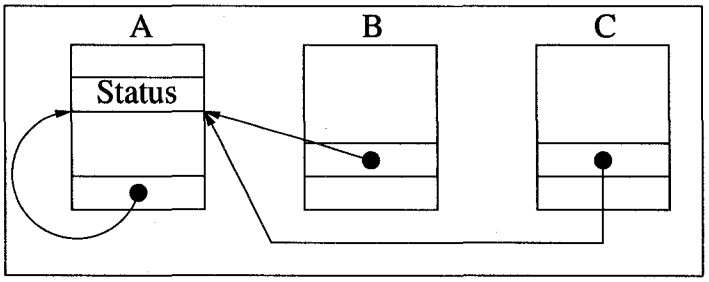
The Mark-Sweep (MS) collector we use for the comparison comes with MMTk. The collector uses segregated free lists [53] to manage the heap memory, with 51 different size classes. It uses a separate size class for every 4 bytes for the range $(8,63)$, every 8 bytes for the range $(64,127)$, every 16 bytes for range $(128,255)$, every 32 bytes for the range $(256,511)$, every 256 bytes for the range $(512,2047)$, and every 1024 bytes for the range $(2048,8192)$. The MMTk MS collector divides the entire heap into a pool of blocks, each of which can be assigned to a free list for any of the size classes. An object is allocated in the free list for the smallest size class that can accommodate it. After garbage collection, if a block becomes empty, MS returns it to the free block pool.

3.4.2 Generational Mark-Sweep

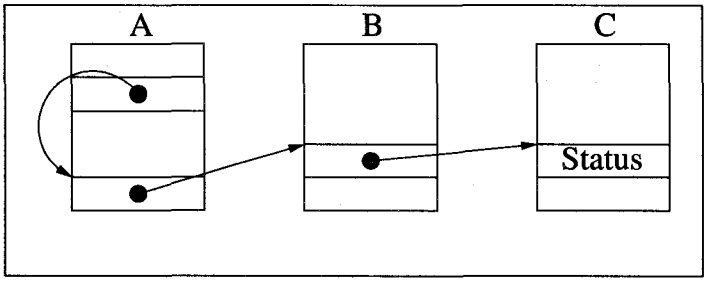
The MMTk generational mark-sweep collector (GMS) divides the heap space into two regions. The region with lower addresses contains the old generation, and the region with higher addresses contains the nursery. The write barrier records, in a remembered set, pointers that point from outside the nursery into the nursery. The write barrier is partially inlined [14]: the infrequently executed code that tests for a store of an interesting pointer is inlined, while the code that inserts interesting pointers into the remembered set is out of line. The nursery uses bump pointer allocation, and the collector copies nursery survivors into an old generation managed by mark-sweep collection.

3.4.2.1 Generational Mark-Compact

We implemented a mark-(sweep)-compact generational collector (MSC), based on the threaded algorithm described by Martin [42]. The collector uses copying collection in the nursery and threaded compaction in the old generation. Threaded compaction does not require any additional space in the heap, except when handling internal pointers. While this is not a problem for Java objects, since Java does not allow internal pointers, Jikes RVM allocates code on the heap that contains internal code pointers. However, the space requirement for these pointers is not very high, since there is only one code pointer per stack frame. MSC also requires a bit map (space overhead of about 3%) in Jikes RVM, because the object layout with scalars and arrays in opposite directions does not allow a sequential heap scan (this has been changed in the most recent release of JikesRVM). MSC divides the heap into nursery, old generation, and bit map regions. It uses the same write barrier as GMS. Its compaction operates in two phases. During the *mark* phase, the collector marks reachable objects. At the same time it identifies pointers that point from higher to lower addresses. These pointers are chained to their target object starting from the status word of the target (Jikes RVM uses a status word in every object that stores lock, hash and GC



(a) Before threading



(b) After threading

Figure 3.2. An example of pointer threading performed by the Mark-Compact collector.

information). For internal pointers, we use extra space to store an additional offset into the target object.

Figure 3.2 shows an example illustrating how MSC performs threading during the first phase. A, B, and C are three objects in the heap. A contains a self-referential pointer (considered a backward pointer), and B and C contain one pointer each to A. The collector creates a linked list starting from the status word for A. The status (which is distinguished by having non-zero least significant bits) is stored in the slot at the end of the linked list.

At the end of the mark phase, the collector has identified all live objects. Also, it has chained to each object all backward pointers to that object, and they can be reached by traversing a linked list starting from the object header. During the second phase, MSC performs the actual compaction. As it moves an object, it updates all pointers referring to the object and copies the original status word into the header of the new copy. It also chains

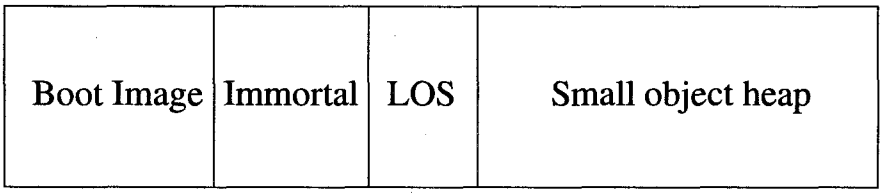


Figure 3.3. Heap layout in an MMTk generational collector. The old write barrier records pointers into the nursery from objects outside the nursery. The new write barrier additionally records mutations to boot image objects.

all (forward) pointers from the object to target objects not yet moved, so that it will update these pointers when it moves the target object later in the phase.

3.4.2.2 Non-Incremental Collector Improvements

We describe here a couple of refinements we make to the MMTk GMS collector and our MSC collector described above. These refinements significantly improve execution and pause times in small and moderate size heaps.

Boot image remembered sets: During a full collection, the MMTk GMS collector scans the entire boot image to find pointers from boot image objects into the heap. The collector needs to do this since it does not record mutations to boot image objects. While the barrier is simple to implement and fast, it is inefficient for small and moderate size heaps, when full collections are frequent. This is because most pointers within the boot image reference boot image objects, and the collector spends a good portion of the collection time following these pointers. Our measurements show that only about 0.4% of all boot image pointers reference objects in the heap. The issue with boot image scanning has also been discussed by Bacon et al. [6].

We modified the MMTk GMS collector to avoid scanning the boot image during full collections. The modified MS collector uses a new write barrier. Figure 3.3 shows the layout of the heap in MMTk and the pointers that are recorded by the old and new write barriers. Figure 3.4 shows the old MMTk GMS write barrier code and Figure 3.5 shows

```
private final void writeBarrier(
    VM_Address srcSlot, VM_Address tgtObj)
  throws VM_PragmaInline {
  // Record pointers from outside the nursery
  // that point to objects in the nursery
  if (srcSlot.LT(NURSERY_START) &&
      tgtObj.GE(NURSERY_START))
    nurseryRemset.outOfLineInsert(srcSlot);
  VM_Magic.setMemoryAddress(srcSlot, tgtObj);
}
```

Figure 3.4. Original MMTk generational write barrier.

the new write barrier code (for a uniprocessor environment). The old write barrier records pointers into the nursery from objects that lie outside the nursery. The new barrier records all boot image objects that contain pointers into the heap, in addition to recording pointers into the nursery. During a full collection, the mutated boot objects are scanned to find heap pointers, and the rest of the boot image is not touched. The barrier does not record immortal pointers since many of these are application specific and the space required to store them cannot be estimated in advance.

The modified MS collector improves the full collection pause time by up to a factor of 4 for small benchmarks since boot image scanning dominates collection time for these benchmarks. It usually has lower execution times in small and moderate size heaps. In large heaps, when fewer collections occur, the execution time is about the same since the reduction in collection time is not significantly larger than the increase in mutator time (due to the more expensive write barrier). We also used this technique in a modified version of the generational mark-compact collector, and found improvements in execution and pause times.

Code Region: MMTk collectors place compiled code and data in the same space. We found that this can cause significant degradation in the performance of MSC due to poor code locality. We modified the collector to allocate code and data in different spaces. We achieved this by maintaining a separate code space to store code objects which we compact

in place. We do not use this technique for GMS, since the use of a separate GMS space for code would not help significantly (objects of different size are not placed together), and would probably increase fragmentation. While the write barrier technique improves GC time (at a small expense in mutator time), the separate code region improves mutator time for copying collectors due to improved code locality.

Figure 3.6 shows MSC GC times for two benchmarks. The three curves represent MSC with a common space for data and code, with a separate code space (MSC-CODE), and MSC with a separate code space and the new write barrier (MSC-CODE-NEW-WB). Figure 3.7 shows execution times for the three MSC variants. The GC times for MSC and MSC-CODE are almost identical. Our performance counter measurements show that MSC-CODE performs better due to improved code locality (fewer ITLB misses). The graphs also show that MSC-CODE-NEW-WB performs better than MSC-CODE in small and moderate heaps because of lower GC times. In large heaps, the two collectors have equivalent performance.

```
private final void writeBarrier(VM_Address srcObj,
        VM_Address srcSlot, VM_Address tgtObj)
  throws VM_PragmaInline {
  if (srcObj.LT(NURSERY_START) &&
      tgtObj.GE(LOS_START))
    slowPathWriteBarrier(srcObj, srcSlot, tgtObj);
  VM_Magic.setMemoryAddress(src, tgtObj);
}

private final void slowPathWriteBarrier(
        VM_Address srcObj, VM_Address srcSlot,
        VM_Address tgtObj)
  throws VM_PragmaNoInline {
  // If source object is in the boot image
  if (srcObj.LT(IMMORTAL_START)) {
    // Check if object has already been recorded.
    // If not, insert object address into boot
    // remset and mark the object
    VM_Word status =
       VM_Interface.readAvailableBitsWord(srcObj);
    if (status.and(Hdr.MUTATE_BIT).isZero()) {
      VM_Interface.writeAvailableBitsWord(srcObj,
                   status.or(Hdr.MUTATE_BIT));
      bootRemset.inlinedInsert(srcObj);
    }
  }
  // Record slots outside nursery that point to
  //  nursery objects
  if (tgt.GE(NURSERY_START))
    nurseryRemset.inlinedInsert(src);
}
```

Figure 3.5. New generational write barrier.

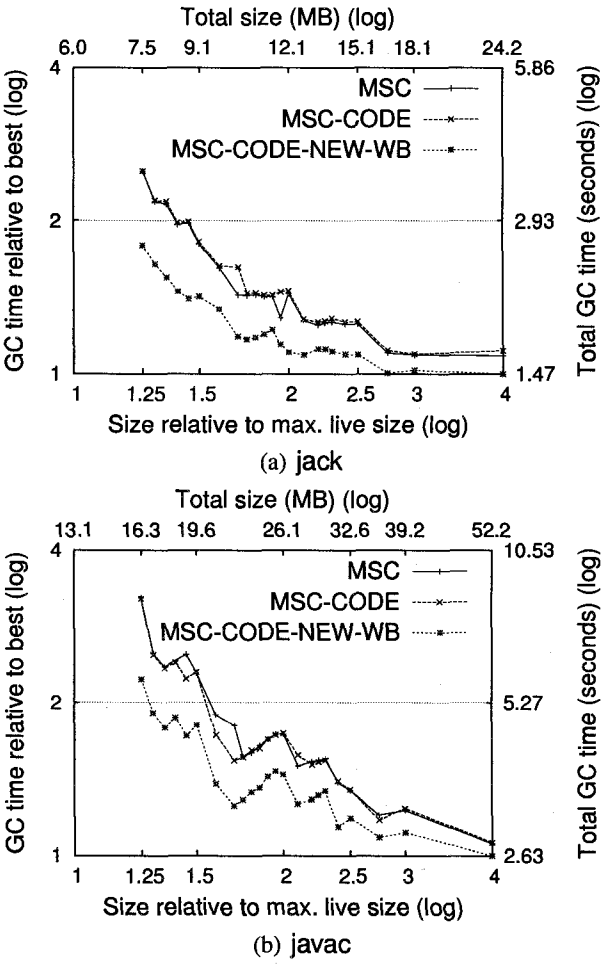


Figure 3.6. GC time relative to best GC time for MSC, MSC with a separate code region (MSC-CODE), and MSC with a separate code region and a new write barrier (MSC-CODE-NEW-WB). The new write barrier lowers execution time significantly.

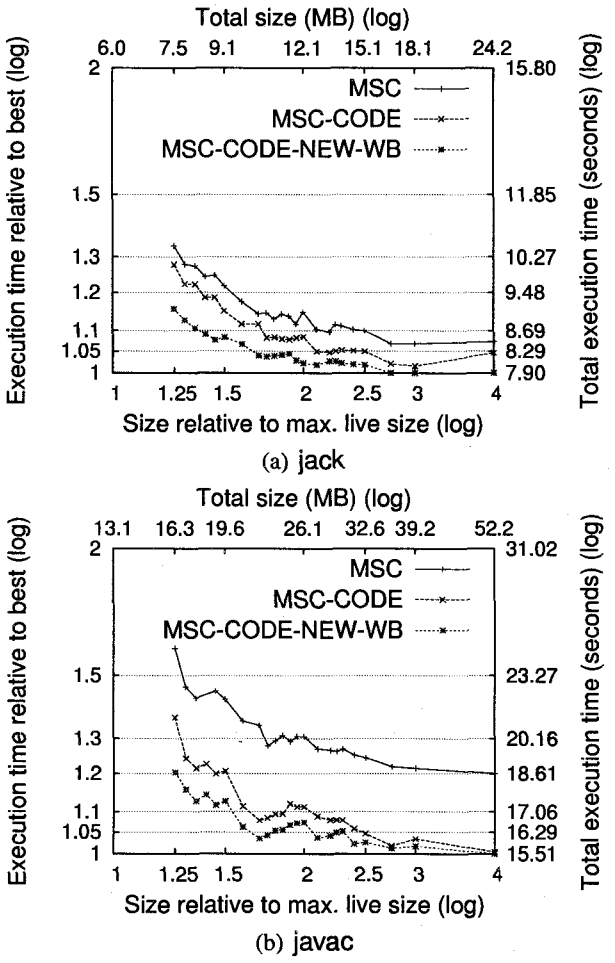


Figure 3.7. Execution time relative to best execution time for MSC, MSC with a separate code region (MSC-CODE), and MSC with a separate code region and a new write barrier (MSC-CODE-NEW-WB). The separate code region lowers execution time significantly by reducing mutator time. The lower GC times with the new write barrier reduce execution time significantly.

CHAPTER 4

THE TRAIN COLLECTOR

A generational copying collector can provide low average pause times, since a majority of its collections target objects in a bounded size nursery. However, the collector needs to collect objects in the oldest generation periodically. The old generation collection requires scanning and copying every live object in the heap, causing occasional long pauses.

The Train collector (mature object space collector, MOS) was designed to overcome this problem. It collects objects in the oldest generation non-disruptively, providing short maximum pause times. An additional important property of MOS is that it does not require the factor-of-two space overhead of a regular copying collector. Since MOS has good space utilization properties and can provide short pauses, we first study its performance to see if it can be used for memory constrained systems.

We describe here details of the MOS algorithm [34] and our implementation of the collector in Jikes RVM, and then present a detailed performance evaluation of the collector. We show that while the collector can provide short pause times and can run with low space overheads, it cannot provide high throughput in constrained memory.

4.1 MOS Algorithm

We describe here the heap layout used by the train collector, details of the metadata used by the collector, the collection procedure, and the collector's write barrier. We also describe an error condition encountered by the original algorithm that Seligmann and Grarup [54] first observed and solved.

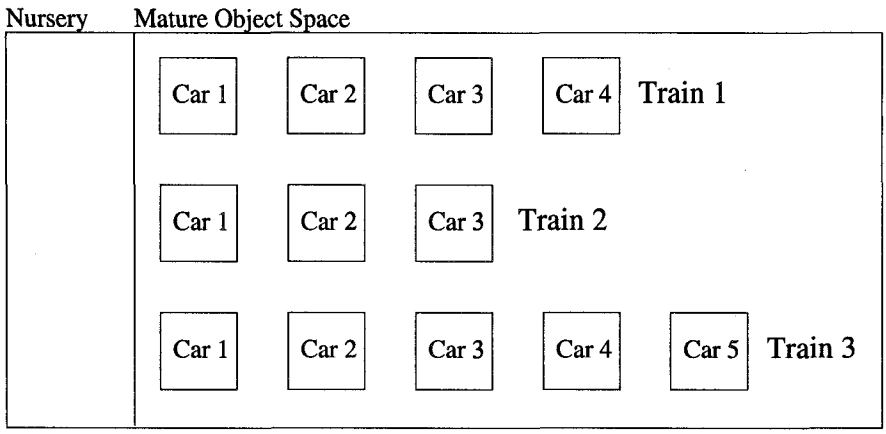


Figure 4.1. An example of the heap layout for a two generational MOS collector.

4.1.1 Heap Layout

Unlike a generational copying collector that divides all generations into semi-spaces, MOS divides the oldest generation into a number of equal size regions called *cars*. A car is the smallest region within the oldest generation that the collector can copy independently. By making the size of a car appropriately small, the collector can bound the amount of copying it performs in any single invocation, thus generally achieving low pause times. In addition to this subdivision of the heap, MOS groups one or more cars in logical units called *trains*. The purpose of this logical division is to allow collection of garbage that spans multiple cars, as we will explain below.

Figure 4.1 shows the heap layout for a two generational MOS collector with three trains and twelve cars in the old generation. Every car belongs to a single train. Trains do not necessarily contain the same number of cars, and the number of cars in a train depends both on collector policy and application data characteristics. Whenever MOS allocates an object in a train, it places the object at the end of the last car in the train. If the object is larger than the available space in the car, MOS adds a new car to the train and allocates the object in the new car.

MOS maintains a global ordering of all cars and trains in the old generation. It collects trains in the order in which it creates them (lower-numbered to higher-numbered trains). Within a train, MOS again collects cars in the order in which it creates them. In the example in Figure 4.1, MOS collects cars in Train 1 followed by cars in Trains 2 and then Train 3.

4.1.2 Remembered Sets

For each car, MOS maintains a *remembered set*, that is, a list of references into the car. This list allows the collector to update pointers efficiently when it copies objects out of a car. This is similar to the remembered set used by a regular generational collector to scavenge nursery objects efficiently. MOS maintains *unidirectional* remembered sets. These sets record pointers only in a single direction, from objects in higher numbered cars or trains to objects in lower numbered cars or trains. Since the order in which the collector scavenges cars is based on the global car ordering, this unidirectional technique ensures that it records all pointers into a car before it collects the car.

Unidirectional remembered sets have two advantages. First, the size of unidirectional remembered sets is usually smaller than *bidirectional* remembered sets (sets that record every pointer into a car irrespective of pointer direction). Second, when MOS scavenges a car it needs to process the remembered set for only that car. It does not have to update remembered sets of other cars since those remembered sets cannot contain references from the scavenged car.

In addition to per-car remembered sets, the collector maintains for each train a count of the number of references into the train from objects in higher-numbered trains. The collector uses this information to detect whether a train contains only garbage.

4.1.3 Collection Procedure

MOS triggers mature space collection at some point during program execution based on implementation policy, such as when a certain fraction of the heap is full. At this point, it designates the lowest numbered train as the collection train and *closes* the train. The

collector does not perform any new allocation into a closed train. If only one train exists in the heap when collection begins, MOS creates a new train for future allocation.

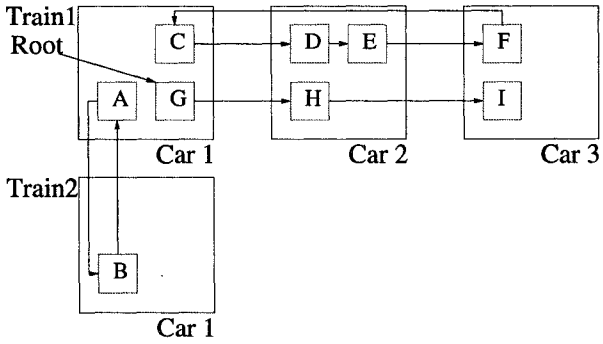
MOS typically scavenges one car when it invokes old generation collection. This is the lowest-numbered car in the lowest-numbered train. In addition, it collects objects in all lower generations since, being a generational collector, it does not maintain a list of references from younger to older generations.

At the start of collection, MOS checks if the number of external references into the train being collected is zero. If this is true, it reclaims all space used by the train since it contains only garbage. If the train has at least one external reference, MOS finds and copies reachable objects in the car selected for collection.

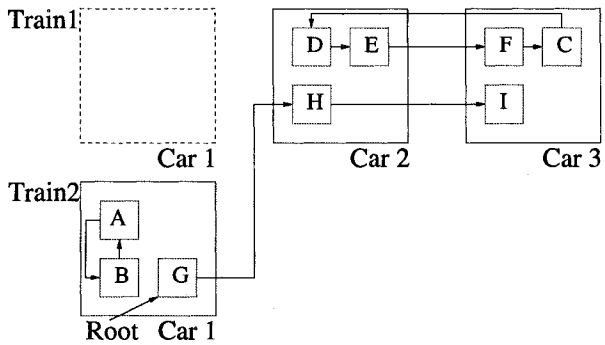
The collector determines where to copy an object being copied using the following rules. It copies objects that are reachable from other trains to those trains. Ideally, if an object is reachable from multiple trains, the collector would copy it to the highest numbered train, but this not a requirement. However, at the very least, when an object is reachable both from outside and inside the collection train, it must copy the object out of the collection train to ensure progress. MOS copies objects reachable from outside mature object space to any train other than the collection train. After the collector has copied all externally reachable objects it copies objects reachable from higher numbered cars in the collection train into the last car in the collection train. All remaining objects in the collection car are garbage and the collector reclaims the space occupied by the car and terminates the collection cycle.

4.1.4 MOS Example

Figure 4.2–Figure 4.4 show an example of how the MOS collector functions. In the example, the mature space contains two trains and MOS is starting to collect Train 1. Train 1 initially contains three cars, and Train 2 contains one car. The example assumes that all objects are of the same size and each car can hold three objects.



(a) Before collection 1

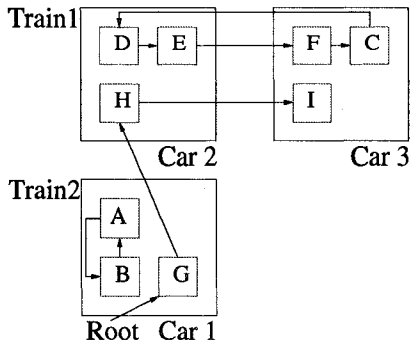


(b) After collection 1

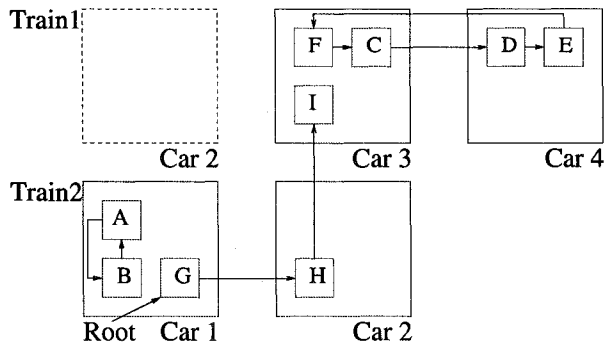
Figure 4.2. MOS collector example.

MOS first selects Car 1 in Train 1 for collection (Figure 4.2(a)). Object A is reachable from Train 2 so MOS copies it to Train 2. Object G is reachable from outside mature space and MOS copies it to Train 2 (a train other than the collection train). Object C is reachable only from within Train 1. MOS copies it to the last car in Train 1 and reclaims space occupied by Car 1 (Figure 4.2(b)).

In the next collection (Figure 4.3(a)), MOS copies object H into Train 2 since it is reachable from object G in train 2. It adds a car (Car 2) to Train 2 to accommodate object H. It then copies objects D and E to the end of Train 1, adding a new car (Car 4) since the last car (Car 3) is full. MOS then reclaims the space occupied by Car 2 (Figure 4.3(b)).



(a) Before collection 2



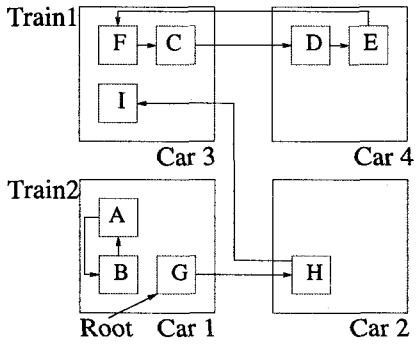
(b) After collection 2

Figure 4.3. MOS collector example (continued).

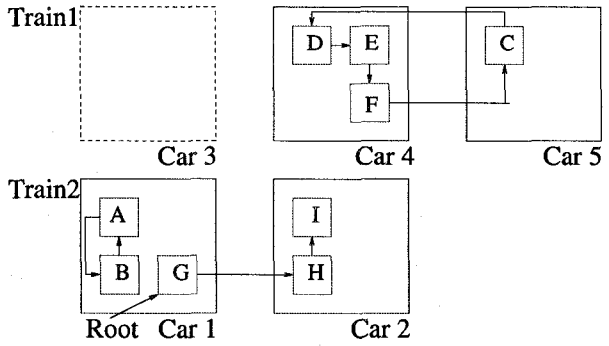
In the third round of collection for Train 1 (Figure 4.4(a)), MOS first copies object I to Train 2, since it is reachable from object H in Train 2. Objects F and C are reachable only from within Train 1. MOS copies them to the end of Train 1 and reclaims space occupied by Car 2 (Figure 4.4(b)).

Now all objects in Train 1 are unreachable from outside Train 1. At the start of the next collection, MOS detects that the external reference count for Train 1 is zero and it reclaims Train 1 entirely.

The example shows that, although the collector scavenges only a single car at every collection it is able to detect and reclaim space occupied by garbage occupying multiple cars, including cyclic garbage. It achieves this by copying objects to trains from which they



(a) Before collection 3



(b) After collection 3

Figure 4.4. MOS collector example (continued).

are referenced, thus ensuring that all objects in a garbage chain or cycle eventually end up in the same train. When the objects all reside in the same train, and MOS collects that train, it reclaims space occupied by the garbage cycle when the external reference count of the train drops to zero (i.e., when it has copied all reachable objects to other trains).

4.1.5 Collection Progress

We noted earlier that when an object is referenced from multiple objects, and one of the references is from outside the collection train, it is essential to copy the object out of the collection train to ensure progress. Figure 4.5 shows an example demonstrating the collector not making progress when it fails to follow this rule. In the example, each car

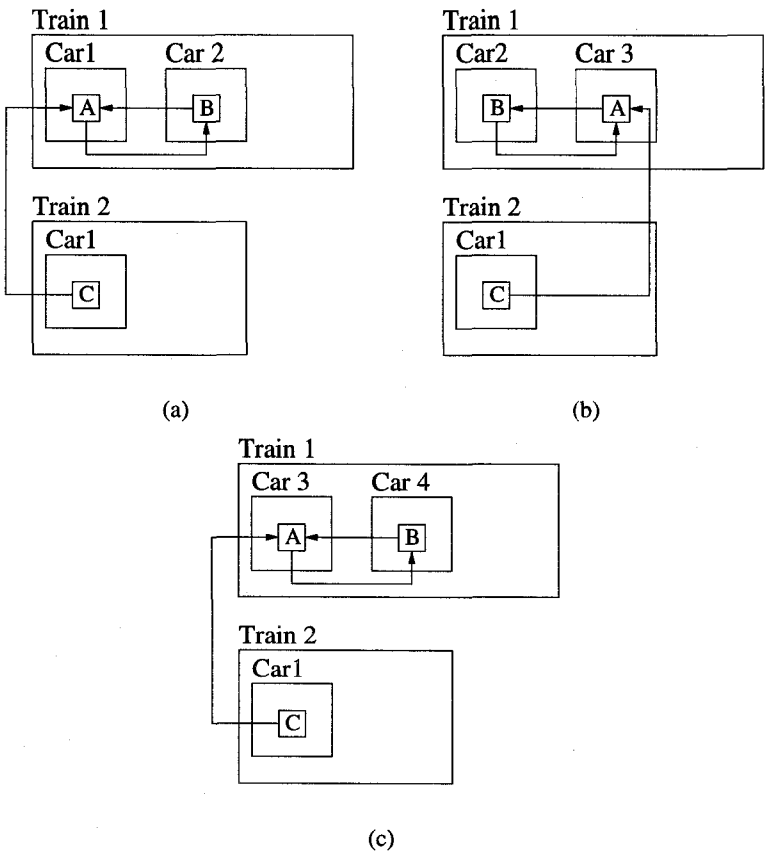


Figure 4.5. MOS example showing that order of pointer processing affects collector progress.

holds a single object. Object A in the figure is referenced by objects B and C. When MOS collects Car 1 in Train 1 (Figure 4.5(a)), if it finds the reference from B first, it copies A to the end of Train 1 (Figure 4.5(b)). In the next collection, when MOS collects B, since B is referenced only from A, MOS copies it to the end of Train 1 (Figure 4.5(c)). Clearly, the collector can continue to repeat this process forever without reclaiming any space. However, if it processes references from outside the collection train first, it will evacuate A and B to Train 2, and reclaim A, B, and C when it starts collecting Train 2.

4.1.6 Write Barrier

The MOS collector uses a write barrier to maintain the per-car remembered sets. Since MOS is a generational collector, the write barrier also needs to record pointer stores from old generations into younger generations. As we discussed earlier, uni-directional remembered sets are generally preferable since they usually occupy less space than bi-directional remembered sets. Uni-directional remembered sets are also easier to manage.

Figure 4.6 shows pseudo-code for the write barrier that MOS uses to manage uni-directional remembered sets. The first test in the barrier checks if the source resides in a higher generation than the target object. If so, it records the source in the remembered set for the generation containing the target object. Otherwise, if the source and target reside in mature object space, the barrier records the pointer if the source is in a higher numbered train, or higher numbered car of the same train, than the target.

```
WriteBarrier(Address source, Address target) {
  if ((generation(source) > generation(target)))
    Record source in remembered set for generation(target)
  else {
    if ((generation(source) == OLDEST_GENERATION) and
        ((train(source) > train(target) or
           ((train(source) == train(target)) and
            (car(source) > car(target))))))
        Record source in remembered set for car(target)
  }
}
```

Figure 4.6. Pseudo-code for the MOS write barrier.

4.1.7 Error Condition

Seligmann and Grarup found an error in the original MOS algorithm. Under certain circumstances, the algorithm may not succeed in collecting any garbage at all. Figure 4.7 shows an example of such an error. The collection train in the figure contains one external reference to object B. When MOS collects the train (Figure 4.7(a)), it copies A to the last

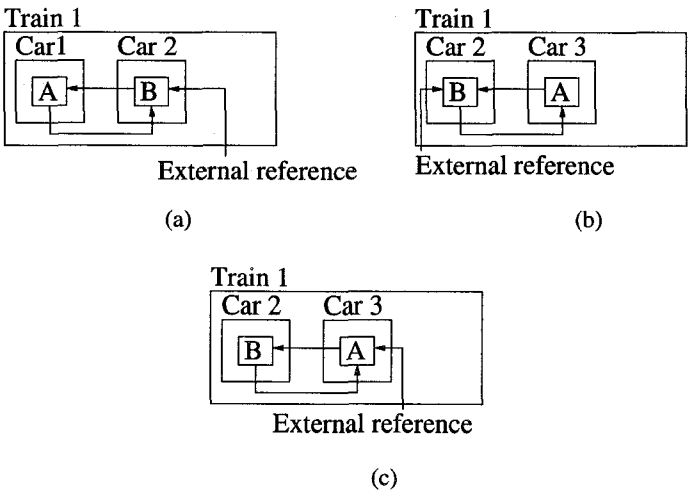


Figure 4.7. MOS error.

car in the train (Figure 4.7(b)), since it is reachable only from B. After the collection, the program switches the external pointer from B to A (Figure 4.7(c)). This will cause MOS to copy B to the last car in the collection train. This pattern can repeat for ever, causing the collector to copy objects repeatedly within the collection train and not make any progress.

Seligmann and Grarup suggest a solution to the problem that ensures that at least one object is evacuated or reclaimed over each pass of a train. At the end of a collection, if the collector does not make progress (number of objects in the collection train does not change), it records some externally referenced object in the collection train. If one has already been recorded it is not overwritten. If a collection makes progress the collector discards any recorded reference. At the start of a collection, the collector checks if it has recorded an object previously. If it has and the object resides in a car it is collecting, it evacuates the object out of the collection train, thus ensuring progress.

4.1.8 Non-Generational Collection

The MOS collector was designed to be a generational collector. Its main goal was to overcome the long pauses of generational copying collection. While it is possible to

make MOS non-generational, and to use it as a full-heap collector, MOS would likely be inefficient as a non-generational collector. The nursery in a generational collector filters out short-lived objects, thus greatly reducing the rate of allocation into mature object space. Without a nursery, the allocation rate into space managed by MOS is likely to be 10 times or more higher, thus requiring MOS to initiate collections much sooner in order to reclaim space before the heap becomes full. The collector would be constantly churning through mature space, copying more data than it normally would. The nursery allows the collector to defer collection until the heap is 70% or more full, thus allowing enough time for even mature objects to die, and giving the collector enough time to churn through mature data before the old generation fills.

4.2 Implementation Details

We describe here the details of our implementation of the MOS collector in MMTk. The collector is two generational. It divides the available virtual memory space into a nursery space and a mature object space. We first describe the nursery layout of the collector. We then explain how the collector implements mature object space using frames and cars. Next, we describe the logical addressing scheme we use. After that, we describe how the collector manages large objects. Finally, we describe our remembered set data structures and the write barrier we use to maintain the remembered sets.

4.2.1 Nursery Space

The collector uses a *bounded-size* nursery with an upper size limit of 1MB. Unlike a fixed-size nursery, the bounded-size nursery grows and shrinks depending on the amount of free space available. The minimum nursery size is 256KB. Like other MMTk collectors, the MOS collector uses a sequential store buffer (SSB) [35] as the remembered set to record pointers into the nursery from objects in the old generation and immortal space. MMTk

allocates space to the nursery remembered set in page size units. After a nursery collection, MMTk reclaims all pages used by the nursery remembered set.

4.2.2 Frames and Cars

The collector divides the virtual memory space available to the old generation into *frames*, each of size 8MB. A frame is the largest contiguous chunk of memory into which the collector performs allocation.

In our current implementation, for each frame we map an amount of memory space that is at most a hundredth of the total heap space. Any remaining virtual memory space in a frame is left unmapped and is not used. Hence the smallest amount in the old generation that the collector can copy in one increment is 1% of the total space. Since frames are power-of-two aligned, only a single shift is required to find the frame number of a heap object during garbage collection.

Cars consist of one or more frames. However, a frame can contain objects of only one car. Normally, when the collector performs mature space collection, it collects a group of frames whose occupancy equals the specified car size. The collector also places a limit on the number of pointers it will update in a single collection. Occasionally, when the pointer count for one or more frames is high, the collector scavenges fewer frames than usual. The size of a car thus varies dynamically, based both on a maximum data that the collector collects in an increment and the threshold on the total number of pointers it may update.

The collector maintains a linked list of free frames. It also maintains a linked list of frames for each train. Every time a train requires a new frame, the collector removes a frame from the free list and adds it to the end of the list for the train. After the collector scavenges frames in the collection train, it returns all processed frames to the free list.

4.2.3 Logical Addresses

Each frame has an associated logical address. The logical address consists of a train number and a frame number. Logical addresses are two bytes long. The high seven bits

store the train number and the other nine bits store the frame number. The collector maintains an array that stores the logical addresses of all frames. To find the logical address for an object, the collector performs a shift to obtain the frame containing the object and then does a lookup in the logical address array. To find the train for an object, the collector performs an additional shift on the logical address.

4.2.4 Large Objects

We use the standard MMTk large object allocator to manage large objects. MMTk allocates objects larger than 8KB into a large object region. It rounds up the size of large objects to whole pages (4 KB) and allocates and frees them in page-grained units. MOS maintains a separate doubly-linked list of large objects for each train. When the collector allocates a large object in a train, it associates the object with the last frame in the train. If the space currently used in the frame plus the size of the large object is greater than the total space allowed in a frame, the collector adds a new car to the train.

In addition to the standard MMTk large object header, our collector adds three fields. A frame number field stores the number of the frame with which the large object is associated. A logical address field stores the logical address of the associated frame. A third card number field is described below in Section 4.2.5.

The collector does not copy large objects. To move a large object from one frame to another, the collector updates the logical address and frame fields in the header. If the collector moves the object from one train to another, it removes the object from the linked list for the original train, and places it in the linked list for the new train.

4.2.5 Remembered Sets

For each frame, our collector maintains an SSB called the *immortal remembered set* that stores pointers into the frame from objects in the boot image and immortal space. For pointers between mature space cars we implement and evaluate two schemes, one that uses

sequential store buffers and another that uses a card table. For large object pointers we use only a card table.

4.2.5.1 Sequential Store Buffer

For the SSB implementation, the collector assigns two separate SSB instances to each frame. The smallest unit of allocation in each SSB is a page. The first SSB for each frame, called the *external remembered set*, records all pointers into the frame from objects that reside in a higher numbered train. The second SSB for each frame, called the *internal remembered set*, records all pointers into the frame from objects that reside in the same train.

When the collector scavenges a set of frames, it first processes the immortal remembered sets. It copies all objects referenced from immortal slots into the youngest train and then repeatedly copies over objects reachable from the just copied objects. Next it processes the external remembered set and copies objects to the trains from which they are referenced. Finally, it processes the internal remembered sets and copies referenced object to the end of the collection train.

If an object is referenced from multiple trains, it is copied to the train corresponding to the first slot that references it. This could cause an object to be copied multiple times until it eventually reaches the highest numbered train that references it, thus increasing the copying cost. Some alternatives are either to sort the external remembered set before processing it or to perform multiple passes over the remembered set, one for each train. We deem both alternatives to be too expensive.

4.2.5.2 Card Table

We also implemented remembered sets using the scheme described by Azagury et al. [5]. We divide each frame into cards of size 128 bytes and allocate a per-frame card table. Each card table entry is two bytes long, and it stores the logical address of the first (lowest numbered) car that contains an object referenced from the card. The collector also

maintains a frame level summary table that stores the smallest logical address stored in the frame's card table. The total space occupied by all the card tables equals 1.5% of the total heap space ($2\ bytes \times (1/128)$).

We also need to maintain a per-frame bitmap to keep track of the locations of objects. Every time the collector copies an object into an old generation frame, it sets a bit in the frame's bitmap indicating the start address of the object. Since objects are word aligned, the bitmap requires 1 bit for every 32 bits of heap space, adding up to approximately 3% of the total heap space.

When the collector performs mature space collection, it scans the summary table starting from frames in the youngest (highest numbered) train moving down eventually to frames in the collection train. For each source frame SF that contains a reference into the target frames being collected (TF), the collector scans the card table corresponding to SF to find cards within SF that contain references into TF. It then scans every object in each of these cards to find the exact slots that point into TF. Every time the collector processes a card, it updates the corresponding card table entry with a new value that is the logical address of the first frame into which it contains a pointer.

While maintaining and processing card tables tends to be somewhat slower than SSBs, they provide a couple of advantages. First, the amount of space required by a card table is bounded. SSBs can occasionally grow to be large, as large as the heap in the worst case. If the collector does not eliminate duplicates, the set could be even larger. A second advantage is that since we can control the order in which we process card tables, we can process references from higher numbered trains before we process references from lower numbered trains. So, if an object is referenced from multiple trains it will always be copied to the highest numbered train that references it, Thus preventing the object from being copied multiple times.

4.2.5.3 Large Object Cards

MMTk assigns a separate virtual memory region to store large objects. It assigns space in this region in page-grained units and maps and unmaps pages as objects are allocated and freed. The virtual memory space could thus be fragmented and so we do not use a single card table to manage large object space (LOS). Instead we maintain a per-object card table.

When the collector allocates a large object, it allocates additional space at the end of the object to serve as the card table for the object. We divide each large object into cards of size 128 bytes and the card table functions exactly like the table we described in the previous section. Each large object also contains a field in its header that stores the card table summary information.

4.2.6 Popular Objects

For some programs, large remembered sets arise because of a few very highly referenced objects. For example, in javac, occasionally a large remembered set occurs when a small number of objects (representing basic types and identifiers of Java) appear in the same window. Our collector regularly checks the size of all remembered sets. If it finds a large remembered set, it counts the number of pointers to each of the objects in the corresponding frame. It uses an integer array to maintain a count of the number of references to each object in the frame. Since the collector always reserves at least one car worth of free space, there is always enough space for the integer array without increasing the space budget. As the collector scans the SSB, it calculates the offset of each referenced object, and increments the corresponding entry in the byte array. When the count for any object exceeds 1000, the collector marks it as popular.

During the copy phase, the collector treats frames containing popular objects specially. Any frame that has popular objects is collected separately. Processing a single frame helps reduce the amount of data copied and focuses collection effort on updating the large number of references into the frame, hence lowering the pause time. The collector copies popular

objects into a separate train. It treats objects in this train as immortal and does not maintain a remembered set for frames in the train.

While our current implementation does not garbage collect popular objects, they can be added back to the regular heap. In order to do this, the popular objects have to be scanned to find pointers between frames in the "popular train". Once the collector constructs remembered sets for these frames, it must make this train the youngest train in the heap. As the collector copies objects into trains that are younger than the popular train, it will construct remembered sets containing pointer into the popular train. Eventually, when the popular train becomes the oldest train in the heap, the collector scavenges objects in the train in the same manner as other objects in the heap.

4.2.7 Write Barrier

The write barrier we use keeps track of two events. First, it records any pointers from outside the nursery to objects within the nursery. It keeps track of the exact slots that point into the nursery in an SSB. Second, it keeps track of all objects outside the nursery that have been mutated to point to an object in the old generation. Rather than record every single slot in the remembered set for the target object's frame, the write barrier records mutated objects exactly once, and processes the pointers in the objects at GC time. It thus ignores multiple writes to the same slot and uses only the last value in a slot that is found during object processing.

Figure 4.8 shows the mutator write barrier code. The inlined portion contains two boundary checks with constant address values. It filters out pointer stores to objects in the nursery and stores whose target lies in immortal space. All other stores are processed in an out-of-line write barrier subroutine. Figure 4.9 and Figure 4.10 show the write barriers used at GC time by the SSB and card implementations. For each slot in a mutated object, the write barriers check if the slot is in a higher numbered car than the target of the slot. If so, they record the entry in the remembered set.

```
private final void writeBarrier(VM_Address srcObj,
                VM_Address src, VM_Address tgtObj)
  throws VM_PragmaInline {
  //interesting if source lies outside the nursery
  //and the target is either in the nursery or the
  //old generation
  if (src.LT(NURSERY_START) && tgtObj.GE(LOS_START)) {
    slowPathWriteBarrier(srcObj, src, tgtObj);
    VM_Magic.setMemoryAddress(src, tgtObj);
  }

private final void slowPathWriteBarrier(
  VM_Address srcObj, VM_Address src, VM_Address tgtObj)
  throws VM_PragmaNoInline {
  //if target is in the old generation, record the
  // source if it hasn't already been mutated
  if (tgtObj.LT(NURSERY_START)) {
    VM_Word status =
      VM_Interface.readAvailableBitsWord(srcObj);
    if (status.and(Header.MUTATE_BIT_MASK).isZero()) {
      VM_Interface.writeAvailableBitsWord(srcObj,
        status.or(Header.MUTATE_BIT_MASK));
      matureWriteBuf.insert(srcObj);
    }
  } else // record slot if target is nursery object
    nurseryWriteBuf.insert(src);
}
```

Figure 4.8. MOS mutator write barrier code.

```
private static final void matureTgtWriteBarrier(
  VM_Address tgtObj, VM_Address location)
    throws VM_PragmaInline {

    VM_Address tgtAddr = VM_Interface.refToAddress(tgtObj);
    int tgtFrame = getFrame(tgtAddr);
    int srcFrame = getFrame(location);
    if (srcFrame != tgtFrame) {
      VM_Address tgtLogical = getLogicalFromFrame(tgtFrame);
      VM_Address srcLogical = getLogicalFromFrame(srcFrame);
      if (srcLogical.GT(tgtLogical)) {
        int tgtTrain = tgtLogical.toInt()>>>TRAIN_SHIFT;
        int srcTrain = srcLogical.toInt()>>>TRAIN_SHIFT;
        if (srcTrain > tgtTrain)
          extRemsetInsert(tgtFrame-FIRST_MATURE_FRAME,
                          location, tgtTrain);
        else
          intRemsetInsert(tgtFrame-FIRST_MATURE_FRAME,
                          location);
      }
    }
  }
```

Figure 4.9. MOS-SSB collector write barrier code.

```
private static final void matureTgtWriteBarrier(
  VM_Address tgtObj, VM_Address srcObj)
  throws VM_PragmaInline {

  VM_Address tgtAddr = VM_Interface.refToAddress(tgtObj);
  int tgtFrame = getFrame(tgtAddr);
  VM_Address srcAddr = VM_Interface.refToAddress(srcObj);
  int srcFrame = getFrame(srcAddr);
  // If source and target are not in the same frame
  if (srcFrame != tgtFrame) {
    VM_Address tgtLogical = getLogicalFromFrame(tgtFrame);
    VM_Address srcLogical = getLogicalFromFrame(srcFrame);
    if (srcLogical.GT(tgtLogical))
      matureCardMark(srcAddr, srcFrame-FIRST_MATURE_FRAME,
                     tgtLogical,
                     tgtFrame-FIRST_MATURE_FRAME);
  }
}

private static final void matureCardMark(VM_Address srcAddr,
                                         int srcFrame,
                                         VM_Address tgtLogical,
                                         int tgtFrame)
  throws VM_PragmaInline {
  incPointerCount(tgtFrame);
  // Find offset for source slot within frame
  int offset =
    srcAddr.diff(matureFrameToAddress(srcFrame)).toInt();
  // Calculate offset in card table
  int cardOffset =
    offset>>>LOG_MATURE_CARD_SIZE<<LOG_BYTES_IN_SHORT;
  VM_Address tgtAddress =
    cardTable.get(srcFrame).add(cardOffset);
  VM_Address curTgtLogical = getCardLogical(tgtAddress);
  // If target logical is lower than current card min.
  if (tgtLogical.LT(curTgtLogical)) {
    setCardLogical(tgtAddress, tgtLogical);
    // Update summary information for source frame
    VM_Address frameMinLogical = getMinLogical(srcFrame);
    if (tgtLogical.LT(frameMinLogical))
      setMinLogical(srcFrame, tgtLogical);
  }
}
```

Figure 4.10. MOS-CARD collector write barrier code.

Benchmark	Live Size (KB)	Total Allocation (MB)
_202_jess	5872	319
_209_db	12800	93
_213_javac	13364	280
_227_mtrt	12788	163
_228_jack	6184	279
pseudojbb	30488	384

Table 4.1. Benchmarks used for the evaluation of MOS.

4.3 Experimental Results

We present here results comparing the performance of our implementation of the bounded-nursery MOS (BG-MOS) collector with two non-incremental collectors, a generational copying collector with a bounded nursery (BG-COPY) and a bounded-nursery generational mark-compact (BG-MSC) collector. We compare space overheads, copying costs, GC times, total execution times, and pause times for the collectors. All the collectors use a bounded nursery of size 1MB. Table 4.1 shows the benchmarks we use, their live sizes, and the total allocation they perform. We run all experiments on JikesRVM 2.3.2.

4.3.1 Space Accounting

We account for the space used by the three collectors as follows. BG-COPY and BG-MSC use the specified heap size to allocate all heap objects and metadata. BG-MOS-SSB, the BG-MOS collector that implements remembered sets with SSBs, uses the specified heap size to account only for heap objects. It keeps track of the remembered set overhead and accounts for it separately. BG-MOS-CARD, the BG-MOS collector that uses a card table, uses the specified heap size to account for heap objects, and accounts for the remembered set space used to record immortal pointers separately.

4.3.2 MOS Configurations

The BG-MOS collector we implemented accepts three parameters as input: *car size*, *train size*, and *collection threshold*. All three parameters are expressed as a percent of the

	BG-MOS-SSB						BG-MSC			BG-COPY		
HS	CS	TS	CT	LPT	HPT	APT	HPT	APT	ET	HPT	APT	ET
1.4	4	25	70	51.0	213.9	100.6	330.3	171.7	0.73	–	–	–
1.5	4	25	70	36.4	150.7	78.3	310.0	167.9	0.76	–	–	–
1.6	4	25	85	40.6	72.3	53.3	315.9	180.2	0.84	–	–	–
1.7	4	25	70	29.0	54.6	46.1	305.9	174.2	0.85	–	–	–
1.75	4	20	85	38.8	55.7	48.4	305.3	165.8	0.87	–	–	–
1.8	4	25	80	18.8	68.7	45.0	315.3	173.5	0.91	–	–	–
1.9	3	25	80	40.1	50.8	44.7	318.2	151.0	0.89	–	–	–
2.0	3	20	70	18.0	49.3	33.1	315.7	150.6	0.89	205.2	111.7	1.00
2.25	2	10	80	15.3	47.1	30.8	310.3	154.1	0.91	200.9	116.7	0.98
2.5	2	20	70	15.4	48.0	32.6	319.5	164.5	0.92	197.6	118.4	0.95
3.0	2	25	80	20.5	58.0	43.8	318.3	185.3	0.92	198.4	113.5	0.96

Table 4.2. Best BG-MOS-SSB configurations in heaps (HS) ranging from 1.4–3 times the maximum live size. For each heap size, the table shows the BG-MOS-SSB configuration that yields the lowest pause time (CS = car size, TS = train size, CT = collection threshold). The table shows lowest max. pause time (LPT) for BG-MOS-SSB, and the highest max. pause time (HPT) and the geometric mean of the maximum pause time (APT) across all benchmarks for BG-MOS-SSB, BG-MSC, and BG-COPY. Pause times are all in milliseconds. It also shows the geometric mean of the execution time (ET) for BG-MSC and BG-COPY relative to BG-MOS-SSB.

heap size. The car size is the minimum amount of space to collect every time the collector scavenges the old generation. The train size is the size at which a train is closed (no new allocation into the train). However, a train can grow larger than the specified train size because of objects copied into the train. The collection threshold is the minimum heap occupancy that causes old generation collection to be triggered.

We vary the car size from 1–4% of the heap size. Parameters for train size are 5%, 10%, 20%, 25% and 50%. A train size of 5% implies that there can be no more than 20 (1/5) trains in the old generation. Similarly 10%, 20%, 25% and 50% imply that there can be no more than 10, 5, 4 and 2 trains respectively. The collection threshold values that we experiment with are 70%, 80%, 85% and 90%. We refer to a particular BG-MOS configuration as *BG-MOS.cs.ts.th*, where *cs*, *ts*, and *th* are the car size, train size, and collection threshold respectively. For example BG-MOS.3.25.85 stands for the configuration with a car size of 3%, a train size of 25% and a collection threshold of 85%

4.3.3 Best Configurations

We first look at the performance of BG-MOS-SSB at a number of heap sizes ranging from 1.5 times the maximum live size to 3 times the maximum live size for each benchmark. We compare the relative performance of the BG-MOS-SSB and BG-MOS-CARD in Section 4.4. At each heap size, we consider only those benchmarks for which BG-MSC performs at least one full collection. .

Since BG-MOS-SSB uses remembered sets in addition to the heap space, for each heap size H and BG-MOS-SSB configuration C, we find the heap size H' at which $H' + MD(C) >= H$, where $MD(c)$ is the metadata overhead for C at heap size H'. We then use H' to evaluate the performance of configuration C at heap size H. At each of the heap sizes we look at the BG-MOS-SSB configurations that perform the best.

Table 4.2 shows a summary of the best BG-MOS-SSB configurations across a range of heap sizes, and a comparison of its performance with BG-MSC and BG-COPY. At each heap size, the best configuration has the lowest maximum pause time across all configurations. The average pause time for a configuration is the geometric mean of the pause times for the configuration across all benchmarks. The table shows the maximum and average pause times for all three collectors, and the execution times for BG-MSC and BG-COPY relative to BG-MOS-SSB.

The first observation we make is that BG-MOS-SSB is capable of running all benchmarks in heap smaller than twice the maximum live size. This is because it requires a copy reserve that is only little more than a car-full, and the metadata overheads are usually not very high.

In the smaller heaps (1.4–1.5 times the live size), the performance of BG-MOS-SSB is considerably lower than that of BG-MSC. It performs 24–27% worse than BG-MSC on average. However, the average pause times for BG-MSC are 1.8–2.2 times higher. In somewhat bigger heaps, between 1.6–1.7 times the live size, the performance is still 15–16% worse and average pause times are 3–4 times lower. In heaps between 1.8–2 times

the live size, the performance of BG-MOS-SSB is between 9–11% worse with an average pause time that is 4–5 times lower. In moderate size heaps between 2.25–3 times the live size, BG-MOS-SSB is 8–9% slower than BG-MSC on average. Average pause times are about 5 times lower and the highest maximum pause time is about 7 times lower.

When compared with BG-COPY, BG-MOS-SSB runs in much smaller heaps. While BG-COPY can run some benchmarks in heaps smaller than twice the live size, it runs all benchmarks only in a heap that is twice the live size. In moderate and large heaps, BG-MOS-SSB performs 3–5% worse than BG-COPY with pause times that are 3–3.5 times lower on average. The highest maximum pause time for BG-MOS-SSB is about a factor of 4 lower.

We also make a few observations about the best configurations. In smaller heaps, configurations with a larger car size (4) tend to yield lower pause times, while in larger heaps, configurations with a smaller car size (3, 2) perform better. This is probably because the larger car sizes allow the collector to reclaim space in the old generation sooner, thus preventing the collector from running low on space and thus having to collect many cars without any allocation in between. In larger heaps, collecting smaller cars works well because of the additional amount of space available. The optimal car size depends both on the heap size and the rate of allocation into the old generation. We could make the smaller car sizes provide short pauses in small heaps. However, this would require lowering the rate of allocation into the old generation and thus shrinking the nursery size. This would cause a significant reduction in overall throughput.

Overall, moderate size trains (20–25%) appear to perform well. A low collection threshold (70%) usually performs better (lower pause times) in very small heaps. This again is probably because this threshold allows the collector to reclaim old generation space sooner and thus prevents the collector from running into low memory situations. In some larger heaps, higher thresholds (80–85%) also work well.

Benchmark	Heap size relative to maximum live size					
	1.5	1.75	2	2.25	2.5	3
_202_jess	10.1%	8.2%	9.0%	7.1%	8.1%	6.2%
_209_db	2.1%	1.7%	1.5%	1.3%	1.1%	0.9%
_213_javac	9.9%	8.9%	10.1%	11.1%	9.1%	10.1%
_227_mtrt	5.8%	5.6%	5.7%	4.9%	4.4%	–
_228_jack	9.5%	7.6%	7.3%	6.5%	6.2%	6.0%
pseudojbb	3.8%	3.5%	3.1%	2.8%	3.8%	3.3%

Table 4.3. Maximum remembered set size as a percent of the heap size, for BG-MOS.3.20.70.

For each BG-MOS configuration with which we experiment, we measure an overall execution time. We do this by first obtaining a vector that contains for each benchmark and heap size, the execution time for the configuration relative to best execution time for that heap size achieved across all configurations. The configuration with the best overall execution time has the lowest sum-of-squares for its vector elements.

In the following sections, we look at the overall performance of two configurations, BG-MOS.4.20.85 and BG-MOS.3.20.70. BG-MOS.4.20.85 provides the best execution times across all configurations in heaps ranging from 1.5–3 times the live size. BG-MOS.3.20.70 provides good pauses in heaps that are twice the live size or larger, and generally provides better execution times than the other configurations with low pause times in this space. We look at the metadata overheads for BG-MOS, and compare the copying costs, GC times, and total execution times of the BG-MOS configurations with BG-MSC and BG-COPY. We also look at the pause time distributions for BG-MOS.3.20.70.

4.3.4 Metadata Overheads

Table 4.3 shows the remembered set overheads for BG-MOS.3.20.70 for heaps ranging from 1.5–3 times the live size. The numbers in the table show the overhead in terms of percent of heap size and represent the maximum space occupied by the remembered sets. An entry with a '–' indicates that BG-MOS does not perform a full collection at the heap

size. In addition to the overhead shown here, BG-MOS.3.2.70 has an overhead for the large object card tables that is equal to 1.5% of the total space occupied by large objects.

The remembered set overheads for jess, javac, and jack are high in small heaps, about 10% of the heap space. The overhead for javac is between 9–11% at all heap sizes. Overheads for jess and jack reduce a little in large heaps. mtrt and pseudojbb have moderate size remembered sets, occupying 3–6% of the space. db mostly contains pointers in large objects that are handled using the large object card table. The small object overhead is only 1–2% of the total space.

The remembered set overhead for BG-MOS does not decrease substantially as the heap grows. The reason for this is that BG-MOS records every reference from a higher numbered to a lower numbered car. So the overhead is not proportional to the live size of the benchmark. It is proportional to the number of cross-car pointers in the old generation, thus causing the percent occupied to be high even in large heaps.

4.3.5 Copying Costs

Figure 4.11, Figure 4.12, and Figure 4.13 show the copying costs for BG-MOS.3.20.70, BG-MSC, and BG-COPY. The horizontal axis for each of the graphs represents heap size relative to maximum live size. The vertical axis represents the *mark/cons* ratio, the ratio of total bytes copied to total bytes allocated. Both axes are drawn to a logarithmic scale.

In the smaller heaps, the copying cost for BG-MOS is substantially higher than that of BG-MSC. For jess, at about 1.5 times the live size, the copying cost for the BG-MOS collectors is about 5–10 times higher. For db, copying costs are about twice as high in small heaps. For javac, mtrt, and jack BG-MOS copies about three times as much as BG-MSC in small heaps. For pseudojbb BG-MOS copies about twice the amount BG-MSC copies. The additional copying is caused by the need for BG-MOS to copy related objects several times before they can be clustered together in a single train. Even if a data structure

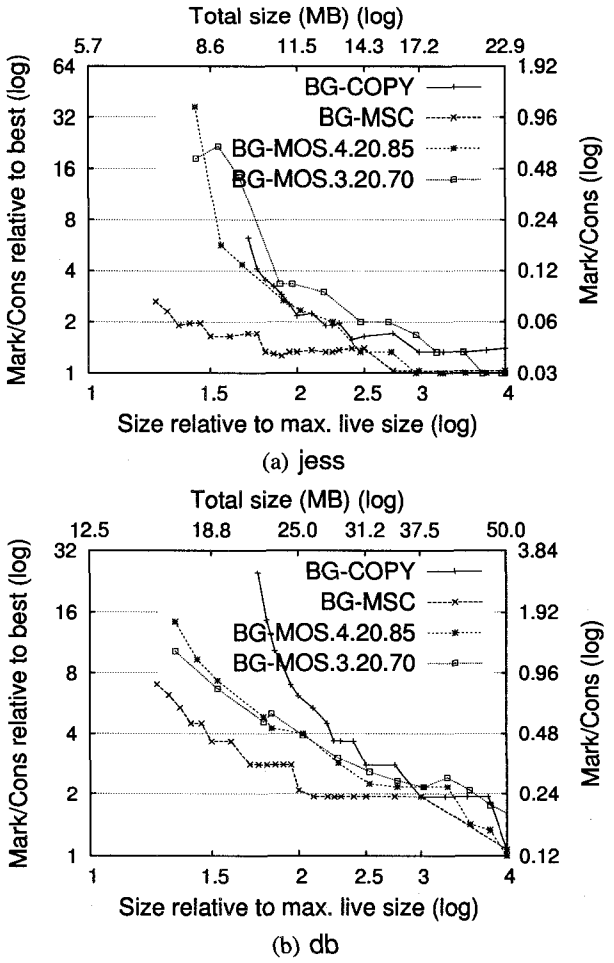


Figure 4.11. BG-MOS, BG-MSC and BG-COPY copying costs for jess and db.

is unreachable when BG-MOS starts collection, portions of it may be copied several times before they all converge into a single train and BG-MOS reclaims them.

In moderate size heaps (2–3 times the live size), the copying costs for BG-MOS are typically much lower. For jess and jack, copying costs are up to 2 times higher than BG-MSC. For db and pseudojbb, BG-MOS copies about 1.5 times as much as BG-MSC. mtrt is not interesting in this range since BG-MSC does not perform any full collections. In large heaps, the copying costs for these benchmarks drop down to 1–1.5 times that of

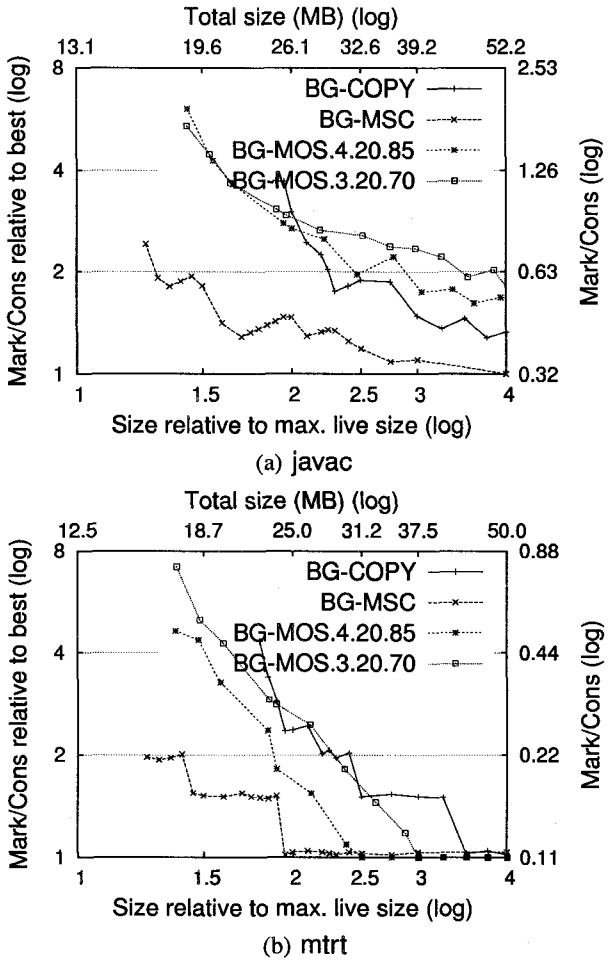


Figure 4.12. BG-MOS, BG-MSC and BG-COPY copying costs for javac and mtrt.

BG-MSC. At the points where copying costs are equal, neither collector is performing full collections.

javac is a particularly difficult benchmark for BG-MOS. It contains large cyclic structures, which cause the collector to copy large amounts of data and function poorly. As a result the copying overhead even in large heaps is high, up to a factor of two higher than that of BG-MSC.

When compared with BG-COPY, BG-MOS runs in smaller heaps as we observed earlier. BG-COPY, in the worst case, requires space equal to at least twice the live data size in

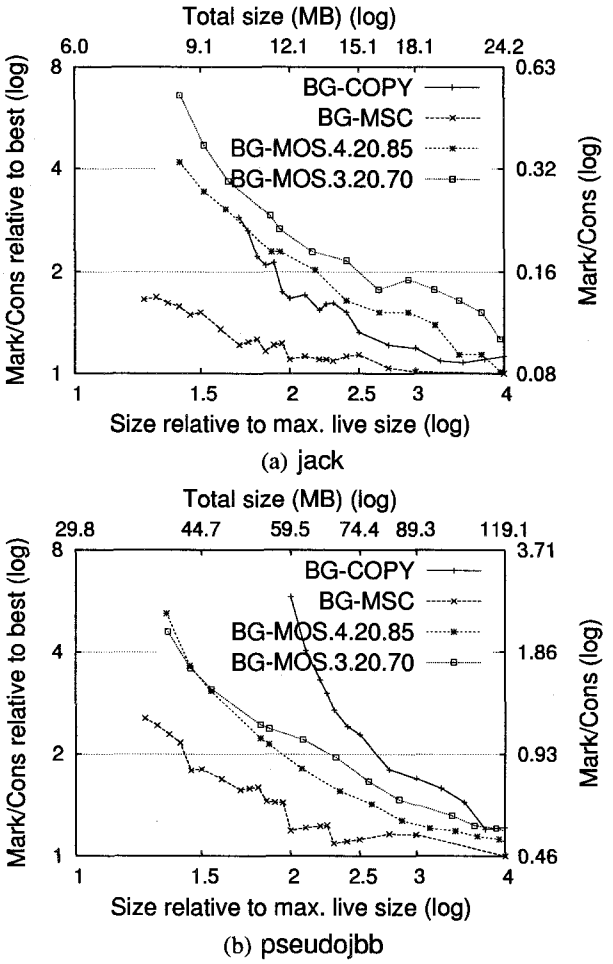


Figure 4.13. BG-MOS, BG-MSC and BG-COPY copying costs for jack and pseudojbb.

order to run. However, since the MMTk collector uses a large object and immortal space that do not require any space for copying, it requires space between 1.75–2.0 times the live size depending on the fraction of the live size that resides in large object and immortal space.

In very small heaps, the copying costs for BG-MOS are usually lower then BG-COPY. We observe this for db, javac, mtrt, and pseudojbb. db and pseudojbb contain a significant amount of permanent data, causing BG-COPY to perform poorly. In moderate size heaps, BG-COPY performs better than BG-MOS for javac, and jack. For jess, BG-COPY

copying costs are slightly lower than BG-MOS.4.25.80 until the BG-MOS collector stops performing full collections. For mtrt costs are about the same as BG-MOS.3.20.30, and somewhat higher than BG-MOS.4.25.80. For db and pseudojbb, copying costs for BG-COPY are consistently a little higher. In large heaps, copying costs are the same for jess, db, and mtrt since the collectors do not perform full collections in this space. Only for javac, are BG-MOS copying costs significantly higher than BG-COPY.

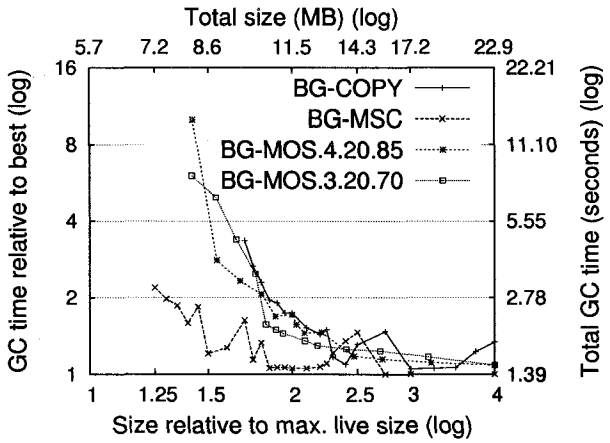
BG-MOS.4.20.85 generally has lower copying costs than BG-MOS.3.20.70. The lower cost is possibly because the larger car size allows it to copy fewer data structures that span multiple frames. The difference is small in smaller heaps, and is higher in moderate and large heaps where the difference in absolute car size is bigger.

Summary: BG-MOS performs substantially more copying than BG-MSC in small heaps, at least twice as much and up to eight times as much. In moderate size heaps, copying overhead is generally about twice as much as that of BG-MSC, and in large heaps it is usually at most 1.5 times higher. javac is an exception, with copying costs at least twice as high even in large heaps.

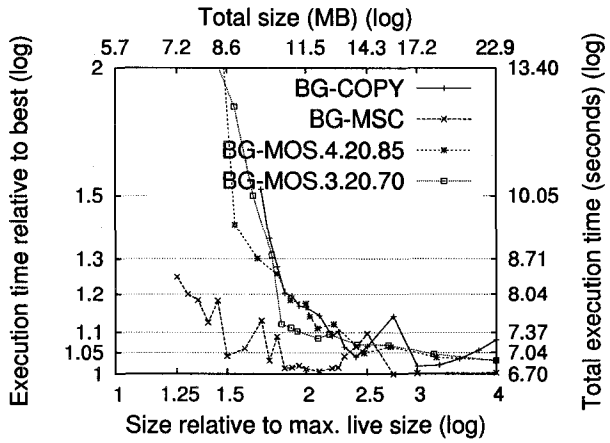
BG-MOS runs in much smaller heaps than BG-COPY, because of the smaller amount of space it reserves for copying. In heaps between 1.75–2.0 times the live size, BG-MOS typically copies less data. However, in moderate and large heaps, BG-MOS can have somewhat higher copying overheads than BG-COPY. The db and pseudojbb benchmarks contain significant amounts of permanent data, and are exceptions. BG-COPY almost always has higher copying overheads for these benchmarks. However, lower copying costs do not necessarily translate to lower GC and execution times for BG-MOS, because BG-MOS has additional write barrier costs, even when it is not collecting the old generation.

4.3.6 GC and Execution Time

Figure 4.14–Figure 4.19 show GC and total execution times for the three collectors over the six benchmarks. For all the graphs, the horizontal axis represents heap size relative to



(a) jess GC times



(b) jess total execution times

Figure 4.14. jess GC and total execution times for BG-MOS, BG-MSC, and BG-COPY.

maximum live size. For the GC plots, the vertical axis represents GC time relative to the best. For execution time plots, the vertical axis represents total execution time relative to the best. All axes are drawn to a logarithmic scale. We look at the performance of the collectors for each benchmark below.

jess: In small heaps between 1.5–1.75 times the live size, GC time for BG-MOS is 2–4 times higher than BG-MSC. Total execution times are 15–45% higher. In moderate-size heaps between 2–3 times the live size, GC time for BG-MOS is at most 1.5 times higher and total execution time is 5–10% higher. At a couple of points, the GC time for BG-MSC

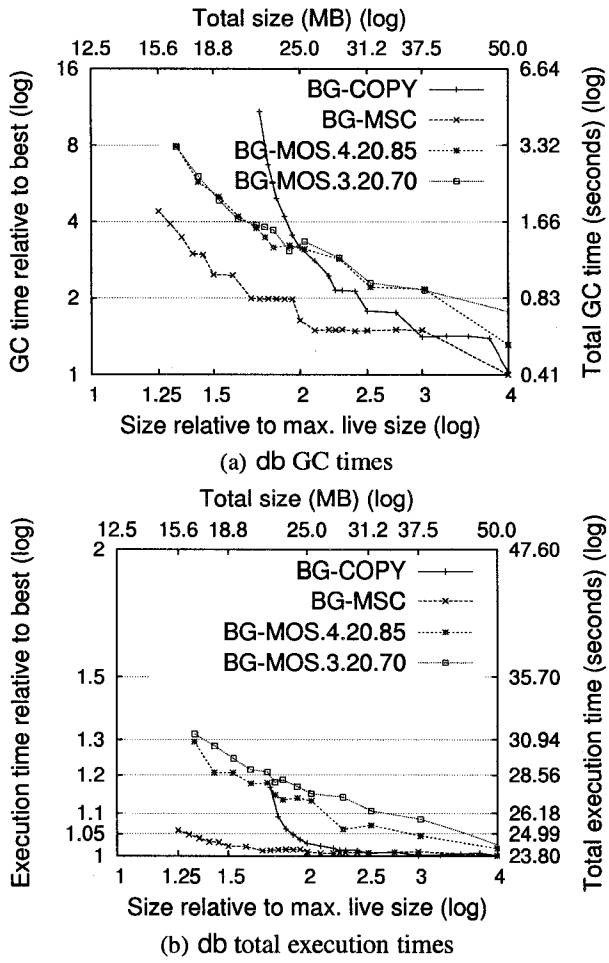


Figure 4.15. db GC and total execution times for BG-MOS, BG-MSC, and BG-COPY.

is slightly higher, possibly due to locality effects during GC, since the copying costs for BG-MSC are about the same or lower. In large heaps, the GC cost for BG-MOS is slightly higher and execution times are 4–5% higher.

GC and total execution time for BG-COPY are slightly higher than BG-MOS in small heaps. In moderate and large heaps, GC and total execution times are about the same. BG-COPY also experiences a couple of spikes and at these points it performs about 5% worse than BG-MOS.

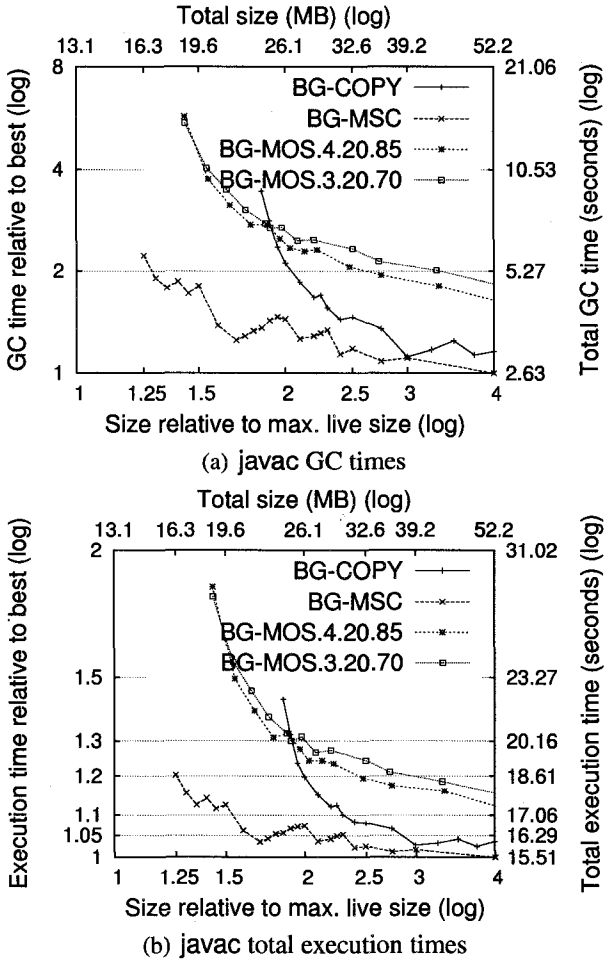
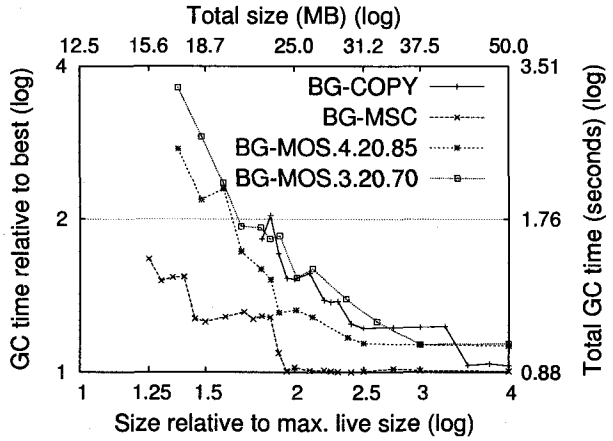
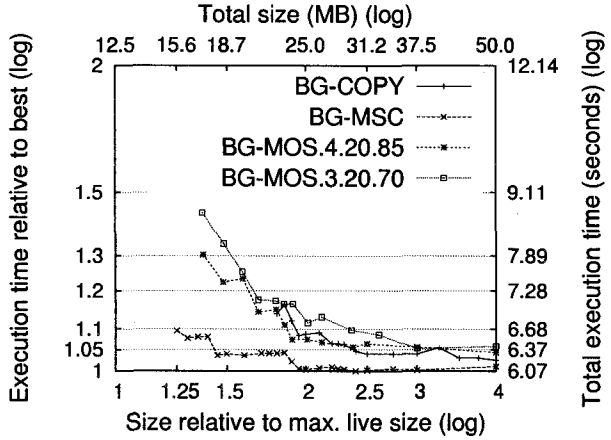


Figure 4.16. javac GC and total execution times for BG-MOS, BG-MSC, and BG-COPY.

db: GC times for BG-MOS are consistently about twice as high as BG-MSC in small and moderate heaps, and about 1.5 times higher in larger heaps. Total execution times are 15–25% higher in small heaps, 5-10% higher in moderate size heaps, and about 5% higher in large heaps. BG-COPY has significantly higher GC times than BG-MOS in small heaps, up to twice as high. In moderate and large heaps, BG-MOS has a 50% higher cost. However, execution times for BG-COPY are usually lower. BG-MOS performs worse because db performs a large number of mutations in the old generation causing the BG-



(a) mtrt GC times

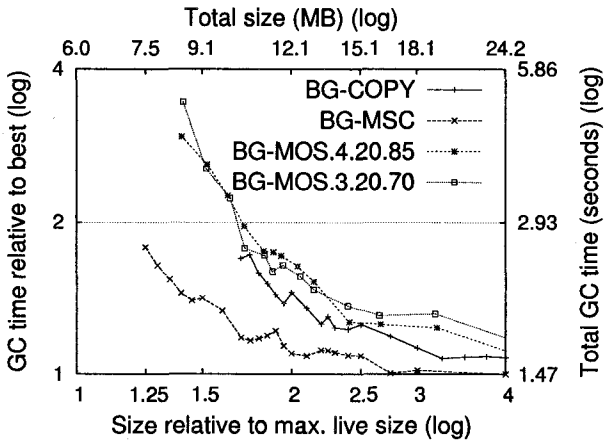


(b) mtrt total execution times

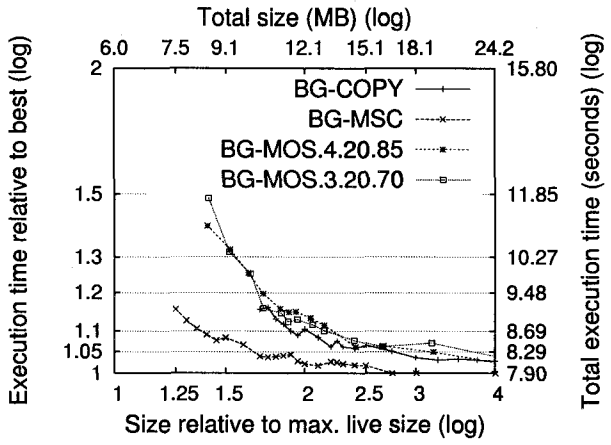
Figure 4.17. mtrt GC and total execution times for BG-MOS, BG-MSC, and BG-COPY.

MOS mutator write barrier cost to be very high, and also because db is very sensitive to locality effects.

javac: As we explained earlier, javac with its large, cyclic structures causes BG-MOS to perform poorly. GC times for BG-MOS are about 3 times higher than BG-MSC in small heaps, and twice as high in moderate and large heaps. Execution times are up to 70% higher in small heaps, and 15–20% higher in moderate and large heaps. BG-COPY also typically has better performance, and execution times for BG-MOS are 5-15% higher than BG-COPY.



(a) jack GC times



(b) jack total execution times

Figure 4.18. jack GC and total execution times for BG-MOS, BG-MSC, and BG-COPY.

mtrt: GC costs for BG-MOS are about twice as high in small heaps, 1.5 times higher in moderate heaps and about 20% higher in large heaps. Execution times for BG-MOS are 20-30% higher in small heaps, 5-10% higher in moderate size heaps and about 5% higher in large heaps. BG-COPY has slightly higher GC and execution times in small heaps. In moderate-size heaps it performs a few percent better than BG-MOS.3.20.70 and a few percent worse than BG-MOS.4.20.85 in moderate size heaps. In large heaps, BG-COPY performs 2–3% better than BG-MOS.

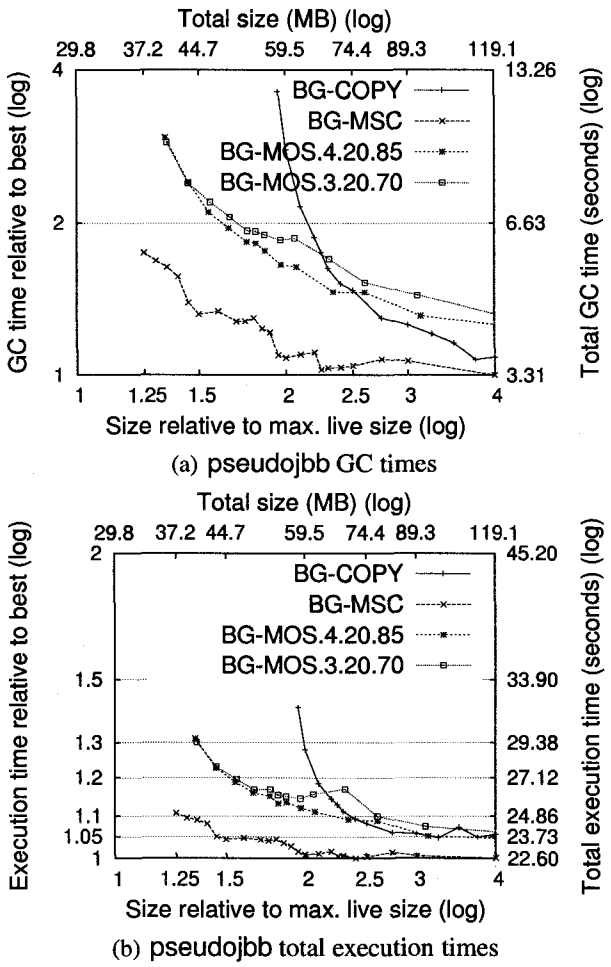


Figure 4.19. pseudojbb GC and total execution times for BG-MOS, BG-MSC, and BG-COPY.

jack: GC times for BG-MOS are twice as high as BG-MSC in small heaps and 20-50% higher in moderate and large heaps. Execution times are 10-40% higher in small heaps, and about 5% higher in moderate-size heaps. BG-MOS performs only a few percent worse than BG-COPY at most heap sizes.

pseudojbb: BG-MOS has GC times that are about twice as high as BG-MSC in small heaps, 50–80% higher in moderate size heaps, and 40% higher in large heaps. Execution times for BG-MOS are 10-20% higher in small heaps, 5–10% higher in moderate-size heaps, and 5% higher in large heaps. BG-COPY has execution times that are up to 25%

higher in small heaps, because of its high copying costs. In moderate and large heaps, BG-COPY performs about the same or a few percent better than BG-MOS.

Summary: BG-MOS has substantially higher GC costs than BG-MSC in small heaps, because of higher copying costs and remembered set maintenance costs. These costs cause the execution times to be generally 15-20% higher and up to 70% higher in the case of javac. In moderate size heaps, execution times are generally 5-10% higher, and 15-20% higher for javac. In large heaps, BG-MOS usually performs about 5% worse than BG-MSC.

When compared with BG-COPY, BG-MOS runs in smaller heaps. The performance of the two collectors is about the same in small heaps between 1.75–2 times the live size. db with its high mutation rates is an exception, and BG-COPY always performs better than BG-MOS for this benchmark. Execution times for jess, mtrt, jack, and pseudojbb are comparable in moderate and large size heaps. For javac, BG-COPY performs 5–15% better than BG-MOS in moderate and large size heaps.

BG-MOS.3.20.70, the configuration that yields good pause times and throughput in moderate size heaps, is usually slower than BG-MOS.4.20.85, the configuration that provides best overall throughput. However, the difference is not substantial, and total execution times for BG-MOS.3.20.70 are a few percent higher at most.

4.3.7 Pause Time Distribution

Figure 4.20–Figure 4.22 show the pause time distributions for BG-MOS.3.20.70 in a heap that is twice the live size. BG-MOS.3.20.70 has the lowest maximum pause times at this heap size. The graphs show the durations of the median pause, and of the 90th and 95th percentile pauses. The x-axis on all graphs shows the actual pause times and the y-axis shows the frequency of occurrence of each pause time. The y-axis is on a *logarithmic scale*, allowing one to see clearly the less frequently-occurring longer pauses.

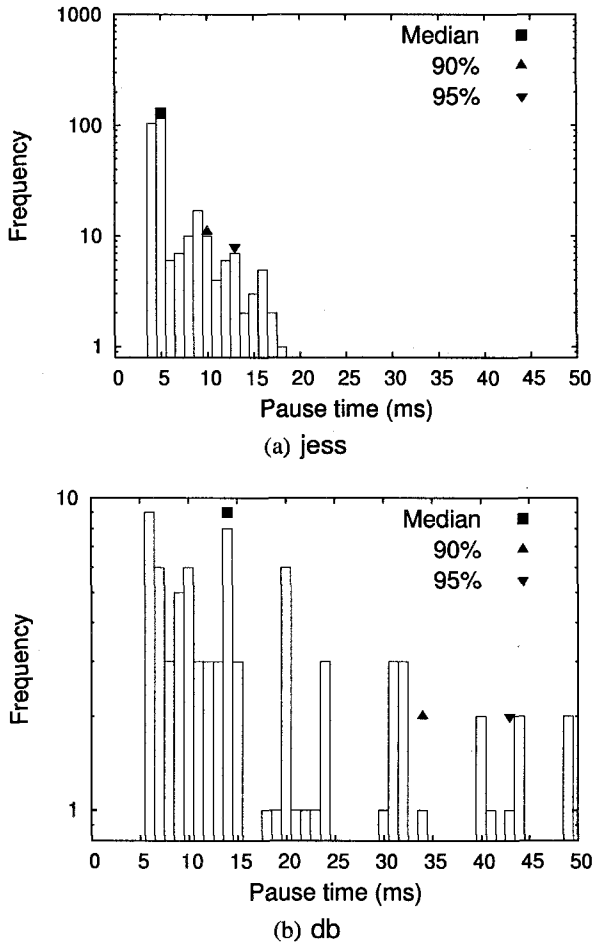


Figure 4.20. BG-MOS.3.20.70 pause time distributions for jess and db in a heap that is twice the maximum live size.

For jess, all pauses are of length 18ms or lower. Half the pauses are 5ms long or lower, 90% of the pauses are 10ms or lower. For db, the highest pause is 49ms long, and the median pause is 14ms long. 36% of the pauses are 10ms or lower, 81% are 30ms or lower, and 90% are 34ms or lower.

javac has a maximum pause time of 48ms. Only 13% of the pauses are 10ms or lower. The median pause is 22ms long. 78% of the pauses are 30ms or lower, and 90% are 35ms or lower. mtrt has a maximum pause time of 31ms, with a median pause that is 4ms long. 65% of the pauses are 10ms or lower, and 99% are 30ms or lower.

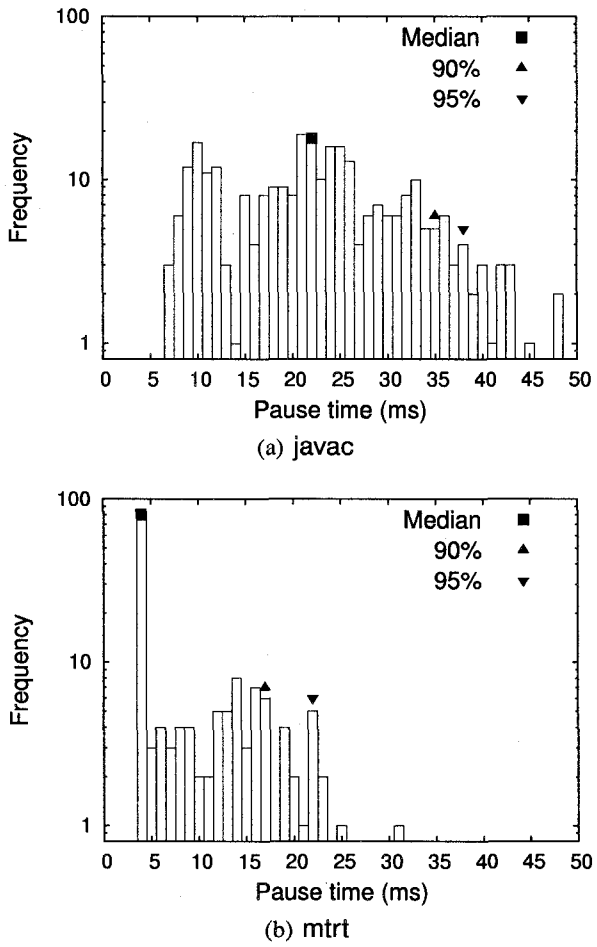


Figure 4.21. BG-MOS.3.20.70 pause time distributions for javac and mtrt in a heap that is twice the maximum live size.

For jack, the maximum pause time is 22ms long, the median pause is 7ms long, and 80% of the pauses are 10ms or shorter. pseudojbb has a maximum pause time of 39ms, with a median pause of 17ms. 31% of the pauses are 10ms or lower, and 93% of the pauses are 30ms or lower.

Summary: BG-MOS can provide short maximum pauses in a heap that is only twice the program maximum live size. Across the six benchmarks, the maximum pause time is 49ms. For jess and jack 80–90% of the pauses are 10ms or lower. For mtrt, 65% of the pauses are of length 10ms or shorter. For pseudojbb and db, 31% and 36% of the pauses

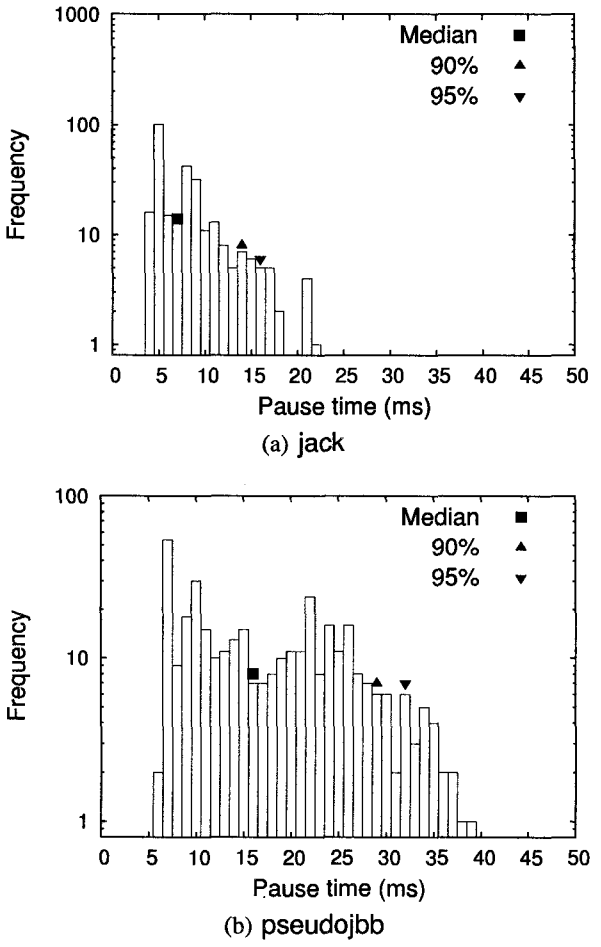


Figure 4.22. BG-MOS.3.20.70 pause time distributions for jack and pseudojbb in a heap that is twice the maximum live size.

are 10ms or shorter. javac tends to have somewhat longer pauses on average. Only 13% of it pauses are 10ms or lower, and at 22ms, its median pause is the longest across the benchmarks.

4.3.8 Mutator Utilization

We now look at the pause time characteristics of the collectors. We consider more than just the maximum pause times that occurred, since these do not indicate how the collection pauses are distributed over the running of the program. For example, a collector might

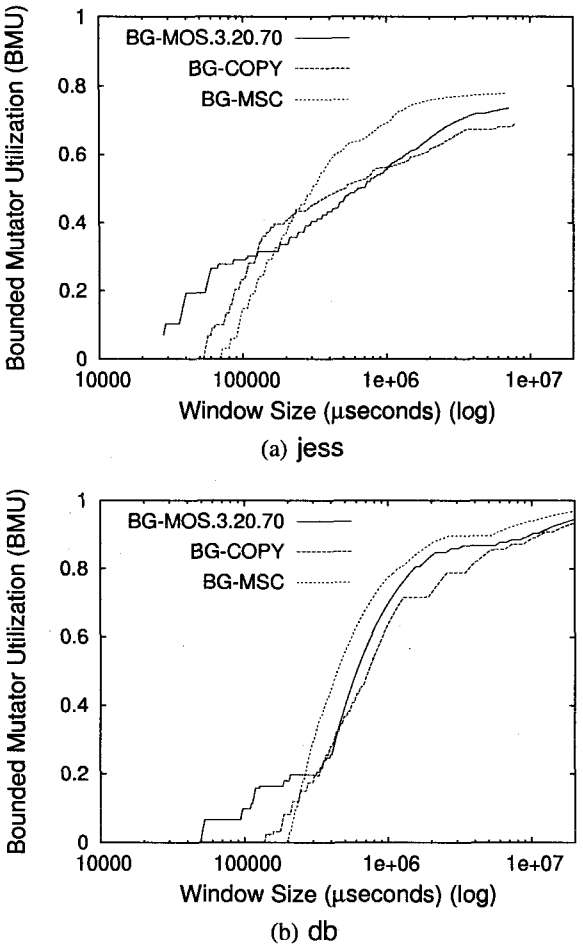


Figure 4.23. BG-MOS.2.3.70, BG-MSC, and BG-COPY mutator utilization curves for jess and db in a heap that is twice the maximum live size.

cause a series of short pauses whose effect is similar to a long pause, which cannot be detected by looking only at the maximum pause time of the collector (or the distributions).

We present *mutator utilization* curves for the collectors, following the methodology of Cheng and Blelloch [24]. They define the *utilization* for any time window to be the fraction of the time that the mutator (the program, as opposed to the collector) executes within the window. The minimum utilization across all windows of the same size is called the *minimum mutator utilization* (MMU) for that window size. An interesting property of this definition is that a larger window can actually have *lower* utilization than a smaller one. To

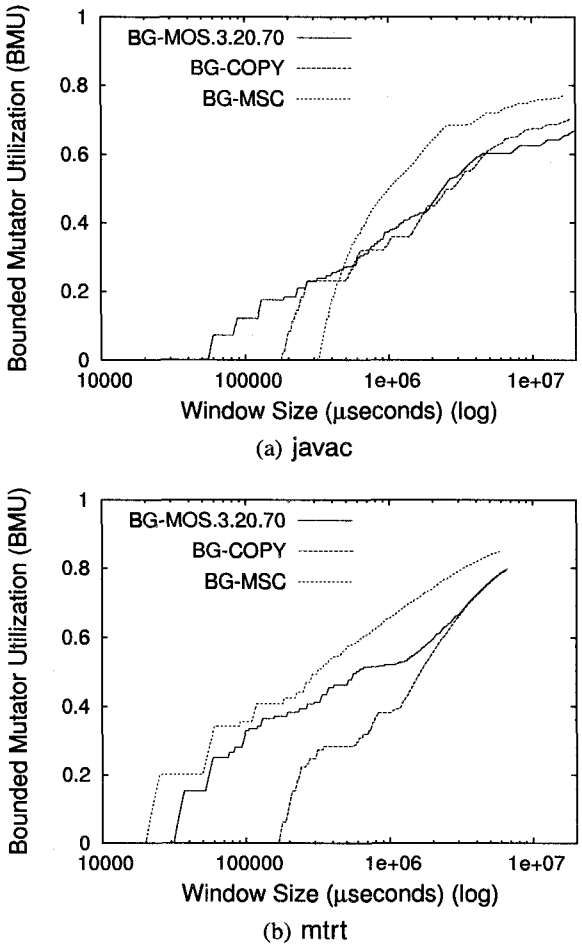


Figure 4.24. BG-MOS.2.3.70, BG-MSC, and BG-COPY mutator utilization curves for javac and mtrt in a heap that is twice the maximum live size.

avoid this, we extend the definition of MMU to what we call the *bounded minimum mutator utilization* (BMU). The BMU for a given window size is the minimum mutator utilization for all windows of that size *or greater*.

Figure 4.23–Figure 4.25 shows BMU curves for the six benchmarks for a heap size equal to twice the benchmark live size. The x-intercept of the curves indicates the maximum pause time, and the asymptotic y-value indicates the fraction of the total time used for mutator execution (average mutator utilization).

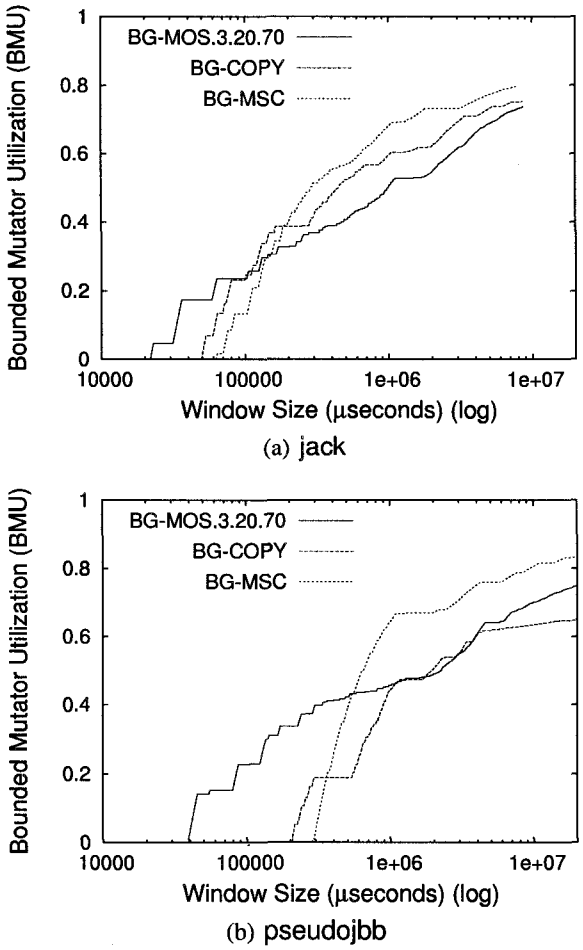


Figure 4.25. BG-MOS.2.3.70, BG-MSC, and BG-COPY mutator utilization curves for jack and pseudojbb in a heap that is twice the maximum live size.

Since it is difficult to factor out the write barrier cost, the curves actually represent utilization inclusive of the write barrier. The real mutator utilization will be a little lower. These graphs also do not show the effects of locality on the overall performance.

The three curves in each graph are for BG-MOS.3.20.70, BG-COPY, and BG-MSC. For all benchmarks, BG-MOS allows some mutator utilization even for very small windows. This is because of the low pause times for the collector. For these benchmarks, the mutator can execute for up to 10–25% of the total time in the worst case, for time windows that are about 55ms long. The non-incremental collectors, on the other hand, allow non-zero

utilization in the worst case only for larger windows, since they have large maximum pause times. An exception is mtrt, where the maximum pause time for BG-MSC is lowest since it is not performing any full collections.

BG-MOS can provide higher utilization than BG-COPY in windows of time up to 100ms for jess, windows up to 400ms for db, 300ms for javac, 1s for pseudojbb, 100ms for jack, and all window sizes for mtrt. When compared with BG-MSC, utilization is higher for windows up to 200ms, 250ms, 500ms, 150ms, and 500ms for jess, db, javac, jack, and pseudojbb respectively.

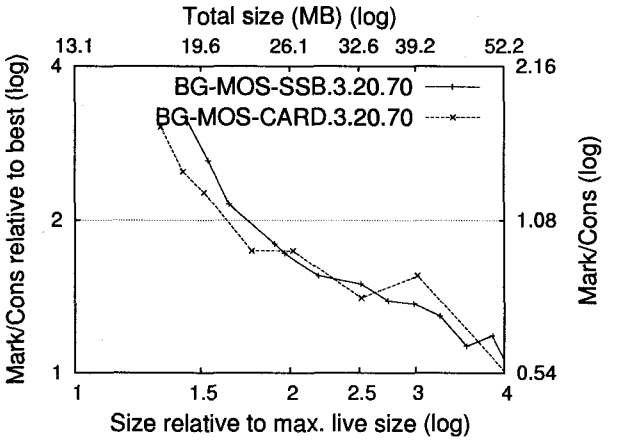
Beyond that, the utilization provided by BG-MOS is always lower than BG-MSC. When compared with BG-COPY, the overall utilization is higher for jess and pseudojbb, lower for javac and jack, and about the same for mtrt. Although the utilization appears to be close for db, the high cost of the BG-MOS write barrier and locality effects cause the utilization to be lower than BG-COPY.

Summary: The mutator utilization curves for the collectors in a heap that is twice the live size show that BG-MOS spreads its pauses well. As a consequence, BG-MOS provides higher mutator utilization for small windows of time. This holds even for windows of time that are larger than the maximum pause times for the non-incremental collectors. In large windows, BG-MOS typically performs substantially worse than BG-MSC, and its performance is comparable with that of BG-COPY.

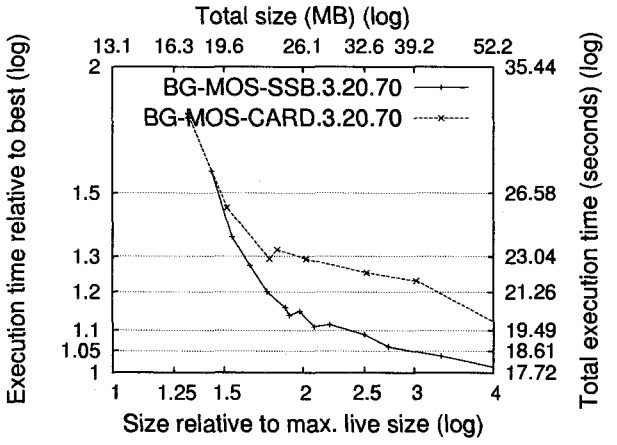
4.4 SSB vs. Card Table

We compare in this section the performance of BG-MOS implemented using SSBs (BG-MOS-SSB) and BG-MOS implemented using a card table (BG-MOS-CARD). Figure 4.26 and Figure 4.27 show the copying costs and total execution times of the collectors using the same configuration for two benchmarks, javac and pseudojbb.

We had stated in Section 4.2.5.2 that we expect the copying costs for BG-MOS-CARD to be a little lower than that of BG-MOS-SSB because of the order in which BG-MOS-



(a) javac copying costs



(b) javac total execution time

Figure 4.26. BG-MOS-SSB and BG-MOS-CARD copying costs and total execution times for javac.

CARD processes pointers. We see in Figure 4.26(a) and Figure 4.27(a) that BG-MOS-CARD has slightly lower copying costs at most heap sizes.

However, other costs of using the card table tend to be much higher than that of an SSB. These include the GC write barrier that records pointers, card table scanning costs, and the overhead of bitmap maintenance. The write barrier is particularly expensive when compared with the SSB write barrier, which simply inserts into a buffer. The card table scanning costs include both the actual scan and the cost of processing every object in a dirty card regardless of whether it contains a pointer into the car being collected. The

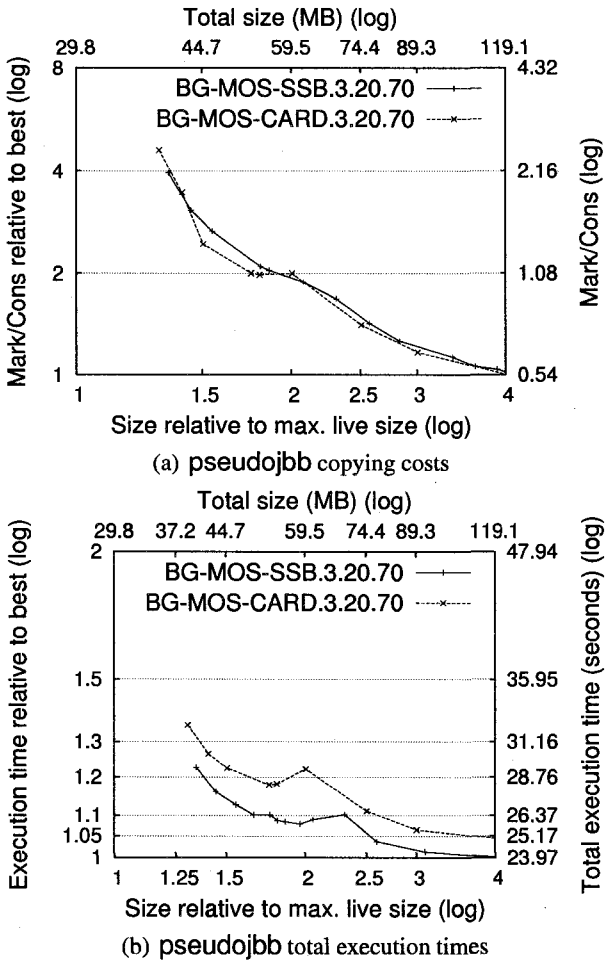


Figure 4.27. BG-MOS-SSB and BG-MOS-CARD copying costs and total execution times for pseudojbb.

execution time curves show that the difference can be substantial for javac, and 5–10% for pseudojbb.

However, card tables allow the collector to have a bounded metadata overhead, and as we saw in Section 4.3.4, SSB overheads can be high occasionally. BG-MOS might thus benefit from using a hybrid SSB/card table remembering scheme that we describe in Chapter 6.

4.5 Conclusions

The MOS collector meets its goal of extending generational copying collection and yielding short pauses. We showed that the collector can consistently provide short maximum pause times in a heap that is twice the maximum live size or larger. MOS can also run with low space overhead. This is because it reserves a small fraction of the heap space for copying, unlike a generational copying collector that reserves up to half the total space on the heap. We demonstrated that the MOS collector can run in heaps as small as 1.3 times the program live size.

However, the collector incurs a substantial overhead in throughput especially in small heaps. In heaps between 1.4–1.5 times the live, performance for the configuration with the lowest maximum pause time is 24–27% worse than a mark-compact collector. In heaps between 1.6–1.75 times the live size, performance is 13–16% lower, and in heaps between 1.8–3 times the live size, the configurations with the best pause times are 8–11% slower.

The cost in throughput is caused primarily by the high copying costs of the collector. In order to collect an unused data structure that spans multiple collection regions in the heap, the collector needs to copy objects in the structure repeatedly until they all reside in a single logical unit. Additionally, the collector also incurs a cost for managing metadata even when it is not collecting the old generation. This results in a moderately high number of dead entries in remembered sets. While there is a time and space cost associated with recording and maintaining these entries, the dead entries further increase the amount of copying the collector performs.

We conclude that while the MOS collector can provide short pauses, the throughput overhead is substantial in small heaps, thus making it unsuitable for applications running on memory constrained devices.

CHAPTER 5

MARK-COPY

We saw in the previous chapter that copying collection techniques typically cannot be used in constrained memories. Regular semi-space copying collection and generational copying collection require a factor-of-two space overhead ruling them out automatically. The MOS collector can run in small heaps, but has a high throughput cost in small heaps.

We present here a new copying collection technique, *Mark-Copy*, that overcomes this space limitation of previous copying collectors. Mark-Copy (MC) can run in small heaps and provide good throughput. It achieves this by dividing the heap into a number of small regions in a manner similar to that of the MOS collector. However, it achieves completeness by using a simple heap layout and a separate mark phase rather than the car/train technique used by MOS. It thus copies less data and provides higher throughput.

MC is a non-incremental collector and it does not provide short pauses. We describe in the next chapter a collector that extends MC and provides short pauses and high throughput in constrained memory.

The layout of this chapter is as follows. We first describe details of the mark-copy collection algorithm. We then study its space utilization properties, showing that it can run with little space overhead. Next we look at the metadata used by the collector and describe a technique that bounds space occupied by metadata. Finally we describe implementations of the collector in MMTk and present results comparing the performance of the collector with a generational copying collector, a mark-sweep collector, and a generational mark-sweep collector.

5.1 Mark-Copy Algorithm

Mark-Copy (MC) extends generational copying collection. Like generational copying collection, it can be implemented with a variable-size nursery (VG-MC), a bounded-size nursery (BG-MC), or a fixed-size nursery (FG-MC). While MC collection has no restriction on the number of generations, we consider only two generations.

5.1.1 Heap Layout

MC divides the heap into two areas: a nursery, and an old generation that has two regions. The nursery is identical to a generational collector's nursery, but the old generation regions are broken down into a number of *windows*. The windows are of equal size, and each window corresponds to the smallest increment of collection in the old generation. The windows are numbered from 1 to n, with lower numbered windows collected before higher numbered windows. Given a heap of size H and an old generation with n windows, the amount of copy reserve space required is H/n (one window), and the old generation can grow to size $H - H/n$. MC windows correspond to cars used by the MOS collector. However, the overall heap layout for MC is much simpler than that for MOS. Unlike MOS, which places cars into logical trains, MC treats all windows alike and does not partition them into separate logical units.

5.1.2 Collection Procedure

MC performs all allocation in the nursery, and promotes survivors of nursery collection into the old generation. Like a generational copying collector, MC uses a write barrier to record pointer stores that point from the old generation into the nursery. This is done to avoid having to scan the entire heap to find nursery survivors. After each nursery collection, the collector checks the amount of free space remaining in the heap. If it finds that free space in the heap has dropped to the size of a single window (a little more than that in practice), it invokes a full collection.

As the first step in the full collection, the collector performs a full heap mark starting from the roots (stack(s), statics). While the marking is in progress, the collector calculates the total volume of live objects in each old generation window. At the same time, it constructs remembered sets for each of these windows. These remembered sets are unidirectional: they record slots in higher numbered windows that point to objects in lower numbered windows. The point is to record pointers whose target may be copied before the source, a condition that requires updating the source pointer when the collector copies the target object. At the end of the mark phase, the collector knows the exact amount of live data in each window. It has also constructed a remembered set for each window. The remembered set entries for each window have the following properties: they are live (not an overestimate of the live set); they are current (i.e., they really point into the window); and they are unique (i.e., the remembered set does not contain any duplicates).

Once the old generation mark phase is complete, the collector performs the copy phase. The copy phase is broken down into a number of *passes*, each pass copying a subset of the windows in the old generation. The collector has the option either of copying the old generation windows one after the other without performing any allocation in between, or of interleaving copying with nursery allocation. We focus on the first technique here, as our goal is increased throughput and not lower pause times.

Since the collector knows the exact volume of live data in each old generation window and also the total free space available in the heap, it may collect multiple windows in a single pass. However, since the remembered sets are unidirectional, the windows must be collected strictly in order of window number, and not in any arbitrary order. One way to overcome this restriction is to build bidirectional remembered sets, which store pointers into each window from objects in all other windows. However, this would increase the space overhead of the remembered sets and complicate their management. We do not support bidirectional remembered sets in our current implementation.

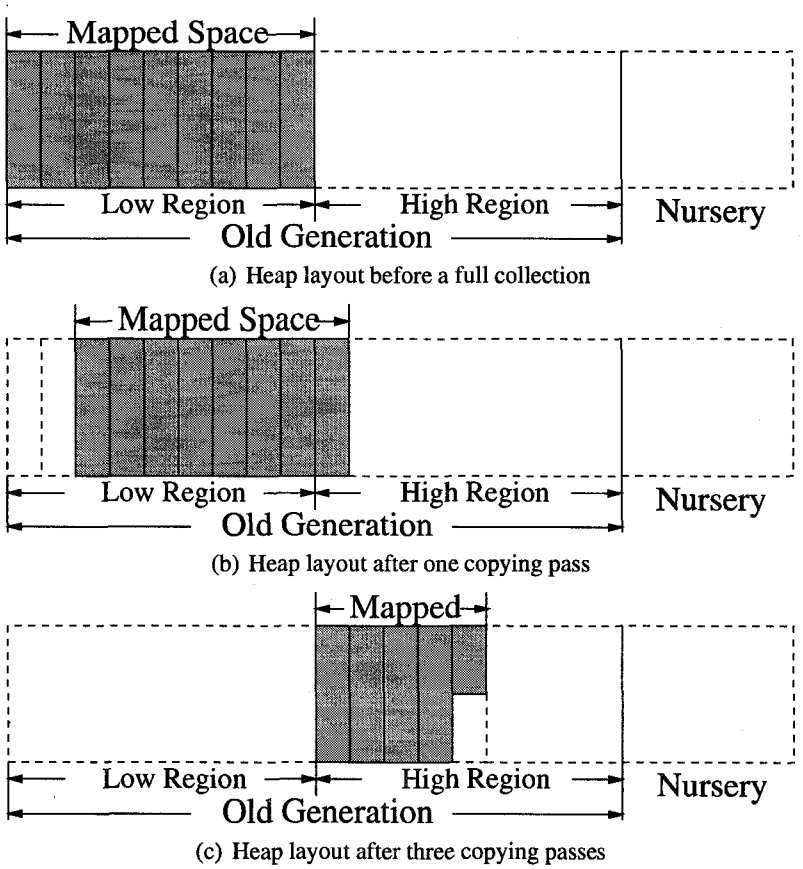


Figure 5.1. Heap layout during full MC collection, with a 50% survival rate in each window.

5.1.3 Virtual Memory Layout

Figure 5.1 shows the virtual memory layout of the collector performing copying in a heap whose old generation regions are divided into nine windows each. Since each old generation region is divided into nine windows, the old generation can occupy up to 90% of the total heap space. Before copying commences, the nine windows in the low region are all full and the heap has enough free space for a single window in the high region (Figure 5.1(a)). In this particular case, we assume that the amount of live data in each low region window is exactly 50%. The first copy pass scans live objects in the first two windows of the low region and copies them into the first window of the high region. The roots

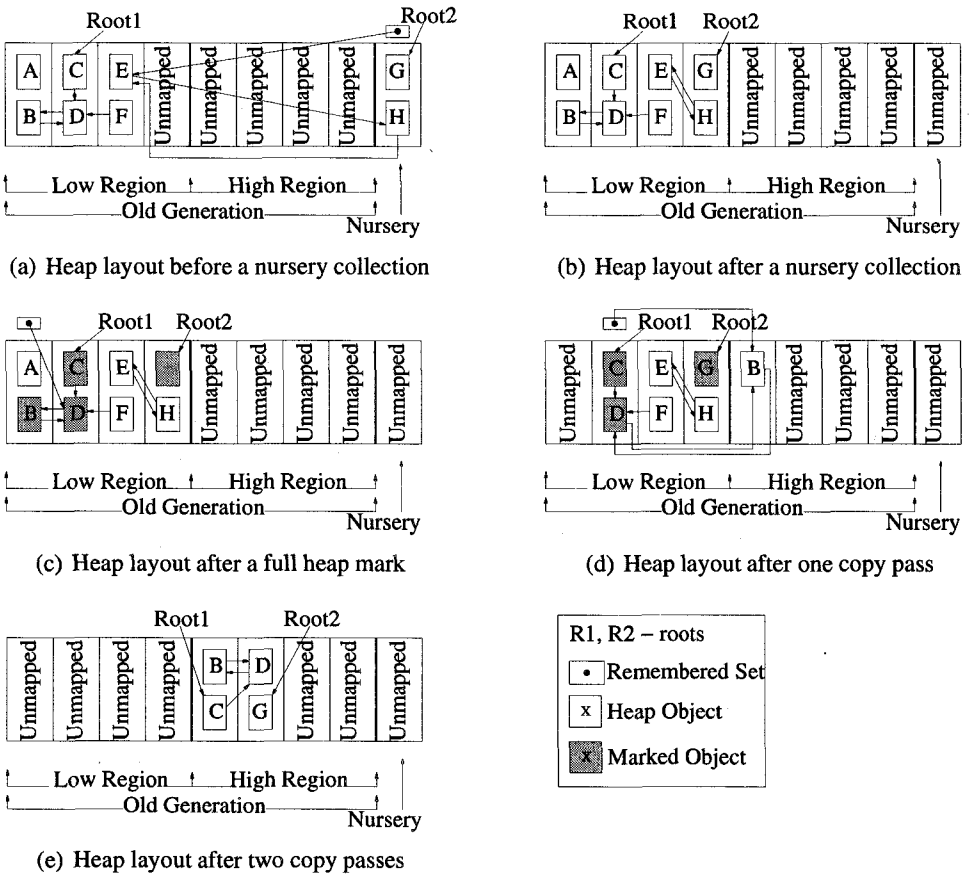


Figure 5.2. An example of a nursery collection followed by a full collection for MC.

for the collection are the stack(s), statics, and remembered set slots. When copying is complete, the total amount of mapped virtual memory equals the total heap space. At this point, the space consumed by the first two windows in the low region is released (unmapped), as shown in Figure 5.1(b). This means that the mapped space after the completion of one copying pass is 80% of the maximum total heap space. The free space in the heap now allows objects from four windows in the low region to be copied in the next pass. Finally, in the third pass the objects from the last three windows are copied, leaving the heap as shown in Figure 5.1(c).

5.1.4 Mark-Copy Example

Figure 5.2 shows a detailed example of a full collection using MC. For this example we assume that all objects allocated in the heap have the same size, and that the heap can accommodate at most 10 objects. The heap consists of an old generation with 4 windows. Each of these windows can hold exactly 2 objects. Root1 and Root2 are root pointers. Figure 5.2(a) shows a nursery collection, which copies objects G and H into the old generation. G is copied because it is reachable from a root, and H is copied because it is reachable from an object (E) in the old generation. At this point, the old generation is full (Figure 5.2(b)). MC first performs a full heap mark and finds objects B, C, D, and G to be live. During the mark phase it builds a unidirectional remembered set for each window. After the mark phase (Figure 5.2(c)), the remembered set for the first window contains a single entry (D→B). All other remembered sets are empty, since there are no pointers into the windows from live objects in higher numbered windows. (If we used bidirectional remembered sets, the second window would contain an entry to record the pointer from B to D.) In the first copy pass, there is enough space to copy two objects. Since the first window contains one live object (B) and the second window contains two live objects (C, D), only the first window can be processed in this pass. MC copies B to a high region window and then unmaps the space occupied by the first window (Figure 5.2(d)). It also adds a remembered set entry to the second window, to record the pointer from B to D (since B is now in a higher numbered window than D, and B needs to be updated when D is moved). The old generation now contains enough free space to copy three objects. In the next copying pass, MC copies the other three live objects and then frees up the space occupied by windows 2, 3, and 4 (Figure 5.2(e)).

The mark phase of the MC collector thus serves two purposes:

- By calculating the free space in each window, the mark phase minimizes the number of passes the copy phase needs to make in order to copy all the data. For example, for an old generation with nine windows and an overall survival rate of 10%, MC

needs only one copying pass. This is because the copy reserve of 10% is enough to accommodate all the collection survivors. However, in the worst case, when the survival rate is over 90%, it will need nine passes to copy the data. One should, however, note that for survival rates over 50% a standard copying collector would not even be able to run the program in this space.

- By building remembered sets, the mark phase ensures that the copy phase needs to perform only a single scan over the old generation objects in spite of having to perform the copying in multiple passes.

5.1.5 Collector Properties

From the above description, we can see that the amount of space reserved for copying can be made extremely small by increasing the number of old generation windows. However, the minimum copy reserve space needs to be at least as large as the largest object allocated in the heap. By allowing control over the copy reserve, the collector is able to run in very tight heaps. It also makes the memory footprint of the collector much more predictable than that of a typical copying collector. For a heap of size H, the footprint of a generational copying collector varies between $H/2$ and H, i.e., by a factor of 2. However, given an old generation that has n windows, the footprint of the MC collector varies between $H - H/n$ and H, a factor of $n/(n-1)$. Since we can control the value of n, we can minimize the variation (subject to the maximum object size).

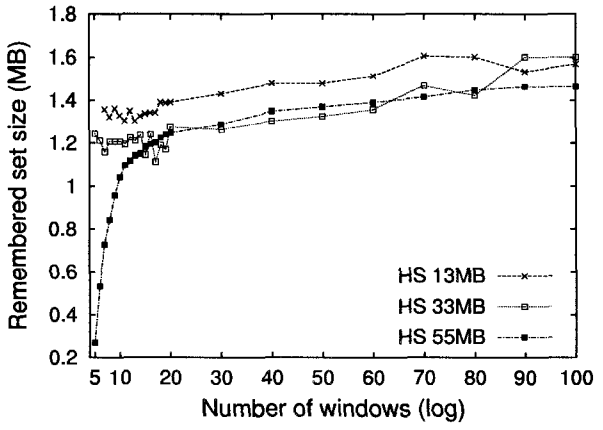
MC defers full collections until the free space drops down to one window. Standard copying collectors perform full collections when the heap becomes half full. Hence, when MC and other copying collectors are given the same amount of space, MC will perform many fewer full collections. This decreased full collection frequency gives objects in the old generation a longer time to die, resulting in much less copying than regular copying collection. Another way to think of this is that MC increases the *effective* heap size.

MC collectors do have some disadvantages compared with generational copying collectors. Each full collection for MC will scan every live object in the heap twice, once while marking and once while copying. The mark phase requires additional space for a mark stack and there is a copying remembered set overhead. Also, for an old generation consisting of n windows, up to n passes (but generally many fewer) over the stacks and statics may be required. This could be significant if there are a large number of threads or the thread stacks are very deep. This overhead could be reduced by storing these slots in the remembered sets during the mark phase. However, doing so would require additional space. While we do not offer details here, we found that scanning stacks and statics did not make a large contribution to GC time for our benchmark suite.

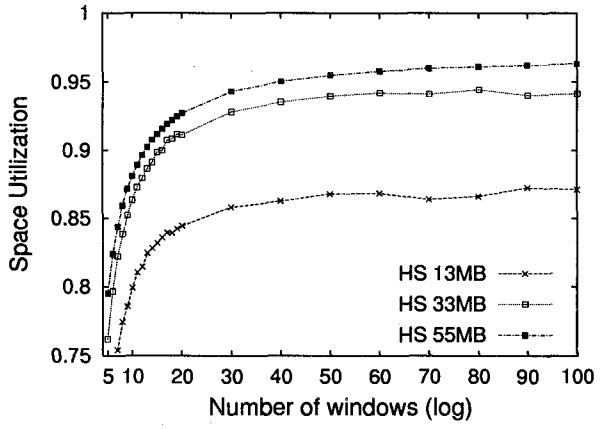
5.2 Remembered Sets

The MC collector builds remembered sets at two different points during program execution. While the mutator is running, the write barrier inserts slots into the nursery remembered set. This is identical to what a generational copying collector does and occupies the same amount of space. During the mark phase of a full collection, MC constructs unidirectional remembered sets to record pointers between windows in the old generation. These remembered sets are not required by a generational copying collector and are an additional space overhead for MC. However, the nursery and old generation remembered sets do not co-exist in a non-incremental MC collector, since MC performs old generation marking only after a nursery collection completes (this policy causes nursery survivors to be copied twice). We look at the overhead of old generation remembered sets here.

Normally we would expect the remembered set space overhead to increase as the number of old generation windows increases. This does not necessarily happen, for a couple of reasons. First, the set size depends on the locations of the objects in the old generation, i.e., if objects that reference each other lie close together in the old generation, then increasing the number of windows will not have a significant impact. Second, an increase in the num-



(a) Remembered set size



(b) Space utilization

Figure 5.3. Effect of the number of old generation windows on the space usage of MC for javac in heaps of size 13MB, 33MB and 55MB.

ber of windows increases the amount of usable space in the heap. This in turn changes the points at which collection occurs, and hence the amount of live data in the heap during full collection. Since the remembered set entries are accurate and depend on the amount of live data in the heap, a change in the volume of live data could reduce the remembered set size.

5.2.1 Remembered Set Analysis for javac

Figure 5.3(a) shows the variation in the maximum overall remembered set size as one varies the number of windows, for heap sizes which are 1.2, 3, and 5 times the maximum live size (13MB, 33MB, 55MB), for javac. The horizontal axis represents the number of windows. We look at the remembered set sizes for an old generation with number of windows ranging from 5 to 100. Although we do not use more than 20 windows in the old generation with the non-incremental collector, it is important to see what the remembered set overhead is with a large number of windows, since an incremental collector might need to use 100 or more windows to ensure a low pause time.

The remembered set sizes drop initially in the smaller heaps, and then generally increase as the number of windows is increased. The size in the smallest heap is about 1.35MB with 7 windows and about 1.57MB with 100 windows. The size in the largest heap is about five times larger with 100 windows than with 5. However, if we look at the remembered set size as a fraction of the total heap space, it is not very large even when we use 100 windows (12% in the smallest heap and 2.6% in the largest heap).

Figure 5.3(b) shows the space utilization for MC as we increase the number of old generation windows. We calculate the space utilization for a given number of windows n using the formula $(H-(H/n)-R)/H$, where H is the heap size, H/n is the copy reserve size, and R is the maximum remembered set size. For all heap sizes, we obtain significant gains in space utilization until the number of old generation windows grows to about 40. This is because the copy reserve is reduced by a larger amount than the corresponding increase in remembered set size. Increasing the number of windows beyond 50 causes the space utilization to improve slightly or even to deteriorate a little.

5.3 Mark-Copy without Remembered Sets

Although the remembered set overhead for MC is not usually very high, in the worst case remembered sets could grow to be as large as the heap. We describe in this section

a variant of MC that uses an extra header word per object in the old generation (MCHW), eliminating the need for remembered sets and thus bounding the worst case space utilization. However, unlike MC, MCHW can be used only as a non-incremental collector.

The heap layout for MCHW is identical to the layout for MC. MCHW performs allocation in the nursery and promotes survivors to the old generation. MCHW adds an extra header word to each object that is copied into the old generation. When the free space in the heap drops down to the size of a single window, it invokes a full collection. Like MC, MCHW performs a full collection in two phases.

During the mark phase, MCHW marks all objects reachable from roots. While performing the marking, it maintains a *live offset array* that stores the sum of the sizes of the live objects found so far in each window. It also maintains a bitmap that indicates the locations of the live objects. While marking a reachable object, the collector computes a *logical address* for the object. The logical address is a combination of the window to which the object belongs and the live offset of the object within the window. The logical address is stored in the extra header word reserved for the object and it indicates the location to which the object will be copied in the next phase. Every time a slot referencing the object is found, the contents of the slot are replaced with the logical address of the object. Figure 5.4 shows the logical address layout used for MCHW. The lowest bit is set to 1 to indicate that the value stored is a logical address. (Objects are word aligned and the machine is byte addressed, so ordinary object pointers always have zeroes in the two low order bits.) The middle bits store the live offset, and the high order bits store the window number.

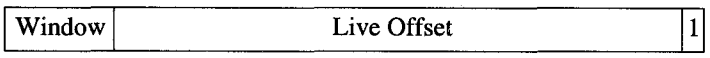


Figure 5.4. Logical address layout for MCHW.

Once the mark phase is complete, the live offset array contains the sum of sizes of the live objects in each window. The collector now calculates a cumulative live offset (CLO),

which for each window is the sum of the offsets of all preceding windows. Using this information, the new address for any heap object is calculated using the logical address (LA) of the object and the start address of the copy space (CSA) using the formula:

$$CSA+CLO[Window(LA)]+LiveOffset(LA)$$

The MCHW collector requires that the entire copy space be contiguous (in virtual memory). Otherwise it will need to perform an extra pass over the live objects to determine the correct cumulative live offset (since objects may span two windows). Once the offsets are calculated, the collector starts the copy phase. It first scans the roots and updates pointers into the heap using the above formula. It then scans the bitmap, processing one window at a time. For each live object in the window being processed, it updates any pointers that the object contains using the above formula, and then copies the object to the to-space location computed from its logical address. After processing all live objects in a window, it unmaps the space occupied by the window and then processes the next. The copy phase terminates when all windows have been processed.

As can be seen from the description, MCHW does bound the worst case space utilization. It requires one additional word per object plus space for a bitmap. The size of the bitmap is approximately 3% of the heap space when objects are word aligned (one bit per 32-bit word). However, since it adds an extra word to each object, the layout of the old generation is not as compact as that for MC, which could cause negative locality effects and also reduce the effective heap size.

5.4 Mark-Copy vs. MOS

We look here at the possible reasons for MC outperforming MOS in terms of throughput. We defer a quantitative comparison of our technique with MOS to the next chapter, where we describe an extension to MC that achieves short pauses in addition to good space utilization and throughput.

Figure 5.5 shows a simple example illustrating the differences in the copying and pointer remembering overheads of the two collectors. Figure 5.5(a) shows the heap layout for MOS just before it initiates an old generation collection. The heap consists of one doubly linked list that spans two trains. Figure 5.5(b) shows the heap layout for MC just before it starts a full collection. The heap contains the same doubly linked list data structure.

The MOS collector will function as follows. It first copies object A to the end of Train 1, since it it only reachable internally. It then copies B to the end of Train 1. When it reaches C, it finds that C is reachable from Train 2 and hence it copies C to the end of Train 2. When it collects A next, it copies A again to the end of Train 1 since it is only reachable internally. It then copies B to the end of Train 2, and eventually in the next pass copies A to the end of Train 2. In the next collection, it detects that the reference count for Train 1 is zero, and it reclaims the entire data structure. However, in order to group the data structure into a single train and eventually reclaim it, the collector needs to copy six objects. In addition, it needs to maintain remembered set entries for every pointer from a higher numbered to a lower numbered car.

MC on the other hand does not mark any of the data since it is all unreachable. Hence it does not need to maintain remembered set entries for any of the pointers, and it does not copy any of the objects in the data structure.

The example illustrates two important differences between MOS and MC. MC marks and copies only data that is reachable at the point when it starts old generation collection. In addition, it creates remembered set entries only for live pointers, and does not maintain remembered sets when it is not performing any collection.

However, MOS must always maintain remembered sets even when it is not performing collection. When it actually starts collecting data, the remembered sets may contain entries that belong to currently unreachable objects. There is no way the collector can tell that it is processing a slot that belongs to a dead object unless it performs a full heap scan. This

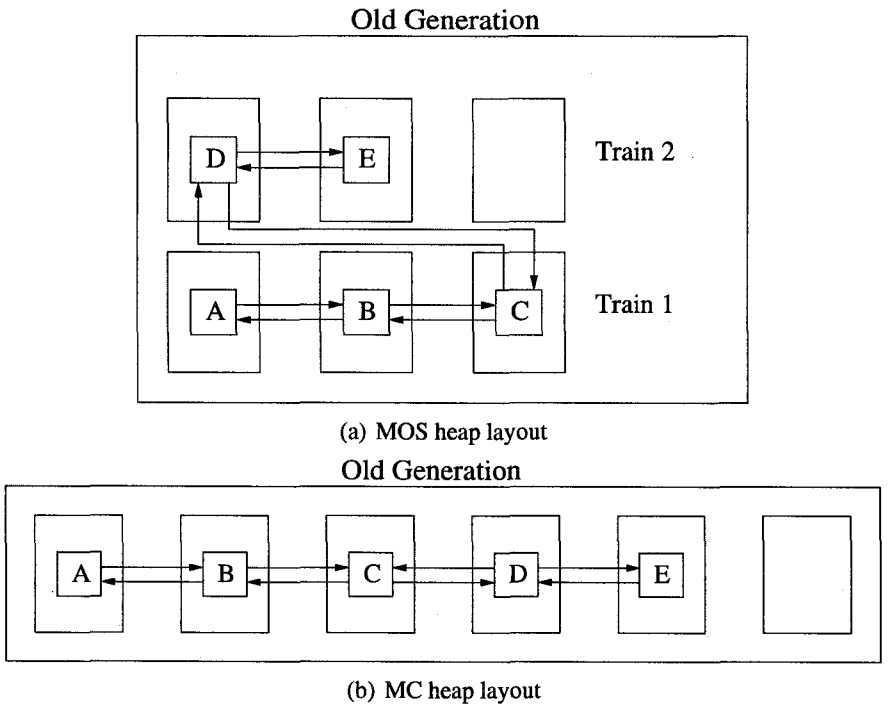


Figure 5.5. An example illustrating the difference in collection costs between MOS and MC.

property, combined with the fact that MOS may need to copy an object several times before it can place it in the appropriate train, causes the copying cost of MOS to be high in the presence of large data structures, (especially large dead data structures).

5.5 Implementation Details

Our implementation of MC divides the entire usable virtual address space into a number of *regions*. The lowest region stores boot image objects, the next region stores immortal objects, and a third region stores large objects. All these regions come by default in the MMTk framework, for any collector. MC allocates objects larger than 8KB into the large object region, and this region is managed by the MMTk large object allocator.

The type information block (TIB) objects, which include the virtual method dispatch vectors, etc., and which are pointed to from the header of each object of their type, are allocated by MMTk into immortal space. This has a couple of benefits for the MC collector. First, the TIB objects do not have to be copied when performing collection. Second, the remembered set overhead is reduced significantly since we do not store pointers into immortal space in the remembered sets.

MC creates two more regions, one for the old generation and one for the nursery. The old generation is divided into a number of fixed size *frames*. A frame is the largest contiguous chunk of memory into which allocation can be performed. The frame size in our implementation is 8MB. Each frame accommodates at most one old generation window. Old generation windows can, however, span multiple frames. The portion of the address space within a frame that is not occupied by a window is left unused. Since the frames are power-of-two aligned, only a single shift is required to find the frame number of a heap object during garbage collection.

MC uses a fast write barrier that records only pointer stores that cross the boundary separating the nursery region from the rest of the heap; the same barrier as in the generational copying collectors. The write barrier is partially inlined [14]: the code that tests for a store of an interesting pointer is inlined, and the code that inserts interesting slots into a remembered set is out of line. Each frame has an associated remembered set. The remembered set is implemented as a sequential store buffer (SSB) [35]. Remembered sets store the addresses of slots residing in the heap and the boot image, and that contain interesting pointers.

The MCHW implementation uses the same memory layout as MC, except for the layout of the old generation. The old generation for MCHW is not divided into power-of-two aligned frames. Instead, it is divided into page-size aligned windows when a full collection is performed. This ensures that objects are allocated contiguously in the old generation (which avoids an extra pass over the heap to calculate addresses). This condition does

Benchmark	Live Size (MB)	Total Allocation (MB)
_202_jess	4.0	291
_209_db	10.5	85
_213_javac	11.0	285
_222_mpegaudio	3.0	27
_227_mtrt	10.0	145
_228_jack	4.5	329
pseudojbb	28.0	334
health	67.5	552

Table 5.1. Benchmarks used for the evaluation of MC.

not necessarily hold true when frames are used since the window size and frame size are usually not equal.

5.6 Experimental Results

We present in this section experimental results comparing MC with state-of-the-art non-incremental collectors. We compare space overheads, copying costs, GC times, and total execution times for a generational copying collector with a variable-size nursery (VG-COPY), VG-MC, and VG-MCHW collectors. We also compare GC times and total execution times of VG-MC with a (non-generational) mark-sweep collector (MS) and a hybrid copying/mark-sweep generational collector that uses a variable size nursery (VG-MS). Since the focus here is on throughput, we use only variable sized nurseries for the generational collectors.

For the MC collectors, we use 20 windows in the old generation. For each benchmark, we run the collectors in heaps ranging from the minimum space required to run the benchmark to 8 times the maximum live size. We ran all experiments on a Macintosh Power PC running Linux. We offer details of the machine configuration in Chapter 3. Table 5.1 lists the benchmarks we use and their live sizes, and the total allocation they perform. We run all experiments on JikesRVM 2.2.3.

Benchmark	Heap size relative to maximum live size						
	1.1	1.2	1.5	2	3	4	6
_202_jess	8.8%	8.3%	5.1%	1.4%	–	–	–
_209_db	1.6%	1.5%	1.0%	0.8%	0.3%	–	–
_213_javac	10.3%	9.2%	7.1%	4.8%	3.7%	2.8%	–
_222_mpegaudio	4.4%	3.7%	2.9%	2.0%	–	–	–
_227_mtrt	2.9%	2.1%	1.6%	–	–	–	–
_228_jack	5.1%	2.7%	2.6%	0.9%	0.6%	–	–
pseudojbb	1.7%	1.6%	1.2%	0.8%	–	–	–
health	14.2%	11.9%	–	–	–	–	–

Table 5.2. Maximum remembered set size as a percent of the heap size, for MC using 20 windows in the old generation. An entry with a '–' indicates that MC did not perform a full collection for the heap size, and hence did not construct remembered sets.

5.6.1 Space Accounting

In all the experiments we ran, the VG-COPY collector used the specified heap space to allocate heap objects and nursery remembered set entries. The nursery remembered set space was not accounted for separately. MC also used the heap space in the same manner. However, the old generation remembered sets were accounted for after each run completed, by adding the maximum space occupied by the remembered sets to the heap space. MCHW used the specified heap space to store heap objects (along with an extra header word), the nursery remembered set, and a bitmap.

Benchmark	Header Word overhead
_202_jess	9.0–10.7%
_209_db	13.5–13.7%
_213_javac	13.9–14.1%
_222_mpegaudio	9.7–11.0%
_227_mtrt	14.5–15.9%
_228_jack	11.3–11.6%
pseudojbb	5.2–6.6%
health	18.2–18.5%

Table 5.3. Average header word space overhead for MCHW.

5.6.2 Mark-Copy Space Overheads

Table 5.2 shows the remembered set space overhead for all eight benchmarks, for an MC collector that has 20 windows in the old generation. For each benchmark, the table shows the remembered set size for heap sizes ranging from 1.1 to 6 times the live size. The heap sizes do not include the remembered set size. They do, however, include a 7.5% copy reserve (we trigger a full collection when the free space drops down to 1.5 windows, so that the collector does not use nurseries smaller than 2.5% of the heap space). Each entry in the table represents the remembered set size as a percent of the heap size. An entry with a '–' indicates that MC did not perform a full collection for the heap size, and hence did not construct remembered sets.

For four of the eight benchmarks, the remembered set overhead for MC is always less than 5% of the maximum live size. For javac the overhead is about 11% of the live size, and for health the overhead is about 15%. The total space overhead for MC varies between 12% and 25% of the maximum live size.

The header word overhead for MCHW is higher than the remembered set overhead for MC. Table 5.3 shows the overhead for the eight benchmarks. We calculate the overhead by dividing the header word size (4 bytes) by the average old generation object size (excluding the header word). This is because the header word is added only to old generation objects. The space overhead for MCHW is about 5–10% higher than the overhead for MC, still considerably lower than the generational copying overhead.

5.6.3 Copying Costs

We examine here the copying characteristics of VG-MC and VG-COPY. Although the performance of copying collectors is influenced by factors such as locality, the total amount of data copied is a good performance indicator and can explain differences in overall performance. For generational collectors, the amount of copying is strongly related to the

number of full collections. We first look at the full collection behavior of VG-MC and VG-COPY for two benchmarks, and then see how this affects the amount of data copied.

Figure 5.6 shows the number of full heap collections performed by VG-MC and VG-COPY for javac and pseudojbb. The horizontal axis represents the total space used relative to the maximum live size, and is drawn to a logarithmic scale. The vertical axis represents the total number of full collections performed. For javac, VG-MC has a significant advantage until the heap has grown to 4.5 times the maximum live size. In space that is about 2.3 times the maximum live size, VG-MC performs as few as three full collections. VG-COPY performs approximately five times as many full collections at this point. The performance for pseudojbb is even better, with VG-MC performing only one full collection in space that is about two times the maximum live size. VG-COPY catches up with VG-MC only when the space grows to five times the maximum live size.

VG-MC performs many fewer full collections because it defers full collection until the heap is almost full, while VG-COPY has to perform them when the heap is half full. The survival rate of objects in the old generation is much higher than nursery objects, since the old generation typically contains long lived objects. For example, for VG-COPY running javac in a heap that is 2.3 times the live size, 66% of old generation data survives on average, while only 21% of data in the nursery survives. By deferring full collections, VG-MC gives the old generation objects a longer time to die, thus significantly reducing the survival rate, which in turn reduces the total amount of copying work. Therefore, even though VG-MC potentially does more work at each full collection, the lower frequency allows it to do much less work overall.

Figure 5.7 shows the effect of the number of full collections on the amount of copying performed by the collectors. The graphs show the relative *mark/cons* ratio, the ratio of total bytes marked and copied ("marked") to total bytes allocated ("cons'ed"), for VG-MC and VG-COPY, for javac and pseudojbb. For javac, VG-COPY copies approximately two times more data than VG-MC in space that is about 2.3 times the maximum live size. At

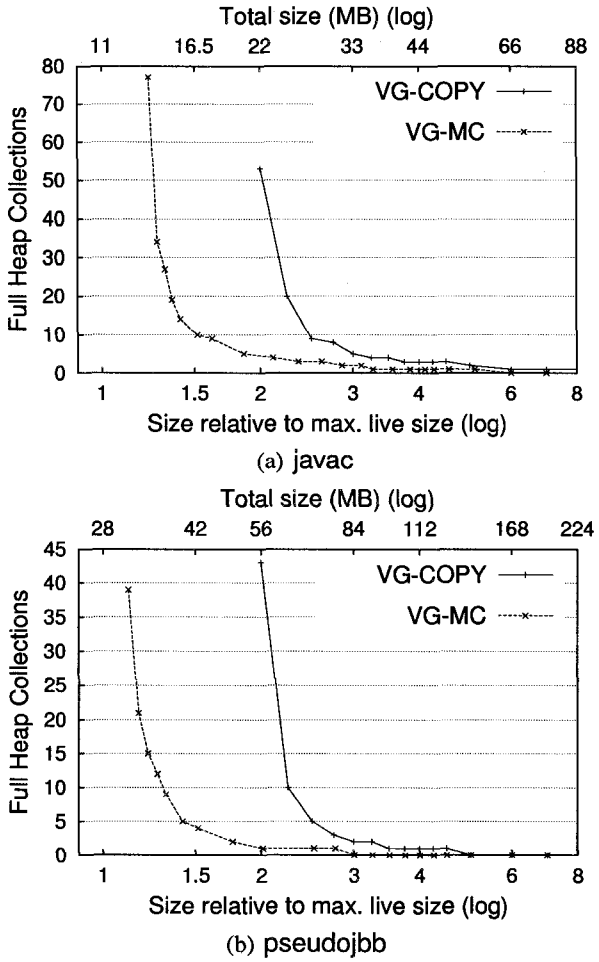


Figure 5.6. Number of full heap collections performed by Mark-Copy (VG-MC) and a generational copying collector (VG-COPY) for javac and pseudojbb.

four times the live size, VG-COPY copies 30% more data than VG-MC. In heaps that are six times the live size or larger, VG-MC does not perform full collections and the copying cost for VG-COPY is 20-25% higher. For pseudojbb, the copying cost for VG-COPY is twice as much as that for VG-MC, in space that is about 2.5 times the maximum live size. In heaps 2.5–4.5 times the live size, the cost is at least 50% higher. Eventually, at six times the live size, the collectors perform roughly equal amounts of copying. In the next section we look at the effect this reduced copying has on overall performance of the collectors.

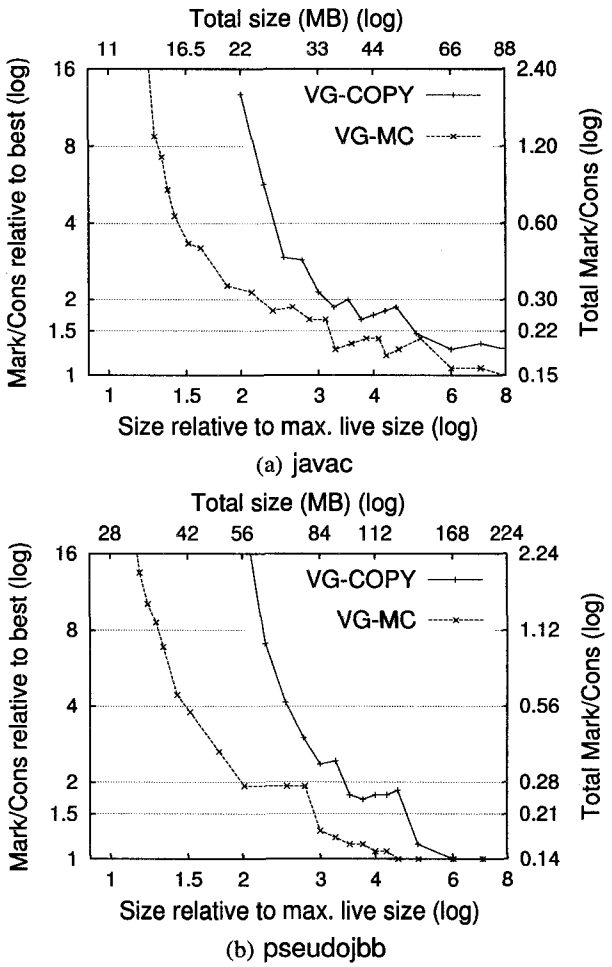


Figure 5.7. Relative Mark/Cons ratio for Mark-Copy (VG-MC) and a generational copying collector (VG-COPY) for javac and pseudojbb.

5.6.4 Mark-Copy vs. Copying Collection

Figure 5.8–Figure 5.15 shows GC times and total execution times for VG-MC, VG-MCHW, and VG-COPY, relative to the best times for the eight benchmarks. Although we measured the performance of BG-MC, we omit the results in the graphs in order to make them legible. We briefly discuss the results for BG-MC at the end of the section. In the graphs, the vertical axis represents relative times, and is drawn to a logarithmic scale. The horizontal axis represents total space relative to the maximum live size, and is also drawn to a logarithmic scale, so as to show greater detail for smaller heap sizes.

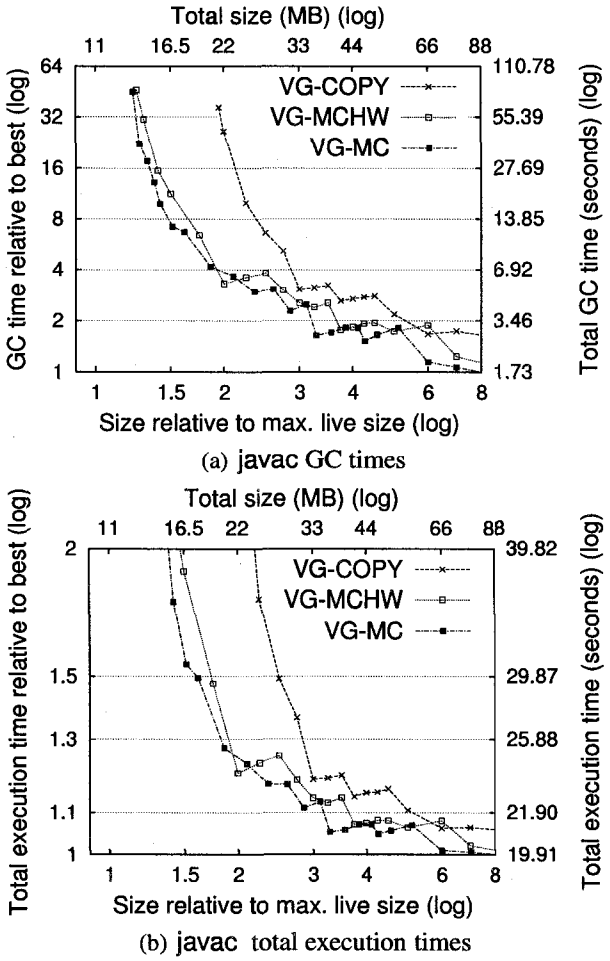


Figure 5.8. javac GC and total execution times relative to the best time for VG-MC, VG-MCHW, and VG-COPY.

For all eight benchmarks, the MC collectors clearly run in much smaller space than VG-COPY. Since VG-COPY uses a large object space (which is managed by a mark-sweep collector), it can run in heaps slightly smaller than twice the maximum live size (it does not have to reserve any space for copying large objects). For all of the benchmarks, the VG-COPY collector does much worse than the MC collectors when the space available is around twice the maximum live size.

javac: When the space available is about 2.3 times the maximum live size, the GC time for VG-COPY is twice as high as that for VG-MC. In space that is about four times the

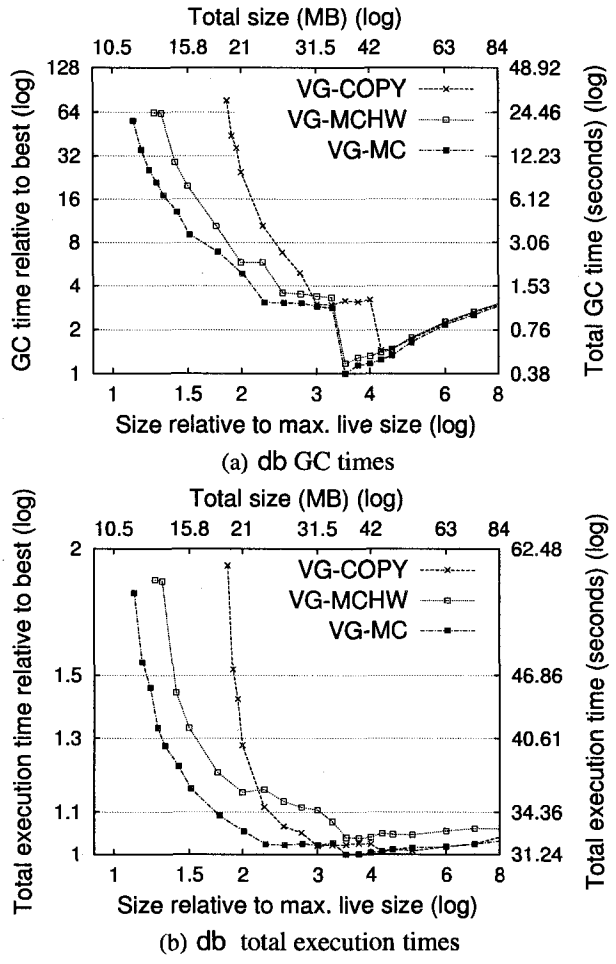


Figure 5.9. db GC and total execution times relative to the best time for VG-MC, VG-MCHW, and VG-COPY.

maximum live size, the GC time for VG-COPY is 1.5 times higher. When the total space grows to about five times the live size, the GC time for VG-COPY is 15% higher than that for VG-MC. The GC time for VG-MC drops sharply at this point because it does not perform any full collections. The GC time for VG-COPY is 50–60% higher than that for VG-MC in space that is six times the maximum live size or larger.

The shapes of the curves for the relative execution times are similar. In space that is about 2.3 times the maximum live size, total execution time with VG-COPY is 40% higher than with VG-MC. VG-COPY consistently performs within 10% of VG-MC only in heaps

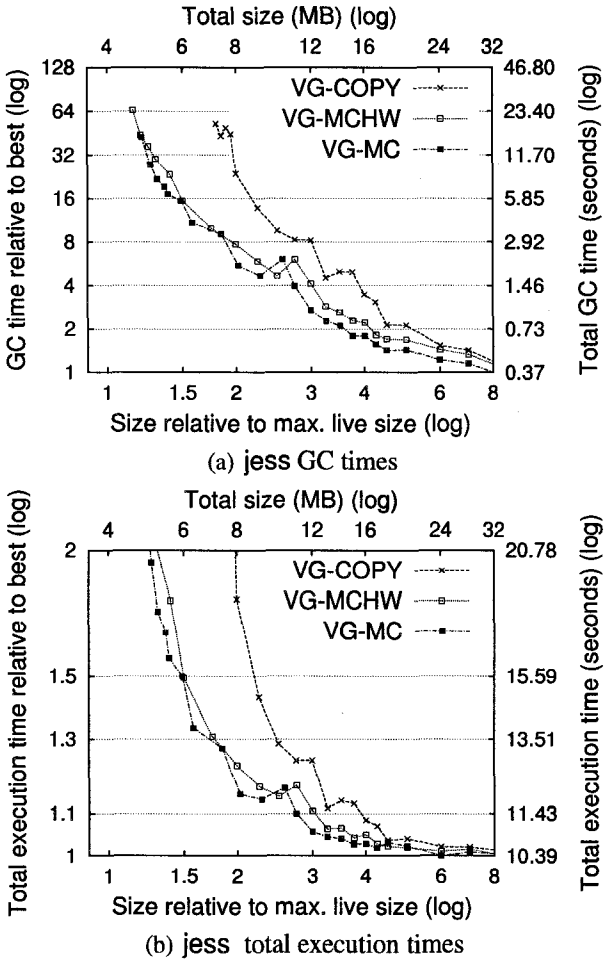
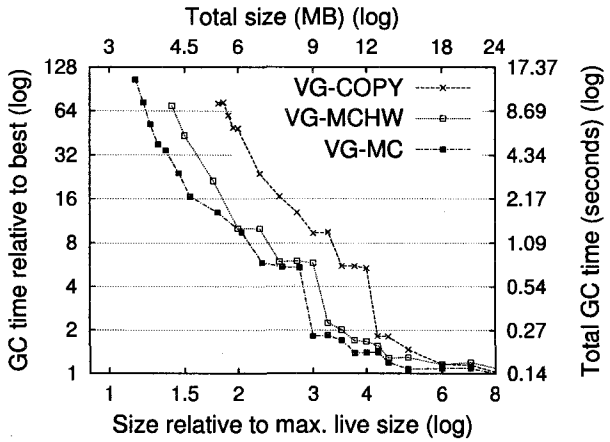


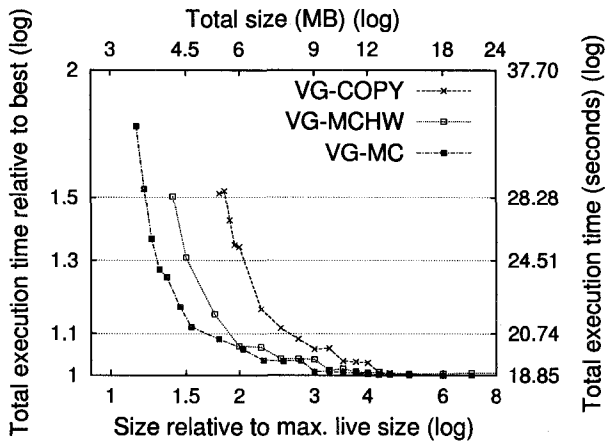
Figure 5.10. jess GC and total execution times relative to the best time for VG-MC, VG-MCHW, and VG-COPY.

larger than 4.5 times the live size. At eight times the live size, the total execution time for VG-COPY is 5% worse than that for VG-MC. The GC and total execution time curves for VG-MC and VG-COPY are very similar in shape to the curves for the copying costs, which shows that VG-MC obtains the performance improvement by reducing copying cost significantly.

VG-MCHW, which uses an extra word per object in the old generation, has a slightly higher space overhead than VG-MC. In tight heaps, it is significantly slower than VG-MC. When the space available is larger than twice the live size, it performs at most 7% worse



(a) mpegaudio GC times

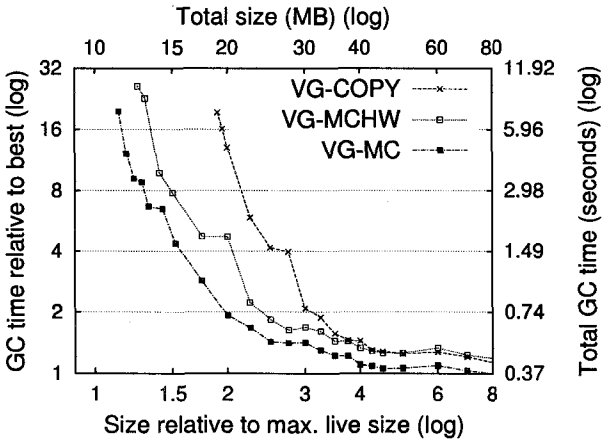


(b) mpegaudio total execution times

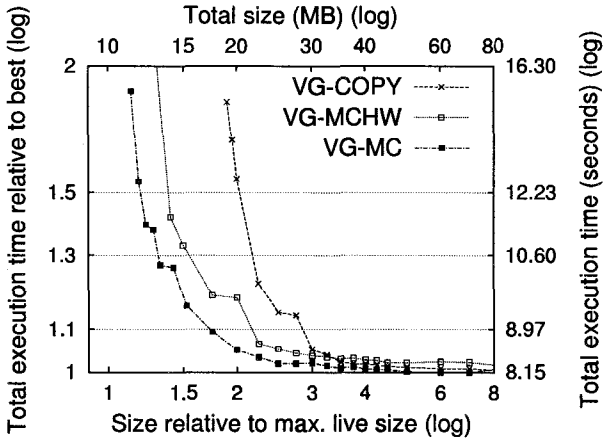
Figure 5.11. mpegaudio GC and total execution times relative to the best time for VG-MC, VG-MCHW, and VG-COPY.

than VG-MC, and is typically within a few percent of VG-MC. It outperforms VG-COPY at most heap sizes.

db: This benchmark builds a large linked list, which it repeatedly traverses during execution. This leads to locality issues. VG-MC performs better than VG-COPY until the space grows to about three times the maximum live size. The performance of all collectors degrades in large heaps (possibly due to locality effects on TLB misses). VG-MCHW is significantly slower than VG-MC in small heaps and about 5–8% worse than VG-COPY



(a) mtrt GC times

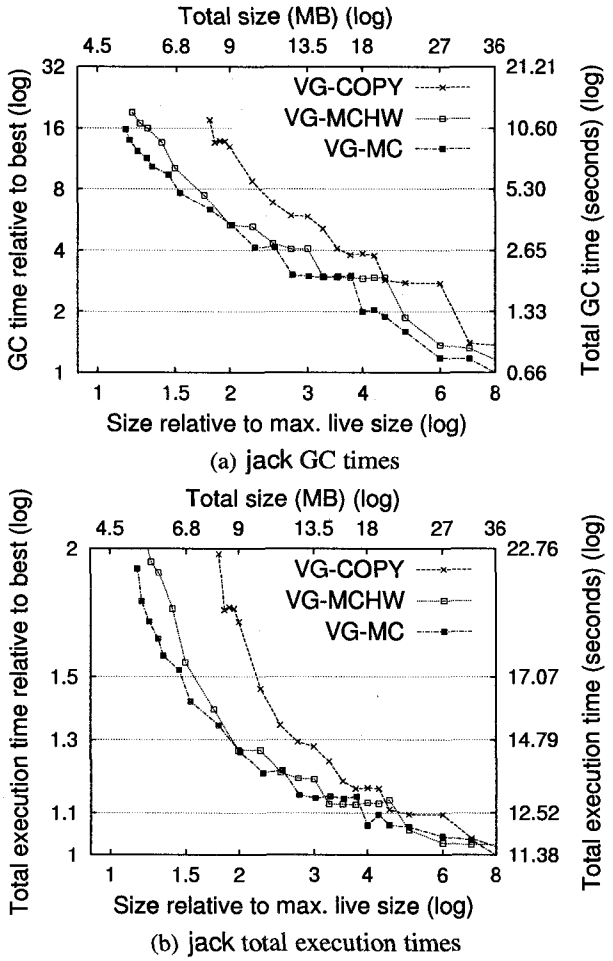


(b) mtrt total execution times

Figure 5.12. mtrt GC and total execution times relative to the best time for VG-MC, VG-MCHW, and VG-COPY.

and VG-MC in heaps that are 2.25 times the live size or larger. The higher execution times are possibly because of the effect the header word has on the cache.

jess, mpegaudio, mtrt, and jack: The results for these benchmarks are similar. VG-COPY starts off with total execution times about 25–50% worse than VG-MC. Its performance is typically within 10% of VG-MC only in heaps that are 2.5–3 times the live size or larger, and it performs almost as well as VG-MC at eight times the maximum live size. VG-MCHW generally performs within a few percent of VG-MC in heaps larger than twice the live size, and outperforms VG-COPY in heaps that are 3–4 times the live size or smaller.



(a) jack GC times

(b) jack total execution times

Figure 5.13. jack GC and total execution times relative to the best time for VG-MC, VG-MCHW, and VG-COPY.

pseudojbb: The total execution time for VG-COPY at 2.25 times the live size is 30% worse than that for VG-MC. VG-COPY comes within 10% of VG-MC in space that is 2.75 times the live size. VG-COPY performs almost as well as VG-MC in space that is five times the live size or larger. VG-MCHW is 5–20% slower than VG-MC in tight heaps. In heaps larger than twice the live size, its performance is about the same as VG-MC.

health: Here, the curve for VG-MC is noisy in the smaller heap sizes. This is because, for some of the small heap sizes, the remembered set overhead for VG-MC is very large

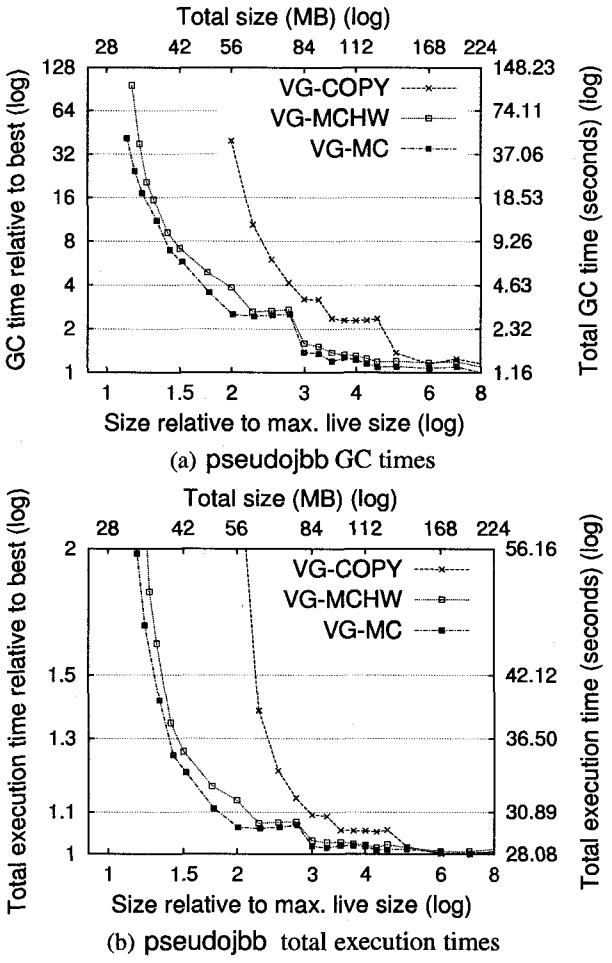
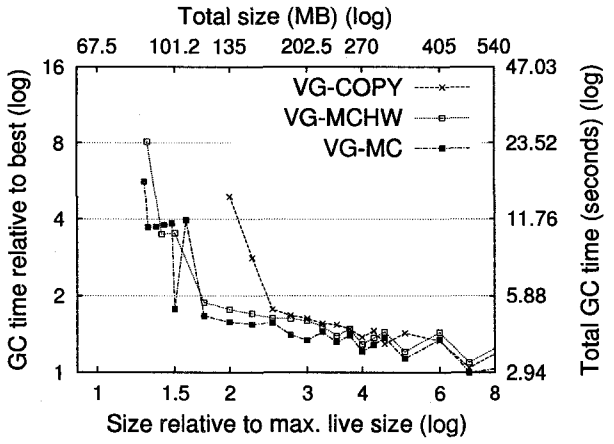


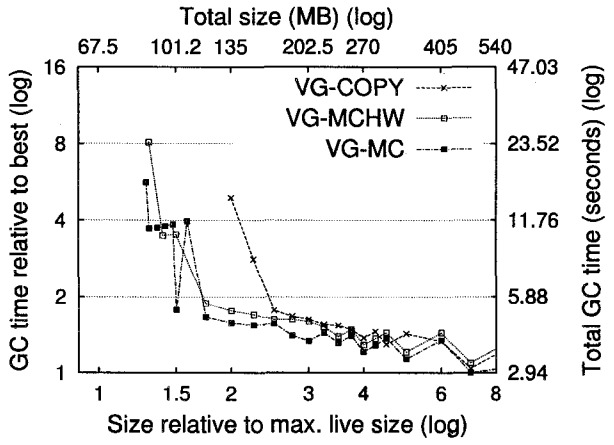
Figure 5.14. pseudojbb GC and total execution times relative to the best time for VG-MC, VG-MCHW, and VG-COPY.

(up to 14MB). This overhead causes configurations that use less heap space to use up a large amount of total space.

The execution time of VG-COPY converges much more quickly for health than for the other benchmarks. At about 2.25 times the live size, VG-COPY performs 5% better than VG-MC, and in heaps larger than three times the live size, VG-COPY performs as well as VG-MC. However, since the live size is quite large, the actual additional memory required by VG-COPY to perform as well as VG-MC is approximately 67MB. VG-MCHW performs about 3–5% worse than VG-MC and VG-COPY in heaps that are 2.5 times the



(a) health GC times



(b) health total execution times

Figure 5.15. health GC and total execution times relative to the best time for VG-MC, VG-MCHW, and VG-COPY.

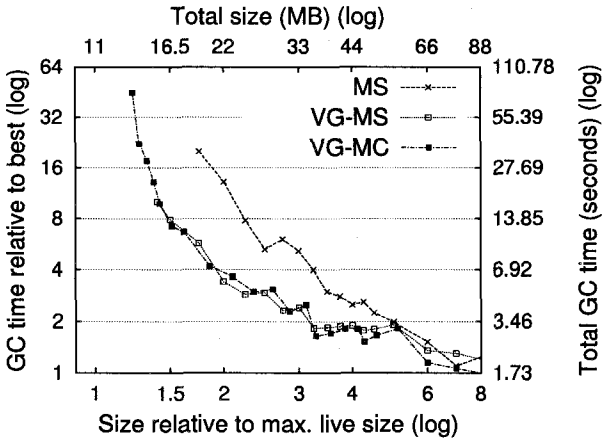
live size or larger, again possibly due to locality effects caused by the additional word per object.

Summary: VG-MC and VG-MCHW are capable of running in much smaller heaps than VG-COPY. The minimum space needed for VG-MC is between 1.12–1.25 times the maximum live size, and 5–10% higher for VG-MCHW. The space advantage is clear in the plots for pseudojbb and health. The minimum space required to run pseudojbb using VG-COPY is 25MB higher than the space required by MC. The additional space required by VG-COPY for health is 50MB.

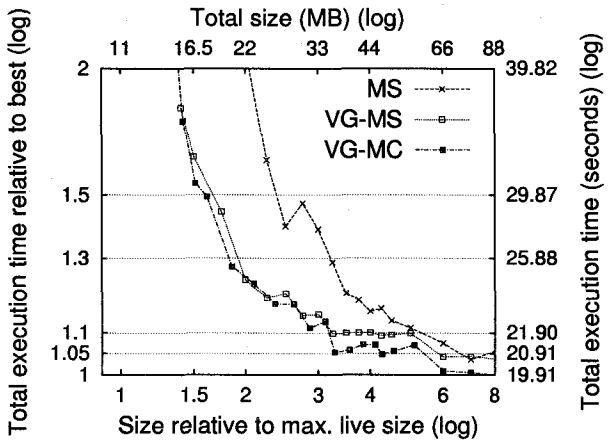
VG-MC consistently performs better than VG-COPY in small and moderate size heaps. For VG-COPY, garbage collection is up to eight times slower in small heaps and is 1.2–2 times slower in moderate size heaps. Overall execution time is significantly higher in small heaps, and is typically 5–10% higher in moderate size heaps. For five of the eight benchmarks, VG-MC performs within 6% of the best execution time, in space that is slightly larger than twice the maximum live size. VG-MC achieves the improved performance by utilizing the heap space better and performing less copying. Full collections for VG-MC take more time (per byte copied) than full collections for VG-COPY. However, VG-MC performs full collections much less frequently, thus reducing the copying and GC overheads considerably in small and moderate size heaps.

The performance of VG-MCHW is usually significantly worse than VG-MC in tight heaps. In heaps larger than twice the maximum live size, it typically performs less than 10% worse, and is usually within a few percent of VG-MC. It does not outperform VG-MC for a couple of reasons. First, the remembered set overhead for the benchmarks we experimented with is smaller than the extra word overhead for VG-MCHW. Second, VG-MCHW suffers from some locality effects because of the extra word added to each object in the old generation. VG-MCHW typically outperforms VG-COPY in small and moderate size heaps. However, VG-COPY performs better than VG-MCHW for db and health because of locality effects.

We also measured the performance of BG-MC, using a nursery that occupied at most 20% of the heap space. However, we did not present the results in this section. We found that BG-MC performs quite well when compared with VG-MC. For all benchmarks, execution times with BG-MC were less than 10% worse than with VG-MC at all heap sizes, and usually within a few percent of VG-MC while providing a bound on the nursery pause times.



(a) javac GC times

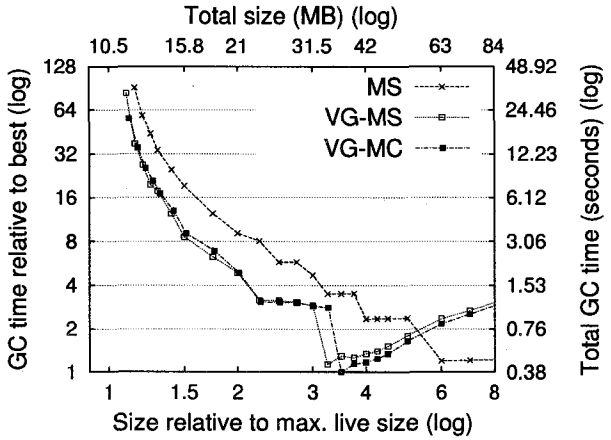


(b) javac total execution times

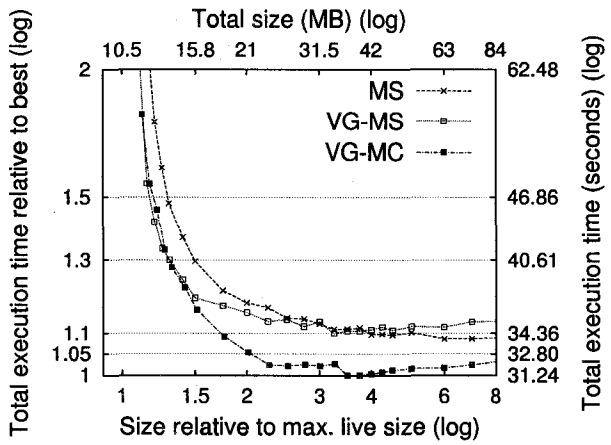
Figure 5.16. javac GC and total execution times relative to the best time for VG-MC, VG-MS, and MS.

5.6.5 Mark-Copy vs. Mark-Sweep

We now relate MC to mark-sweep collection. We compare the performance of a (non-generational) mark-sweep collector (MS) and a hybrid copying/mark-sweep generational collector (VG-MS) with VG-MC. MS collectors can run in much smaller heaps than standard copying collectors. In the best case, to be able to run a program, MS requires space that is slightly more than the maximum live size for a program. However, MS usually suffers from some degree of fragmentation, and the amount of additional space required depends on the degree of fragmentation. MC, on the other hand, does not suffer from frag-



(a) db GC times



(b) db total execution times

Figure 5.17. db GC and total execution times relative to the best time for VG-MC, VG-MS, and MS.

mentation. Like other copying collectors, it avoids fragmentation by regularly copying and compacting live data. However, MC does require space for copying and for remembered sets.

Figure 5.16–Figure 5.23 show GC times and total execution times relative to the best time for VG-MC, VG-MS, and MS for the eight benchmarks. The horizontal axis in all graphs represents total space and is drawn to a logarithmic scale. The vertical axis on the graphs represents relative times and is also drawn to a logarithmic scale. We now describe the performance of the three collectors for each of the eight benchmarks.

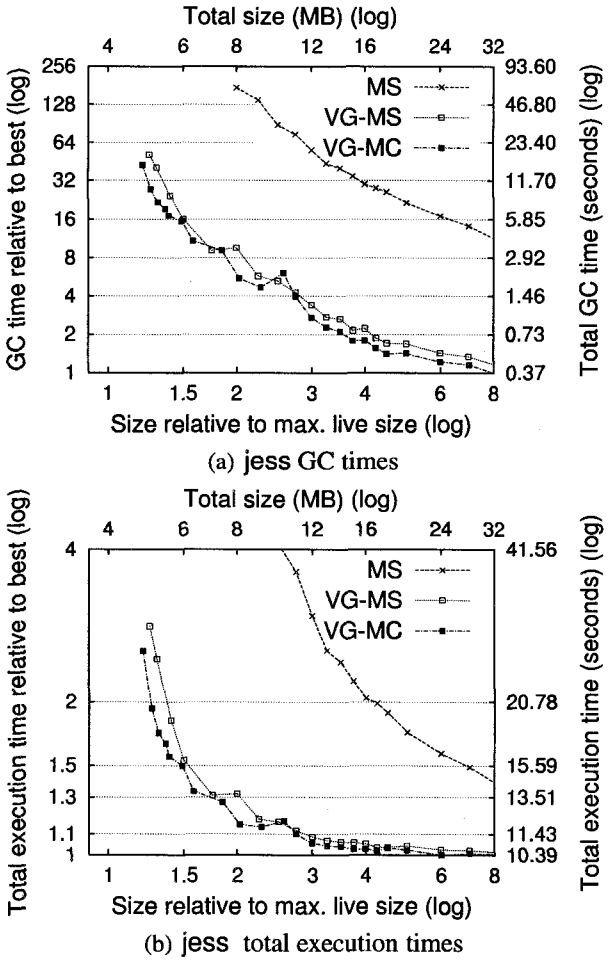
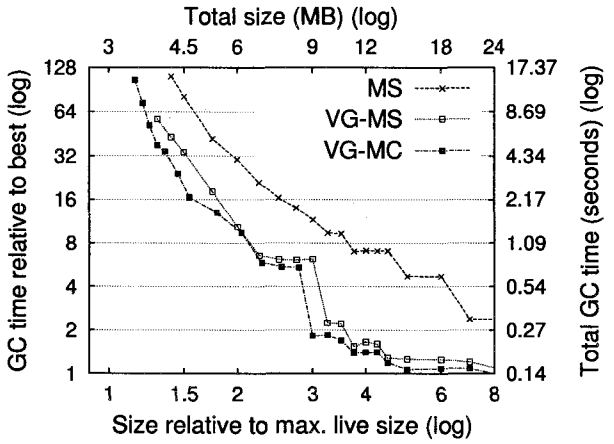
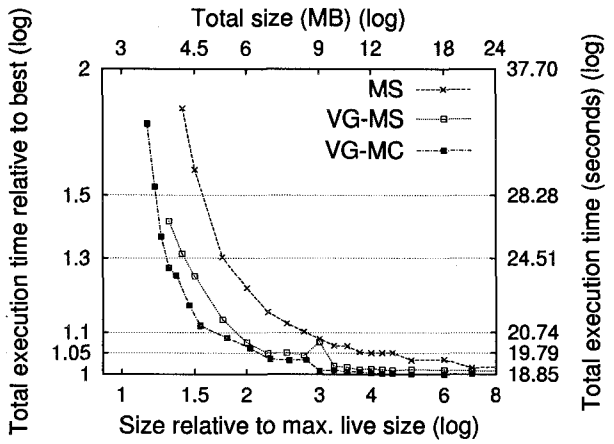


Figure 5.18. jess GC and total execution times relative to the best time for VG-MC, VG-MS, and MS.

javac: In heap sizes between 1.4–3 times the live size, VG-MS performs slightly worse than VG-MC at a few points due to slightly higher mutator (program, as opposed to collector) times. VG-MS performs 2–5% slower than VG-MC in heaps larger than 3.25 times the live size, due to slightly higher GC times and mutator times. Mutator times for VG-MS are higher possibly because VG-MC compacts the old generation data and hence achieves somewhat better locality (both collectors use an identical write barrier). MS is significantly slower in heaps smaller than three times the live size, and it eventually performs as well as VG-MS.



(a) mpegaudio GC times



(b) mpegaudio total execution times

Figure 5.19. mpegaudio GC and total execution times relative to the best time for VG-MC, VG-MS, and MS.

db: In heaps larger than 1.5 times the live size, VG-MC is 5–10% faster than VG-MS. This happens even though the GC times for VG-MC and VG-MS are approximately the same, which suggests that VG-MC might be compacting and ordering objects in a manner that improves locality significantly, thus lowering mutator time. MS has the lowest GC time in heaps larger than five times the live size, but it performs 5–10% worse than VG-MC.

jess, jack, mpegaudio: Total execution times for VG-MS are significantly higher than those for VG-MC in heaps smaller than 1.5 times the live size, and about 1–3% higher than those for VG-MC in heaps larger than 2.5 times the maximum live size. For jess

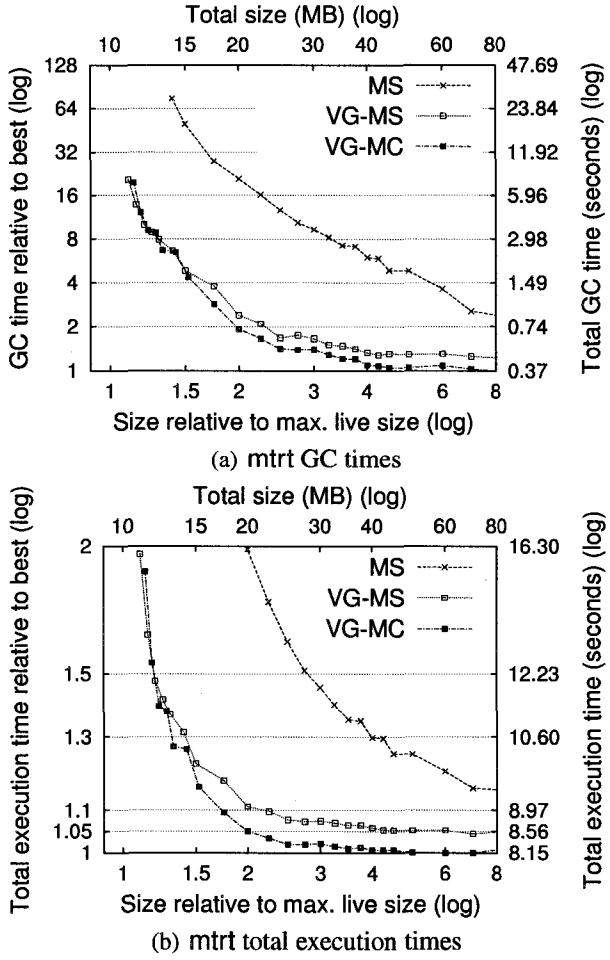
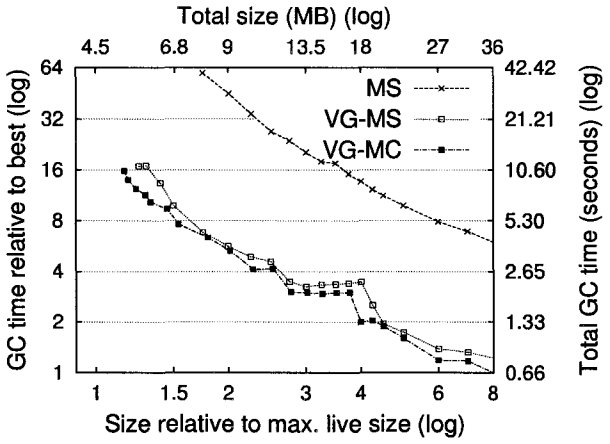


Figure 5.20. mtrt GC and total execution times relative to the best time for VG-MC, VG-MS, and MS.

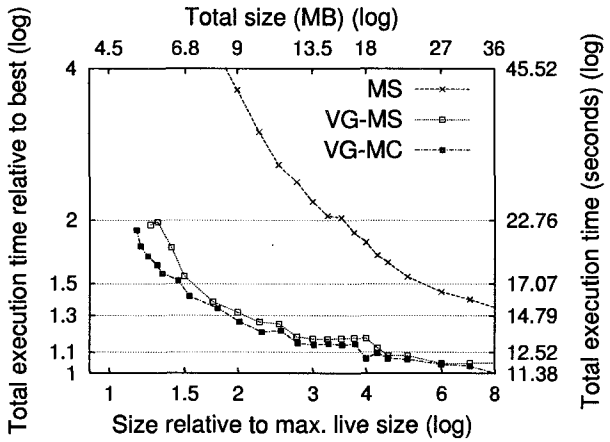
and jack, the performance of MS is significantly worse than the generational collectors at all heap sizes. For mpegaudio, the performance of MS is consistently within 5% of the generational collectors only in heaps larger than 3.5 times the live size.

mtrt: In heaps larger than 1.5 times the live size, VG-MS is 5% slower than VG-MC due to higher GC times and mutator times. MS performs significantly worse than both generational collectors at all heap sizes.

pseudojbb: VG-MS performs 3–5% worse than VG-MC at most heap sizes, again due to a combination of higher GC and mutator times. The performance of MS is worse than



(a) jack GC times

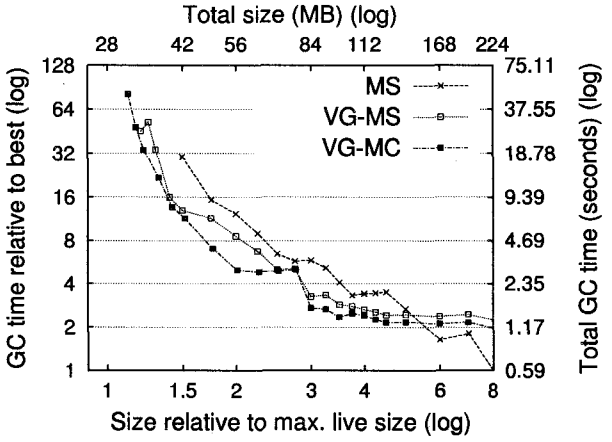


(b) jack total execution times

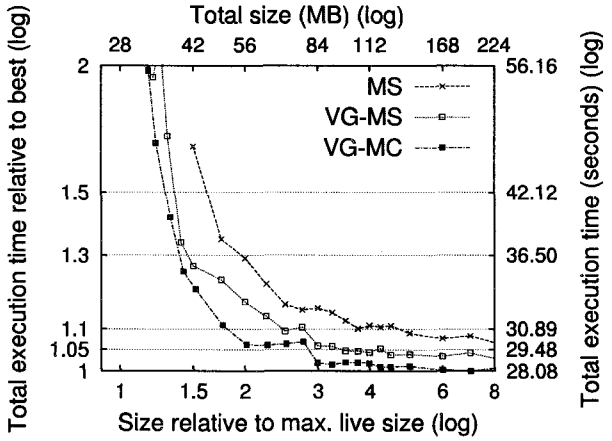
Figure 5.21. jack GC and total execution times relative to the best time for VG-MC, VG-MS, and MS.

the generational collectors at all heap sizes. Interestingly, MS has the lowest GC time in heaps larger than five times the live size. However, it is 6–8% slower than VG-MC due to higher mutator times.

health: VG-MC suffers from large remembered set overheads in small heaps making the curve noisy. It performs 5–10% worse than VG-MS in some small heaps. However, in heaps larger than 2.5 times the live size, VG-MC performs 5% better than VG-MS. MS has the lowest GC time of all three collectors in heaps larger than 4.5 times the live size, but, it has the highest execution times.



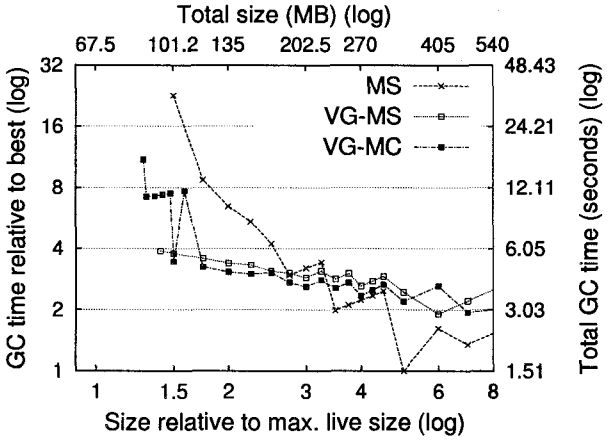
(a) pseudojbb GC times



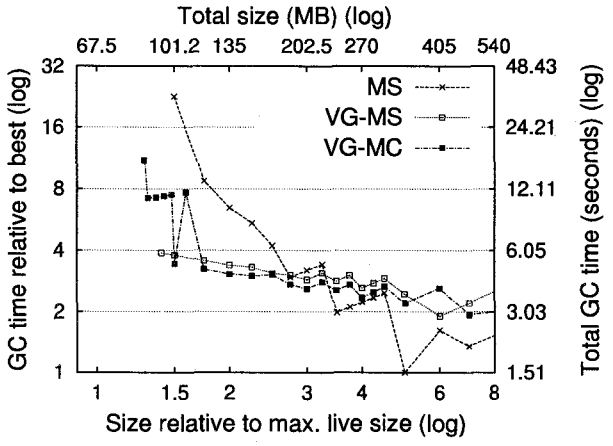
(b) pseudojbb total execution times

Figure 5.22. pseudojbb GC and total execution times relative to the best time for VG-MC, VG-MS, and MS.

Summary: VG-MC can run in space that is comparable to the minimum required by VG-MS. MS usually has a higher fragmentation overhead than VG-MS and thus requires somewhat larger heaps to be able to run successfully. Both VG-MC and VG-MS run significantly faster than MS for most benchmarks in small and moderate size heaps, affirming the generational hypothesis. The GC times for VG-MC are slightly lower or about the same as those for VG-MS. However, for javac, mtrt, pseudojbb, and health the total execution times for VG-MS are up to 5% higher than those for VG-MC. For db, execution times are 5–10% higher. Since both collectors use an identical write barrier, our guess is that



(a) health GC times



(b) health total execution times

Figure 5.23. health GC and total execution times relative to the best time for VG-MC, VG-MS, and MS.

this performance improvement happens because data in the old generation is compacted by VG-MC, because of which it achieves better cache locality and/or TLB performance. For health, VG-MC suffers from somewhat large remembered sets, which makes its performance worse than that of VG-MS in small heaps.

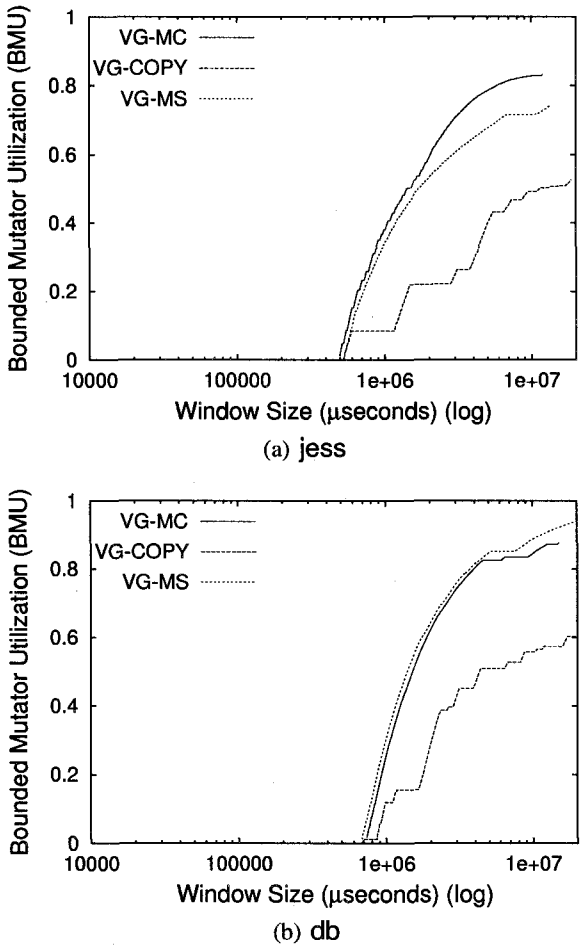


Figure 5.24. jess and db mutator utilization curves for VG-MC, VG-MS, and VG-COPY.

5.7 Pause Times

We now look at the pause time characteristics of MC. We use the methodology we describe in Chapter 3 to obtain BMU curves for the collectors. Figure 5.24–Figure 5.27 shows BMU curves for all benchmarks for heap sizes equal to two times the maximum live size. The x-intercept of the curves indicates the maximum pause time, and the asymptotic y-value indicates the fraction of the total time used for mutator execution (average utilization). The three curves in each graph are for VG-MC, VG-MS, and VG-COPY. Since it is difficult to factor out the write barrier cost, the curves actually represent utilization inclu-

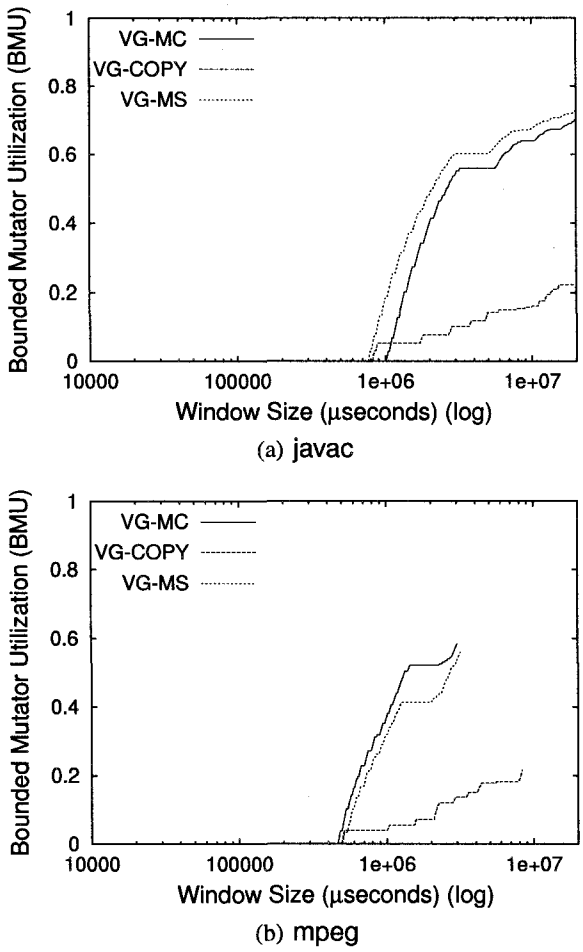


Figure 5.25. javac and mpegaudio mutator utilization curves for VG-MC, VG-MS, and VG-COPY.

sive of the write barrier. The real mutator utilization will be a little lower. These graphs also do not show the effects of locality on the overall performance. It should also be noted that these curves represent pause time distribution for a single run and the maximum pause times depend on the amount of live data in the heap at a collection point.

The maximum pause times for the three collectors are typically in the same range. Occasionally, the pause time for VG-MC is significantly higher. For javac, VG-MC has a maximum pause time that is about 30% higher than that for VG-MS. The pause time for VG-MC is about 22% higher than that of VG-COPY. For pseudojbb, MC has a maximum

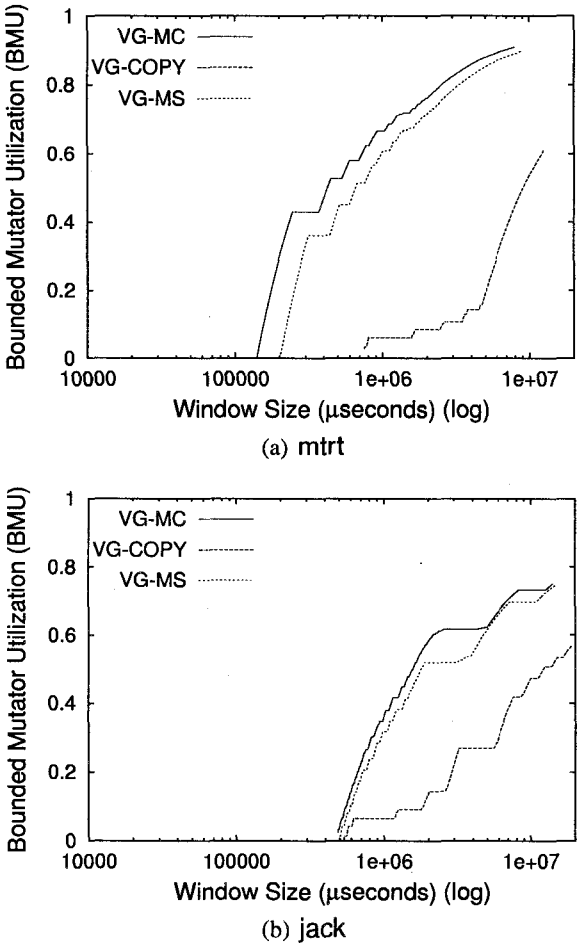


Figure 5.26. mtrt and jack mutator utilization curves for VG-MC, VG-MS, and VG-COPY.

pause time that is 37% and 6% higher than VG-MS and VG-COPY. The VG-MS pause time is lower because it does not copy data. The pause time for VG-COPY is lower in these cases both because its effective heap size is smaller and because it performs collection in a single phase.

Occasionally, as in the case of mtrt, the pause time for VG-MC is significantly lower. This could happen if a full collection for VG-MC occurs at a different point than VG-MS, when the amount of live data is lower. This is not typical, and given the same amount of live data, the pause time for VG-MC will normally be a little higher than that of VG-

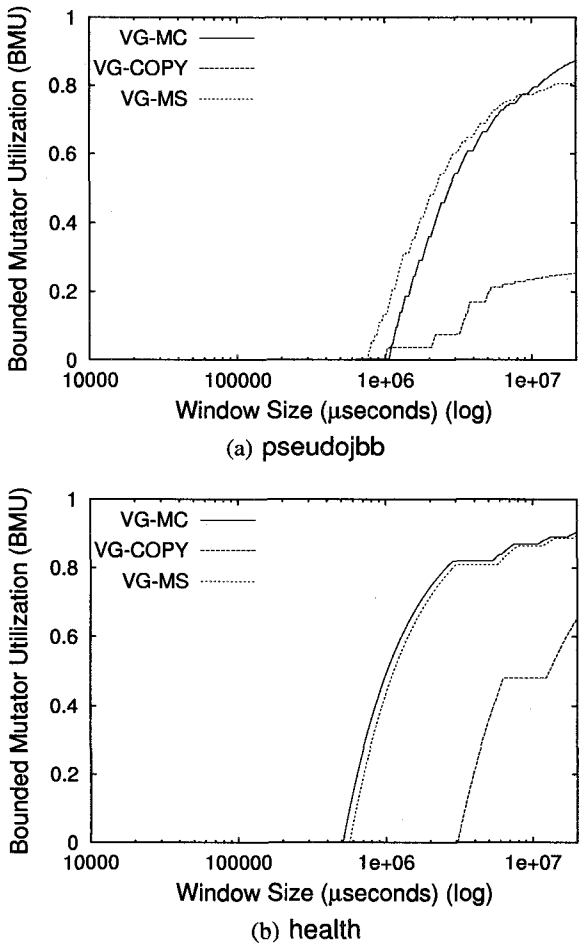


Figure 5.27. pseudojbb and health mutator utilization curves for VG-MC, VG-MS, and VG-COPY.

MS. The overall throughput for VG-MC and VG-MS are close, and that of VG-COPY is significantly lower because of the tight heap space.

5.8 Conclusions

Mark-copy (MC) significantly reduces space requirements compared to standard copying collectors. We showed that MC collectors can run in space that is as small as 1.12–1.25 times the maximum live size of a program, while standard copying collectors usually require two times the maximum live size. Additionally, MC provides high throughput even

in heaps much smaller than twice the maximum live size. This increases the range of applications that can use copying garbage collection.

We saw that MC collectors outperform generational copying collectors in equal sized heaps; they do so by significantly reducing the frequency of full heap collections and thus the amount of copying. Overall performance with MC is significantly better in tight heaps, and is typically around 5–10% better in moderate size heaps.

We also compared MC against a mark-sweep (MS) collector and a hybrid copying/mark-sweep generational collector (VG-MS); we showed that MC can run in heaps that are comparable in size to the minimum for MS and VG-MS. For most benchmarks, MC is significantly faster than MS in small and moderate size heaps. When compared with VG-MS, MC improves total execution time by about 5% for some benchmarks. The improvements are caused both by lower GC times and by better locality.

However, while MC can provide good throughput and space utilization, it still suffers from long pauses. Both mark and copy phases of the collector scan every live object in the heap and proceed non-incrementally. The maximum pause time for MC can be higher than a generational copying collector for two reasons. First, the usable heap when using MC is bigger. Second, MC uses two phases while a generational copying collector has only one. When compared with VG-MS, MC can have a longer pause since it copies and compacts data while VG-MS does not.

CHAPTER 6

MC^2: MARK-COPY WITH SHORT PAUSES

While Mark-Copy (MC) can provide high throughput while running in limited amounts of memory, it has some limitations. Primarily, it suffers from long pause times. This makes it unacceptable for many interactive applications that run on memory-constrained devices. We describe in this chapter MC^2, a collector that extends MC, and provides good throughput and short pause times while running in constrained memory.

This chapter is organized as follows. We first describe the limitations of MC. Next we describe how MC^2 works, and explain how it overcomes each of the limitations of MC. We then present details of the MC^2 implementation in MMTk. Finally, we compare the performance of MC^2 with MOS and three generational non-incremental collectors.

6.1 Limitations of Mark-Copy

We describe here limitations of MC that could prevent its use for applications on memory-constrained devices. First, it maps and unmaps pages in windows while performing the copying phase. It thus depends on the presence of virtual memory hardware, which may not always be available on handheld devices. Second, the collector always copies all live data in the old generation. This is clearly not efficient when the old generation contains large amounts of long-lived data. Third, and perhaps most significantly for many applications, the marking and copying phases can be time-consuming, leading to large pauses. Finally, although the collector usually has low space overheads, it occasionally suffers from large remembered sets. In the worst case, the remembered set size can grow to be as large as the heap.

6.2 MC^2

The new collector, *memory-constrained copying* (MC^2), uses the basic MC technique to partition the heap: there is a nursery and an old generation divided into a number of windows. We do not use MC^2 as a non-generational collector for the same reason we did not use MOS as a non-generational collector, as we explained in Section 4.1.8. A full collection marks live objects in the heap, followed by a copy phase that copies and compacts live data. However, MC^2 overcomes the cited limitations of MC. We describe in successive sections below features of the collector that allow it to obtain high throughput and low pause times in bounded space.

6.2.1 Old Generation Layout

As previously stated, MC^2 divides the heap into equal size windows. It requires that the address space within each window be contiguous. However, the windows themselves need not occupy contiguous memory locations: MC^2 maintains a free list of windows and assigns a new window to the old generation whenever a previously-assigned window cannot accommodate an object about to be copied. Unlike MC, which uses object addresses to determine the relative location of objects in the heap, MC^2 uses indirect addressing to determine this information in order to decouple actual addresses from logical window numbers. It uses a byte array, indexed by high order bits of addresses, to indicate the logical window number for each window.

While this indirection adds a small cost to the mark phase, it has several advantages. First, it removes the need to map and unmap memory every time MC^2 evacuates a window, in order to maintain MC^2's space bound. Second, the indirection removes the need to copy data out of every window; we can assign the window a new logical number and it will be as if the data had been copied. This is important for programs with large amounts of long-lived data. Third, it allows the collector to change the order of collection of windows across multiple collections. We describe the details of these features in Section 6.2.3.

MC^2 also differentiates between *physical windows* and *collection windows*. It divides the heap into a number of small windows (typically 100) called physical windows. MC^2 maintains remembered sets for each of these windows. A collection window defines the maximum amount of live data that MC^2 normally collects during a copying increment. Collection windows are usually larger than physical windows. This scheme allows MC^2 to copy smaller amounts of data occasionally (e.g., when a physical window contains highly referenced objects). In the following sections we use the term "window" to refer to a physical window.

6.2.2 Incremental Marking

MC marks the heap when the free space drops to a single window. While this gives good throughput, it tends to be disruptive: when MC performs a full collection, the pause caused by marking can be long. In order to minimize the pauses caused by marking, MC^2 triggers the mark phase sooner than MC, and spreads out the marking work by interleaving it with nursery allocation.

After every nursery collection, MC^2 checks the occupancy of the old generation. If the occupancy exceeds a predefined threshold (typically 80%), MC^2 triggers a full collection and starts the mark phase. It first associates a bit map with each window currently in the old generation (the collector has previously reserved space in the heap for the bit maps). Marking uses these bit maps to mark reachable objects, and to check if an object has already been marked. Apart from the benefit to locality of marking, the bit maps also serve other purposes, described in Section 6.2.3.

MC^2 then assigns a logical address to each window. Marking uses these addresses to determine the relative positions of objects in the old generation. MC^2 marks data only in windows that are in the old generation when a full collection is triggered. It considers any data promoted into the old generation during the mark phase to be live, and collects it only in the next full collection. After MC^2 assigns addresses to the windows, it cannot alter the

order in which they will be collected for the duration of the current collection. Finally, MC^2 allocates a new window in the old generation, into which it performs subsequent allocation.

After triggering the mark phase, MC^2 allows nursery allocation to resume. Every time a 32KB block in the nursery fills up, MC^2 marks a small portion of the heap.[1] In order to compute the volume of data that needs to be marked in each mark increment, MC^2 maintains an average of the nursery survival rate (NSR). It computes the mark increment size using the following formulae:

$$
\begin{aligned}
numMarkIncrements &= availSpace/(NSR * nurserySize) \\
markIncrementSize &= totalBytesToMark/numMarkIncrements \\
totalBytesToMark &= totalBytesToMark - markIncrementSize
\end{aligned}
$$

MC^2 initializes *totalBytesToMark* to the sum of the size of the windows being marked, because in the worst case all the data in the windows may be live. If the heap occupancy reaches a predefined *copy threshold* (typically 95% occupancy) during the mark phase, MC^2 will perform all remaining marking work without allowing further allocation.

MC^2 maintains the state of marking in a work queue, specifically a list of the objects marked but not yet scanned. When it succeeds in emptying that list, marking is complete.

6.2.2.1 Write Barrier

A problem with incremental marking is that the mutator modifies objects in the heap while the collector is marking them. The collector can thus miss an object that is actually reachable. Using the tri-color invariant [30], we can classify each object in the heap as *white* (unmarked), *gray* (marked but not scanned), or *black* (marked and scanned). The problem arises when the mutator changes a black object to refer to a white object, destroys

[1]We chose 32KB as a good compromise in terms of interrupting mutator work and allocation often enough, but not too often. This value determines the incrementality of marking.

```
writeBarrier(srcObject, srcSlot, tgtObject){
  if (srcObject not in nursery) {
    if (tgtObject in nursery)
      record srcSlot in nursery remset
    else if (tgtObject in old generation) {
      if (srcObject is not mutated) {
        set mutated bit in srcObject header
        record srcObject in mutated object list
      }
    }
  }
}
```

Figure 6.1. MC^2 write barrier.

the original pointer to the white object, and no other pointer to the white object exists. An example demonstrating this problem can be found in Section 2.5.2.

Figure 6.1 shows pseudo-code for the write barrier that MC^2 uses. The write barrier serves two purposes. First, it records pointer stores that point from outside the nursery to objects within the nursery (in order to be able to locate live nursery objects during a nursery collection). Second, it uses an incremental update technique to record mutations to objects in the old generation. When an object mutation occurs, and the target is an old generation object, the write barrier checks if the source object is already recorded as mutated. If so, MC^2 ignores the pointer store. If not, it records the object as mutated. When MC^2 performs a mark increment, it first processes the mutated objects, possibly adding additional objects to the list of those needing to be scanned. After processing the mutated objects, it resumes regular marking.

6.2.3 Incremental Copying

When MC performs a full collection, it copies data out of all windows, without allowing any allocation in between. MC^2 overcomes this long pause by spreading the copying work over multiple nursery collections.

6.2.3.1 High Occupancy Windows

At the end of the mark phase, MC^2 knows the volume of data marked in each window. At the start of the copy phase, MC^2 uses this information to classify the windows. MC^2 uses a *mostly-copying* technique. It classifies any window that has a large volume of marked data (e.g., $> 98\%$) as a *high occupancy* window, and does not copy data out of the window.

6.2.3.2 Copying Data

After MC^2 identifies the high occupancy windows, it resumes nursery allocation. At every subsequent nursery collection, it piggybacks the processing of one old generation *group*. MC^2 groups windows based on the amount of live data they contain. Each group consists of one or more old generation windows, with the condition that the total amount of marked data in a group is less than or equal to the size of a collection window. Since MC^2 scans high-occupancy windows sequentially, and the processing does not involve copying and updating slots, it treats a high-occupancy window as equivalent to copying and updating slots for half a window of live data. MC^2 also allows one to specify a limit on the total number of remembered set entries in a group. If the addition of a window to a group causes the group remembered set size to exceed the limit, MC^2 places the window in the next group.

When MC^2 processes an old generation group, it first checks if the group contains high occupancy windows. If so, MC^2 uses the mark bit map for the windows to locate marked objects within them. It scans these objects to find slots that point into windows that have not yet been copied, and adds the slots to remembered sets for those windows. It then logically moves the windows into to-space. In subsequent collections, MC^2 places these high occupancy windows at the end of the list of windows to be collected. If they still contain large volumes of live data, they do not even have to be scanned, and the copy phase can terminate when it reaches these windows. This technique turns out to be especially helpful

for SPECjvm98 benchmarks such as db and pseudojbb, which allocate large amounts of permanent data.

If a window group contains objects that need to be copied, MC^2 locates marked objects in the window by scanning the root set and remembered set entries. It copies these objects into free windows using a regular Cheney scan [23]. While scanning to-space objects, MC^2 adds slots that reference objects in uncopied windows to the corresponding remembered set.

As in the mark phase, the write barrier keeps track of mutations to old generation objects during the copy phase. It records mutated objects at most once. At every copy phase collection, MC^2 updates mutated slots that reference objects in windows being copied, and adds mutated slots that reference objects in uncopied windows to the corresponding remembered sets.

Example: Figure 6.2 shows an example of the copy phase of the collector in action. The example assumes that the physical window and collection window sizes are identical, i.e., MC^2 will copy at most one physical window worth of data in each pass. Figure 6.2(a) shows the old generation layout before the copy phase starts. The old generation contains four marked windows (W0–W3). MC^2 classifies W1 and W3 as high occupancy since they contain 100% and 98% marked data respectively and places them in separate groups. It also places W0 and W2 in separate groups. In the first copy pass (Figure 6.2(b)), MC^2 copies data from the nursery and W0 into W4. It then adds W0 to the list of free windows. In the second pass (Figure 6.2(c)), MC^2 scans objects in W1 and adds W1 to the end of to-space. It copies nursery survivors into W4 and W0. In the third pass (Figure 6.2(d)), MC^2 copies objects out of the nursery and W2 into windows W0 and W5. It then adds W2 to the list of free windows. At this point, the only remaining window, W3, is high occupancy, so MC^2 adds it to the end of to-space and ends the copy phase.

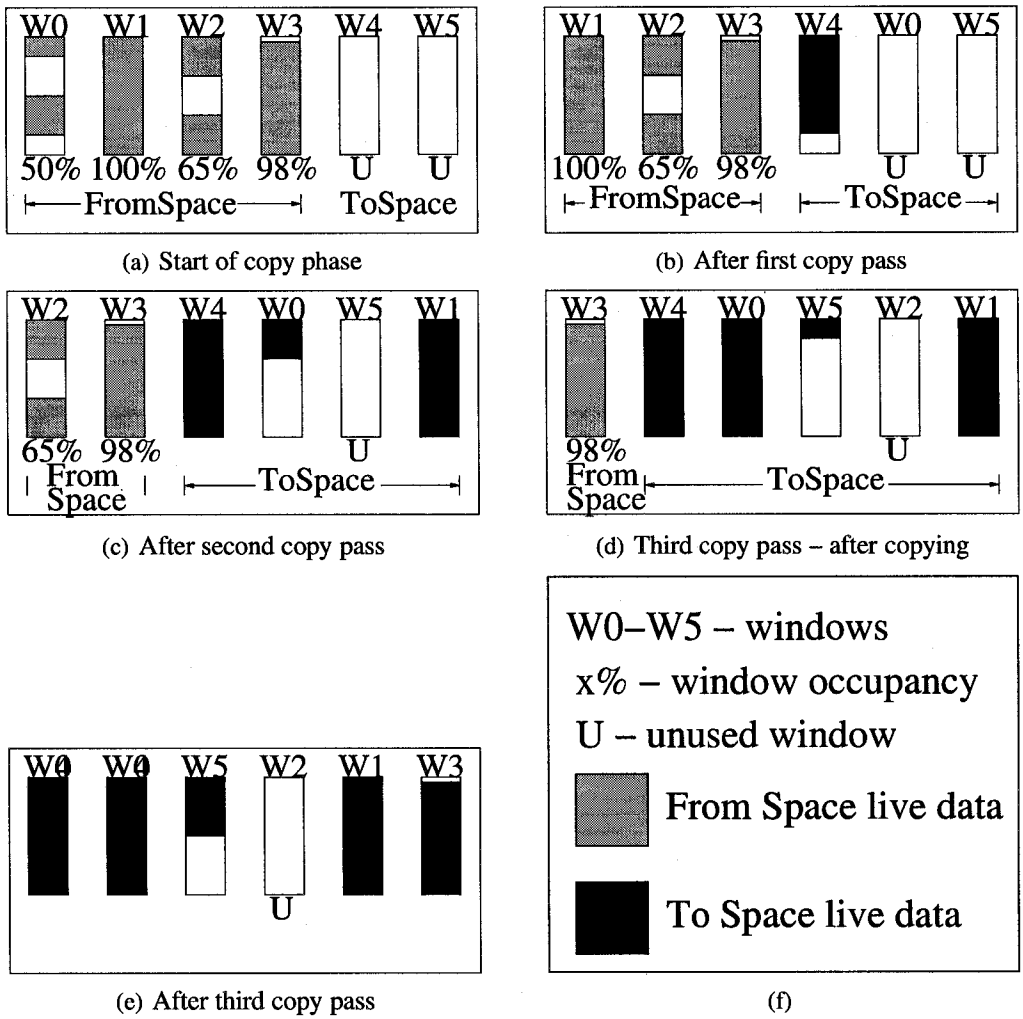


Figure 6.2. Stages in the copy phase for MC^2.

6.2.4 Bounding Space Overhead

The remembered sets created during MC collection are typically small (less than 5% of the heap space). However, they occasionally grow to be large. There are two typical causes for such large remembered sets: *popular* objects (objects heavily referenced in a program), and large arrays of pointers. MC^2 uses two strategies to reduce the remembered set overhead, both of which involve coarsening remembered set entries.

6.2.4.1 Large Remembered Sets

MC^2 sets a limit on the total amount of space that remembered sets can occupy. When the space overhead reaches the limit, it coarsens remembered sets starting from the largest, until the space overhead is below the limit. Also, when the size of a single remembered set exceeds a predefined limit, MC^2 coarsens that particular remembered set. This coarsening involves converting the remembered set representation from a sequential store buffer (SSB) to a card table. Our usual SSB representation for a remembered set for a window W is a sequential list of addresses of slots containing pointers into W. Whenever we detect a reference into W that needs recording, we simply add it to the end of this list. The list may contain duplicates as well as stale entries (slots that used to point into W but no longer do). The physical representation of the list is a chain of fixed sized chunks of memory, where each chunk is an array of slot addresses.

We use the scheme described by Azagury et al. [5] to manage card tables, as we did for MOS. MC^2 divides every window into cards of size 128 bytes. For every window, it creates a card table that contains a byte for each card in the window. The byte corresponding to a card stores the logical address of the first window containing an object that is referenced from the card. The collector also maintains a window level summary table (that stores the lowest logical address contained in each window's card table) .

When it is time to collect a target window TW whose remembered set is a card table the collector proceeds as follows. It first scans the summary table to identify windows that contain references into TW. For each source window SW that contains a reference into TW, the collector scans the card table of SW to find cards within SW that contain references into TW. It then scans every object in each of these cards, to find the exact slots that point into TW. If a particular slot does not point into TW, and it points to a window whose remembered set is a card table, MC^2 records the window number of the slot reference. It uses this information to update the card table with a new value at the end of the scan.

The process of converting the remembered set representation for a window TW is straightforward. MC^2 scans the SSB sequentially, and for every recorded slot, it checks if the contents still refer to an object in TW. If so, it finds the source window and card corresponding to the object that contains the slot. If the current entry for the source window card is larger than the logical address of TW, MC^2 overwrites the entry with the lower logical address.

With a card size of 128 bytes, the size of the card table is about 0.78% of the total heap space. MC^2 ensures that the total sum of the space taken by the SSB remembered sets and the card table does not exceed a specified limit. For example, if the limit is set at 5% of the total space, MC^2 starts coarsening remembered sets when their occupancy reaches 4.2% of the total space. (Another possibility is to use only a card table, and not use SSBs. We briefly compare the performance of MC^2 with both remembering schemes in the results section.)

The bounded space overhead can come at a run-time cost. Setting a byte in a card table is more expensive than inserting into a sequential store buffer. Also, scanning a card table and objects in a card to identify slots that point into a target window is more expensive than simply traversing a buffer sequentially. However, large remembered sets are relatively rare, and when MC^2 creates a large remembered set, we use a technique described below to prevent the situation from recurring.

6.2.4.2 Popular Objects

MC^2 identifies popular objects during the process of converting an SSB to a card table. While performing the conversion, it uses a byte array to maintain a count of the number of references to each object in the window. Since the collector always reserves at least one window worth of free space, there is always enough space for the byte array without increasing our space budget. As MC^2 scans the SSB, it calculates the offset of each refer-

enced object, and increments the corresponding entry in the byte array. When the count for any object exceeds 100, MC^2 marks it as popular.

During the copy phase, MC^2 treats windows containing popular objects specially. Any window that has a coarsened remembered set (and hence popular objects), is placed in a separate collection group, even if the amount of live data in the window is less than one collection window. This helps reduce the amount of data copied and focuses collection effort on updating the large number of references into the window, hence lowering pause time.

MC^2 copies popular objects into a special window. It treats objects in this window as immortal and does not maintain a remembered set for this window in subsequent collections. However, if the occupancy of the window grows to be high, MC^2 can add it back to the list of collected windows. So, if popular objects exist, MC^2 may take a slight hit on pause time occasionally. However, it ensures that these objects are isolated and do not cause another disruption. MC^2 cannot avoid this problem completely. To be able to do so, it would have to know in advance (at the point when a popular object is copied out of the nursery), that a large number of references will be created to the object.

6.2.4.3 Large Reference Arrays

MC^2 divides large reference arrays (larger than 8 KB) into 128-byte cards. Rather than store every array slot in the remembered sets, MC^2 always uses a card table for these large objects. When MC^2 allocates a large reference array, it allocates additional space (one byte for every 128 bytes in the object) at the end of the object. These bytes function as a per-object card table in the exact same manner as the technique described in the previous section. MC^2 also adds a word to the header of the object to store the logical address of the first window referenced from the object. The card scan and update is identical to the scheme described in the previous section.

6.2.5 Worst Case Performance

It is important to note that MC^2 is a soft real time collector, and cannot provide hard guarantees on maximum pause time and CPU utilization. In the worst case MC^2 behaves like a non-incremental collector. For instance, if an application allocates a very big object immediately after the mark phase commences, causing the heap occupancy to cross the heap size limit, MC^2 must perform all marking and copying non-incrementally in order to reclaim space to satisfy the allocation. If such a situation arises, then the program will experience a long pause. MC^2 does not assume any knowledge about peak allocation rates of the running program and provides a best effort service based on statistics it can compute as the program runs. Programs with very high allocation rates during a full collection will experience longer pauses than programs with lower allocation rates.

As described in the previous section, popular objects can also cause MC^2 to exhibit poor performance occasionally. In the worst case, every word on the heap points to a single object, and moving the object requires updating the entire heap, causing a long pause. However, these situations are rare and MC^2 provides good pause time behavior under most conditions.

6.3 Implementation Details

Our MC^2 implementation divides the virtual address space for the old generation into a number of fixed size *frames*. A frame is the largest contiguous chunk of memory into which MC^2 performs allocation. We use a frame size of 8MB (the heap sizes for our benchmarks vary from 6MB–120MB). Each frame accommodates one old generation physical window. Windows are usually smaller than a frame, and the portion of the address space within a frame that is not occupied by a window is left unused (unmapped). The frames are power-of-two aligned, so we need to perform only a single shift to find the physical window number for an old generation object during GC; we use the physical window number as an index into a byte array to obtain the logical address of the window, as previously described.

```
private final void slowPathWriteBarrier(
  VM_Address srcObj, VM_Address src,
  VM_Address tgtObj)
  throws VM_PragmaNoInline {
  if (tgtObj.LT(NURSERY_START)) {
    VM_Word status =
      VM_Interface.readAvailableBitsWord(srcObj);
    if (status.and(Header.MUTATE_BIT_MASK).isZero()) {
      VM_Interface.writeAvailableBitsWord(srcObj,
        status.or(Header.MUTATE_BIT_MASK));
      matureWriteBuf.insert(srcObj);
    }
  } else
    nurseryWriteBuf.insertIL(src);
}

private final void writeBarrier(
  VM_Address srcObj, VM_Address src,
  VM_Address tgtObj)
  throws VM_PragmaInline {
  if (src.LT(NURSERY_START) && tgtObj.GE(LOS_START))
    slowPathWriteBarrier(srcObj, src, tgtObj);
  VM_Magic.setMemoryAddress(src, tgtObj);
}
```

Figure 6.3. MC^2 mutator write barrier code.

```
private static final void gcWriteBarrier(
  int objWindow, int objLogical,
  VM_Address slot, VM_Address srcObj)
   throws VM_PragmaInline {
   int slotWindow = getWindow(slot);
   if ((slotWindow != objWindow) &&
       getLogical(slotWindow) > objLogical) {
     int matureObjWindow = objWindow-FIRST_MATURE_FRAME;
     // check if window uses an SSB or card table. A high
     // reference window uses a card table
     if ((getStatus(matureObjWindow) != HIGH_REF_WINDOW))
       // insert into a buffer
       remsetInsert(matureObjWindow, slot);
     else
       matureCardMark(matureObjWindow, srcObj, slot);
   }
}

 private static final void matureCardMark(
   int tgtWindow, VM_Address srcObj,
   VM_Address slot)
   throws VM_PragmaInline {
   VM_Address srcAddr = VM_Interface.refToAddress(srcObj);
   int srcWindow = getMatureWindow(srcAddr);
   int offset = srcAddr.diff(windowToAddress(srcWindow)).toInt();
   cardMapSet(cardTable.get(srcWindow), offset, tgtWindow);
   int tgtLogical = getMatureLogical(tgtWindow);
   int minLogical = getMinLogical(srcWindow);
   if (tgtLogical < minLogical)
     setMinLogical(srcWindow, tgtLogical);
 }

 private static final void cardMapSet(
   VM_Address table, int offset, int tgtWindow)
   throws VM_PragmaInline {
   int cardOffset = offset>>>LOG_MATURE_CARD_SIZE;
   VM_Address tgtAddress = table.add(cardOffset);
   int curTgtLogical = VM_Magic.getByteAtOffset(
     VM_Magic.addressAsByteArray(tgtAddress), 0);
   int tgtWindowLogical = getMatureLogical(tgtWindow);
   if (tgtWindowLogical < curTgtLogical) {
     VM_Magic.setByteAtOffset(
       VM_Magic.addressAsByteArray(tgtAddress), 0,
                                (byte)tgtWindowLogical);
   }
 }
```

Figure 6.4. MC^2 GC write barrier code.

Benchmark	Live Size (KB)	Total Allocation (MB)
_202_jess	5872	319
_209_db	12800	93
_213_javac	13364	280
_227_mtrt	12788	163
_228_jack	6184	279
pseudojbb	30488	384

Table 6.1. Benchmarks used for the evaluation of MC^2.

Parameter	Value
Max. nursery size	1MB
Collection threshold	80%
Mark frequency	32KB
Num. collection windows	30
Num. physical windows	100
Popularity threshold	100
High Occupancy threshold	98%

Table 6.2. Parameters used for the evaluation of MC^2. The nursery size is common to all collectors. All other parameters are specific to MC^2.

Each window has an associated remembered set, implemented using a sequential store buffer.

Our implementation tags each remembered set entry to indicate whether the entry belongs to a scalar object or an array (this information is required to locate the object containing a slot while converting remembered set representations). If the overall metadata size grows close to 4.2% of the heap, MC^2 converts the largest remembered sets to card tables (we use a card table with a granularity of 128 bytes and place a 5% limit on remembered set size). In our current implementation, we do not coarsen boot image slots since the number of entries is small and limited.

Figure 6.3 shows the mutator write barrier for MC^2. It is identical to the MOS mutator write barrier. MC^2 processes mutated objects at GC time using the write barrier shown in Figure 6.4.

6.4 Experimental Results

We compare space overheads, GC times, total execution times, and pause times for five collectors: bounded-nursery MC^2 (BG-MC^2), a bounded-nursery MOS collector (BG-MOS), a bounded-nursery generational copying collector (BG-COPY), a bounded-nursery generational mark-sweep collector (BG-MS), and a bounded-nursery generational mark-(sweep)-compact collector (BG-MSC). All results are for configurations using a nursery with a maximum size of 1MB and a minimum size of 256KB. BG-MC^2 uses 100 physical windows and 30 collection windows in the old generation. All copying collectors use separate regions for code and data. Table 6.1 shows the benchmarks we use, their live sizes, and the total allocation they perform. We run all experiments on JikesRVM 2.3.2.

Table 6.2 shows the parameters we use for the evaluation of MC^2. All parameters were chosen experimentally and were tuned to a nursery size of 1MB. We choose a nursery size of 1MB as a compromise between larger nurseries which could cause an long pauses, and smaller nurseries which increase execution times and nursery survival rates. The collection threshold needs to to be large enough so that the collector does not collect too much live data. High thresholds (85% or greater) on the other hand tend to give the collector little time to mark the heap before running out of space. Too many collection windows (40 or greater) cause the collector to run out of space before completing the copy phase, and fewer windows (20 or lower) tend to have higher maximum pause times.

We first look at the space overheads of BG-MC^2. We then compare execution times and pause times of BG-MC^2 with those of BG-COPY, BG-MS and BG-MSC and BG-MOS. Finally we look at the pause time distributions for BG-MC^2 and the mutator utilization curves.

6.4.1 MC^2 Space Overheads

BG-MC^2 implemented using an SSB remembered set incurs the following space overheads for its metadata:

Benchmark	Rel. heap size			
	1.5	1.75	2.25	2.5
_202_jess	6.1	4.9	4.2	2.3
_209_db	0.8	0.7	0.4	0.4
_213_javac	6.9	7.4	5.5	5.2
_227_mtrt	3.0	2.3	1.5	0.4
_228_jack	5.6	3.1	2.8	1.9
pseudojbb	2.8	1.7	1.3	1.0

Table 6.3. Remembered set size (without coarsening) as percent of heap size, for BG-MC^2 using 100 physical windows and 30 collection windows.

Benchmark	Rel. heap size			
	1.5	1.75	2.25	2.5
_202_jess	0.61	0.52	0.41	0.36
_209_db	0.45	0.38	0.3	0.26
_213_javac	0.73	0.59	0.46	0.36
_227_mtrt	0.17	0.16	0.11	–
_228_jack	0.59	0.38	0.37	0.27
pseudojbb	0.19	0.11	0.08	0.08

Table 6.4. Maximum mark queue size as percent of heap size, for BG-MC^2 using 100 physical windows and 30 collection windows.

Benchmark	NP	NC	NW
_202_jess	164	263	637
_209_db	0	245	288
_213_javac	312	502	911
_227_mtrt	0	0	159
_228_jack	0	261	610
pseudojbb	0	0	442

Table 6.5. Coarsening and popular object statistics for MC^2 in a heap that is 1.5 times the live size.

- 3.12% of the total heap space for a bit map that is used both to mark objects and locate objects during a window scan.

Benchmark	NP	NC	NW
_202_jess	41	1	135
_209_db	0	0	76
_213_javac	100	3	460
_227_mtrt	0	0	101
_228_jack	0	0	279
pseudojbb	0	0	303

Table 6.6. Coarsening and popular object statistics for MC^2 in a heap that is 2 times the live size.

- At most 5% of the total heap space for window remembered sets. A card table for 128 byte cards occupies 0.78% of the total space. BG-MC^2 coarsens SSB remembered sets when their total size reaches 4.2% of the total space. Table 6.3 shows the maximum remembered set overhead (without coarsening) for the six benchmarks, for heap sizes ranging from 1.5–2.5 times the program live size. To compute the overhead, we run BG-MC^2 without accounting for remembered set space, and compute the maximum space occupied by the remembered sets. jess, jack, javac require some coarsening in small heaps. The overheads shown here do not include pointers from large and immortal objects into the windows. We always use a card table for these objects.

- Work Queue overhead. Our current implementation does not bound the total work queue overhead, but we account for the space taken up by the queue, which is small. Table 6.4 shows the maximum queue overheads for our benchmark suite.

Table 6.5 and Table 6.6 show the number of popular objects and windows coarsened for each of the benchmarks in heaps that are 1.5 and 2 times the live size. In a heap that is 1.5 times the live size, a high number of windows are coarsened for jess, db, javac, and jack. This is because of two reasons. First the fraction of space occupied by remembered sets is high in smaller heaps. Additionally, the collector can reach the copy threshold before it completes marking in small heaps. This forces the collector to coarsen all remembered sets

before completing the marking phase, so that it does not run out of space while marking. In a heap that is twice the live size, few windows need coarsening, less than 1% for jess and javac. The number of popular objects identified also depends on the amount of coarsening performed. Thus, the number of popular objects identified in smaller heaps is larger.

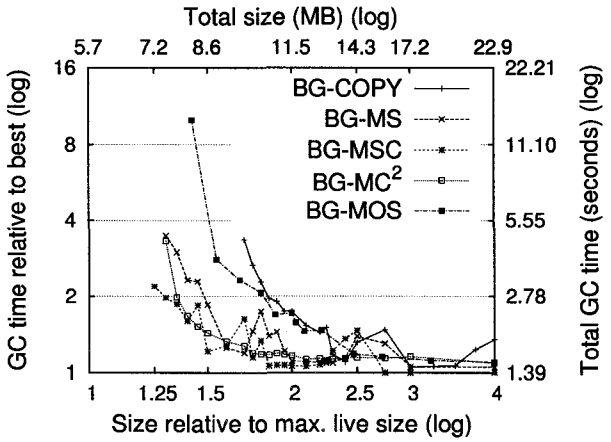
These numbers represent the overhead for uni-directional remembered sets. Our collector does not use bi-directional remembered sets, because these could cause a higher space overhead. Additionally, we cannot use our current coarsening technique with bi-directional remembered sets. Bi-directional sets, however, provide MC^2 with the option of changing the order of copying even after the marking phase has completed.

6.4.2 Execution Times and Pause Times

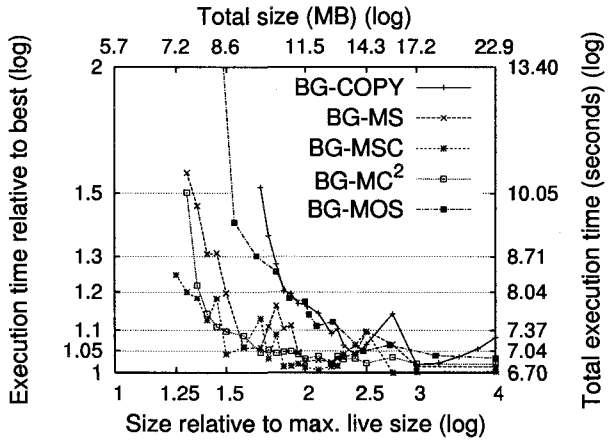
Figure 6.5–Figure 6.10 show GC times and total execution times for all collectors relative to the best time. The x-axis on all graphs represents the heap size relative to the maximum live size, and the y-axis represents the relative times. For BG-MOS, we plot the configuration that has the best overall execution times, BG-MOS.4.20.85.

BG-MSC almost always has the best execution times among the collectors. It always outperforms BG-MS in small and moderate size heaps. BG-MC^2 also generally performs better than BG-MS in small and moderate size heaps. The exceptions are db and javac, where BG-MC^2 performs worse than BG-MS in small heaps. In large heaps, BG-MC^2 and BG-MS have equivalent performance for all benchmarks apart from db and pseudojbb. BG-MC^2 performs significantly better for these benchmarks.

BG-MC^2 has lower GC times than BG-MS for jess and jack. For all other benchmarks, GC times for BG-MC^2 are slightly worse than those for BG-MS. However, the overall performance of BG-MC^2 is usually better due to improved locality. Figure 6.11 shows db and pseudojbb data TLB misses for BG-MC^2, BG-MS, and BG-MSC, and demonstrates the effect that locality has on the performance of the three collectors. The number of data TLB misses for BG-MS is a factor of two higher than that for BG-MC^2 and BG-MSC,



(a) jess GC times

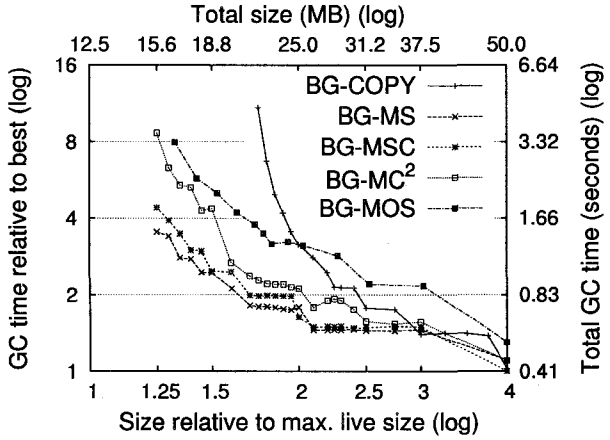


(b) jess total execution times

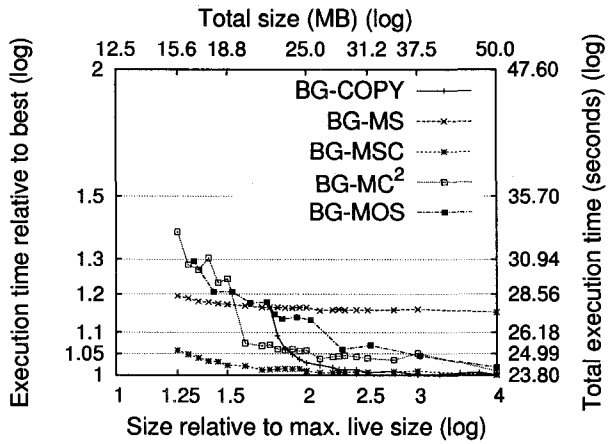
Figure 6.5. jess GC and total execution times for BG-MC^2, BG-MOS, BG-MS, BG-MSC, and BG-COPY.

explaining the difference in performance we observe in the execution time curves for the benchmarks.

For all benchmarks, the performance for BG-MC^2 is usually within 5% of BG-MSC in heaps that are 1.5–1.8 times the program live size or larger. javac again is an exception and BG-MC^2 is within 5-6% of BG-MSC only in heaps that are twice the live size or larger. BG-MSC performs better than the other collectors for a couple of reasons. First, the GC cost for BG-MSC is almost always the lowest. Second, the collector preserves allocation order, which yields better locality in these programs and thus lower mutator times.



(a) db GC times



(b) db total execution times

Figure 6.6. db GC and total execution times for BG-MC^2, BG-MOS, BG-MS, BG-MSC, and BG-COPY.

Both db and pseudojbb contain significant amounts of permanent data, which is advantageous to BG-MC^2. Since it uses a mostly-copying technique, it does not copy large portions of data. It compacts a smaller portion of the heap, which contains transient data. In spite of copying little data, BG-MC^2 does not have lower GC times than BG-MSC, because BG-MSC has the same property. It also does not repeatedly copy permanent data, since permanent data flows toward the lower end of the heap, and BG-MSC does not move any live data at the start of the heap.

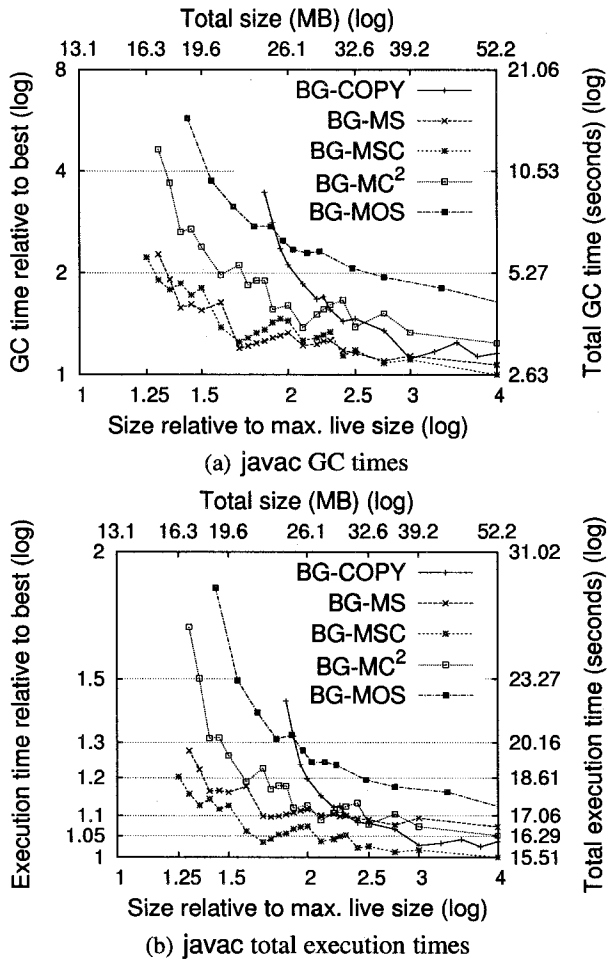


Figure 6.7. javac GC and total execution times for BG-MC^2, BG-MOS, BG-MS, BG-MSC, and BG-COPY.

BG-MOS generally has the highest execution times in small heaps, considerably higher than BG-MC^2. An exception is db for which it has slightly better execution times than BG-MC^2 at a few points. In moderate size heaps, BG-MOS.4.20.85 has execution times about 5–10% higher than BG-MC^2. In large heaps, BG-MOS does about the same when it is not performing full heap collections, and is a few percent slower in most other cases. The exception is javac where execution times are about 7–10% higher even in large heaps.

When compared with BG-COPY, BG-MC^2 is usually better in small and moderate size heaps. Only for db, primarily because of write barrier costs, it performs a few percent

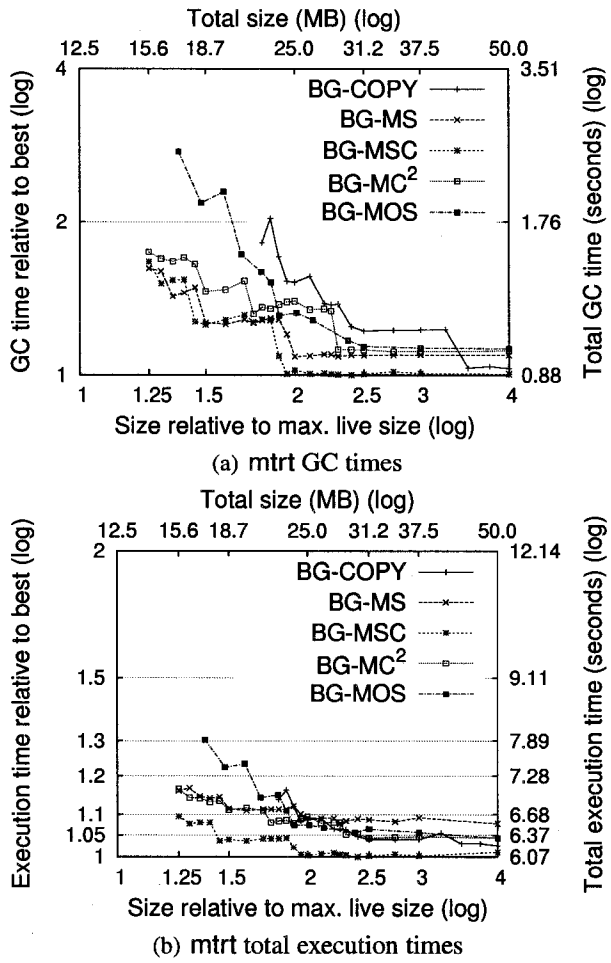


Figure 6.8. mtrt GC and total execution times for BG-MC^2, BG-MOS, BG-MS, BG-MSC, and BG-COPY.

worse. In larger heaps BG-MC^2 performs 3–4% worse than BG-COPY only for db and javac

Table 6.7 shows the maximum pause times for the collectors and relative execution times for BG-MOS, BG-MS, and BG-MSC collectors in a heap that is 1.8 times the program's maximum live size. This is the smallest heap in which BG-MC^2 provides a combination of high throughput and low pause times for 5 of the 6 benchmarks, and is the typical overhead to be expected for good performance.

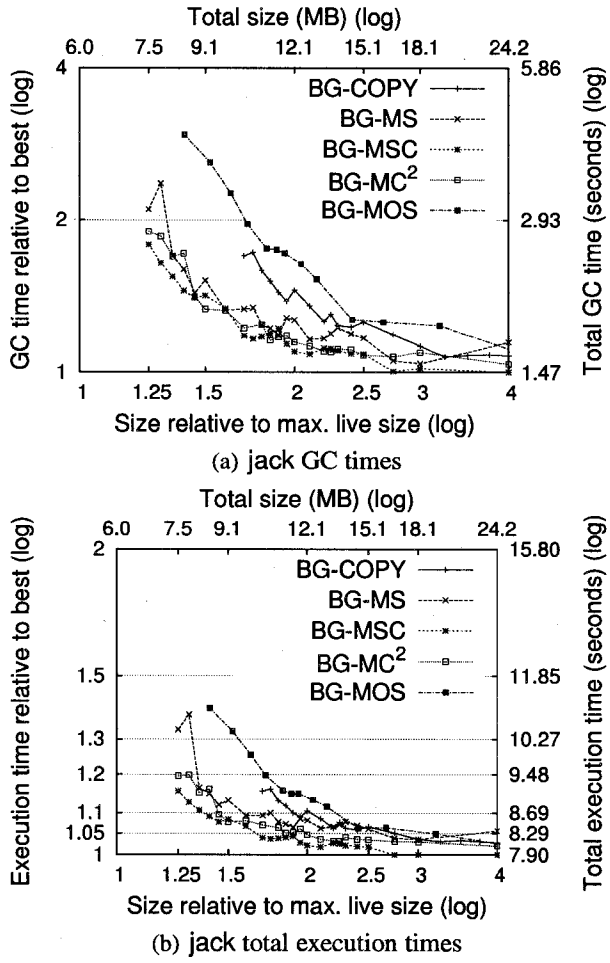
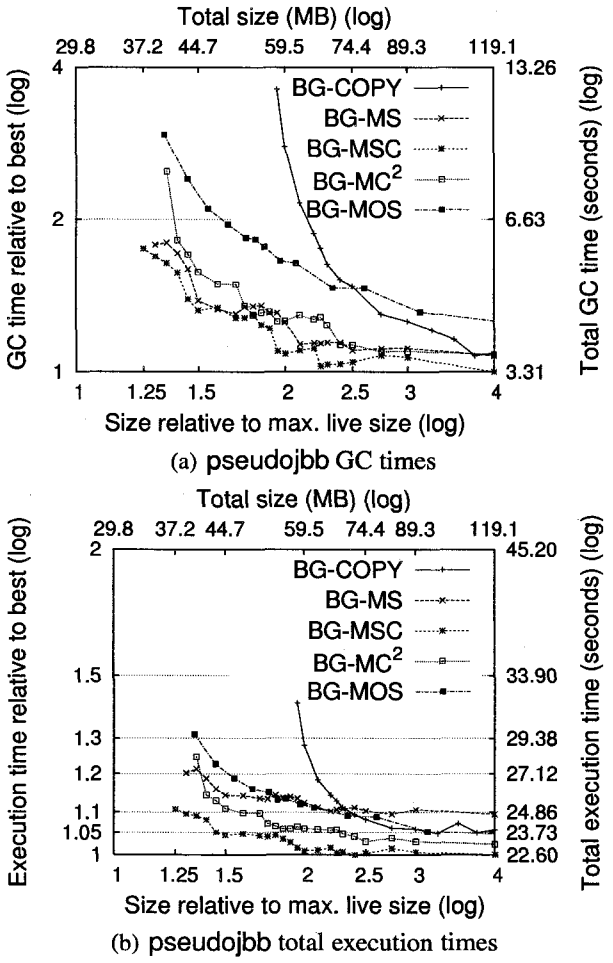


Figure 6.9. jack GC and total execution times for BG-MC^2, BG-MOS, BG-MS, BG-MSC, and BG-COPY.

The data in Table 6.7 show that, for all benchmarks, the maximum pause times for BG-MC^2 are significantly lower than those for BG-MS and BG-MSC, even in a tight heap. The geometric mean of the pause times for BG-MC^2 is 27.5ms. The mean pause time for BG-MS is 109.1, a factor of 4 higher, with a mean performance degradation of 4%. BG-MC^2 is 3% slower then BG-MSC on average but its pause time is a factor of 6 lower. The only benchmark for which the performance of BG-MC^2 is significantly worse is javac. BG-MC^2 requires a slightly larger heap for javac and is usually about 5% worse than BG-MSC in heaps that are twice the live size or larger.



(a) pseudojbb GC times

(b) pseudojbb total execution times

Figure 6.10. pseudojbb GC and total execution times for BG-MC^2, BG-MOS, BG-MS, BG-MSC, and BG-COPY.

Table 6.8 shows minimum, maximum, and geometric means of the pause times across all benchmarks for BG-MC^2 in heaps ranging from 1.4–3 times the live size. It also shows geometric means of pause times and average relative execution times for BG-COPY, BG-MS, and BG-MSC. Table 6.9 shows a comparison between BG-MC^2 and the best BG-MOS configurations in heaps between 1.4–3 times the live size. At each heap size, we only consider those benchmarks for which BG-MSC performs at least one full collection.

In a heap that is 1.4–1.5 times the live size, BG-MC^2 is 8–9% slower than BG-MSC on average, at most 1% slower than BG-MSC on average, and it has a maximum pause time

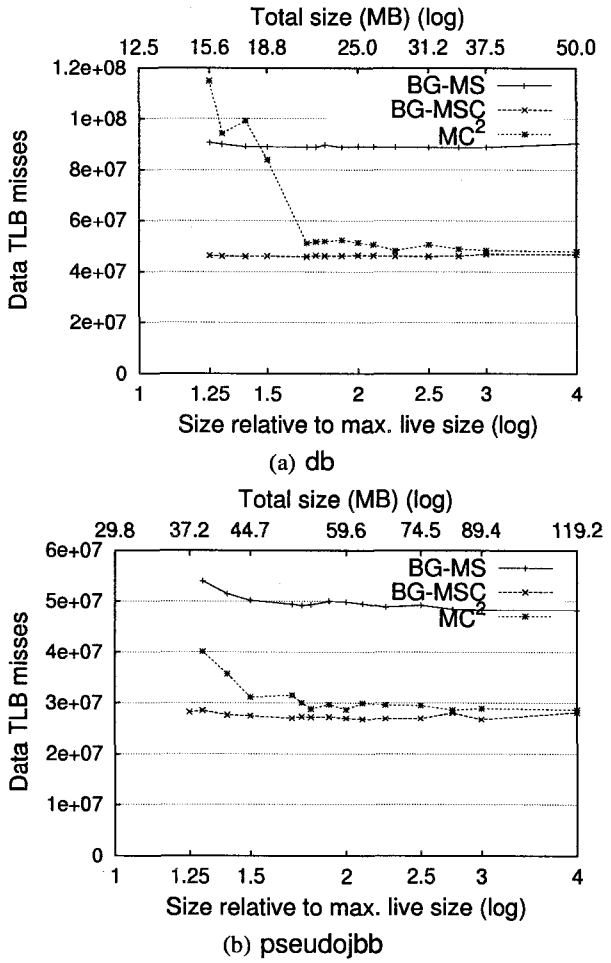


Figure 6.11. db and pseudojbb data TLB misses for BG-MC^2, BG-MS, and BG-MSC.

of 119.47ms (this long pause occurs for javac because a few copying passes are clustered together). Maximum pause times for BG-MS and BG-MSC are about a factor of 2 and 3 higher, and average pause times are a factor of 3 and 3.5 higher. At 1.6 times the live size, only db (84.3ms) has a pause time over 50ms. Average execution times for BG-MC^2 are within 5% of BG-MSC.

In heaps between 1.8–2.5 times the live size, the maximum pause time for BG-MC^2 is under 50ms for all benchmarks. The average pause time varies from 27–30ms. BG-MC^2 is 3–5% faster than BG-MS and 1–4% slower than BG-MSC on average. Average pause

Benchmark	BG-MC^2 MPT	BG-MOS		BG-MS		BG-MSC	
		MPT	ET	MPT	ET	MPT	ET
_202_jess	17.3	18.8	1.04	54.8	1.11	65.7	1.04
_209_db	20.6	68.7	1.05	124.9	1.10	199.4	0.96
_213_javac	43.1	48.6	1.12	171.9	0.93	310.5	0.89
_227_mtrt	31.4	38.9	1.04	138.1	1.03	226.8	0.96
_228_jack	21.0	68.5	1.07	60.8	1.01	93.8	0.98
pseudojbb	42.2	49.7	1.09	170.6	1.07	315.2	0.98
Geo. Mean	27.5	45.0	1.07	109.1	1.04	173.5	0.97

Table 6.7. Maximum Pause Times (MPT, all in milliseconds) and execution times relative to BG-MC^2 (ET) for BG-MC^2, BG-MS, and BG-MSC, in a heap that is 1.8 times the maximum live size. BG-MSC and BG-MC^2 use separate data and code regions. All collectors use a 1MB nursery and BG-MC^2 uses 100 physical windows and 30 collection windows.

HS	BG-MC^2			BG-COPY			BG-MS			BG-MSC		
	LPT	HPT	APT	HPT	APT	ET	HPT	APT	ET	HPT	APT	ET
1.40	26	106	50	–	–	–	191	111	0.99	330	172	0.91
1.50	17	97	37	–	–	–	184	107	1.00	310	168	0.92
1.60	17	84	34	–	–	–	182	107	1.00	316	180	0.95
1.80	17	43	28	–	–	–	172	109	1.04	315	174	0.97
2.00	16	43	28	205	106	1.08	177	96	1.03	316	151	0.96
2.25	18	49	30	201	112	1.03	183	95	1.03	310	154	0.97
2.50	17	47	28	204	112	1.02	182	100	1.06	320	165	0.99
3.00	27	53	40	198	112	0.99	183	112	1.05	318	185	0.96

Table 6.8. Min. (LPT), max. (HPT), and geometric mean of max. (APT) pause times (milliseconds) across all benchmarks for BG-MC^2 in heaps (HS) 1.4–3 times the live size. Max. pause time, geometric mean of max. pause times (milliseconds), and geometric mean of execution times relative to BG-MC^2 across all benchmarks for BG-COPY, BG-MS, and BG-MSC. For each heap size, we consider only benchmarks that cause invocation of at least one full collection for all collectors. BG-COPY, BG-MSC, and BG-MC^2 use separate data and code regions. All collectors use a 1MB nursery and BG-MC^2 uses 100 physical windows and 30 collection windows.

times for BG-MS are a factor of 3–4 higher, and average pause times for BG-MSC are a factor of 5–6 higher. At 3 times the live size, the average pause times for all collectors increase, because jess, which has the lowest pause times, is not considered, since BG-MSC does not perform any full collections. When compared with BG-COPY, BG-MC^2 performs

	BG-MC^2			BG-MOS			
HS	LPT	HPT	APT	LPT	HPT	APT	ET
1.40	26.2	105.9	49.7	51.0	213.9	100.6	1.25
1.50	16.8	97.0	36.8	36.4	150.8	78.3	1.21
1.60	17.3	84.3	33.7	51.6	63.6	53.3	1.12
1.80	17.3	43.1	27.5	18.8	68.7	45.0	1.07
2.00	16.4	43.3	27.6	18.1	49.3	33.1	1.08
2.25	18.4	49.1	30.4	15.3	47.1	30.8	1.06
2.50	17.0	46.6	27.8	15.4	48.0	32.6	1.08
3.00	26.6	53.3	40.2	20.5	58.0	43.8	1.05

Table 6.9. Min. (LPT), max. (HPT), and geometric mean of max. (APT) pause times (milliseconds) across all benchmarks for BG-MC^2 and BG-MOS (best configurations) in heaps (HS) 1.4–3 times the live size. Geometric mean of execution times relative to BG-MC^2 across all benchmarks for BG-MOS. For each heap size, we consider only benchmarks that cause invocation of at least one full collection for BG-MSC. Both collectors use a 1MB nursery and BG-MC^2 uses 100 physical windows and 30 collection windows.

between 2–8% better in moderate size heaps, and is about 1% worse at 3 times the live size because of its higher write barrier cost.

The best pause time configurations for BG-MOS have execution times that are considerably higher in small heaps, 12–25% on average. From 1.8–2.5 times the live size, average execution times are 6–8% higher, and at 3 times the live size, average execution time is 5% higher. Average pause times are about the same in heaps that are twice the live size or larger. The high pause time for BG-MOS is better at a single point (1.6 times live size).

We also compared the performance of BG-MC^2 (using SSBs with coarsening) with a version of BG-MC^2 that uses only a card table. In very small heaps (1.25 times the live size or lower) the card table technique usually performs better than the SSB technique. This is because it has a smaller space overhead (always 0.78%) than the SSB collector. However, in heaps that are larger than 1.25 times the live size, the SSB technique performs slightly better or about the same as the card technique. The pause times for the SSB collector also tend to be slightly lower than the card technique since it knows the exact location of slots pointing into windows. The card technique must examine every pointer in every object

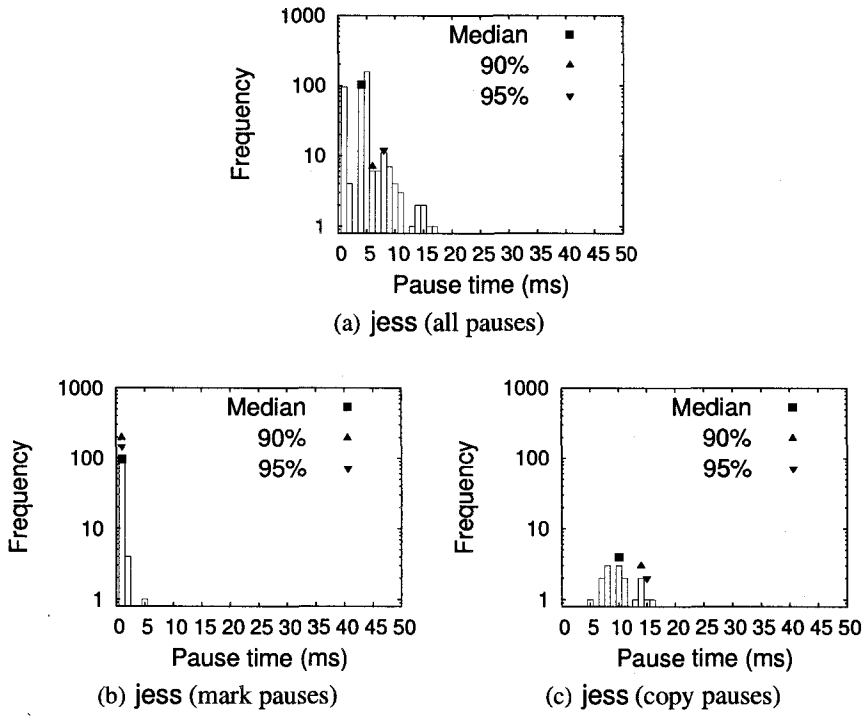


Figure 6.12. BG-MC^2 pause time distributions for jess in a heap that is 1.8 times the maximum live size.

in all cards that contain an object into the window being collected, which can be more expensive.

Summary: The results show that BG-MC^2 can obtain low pause times and good throughput in constrained heaps. The average pause time for BG-MC^2 is 27.5ms in a heap that is 1.8 times the program live size. Importantly, the execution times for BG-MC^2 are good—about 4% better than a well tuned mark-sweep collector and about 3% worse than a well tuned mark-compact collector. BG-MC^2 outperforms BG-MOS significantly in small heaps. In heaps 1.8–2.5 times the live size, BG-MOS has execution times 6–8% higher on average with comparable pause times. In larger heaps, BG-MOS is a few percent worse. BG-MC^2 usually performs better than BG-COPY in small and moderate-size heaps, is a few percent worse for db in moderate-size and large heaps, and for javac in large heaps.

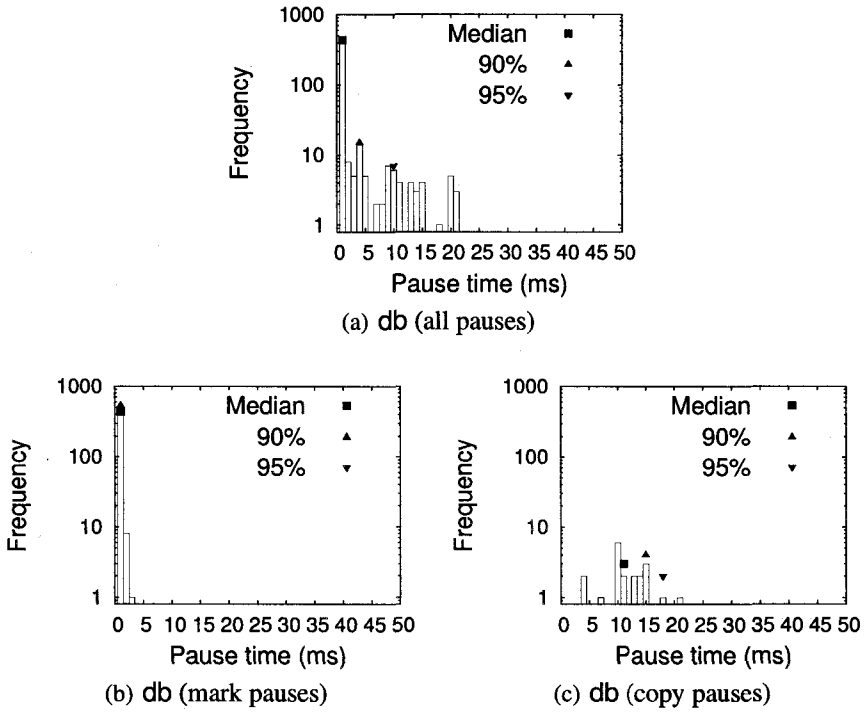


Figure 6.13. BG-MC^2 pause time distributions for db in a heap that is 1.8 times the maximum live size.

6.4.3 Pause Time Distribution

Figure 6.12–Figure 6.17 show the distribution of BG-MC^2 pause times for the six benchmarks in a heap that is 1.8 times the benchmark's maximum live size. The figures contain three plots for each benchmark: the first contains all pauses (nursery collection, mark phase, and copy phase), the second only mark phase pauses, and the third only copy phase (nursery and old generation window copying) pauses. The graphs also show the durations of the median pause, and of the 90th and 95th percentile pauses. The x-axis on all graphs shows the actual pause times and the y-axis shows the frequency of occurrence of each pause time. The y-axis is on a *logarithmic scale*, allowing one to see clearly the less frequently-occurring longer pauses.

For all benchmarks, a majority of the pauses are 10ms or less. 83% of all pauses for javac are 10ms or less, and 93% of pauses for pseudojbb are in the 0–10ms range. For

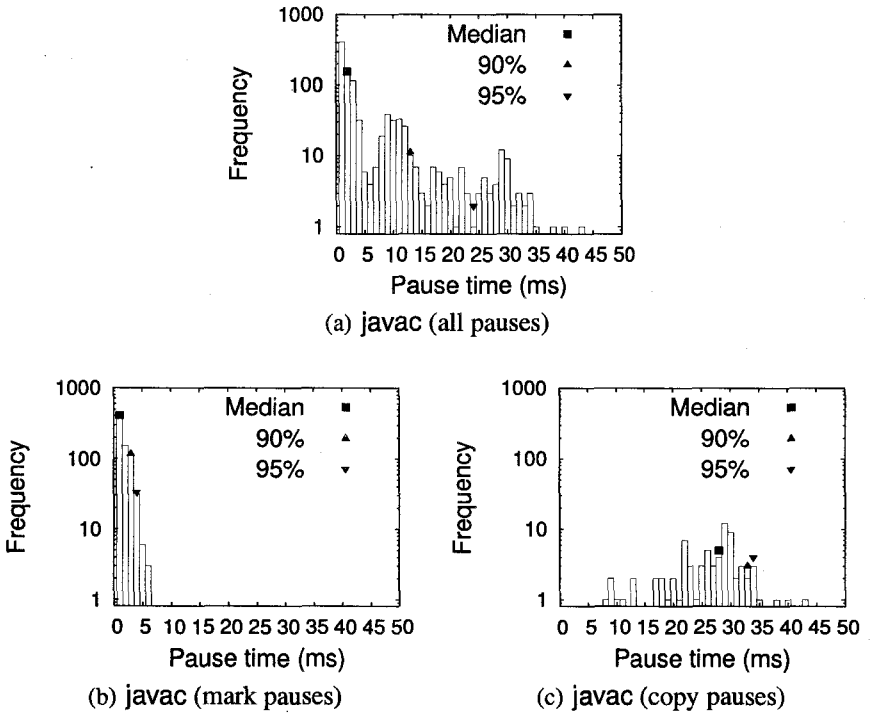


Figure 6.14. BG-MC^2 pause time distributions for javac in a heap that is 1.8 times the maximum live size.

jess, db, mtrt, and jack, pauses that are 10ms or less account for 97%, 95%, 89%, and 93% of all pauses.

Most of these pauses are caused by the mark phase, which performs small amounts of marking interleaved with allocation. These pauses cause the median pause time value to be low. The graphs containing mark-only pauses show that the maximum duration of a mark pause is 9ms, and this occurs for jack. jess, db, mtrt, and pseudojbb have mark pauses that are 5ms or less. All mark pauses for javac are at most 7ms long.

The less frequent, longer pauses (up to 43ms long) typically result from the copy phase. These collections copy objects out of both the nursery and a subset of the old generation windows. The average copy phase collection time for javac is 26ms, and the average copy phase pause time for pseudojbb is 10ms. The longest copy pause times are for javac (43ms) and pseudojbb (42ms). 91% of all copy pauses are shorter than 30ms in duration.

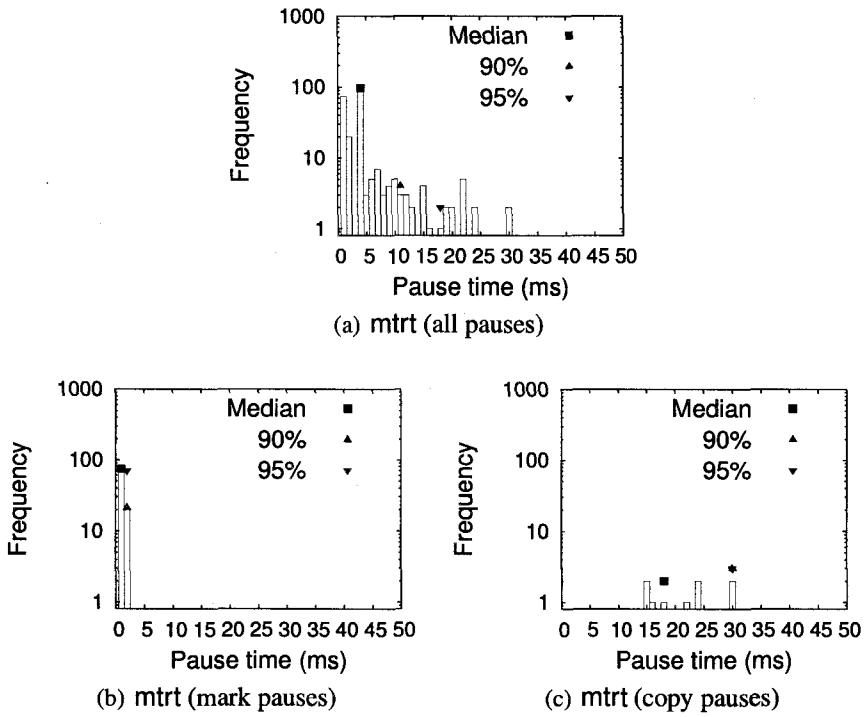


Figure 6.15. BG-MC^2 pause time distributions for mtrt in a heap that is 1.8 times the maximum live size.

One possible technique we could use to reduce copy phase pause times further is to collect the nursery and old generation windows separately. We call this a *split phase* technique. Using split-phase, BG-MC^2 would alternate between nursery collections and old generation window copying, with data from the windows copied when the nursery is half full. However, this technique adds a cost to the write barrier, to keep track of pointers from the nursery into the next set of old generation windows being copied. We have not implemented and evaluated this technique for BG-MC^2.

6.4.4 Bounded Mutator Utilization

We now look more closely at the pause time characteristics of the collectors. We consider more than just the maximum pause times that occurred, since these do not indicate how the collection pauses are distributed over the running of the program. For example,

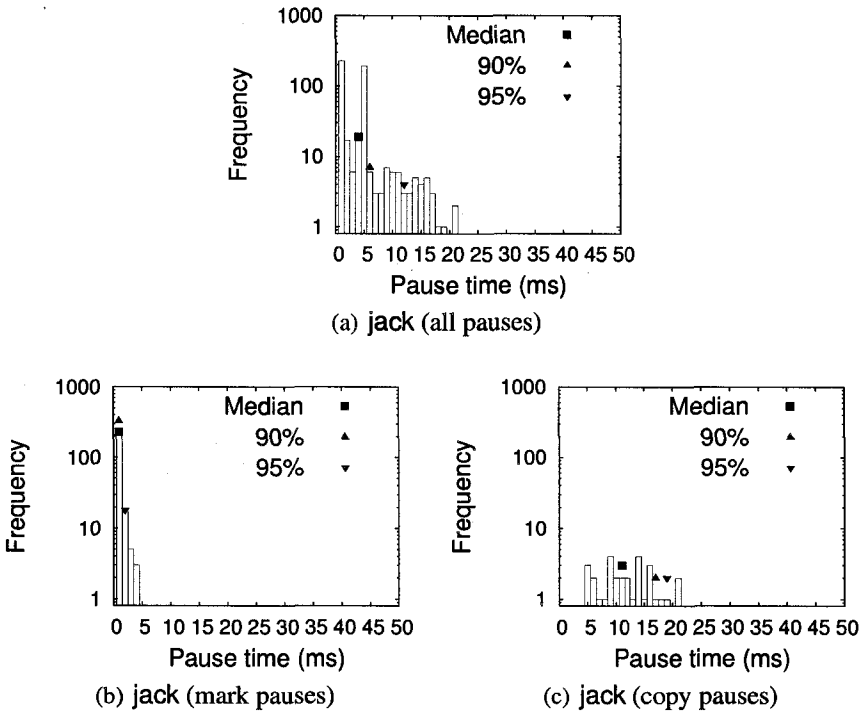


Figure 6.16. BG-MC^2 pause time distributions for jack in a heap that is 1.8 times the maximum live size.

a collector might cause a series of short pauses whose effect is similar to a long pause, which cannot be detected by looking only at the maximum pause time of the collector (or the distributions). We use the methodology described in Chapter 4 to plot BMU curves for the collectors.

Figure 6.18–Figure 6.20 show BMU curves for the six benchmarks for a heap size equal to 1.8 times the benchmark live size. The x-intercept of the curves indicates the maximum pause time, and the asymptotic y-value indicates the fraction of the total time used for mutator execution (average mutator utilization). These graphs do not show the effects of write barrier costs and locality on the overall performance. For instance, for db, BG-MC^2, BG-MOS, and BG-MS have lower throughput. However, since this is caused by higher mutator times (due to write barrier costs and locality effects), and not because of higher GC times, the BMU curves do not reflect the consequences.

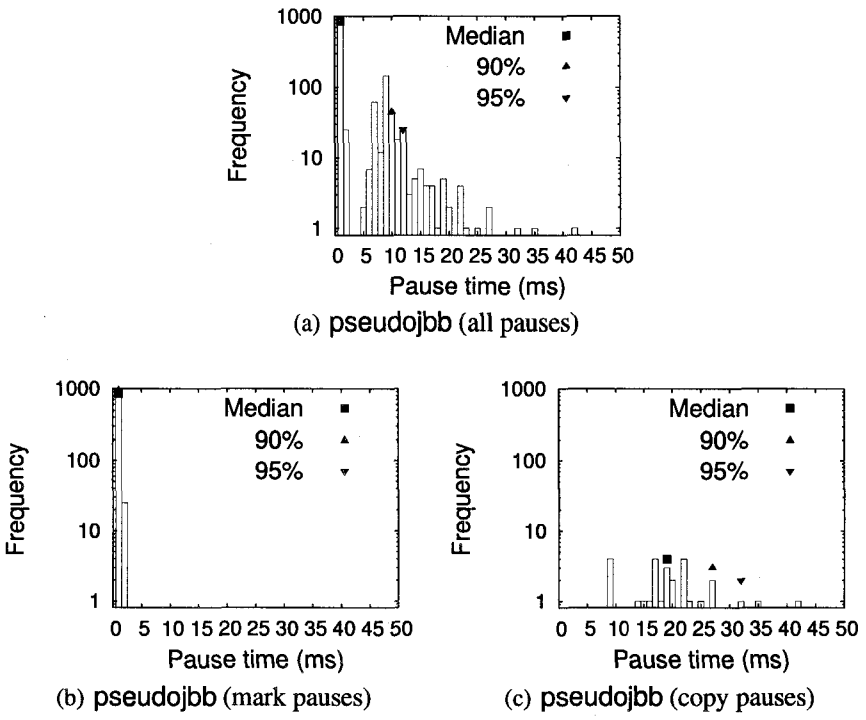


Figure 6.17. BG-MC^2 pause time distributions for pseudojbb in a heap that is 1.8 times the maximum live size.

The five curves in each graph are for BG-MC^2, BG-MOS, BG-COPY, BG-MS, and BG-MSC. The curve with the smallest x-intercept is for BG-MC^2, and the curves for BG-MOS typically have the next smallest x-intercept. BG-MSC has the curve with the largest x-intercept; BG-COPY and BG-MS typically have curves with x-intercepts in between.

For all benchmarks, BG-MC^2 allows some mutator utilization even for very small windows. This is because of the low pause times for the collector. For most benchmarks, the mutator can execute for up to 10–25% of the total time in the worst case, for time windows that are about 50ms long. The non-incremental collectors, on the other hand, allow non-zero utilization in the worst case only for much larger windows, since they have large maximum pause times.

BG-MC^2 allows better utilization than BG-MOS across all window sizes for db, javac, and jack. For jess and mtrt, apart from a couple of points where utilization is a little lower,

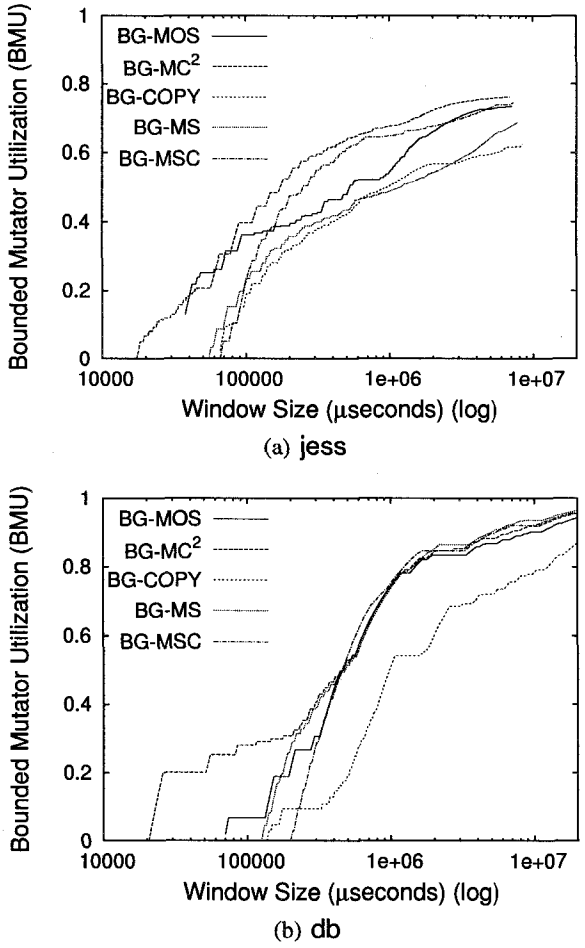


Figure 6.18. BG-MC^2, BG-MOS, BG-COPY, BG-MS, and BG-MSC BMU curves for jess and db in a heap that is 1.8 times the maximum live size.

BG-MC^2 provides better utilization. For pseudojbb, for a range of heap sizes between 100–800ms, utilization is a few percent lower. Utilization is better at all other window sizes.

BG-MC^2 can provide higher utilization than BG-MS in windows of time up to 7 seconds for jess, windows up to 600ms for db, 300ms for javac, mtrt, and pseudojbb, and windows up to 200ms for jack. When compared with BG-MSC, utilization is higher for windows up to 7s, 3s, 800ms, 500ms, 400ms, and 300ms for jess, javac, pseudojbb, mtrt,

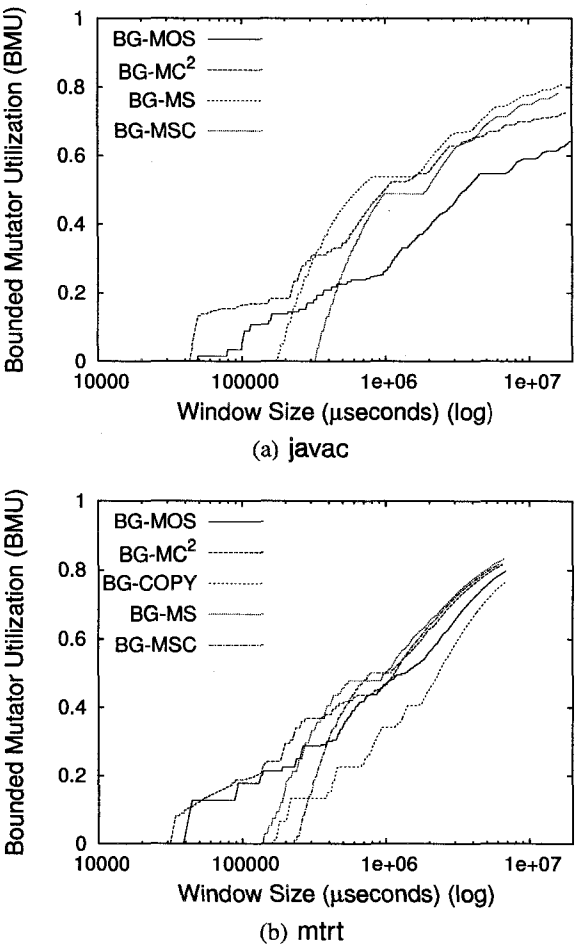


Figure 6.19. BG-MC^2, BG-MOS, BG-COPY, BG-MS, and BG-MSC BMU curves for javac and mtrt in a heap that is 1.8 times the maximum live size.

db, and jack respectively. BG-COPY runs for only 4 out of 6 benchmarks at this heap size, and BG-MC^2 provides better utilization across all window sizes for these benchmarks.

Beyond that, the utilization provided by BG-MC^2 for most benchmarks is about the same, and the asymptotic y-values for the curves are very close. For jack, utilization provided by BG-MC^2 is lower than BG-MSC for windows larger than 300ms but overall utilization is close. Only for javac does BG-MC^2 provide significantly lower overall

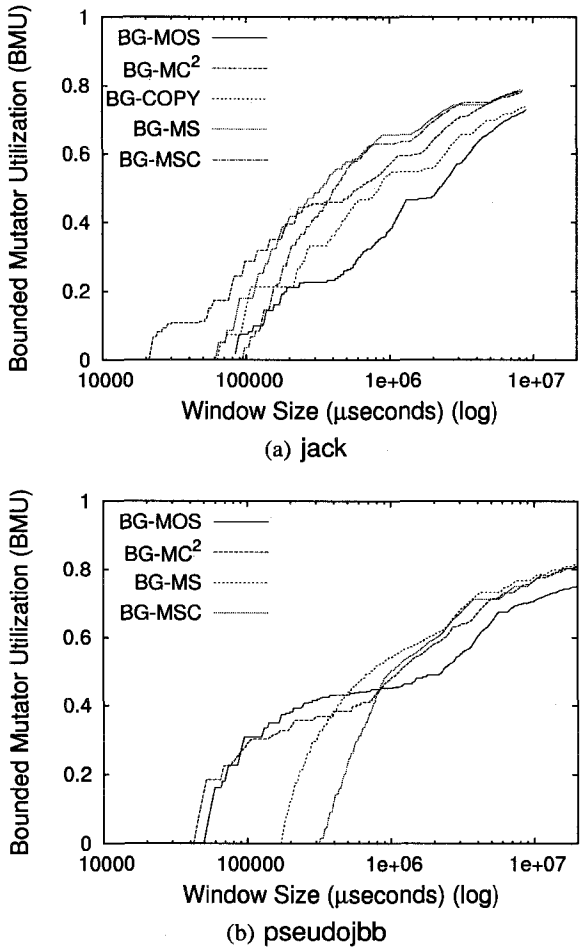


Figure 6.20. BG-MC^2, BG-MOS, BG-COPY, BG-MS, and BG-MSC BMU curves for jack and pseudojbb in a heap that is 1.8 times the maximum live size.

utilization (the overall utilization for javac is much closer in heaps that are twice the live size or higher).

Summary: The mutator utilization curves for the collectors in a heap that is 1.8 times the live size show that BG-MC^2 not only provides shorter pause times, it also provides higher mutator utilization for small windows of time, i.e., it spreads its pauses out well. This holds even for windows of time that are larger than the maximum pause times for the non-incremental collectors. In windows of time that are larger than one second, the utiliza-

tion provided by BG-MC^2, BG-MS, and BG-MSC tends to be about the same. BG-MC^2 provides better utilization than BG-COPY across all window sizes, for the four benchmarks that BG-COPY can run at this heap size.

When compared with BG-MOS, BG-MC^2 is always better for db, javac, and jack. For jess and mtrt, BG-MC^2 is usually better except at a few points. For pseudojbb, utilization is better except for window sizes between 100–800ms. At the points where BG-MC^2 has lower utilization, it is usually only a few percent. Overall utilization for BG-MC^2 is always better.

6.5 Collector Processing Costs

We present in this section, the raw processing costs of the collectors with which we have experimented in the thesis. The costs do not indicate how a collector would perform overall, since this depends on the number of collections invoked and the locality effects of the collector's object layout. Collectors such as MSC and MC^2 optimize collection costs by avoiding copying permanent data that is clustered together. These optimizations are not reflected in the cost of a single collection. Also, since MSC, MC, and MC^2 copy much less data than a semi-space copying collector, they end up performing much less work overall. Additionally, MS performs its sweep phase lazily. Thus the cost of the sweep phase is added incrementally to the nursery collection cost and is not reflected in the full collection cost.

To compute the raw processing cost, we run a fixed number of nursery collections for each collector, without any restriction on heap space. We then trigger a full collection, and measure the GC cost and the bytes processed. To measure the cost of MC/MC^2, we implement MC with logical addressing and perform all marking and copying at once, i.e., if MC/MC^2 requires multiple copying passes, we perform them immediately after each other. For each collector, we present the collector processing cost as the ratio of the collection time to the total bytes processed. Table 6.10 shows processing costs on an Intel P4 machine,

and Table 6.11 shows costs on a PowerPC G4. The configurations for these machines are described in Section 3.3.

The relative performance of the copying collectors is similar across the architectures. MC/MC^2 is 63% more expensive than semi-space copying on average on the Intel machine, and 73% more expensive on PowerPC. MSC costs 49% more than semi-space copying on average on the Intel machine, and 40% more on average on the PowerPC. The additional cost is because of the two passes over the heap. MSC performs better than MC/MC^2 because of better locality during compaction.

However, the relative cost of copying and mark-sweep varies significantly between the architectures. While we find that semi-space copying costs only 6% more on the Intel machine, it costs 60% more on average on the PowerPC. Similarly MSC and MC/MC^2 cost much more relative to MS on the PowerPC. We are not sure of the reasons for the performance difference between the architectures. One possibility is that the code generated for the Intel machine is not of very high quality, thus making the MS tracing and bit operations inefficient. Another possibility is that the performance difference is caused by the gap between processor and memory speeds. With the faster P4 processor and relatively slower memory, locality of data becomes an important factor. Thus, the copying collectors with their better data locality have better relative performance. Fragmentation caused by MS becomes a bigger factor here, causing MS to be more expensive.

The lower MS marking costs on the PowerPC likely improve MS's overall performance. However, we believe that MC^2 will still obtain significant pause time advantage over MS. We estimate that MS pause times will be 2.5 times higher than MC^2's even with the lower mark pause times.

6.6 Conclusions

BG-MC^2 (Memory-Constrained Copying collector) can run in constrained memory and provides both good throughput and short pause times. These properties make the collector

Benchmark	MS	COPY	MSC	MC/MC^2
_202_jess	9.1	9.6	14.3	16.4
_209_db	10.3	11.3	16.3	18.5
_213_javac	13.8	13.9	24.3	24.2
_227_mtrt	12.1	12.4	19.3	20.1
_228_jack	8.7	8.7	11.7	14.7
pseudojbb	6.2	7.4	9.9	11.4

Table 6.10. Cost per byte processed (microseconds) for four generational collectors. The values represent the cost during a full collection for approximately the same amount of data on an Intel 1.7GHz P4.

Benchmark	MS	COPY	MSC	MC/MC^2
_202_jess	16.7	28.1	36.9	44.6
_209_db	14.8	22.6	32.9	38.0
_213_javac	20.4	28.1	44.4	50.9
_227_mtrt	17.0	24.0	36.3	41.1
_228_jack	16.6	26.9	33.7	39.5
pseudojbb	8.0	15.9	20.9	24.5

Table 6.11. Cost per byte processed (microseconds) for four generational collectors. The values represent the cost during a full collection for approximately the same amount of data on a 533MHz PowerPC 7410 G4.

suitable for applications running on handheld devices that have soft real-time requirements. It is also attractive for desktop and server environments, where its smaller and more predictable footprint makes better use of available memory.

We compared the performance of BG-MC^2 with BG-MOS, a non-incremental generational copying collector (BG-COPY), a non-incremental generational mark-sweep collector (BG-MS), and a generational mark-compact collector (BG-MSC). We showed that BG-MC^2 provides throughput comparable to that of all the non-incremental collectors. The pause times of BG-MC^2 are significantly lower than those for BG-COPY, BG-MS, and BG-MSC in constrained memory. Also, BG-MC^2 provides much higher throughput than BG-MOS in small heaps, and 6-8% better overall performance in heaps between 1.8–2.5 times the program live size.

CHAPTER 7

CONCLUSIONS

With the recent increase in popularity of managed run-time environments such as Java and .NET, high performance garbage collection has become a necessity. This, coupled with the explosion of handheld memory-constrained devices that run interactive applications, has made it essential that garbage collectors provide high throughput and short response times with low space overhead. However, many modern garbage collectors tend to impose a significant space overhead in order to provide good throughput and response times. Copying garbage collection, a technique that can provide high performance, has particularly high space requirements, thus making it unsuitable for applications running on memory-constrained devices.

7.1 Contributions

In this thesis we perform a detailed study of two previous copying garbage collection techniques: a generational copying collector and the Train collector. A generational copying collector, like most previous copying collectors, generally has a minimum space overhead of a factor-of-two. It also tends to have occasional long pauses. The Train collector extends generational copying collection to lower minimum space overheads and yields short maximum pauses. However, we show that the Train collector can suffer from high throughput costs, especially in small heaps. This is due to its high copying costs, that are particularly high in the presence of large, cyclic data structures.

We then present a new copying collection technique, Mark-Copy, that extends generational copying collection, and like the Train collector overcomes the minimum factor-of-

two space overhead. Additionally, we show that Mark-Copy provides high throughput in constrained memory. The throughput is higher than that of a generational copying collector in small and moderate-size heaps. It is comparable to that of a generational mark-sweep collector in small heaps, and slightly better for some benchmarks in moderate and large heaps, while compacting data and preventing memory fragmentation.

Finally, we present MC^2, a collector that extends Mark-Copy and provides short response times in addition to high throughput and low space overheads. We show that the collector provides performance that is comparable to that of a generational mark-sweep collector and a generational mark-compact collector in small heaps, while yielding pause times that are a factor of 4–6 lower. We also show that MC^2 outperforms a generational copying collector and the Train collector in small and moderate-size heaps.

This thesis shows that while copying collection can provide high performance in large heaps, its advantages can be leveraged to perform effectively even in constrained memory situations. MC^2, with its high throughput, short pauses, and low space overheads is suitable for memory-constrained devices, and also for other environments including desktops and servers.

7.2 Future Work

A future direction for the Mark-Copy collector would be to study its performance on multiprocessor systems. While, we do not discuss this in the thesis, we believe that parallelizing Mark-Copy can be simpler than parallelizing a mark-compact collector. The advantage of Mark-Copy is that data is not being compacted in place, thus making it easy for multiple threads to copy data in parallel into a single region. A parallel Mark-Copy collector will be suitable for multiprocessor systems that require high throughput without having pause time constraints.

MC^2 has been implemented and evaluated only on a single processor system. Parallelizing MC^2 would be another interesting piece of future work. A parallel MC^2 collector

would be suitable for multiprocessor server machines that require very short pauses. While MC^2 on a single processor yields pause times in the 15–50ms range, parallel MC^2 should be able to provide pauses in the sub 10ms range.

An issue that MC^2 does not handle comprehensively is that of popular objects. While MC^2 identifies and isolates popular objects, it cannot entirely prevent disruptions from occurring due to popular objects. A key requirement for removing such disruptions is to identify and isolate objects before they become popular. A popular object handling technique that eliminates these pauses with minimal performance penalty would allow MC^2 to provide high performance non-disruptive collection in most situations.

Among modern collection techniques, only concurrent collection is capable of consistently providing maximum pauses lower than a millisecond. Eventually, we hope to extend MC^2 to be a fully concurrent collector, with garbage collection and the program executing concurrently on separate processors.

BIBLIOGRAPHY

[1] Alpern, Bowen, Attanasio, C. R., Cocchi, Anthony, Lieber, Derek, Smith, Stephen, Ngo, Ton, Barton, John J., Hummel, Susan Flynn, Sheperd, Janice C., and Mergen, Mark. Implementing Jalapeño in Java. In OOPSLA [44], pp. 314–324.

[2] Alpern, Bowen, Attanasio, Dick, Barton, John J., Burke, M. G., Cheng, Perry, Choi, J.-D., Cocchi, Anthony, Fink, Stephen J., Grove, David, Hind, Michael, Hummel, Susan Flynn, Lieber, D., Litvinov, V., Mergen, Mark, Ngo, Ton, Russell, J. R., Sarkar, Vivek, Serrano, Manuel J., Shepherd, Janice, Smith, S., Sreedhar, V. C., Srinivasan, H., and Whaley, J. The Jalapeño virtual machine. *IBM Systems Journal 39*, 1 (Feb. 2000), 211–238.

[3] Appel, Andrew W. Simple generational garbage collection and fast allocation. *Software Practice and Experience 19*, 2 (1989), 171–183.

[4] Appel, Andrew W., Ellis, John R., and Li, Kai. Real-time concurrent collection on stock multiprocessors. In *Proceedings of SIGPLAN '88 Conference on Programming Languages Design and Implementation* (Atlanta, Georgia, June 1988), vol. 23(7) of *ACM SIGPLAN Notices*, ACM Press, pp. 11–20.

[5] Azagury, Alain, Kolodner, Elliot K., Petrank, Erez, and Yehudai, Zvi. Combining card marking with remembered sets: How to save scanning time. In *ISMM'98 Proceedings of the First International Symposium on Memory Management* (Vancouver, Oct. 1998), Richard Jones, Ed., vol. 34(3) of *ACM SIGPLAN Notices*, ACM Press, pp. 10–19.

[6] Bacon, David F., Cheng, Perry, and Rajan, V.T. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'2003)* (San Diego, CA, June 2003), ACM Press, pp. 81–92.

[7] Bacon, David F., Cheng, Perry, and Rajan, V.T. A real-time garbage collector with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages* (New Orleans, LA, Jan. 2003), vol. 38(1) of *ACM SIGPLAN Notices*, ACM Press, pp. 285–298.

[8] Bacon, David F., Cheng, Perry, and Rajan, V.T. A unified theory of garbage collection. In OOPSLA [46], pp. 50–68.

[9] Baker, Henry G. List processing in real-time on a serial computer. *Communications of the ACM 21*, 4 (1978), 280–94. Also AI Laboratory Working Paper 139, 1977.

[10] Bekkers, Yves, and Cohen, Jacques, Eds. *Proceedings of International Workshop on Memory Management* (St Malo, France, 16–18 Sept. 1992), vol. 637 of *Lecture Notes in Computer Science*, Springer-Verlag.

[11] Ben-Yitzhak, Ori, Goft, Irit, Kolodner, Elliot, Kuiper, Kean, and Leikehman, Victor. An algorithm for parallel incremental compaction. In Detlefs [28], pp. 100–105.

[12] Blackburn, Stephen M., Cheng, Perry, and McKinley, Kathryn S. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *ICSE 2004, 26th International Conference on Software Engineering* (Edinburgh, May 2004), pp. 137–146.

[13] Blackburn, Stephen M., Jones, Richard, McKinley, Kathryn S., and Moss, J. Eliot B. Beltway: Getting around garbage collection gridlock. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation* (Berlin, June 2002), ACM SIGPLAN Notices, ACM Press, pp. 153–164.

[14] Blackburn, Stephen M., and McKinley, Kathryn S. In or Out? Putting Write Barriers in Their Place. In Detlefs [28], pp. 175–184.

[15] Blackburn, Stephen M., and McKinley, Kathryn S. Ulterior reference counting: Fast garbage collection without a long wait. In OOPSLA [45].

[16] Bobrow, Daniel G. Managing re-entrant structures using reference counts. *ACM Transactions on Programming Languages and Systems 2*, 3 (July 1980), 269–273.

[17] Boehm, Hans-Juergen, Demers, Alan J., and Shenker, Scott. Mostly parallel garbage collection. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)* (Toronto, Ontario, Canada, June 1991), vol. 26(6) of *ACM SIGPLAN Notices*, ACM Press, pp. 157–164.

[18] Boehm, Hans-Juergen, and Weiser, Mark. Garbage collection in an uncooperative environment. *Software Practice and Experience 18*, 9 (1988), 807–820.

[19] Brooks, Rodney A. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In Steele [58], pp. 256–262.

[20] Cahoon, Brendon, and McKinley, Kathryn S. Data flow analysis for software prefetching linked data structures in Java. In *IEEE PACT* (2001), IEEE Computer Society, pp. 280–291.

[21] Caudill, Patrick J., and Wirfs-Brock, Allen. A third-generation Smalltalk-80 implementation. In *OOPSLA'86 ACM Conference on Object-Oriented Systems, Languages and Applications* (Oct. 1986), Norman Meyrowitz, Ed., vol. 21(11) of *ACM SIGPLAN Notices*, ACM Press, pp. 119–130.

[22] Chen, G., Kandemir, M., Vijaykrishnan, N., Irwin, M., Mathiske, B., and Wolczko, M. Heap compression for memory-constrained Java environments. In OOPSLA [45], pp. 282–301.

[23] Cheney, C. J. A non-recursive list compacting algorithm. *Communications of the ACM 13*, 11 (Nov. 1970), 677–8.

[24] Cheng, Perry, and Blelloch, Guy. A parallel, real-time garbage collector. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation* (Snowbird, Utah, June 2001), ACM SIGPLAN Notices, ACM Press, pp. 125–136.

[25] Christopher, T. W. Reference count garbage collection. *Software Practice and Experience 14*, 6 (June 1984), 503–507.

[26] Cohen, Jacques, and Nicolau, Alexandru. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems 5*, 4 (1983), 532–553.

[27] Collins, George E. A method for overlapping and erasure of lists. *Communications of the ACM 3*, 12 (Dec. 1960), 655–657.

[28] Detlefs, David, Ed. *ISMM'02 Proceedings of the Third International Symposium on Memory Management* (Berlin, June 2002), ACM SIGPLAN Notices, ACM Press.

[29] Deutsch, L. Peter, and Bobrow, Daniel G. An efficient incremental automatic garbage collector. *Communications of the ACM 19*, 9 (Sept. 1976), 522–526.

[30] Dijkstra, Edsgar W., Lamport, Leslie, Martin, A. J., Scholten, C. S., and Steffens, E. F. M. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM 21*, 11 (Nov. 1978), 965–975.

[31] Fenichel, Robert R., and Yochelson, Jerome C. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM 12*, 11 (Nov. 1969), 611–612.

[32] Fisher, David A. Bounded workspace garbage collection in an address order preserving list processing environment. *Information Processing Letters 3*, 1 (July 1974), 25–32.

[33] Haddon, B. K., and Waite, W. M. A compaction procedure for variable length storage elements. *Computer Journal 10* (Aug. 1967), 162–165.

[34] Hudson, Richard L., and Moss, J. Eliot B. Incremental garbage collection for mature objects. In Bekkers and Cohen [10].

[35] Hudson, Richard L., Moss, J. Eliot B., Diwan, Amer, and Weight, Christopher F. A language-independent garbage collector toolkit. Tech. Rep. COINS 91-47, University of Massachusetts at Amherst, Department of Computer and Information Science, Sept. 1991.

[36] Johnstone, Mark S., and Wilson, Paul R. The memory fragmentation problem: Solved? In *OOPSLA '97 Workshop on Garbage Collection and Memory Management* (Oct. 1997), Peter Dickman and Paul R. Wilson, Eds.

[37] Jones, Richard E. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[38] Jonkers, H. B. M. A fast garbage compaction algorithm. *Information Processing Letters 9*, 1 (July 1979), 25–30.

[39] Lang, Bernard, and Dupont, Francis. Incremental incrementally compacting garbage collection. In *SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques* (1987), vol. 22(7) of *ACM SIGPLAN Notices*, ACM Press, pp. 253–263.

[40] Lieberman, Henry, and Hewitt, Carl E. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM 26(6)* (1983), 419–429.

[41] Lins, Rafael D., and Vasques, Márcio A. A comparative study of algorithms for cyclic reference counting. Tech. Rep. 92, Computing Laboratory, The University of Kent at Canterbury, Aug. 1991.

[42] Martin, Johannes J. An efficient garbage compaction algorithm. *Communications of the ACM 25*, 8 (Aug. 1982), 571–581.

[43] McCarthy, John. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM 3* (1960), 184–195.

[44] *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications* (Denver, CO, Oct. 1999), vol. 34(10) of *ACM SIGPLAN Notices*, ACM Press.

[45] *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications* (Anaheim, CA, Nov. 2003), vol. 38(11) of *ACM SIGPLAN Notices*, ACM Press.

[46] *OOPSLA'04 ACM Conference on Object-Oriented Systems, Languages and Applications* (Vancouver, Oct. 2004), vol. 39(10) of *ACM SIGPLAN Notices*, ACM Press.

[47] Pettis, Karl, and Hansen, Robert C. Profile guided code positioning. In *Proceedings of SIGPLAN '90 Conference on Programming Languages Design and Implementation* (White Plains, New York, June 1990), vol. 25(6) of *ACM SIGPLAN Notices*, ACM Press, pp. 16–27.

[48] Printezis, Tony, and Garthwaite, Alex. Visualising the Train garbage collector. In Detlefs [28], pp. 100–105.

[49] Rogers, Anne, Carlisle, Martin C., Reppy, John H., and Hendren, Laurie J. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems (TOPLAS) 17*, 2 (Mar. 1995), 233–263.

[50] Sachindran, Narendran, and Moss, Eliot. Mark-Copy: Fast copying GC with less space overhead. In OOPSLA [45], pp. 326–343.

[51] Sachindran, Narendran, Moss, J. Eliot B., and Berger, Emery D. MC^2: High-performance garbage collection for memory-constrained environments. In OOPSLA [46], pp. 81–98.

[52] Saunders, Robert A. The LISP system for the Q–32 computer. In *The Programming Language LISP: Its Operation and Applications* (Cambridge, MA, 1974), E. C. Berkeley and Daniel G. Bobrow, Eds., Information International, Inc., pp. 220–231.

[53] Schorr, H., and Waite, W. An efficient machine independent procedure for garbage collection in various list structures. *Communications of the ACM 10*, 8 (Aug. 1967), 501–506.

[54] Seligmann, Jacob, and Grarup, Steffen. Incremental mature garbage collection using the train algorithm. In *Proceedings of 1995 European Conference on Object-Oriented Programming* (University of Aarhus, Aug. 1995), O. Nierstras, Ed., Lecture Notes in Computer Science, Springer-Verlag, pp. 235–252.

[55] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 ed., March 1999.

[56] Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 ed., 2001.

[57] Steele, Guy L. Multiprocessing compactifying garbage collection. *Communications of the ACM 18*, 9 (Sept. 1975), 495–508.

[58] Steele, Guy L., Ed. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (Austin, TX, Aug. 1984), ACM Press.

[59] Darko Stefanović and Kathryn S. McKinley and J. Eliot B. Moss. Age-based garbage collection. In OOPSLA [44], pp. 370–381.

[60] Stefanović, Darko. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. Ph.d. thesis, University of Massachusetts, Department of Computer Science, Amherst, MA, 1999.

[61] Darko Stefanović and Matthew Hertz and Stephen M. Blackburn and Kathryn S. McKinley and J. Eliot B. Moss. Older-first garbage collection in practice: Evaluation in a Java virtual machine. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance (MSP 2002)* (Berlin, Germany, June 2002).

[62] Stoye, Will R., Clarke, T. J. W., and Norman, Arthur C. Some practical methods for rapid combinator reduction. In Steele [58], pp. 159–166.

[63] Thorelli, Lars-Erik. Marking algorithms. *BIT 12*, 4 (1972), 555–568.

[64] Ungar, David M. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments* (1984), vol. 19(5) of *ACM SIGPLAN Notices*, ACM Press, pp. 157–167.

[65] Ungar, David M., and Jackson, Frank. Tenuring policies for generation-based storage reclamation. In *OOPSLA'88 ACM Conference on Object-Oriented Systems, Languages and Applications* (San Diego, CA, Nov. 1988), vol. 23(11) of *ACM SIGPLAN Notices*, ACM Press, pp. 1–17.

[66] Weizenbaum, J. Symmetric list processor. *Communications of the ACM 6*, 9 (Sept. 1963), 524–544.

[67] Wilson, Paul R. Uniprocessor garbage collection techniques. In Bekkers and Cohen [10].

[68] Wise, David S., and Friedman, Daniel P. The one-bit reference count. *BIT 17*, 3 (1977), 351–9.

[69] Yuasa, Taichi. Real-time garbage collection on general-purpose machines. *Journal of Software and Systems 11*, 3 (1990), 181–198.

[70] Zorn, Benjamin G. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, Mar. 1989. Technical Report UCB/CSD 89/544.