MC^2: High-Performance Garbage Collection for Memory-Constrained Environments

Narendran Sachindran, J. Eliot B. Moss, & Emery D. Berger

CMPSCI TR 04-15
September 23, 2003

MC^2: High-Performance Garbage Collection for Memory-Constrained Environments

Narendran Sachindran J. Eliot B. Moss Emery D. Berger
Department of Computer Science
University of Massachusetts
Amherst, MA 01003, USA
{naren, moss, emery}@cs.umass.edu

ABSTRACT

Java is becoming an important platform for memory-constrained consumer devices such as PDAs and cellular phones, because it provides safety and portability. Since Java uses garbage collection, efficient garbage collectors that run in constrained memory are essential. Typical collection techniques used on these devices are mark-sweep and mark-compact. Mark-sweep collectors can provide good throughput and pause times but suffer from fragmentation. Mark-compact collectors prevent fragmentation, have low space overheads, and provide good throughput. However, they cannot be made fully incremental and so can suffer from long pause times.

Copying collectors can provide higher throughput than either of these techniques, but because of their high space overhead, they previously were unsuitable for memory-constrained devices. This paper presents MC^2 (Memory-Constrained Copying), a copying, generational garbage collector that meets the demands of memory-constrained devices with soft real-time requirements. MC^2 has low space overhead and tight space bounds, prevents fragmentation, provides good throughput, and yields short pause times. These qualities make MC^2 also attractive for other environments, including desktop and server systems.

Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors—*Memory management(garbage collection)*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Java, copying collector, generational collector, mark-copy, mark-sweep, mark-compact, memory-constrained copying

1. INTRODUCTION

Submitted to OOPSLA '04 Vancouver, Canada

Handheld consumer devices such as cellular phones and PDAs are becoming increasingly popular. These devices tend to have limited amounts of memory. They also run on a tight power budget, and the memory subsystem consumes a considerable fraction of the total power. As a result, it is extremely important to be able to run applications in constrained memory, and to minimize memory consumption during execution.

Java is becoming a popular platform for these handheld devices. It allows developers to focus on writing applications, without having to be concerned with the diverse and rapidly evolving hardware and operating systems of these devices. Since Java uses garbage collection to manage memory, efficient garbage collection in constrained memory has become a necessity.

While a small memory footprint is an essential requirement, applications in the handheld space make other demands of the garbage collector. Cellular phones now contain built-in digital cameras, and run multimedia applications such as games and streaming audio and video. PDAs run scaled-down versions of desktop applications such as web browsers and e-mail clients. While all these applications require good throughput from the garbage collector, many interactive and communications applications also require the collector to have short pause times. For instance, cellular phones need to code, decode, and transmit voice data continuously without delay or distortion. Hence, in order to meet the demands of applications on handheld devices, modern garbage collectors must be able to satisfy all three of these requirements: bounded and low space overhead; good throughput; and reliably short pause times.

Java implementations on handheld devices [21, 27] typically use mark-sweep (MS) [20, 24], mark-(sweep)-compact (MSC) [11], or generational variants of these collectors [18, 28] to manage dynamically allocated memory. MS collectors can provide excellent throughput, and they can be made fully incremental (provide short pause times consistently). However, they suffer from fragmentation, which affects both space utilization and locality. MS collectors typically need to use additional compaction techniques to lower the impact of these problems. MSC collectors can provide good throughput and their space utilization is excellent. However, they tend to have long pauses making them unsuitable for a range of applications that require good response times.

We present in this paper a memory-constrained copying collector, MC^2, that addresses the problems that the above collectors face. MC^2 provides good throughput and short pause times with low space overheads. The collector is based on the mark-copy collection technique [23]. MC^2 runs in bounded space, thus making it suitable for devices with constrained memory. Since the collector regularly copies data, it prevents fragmentation, minimizes memory requirements, and yields good program locality. The collector also limits the amount of

data copied in every collection, thus obtaining short pause times.

We organize the remainder of this paper as follows. We first describe related work in Section 2. In Section 3, we describe the basic mark-copy technique and its limitations. We explain in Section 4 how the new collector overcomes these limitations, bounding space utilization and providing short pause times. To explore the space of collectors appropriate for memory-constrained environments, we built a generational mark-compact collector. We describe the implementation of this collector, a generational mark-sweep collector, and MC^2 in Section 5. We then compare the throughput and pause time characteristics across the collectors and a range of benchmarks in Section 6, and conclude in Section 7.

2. RELATED WORK

We categorize related work according to collector algorithm, discussing in turn related work in mark-sweep, mark-(sweep)-compact, copying, and generational collection.

2.1 Mark-Sweep

A number of incremental collection techniques use mark-sweep collection. Examples of collectors in this category are the collector by Dijkstra et al. [12] and Yuasa's collector [30]. The problem with mark-sweep collectors is that they suffer from fragmentation. Johnstone and Wilson [16] show that fragmentation is not a problem for carefully designed allocators, for a range of C and C++ benchmarks. However, our experience indicates that this is not necessarily true with Java programs, and this property makes purely mark-sweep collectors unsuitable for devices with constrained memory.

Researchers and implementors have also proposed mark-sweep collectors that use copying or compaction techniques to combat fragmentation. Ben-Yitzhak et al. [5] describe a scheme that incrementally compacts small regions of the heap via copying. However, they require additional space during compaction to store remembered set entries, and do not address the problem of large remembered sets. Further, for a heap containing n regions, they require n marking passes over the heap in order to compact it completely. This can lead to poor performance when the heap is highly fragmented. In order to compact the heap completely our collector requires only a single round of marking.

The only collector we are aware of that meets all the requirements we lay out for handheld devices is the real-time collector proposed by Bacon et al. [3]. They use mark-sweep collection and compact the heap using an incremental copying technique that relies on a read barrier. They demonstrate good throughput and utilization in constrained memory. However, in order to make their read barrier efficient, they require advanced compiler optimization techniques. Our collector does not require compiler support beyond that generally available, such as support for write barriers, and therefore is simpler to implement, especially in the absence of a significant compiler optimization infrastructure. In addition, the extensive optimization that their approach requires may not be suitable in the context of memory-constrained devices, which generally perform little optimization of Java bytecodes.

2.2 Mark-Compact

Mark-(sweep)-compact (MSC) collectors [11, 13, 17, 19] use bump pointer allocation, and compact data during every collection. They prevent fragmentation and typically preserve the order of allocation of objects, thus 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 cannot be made fully incremental. While the mark phase of some MSC collectors can be made incremental, the compaction phase is inherently non-incremental.

MC^2 is similar in many ways to MSC collection, but because its copying is incremental it gains the added benefit of shorter pauses.

2.3 Copying

Purely copying techniques also have incremental versions. The most well-known of these are Baker's collector [4], Brooks's collector [8], and the Train collector [14]. 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. However, our experiments with the Train collector show that it tends to copy large amounts of data, especially when programs have large, cyclic structures. In order to obtain good throughput, we found that the collector typically requires space overheads of a factor of 3 or higher. We conclude that the Train algorithm is not well-suited to memory-constrained environments.

2.4 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 collection is based on the hypothesis that for many applications, a large fraction of objects have very short lifetimes, while a small fraction live much longer. The frequent nursery collections weed out the short-lived objects, and less frequent older generation collections reclaim the space occupied by dead long-lived objects. While generational collectors can provide good throughput and short average pause times, the collection technique used in the old generation determines the overall space requirements and pause time characteristics of the collector (namely, fragmentation, minimum space overhead being twice the maximum live size, and large maximum pause time). The drawbacks of MS, copying, and MSC therefore carry over to generational collection.

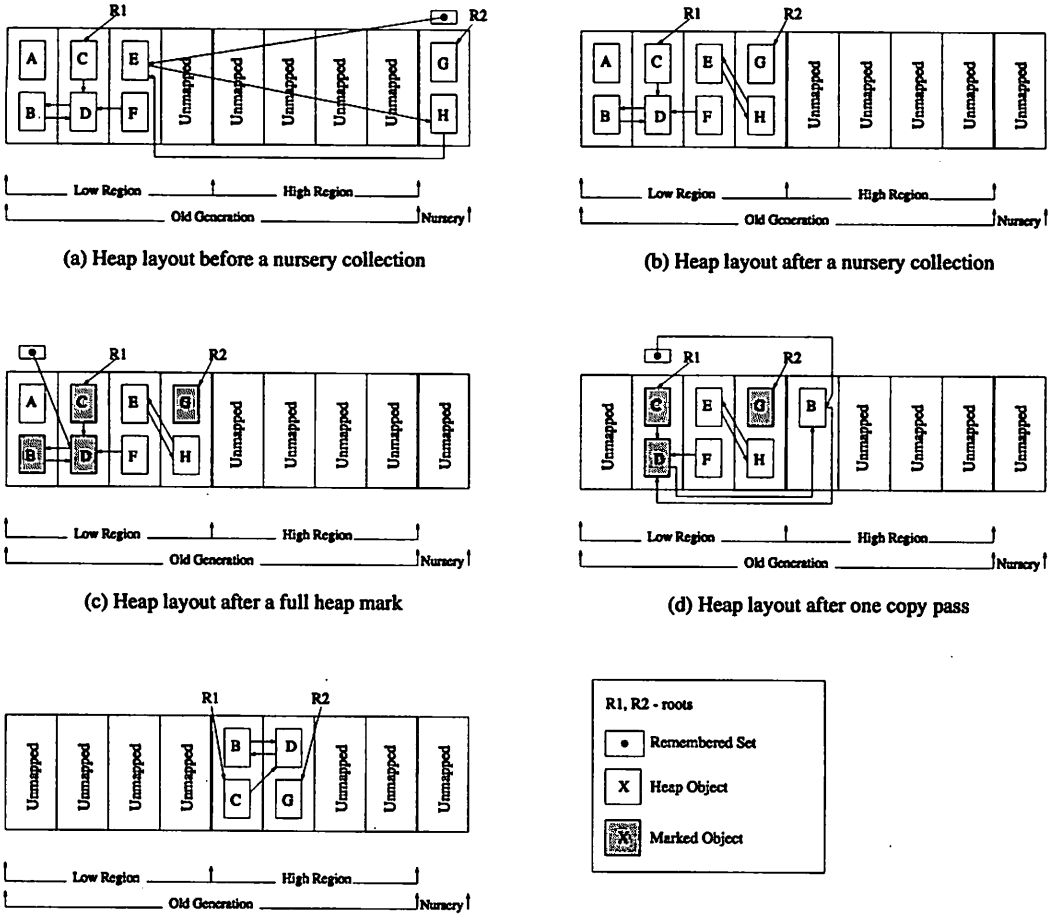
3. BACKGROUND

We introduced the basic mark-copy algorithm, MC, in a previous paper [23]. In this section we summarize MC and describe its limitations with respect to the requirements of memory-constrained environments.

3.1 Mark-Copy

MC extends generational copying collection. It divides the heap into two regions, a nursery, and an old generation. The nursery is identical to a generational copying collector's nursery. MC further divides the old generation into a number of equal size subregions called *windows*. Each window corresponds to the smallest increment that can be copied in the old generation. The windows are numbered from 1 to n, with lower numbered windows collected before higher numbered windows.

MC performs all allocation in the nursery, and promotes nursery survivors into the old generation. When the old generation becomes full (only one window remains empty), MC performs a full heap collection, in two phases, a *mark* phase followed by a *copy* phase.



(a) Heap layout before a nursery collection

(b) Heap layout after a nursery collection

(c) Heap layout after a full heap mark

(d) Heap layout after one copy pass

(e) Heap layout after two copy passes

Figure 1: MC: Full collection example

During the mark phase, MC traverses live objects in the heap, and marks them as reachable. It also performs two other tasks while marking. First, it counts the total volume of live data in each old generation window. Second, it constructs remembered sets for each of the windows. The remembered sets are unidirectional: they record slots in higher numbered windows that point to objects in lower numbered windows. The requirement is to record pointers whose target may be copied (moved) before the source. An implication of using unidirectional remembered sets is that the collector cannot change the order of collection of windows once the mark phase starts. While MC can overcome this limitation by using bidirectional remembered sets (recording all cross-window pointers), this is not generally desirable since bidirectional sets tend to occupy much more space.

Once the mark phase completes, the collector performs a *copy* phase. The copy phase is broken down into a number of *passes*. Each pass copies data out of a subset of the windows in the old generation. Since the collector knows the exact amount of live data in each window, and the total amount of free space in the heap, it can copy data out of multiple windows in each pass. After each copy pass, MC unmaps the pages occupied by the windows evacuated in that pass, thus limiting the total virtual memory mapped at any time during the collection. After finishing all copy passes, the collector resumes nursery allocation and collection.

Figure 1 offers an 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. R1 and R2 are root pointers. Figure 1(a) shows a nursery collection, which results in objects G and H being copied 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 1(b)). MC then 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 1(c)), the remembered set for the first window contains a sin-

gle entry (D→B). All other remembered sets are empty, since there are no pointers into the windows from live objects in higher numbered windows. 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), MC can copy only the first window in this pass. It copies B to the next free window and then unmaps the space occupied by the first window (Figure 1(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 3 objects. In the next copying pass, MC copies the other 3 live objects and then frees up the space occupied by windows 2, 3, and 4 (Figure 1(e)).

3.2 Limitations of Mark-Copy

MC suffers from several limitations. 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. The MC paper ([23]) describes techniques to make the collector incremental, but demonstrates only incremental copying. MC^2 uses the same basic algorithms, but makes several enhancements. 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. The original paper describes a technique that can be used to bound space overhead by using an extra word per object. However, it is not possible to make that technique incremental.

4. 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. 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.

4.1 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

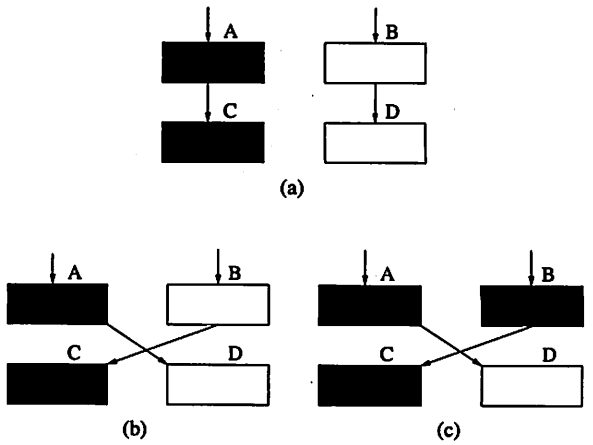


Figure 2: Example of an error during incremental marking

across multiple collections. We describe the details of these features in Section 4.3.

4.2 Incremental marking

MC marks the heap when the free space drops down to a single window full. 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 allocates a bit map for each window currently in the old generation. Marking uses these bit maps to mark reachable objects, and to check if an object has already been marked. Apart from the benefit to marking locality, the bit maps also serve other purposes, described in Section 4.3.

MC^2 then assigns logical addresses to each of the windows. 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:

$$numMarkIncrements = availSpace/(NSR * nurserySize)$$
$$markIncrementSize = totalBytesToMark/numMarkIncrements$$

[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.

totalBytesToMark = *totalBytesToMark* − *markIncrementSize*

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.

Write barrier

A problem with incremental marking is that the mutator modifies objects in the heap while the collector is marking them. This can lead to a situation where the collector does not mark an object that is actually reachable. Using the tri-color invariant [12], 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 the original pointer to the white object, and no other pointer to the white object exists.

Figure 2 shows an example illustrating the problem. In Figure 2(a), the collector has marked and scanned objects A and C, and it has colored them black. The collector has not yet reached objects B and D. At this point, the program swaps the pointers in A and B (Figure 2(b)). When the collector resumes marking, it marks B (Figure 2(c)). Since B points to C, and the collector has already processed C, the collector wrongly concludes that the mark phase is complete, although it has not marked a reachable object (D).

Two techniques exist to handle this problem, termed by Wilson [29] as *snapshot-at-the-beginning* and *incremental update*. Snapshot collectors tend to be more conservative. They consider any object that is live at the start of the mark phase to be live for the duration of the collection. They collect objects that die during the mark phase only in a subsequent collection. Whenever a pointer is overwritten, a snapshot collector records the original target and marks it gray, thus ensuring that all reachable objects are marked. Incremental update collectors are less conservative than snapshot collectors. When the mutator creates a black-to-white pointer, they record either the source or the new target of the mutation. Recording the source causes the collector to rescan the black object, while recording the new target causes the white object to be grayed, thus making it reachable.

Figure 4 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.

```
writeBarrier(sourceObject, sourceSlot, targetObject){
  if (sourceObject not in nursery) {
    if (targetObject in nursery)
      record sourceSlot in nursery remset
    else if (targetObject in old generation) {
      if (sourceObject is not mutated) {
        set mutated bit in sourceObject header
        record sourceObject in mutated object list
      }
    }
  }
}
```

Figure 4: MC^2 write barrier

4.3 Incremental copying

When MC performs a full collection, it copies data out of all windows, without allowing any allocation in between. While this is good for throughput, the pause caused by copying can be long. MC^2 overcomes this problem by spreading the copying work over multiple nursery collections. (The MC paper ([23]) described and offered preliminary results for a version of incremental copying. It did not offer incremental marking, the bounded space guarantee, or the short pause times of MC^2.)

Grouping 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., > 95%) as a *high occupancy* window, and does not copy data out of the window. MC^2 determines the threshold for this classification based on the total space that the copy phase will reclaim. Once MC^2 classifies a window as high occupancy, it discards the remembered set for the window, since no slots pointing to that window will be updated in the current collection.

After separating out the high occupancy windows, MC^2 groups the other windows based on the amount of 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 single old generation window. MC^2 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. MC^2 also creates a separate group for each high occupancy window.

Copying data

After MC^2 groups the marked windows, it resumes nursery allocation. At every subsequent nursery collection, it piggybacks the processing of one old generation window group. In order to pace old generation collection, MC^2 uses information it has about the total space that will be reclaimed by the copy phase. The target for the copy phase is to reduce the heap occupancy below the mark phase threshold. To achieve this goal, MC^2 resizes the nursery at every nursery collection, based on the average survival rate of the nursery and the space that will be reclaimed by compacting the old generation data.

When MC^2 processes an old generation group, it first checks if the group contains a single high occupancy window. If so, MC^2 uses the mark bit map for the window to locate marked objects within it. 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 window 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

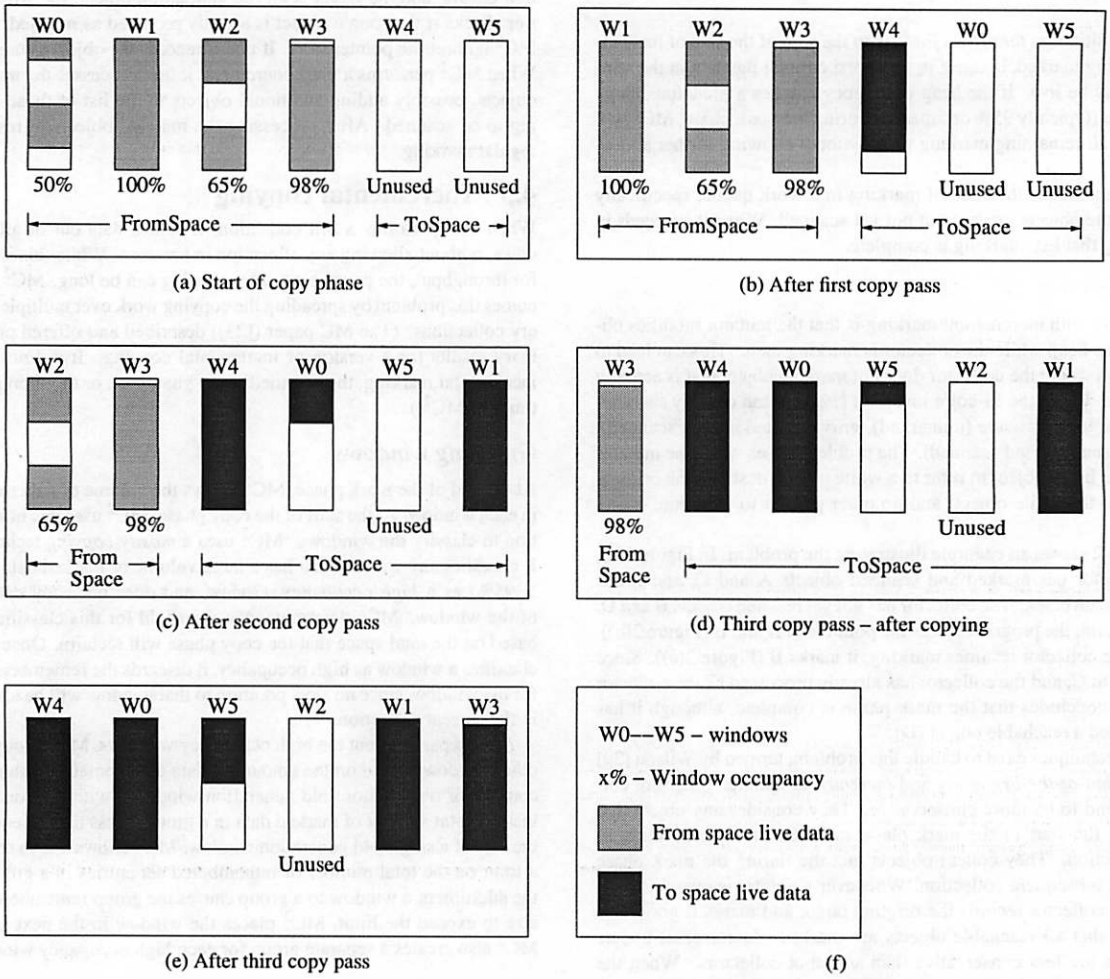


Figure 3: Stages in the copy phase for MC^2

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 [9]. While scanning objects copied into to-space, MC^2 adds slots that reference objects in uncopied windows to the corresponding remembered set.

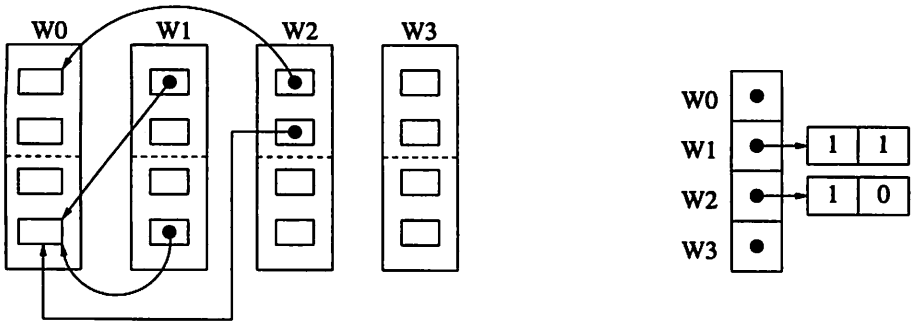
Write barrier

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.

Figure 3 shows an example of the copy phase of the collector. Figure 3(a) shows the heap layout before the copy phase starts. The old generation contains four marked windows (W0–W3). MC^2 classifies W1 and W3 as high occupancy and places them in separate groups, since they contain 100% and 98% marked data respectively. It also places W0 and W2 in separate groups because they are adjacent to high occupancy windows. In the first copy pass (Figure 3(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 3(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 3(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.

4.4 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: large arrays of pointers, and *popular* objects (objects heavily referenced in a program). MC^2 uses two strategies to reduce the remembered set overhead, both of which involve coarsening remembered set entries.



(a) Heap layout (W0–W3 are old generation windows)

(b) Card table for W0

Figure 5: An example of a card table. Each window is broken down into two cards.

Large reference arrays

MC^2 divides large reference arrays (larger than 8 KB) into 512-byte cards. Rather than store every array slot in the remembered sets, MC^2 stores a card address (address with lower 9 bits zeroed out), similar to the technique used by Printezis and Garthwaite [22]. While this technique reduces space overhead significantly, it may increase the remembered set scanning time.

Large remembered sets

MC^2 sets a limit on the total amount of space that remembered sets can occupy. When the space overhead is close to the limit, it coarsens remembered sets starting from the largest, until the space overhead is less than or equal to 95% of 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 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.)

MC^2 divides every window into cards of size 256 bytes. The card table for a target window T is an array of bit maps, one for every other window that contains a reference into T. A bit map in the table corresponding to source window S contains a bit for each card in S. MC^2 sets a bit in the bit map if any object that *starts* in the corresponding card contains a reference into T.

Figure 5 shows an example of a card table. In the example, each window can contain four objects. Each window is divided into two cards and each card can hold two objects. Figure 5(b) shows the card table for window W0. The table contains four entries, one for each window in the heap. The entry for W0 is empty since we do not record intra- window references, and the entry for W3 is empty since W3 contains no references into W0. The bit maps for W1 and W2 contain two entries, one for each card in the windows. Since one object in each card of W1 contains a reference into W0, both bits are set in the bit map for W1. The bit map for W2 has only the first bit set, since no objects in the second card contain a reference into W0.

The process of converting the remembered set representation is straightforward. MC^2 scans the SSB sequentially, and for every recorded slot, it checks if the contents still refer to an object in the target window. If so, it finds the source window and card corresponding to the object that contains the slot. If the card table for the target window does not contain an entry for the source window, it allocates space in the table. It then marks a bit in the table corresponding to the source window card.

With a card size of 256 bytes, the size of the card table for a single window is at most $1/(8 \times 256)$ times the window size, i.e., 0.05% of the total heap space. The collector allows a maximum of 100 windows, thus making the worst case remembered set overhead $100 \times 0.05 = 5\%$ of the total heap space. The bounded space overhead comes at a run-time cost. Marking a bit 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.[2] 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.

Popular objects

Very often, 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. These objects can account for over 90% of all references in a remembered set of size 600KB. MC^2 identifies popular objects during the process of converting an SSB to a card table. 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 exceeding our space budget.) As MC^2 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 100, MC^2 marks it as popular.

During the copy phase, MC^2 copies popular objects into a special window. It treats objects in this window as immortal and does not

[2] While our platform (Jikes RVM) uses an object format that precludes sequential scanning of a region by simple examination of its contents, we use the mark bit map to find (marked) objects within a card. Only the marked objects are relevant.

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 will 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.

5. METHODOLOGY

In order to evaluate garbage collection for memory-constrained environments, we needed to implement garbage collectors described in the literature that are appropriate to that environment. In this section, we describe the implementation of these collectors, followed by our experimental setup.

5.1 Implementation Details

We used the Jikes RVM Java virtual machine [1, 2], release 2.2.3, to implement and evaluate the collectors. We used the Java memory management toolkit (JMTk [6]), standard with Jikes RVM 2.2.3, as the base collector framework. JMTk includes a generational mark-sweep collector, and it provided us with most of the generic functionality required for a copying collector. While none of the collectors *requires* virtual memory mapping support, they happen to use mapping because the JMTk framework uses it. This support speeds up the performance of all collectors by allowing a faster write barrier (no indirection in mapping addresses to logical regions).

JMTk divides the available virtual address space into a number of *regions*. The region with the lowest addresses stores the virtual machine "boot image". The region adjacent to the boot region stores immortal objects—objects never considered for collection. The immortal region uses bump pointer allocation and maps memory on demand in blocks of size 32KB. JMTk allocates all system objects created at run time in the immortal region. It also allocates type information block (TIB) objects, which include virtual method dispatch vectors, etc., in immortal space. Additionally, we allocate all type objects and reference offset arrays for class objects into immortal space, since the mark-compact collector requires that these objects not move during collection.

Next to the immortal region is the region that stores large objects. All collectors allocate objects larger than 8KB into this region. JMTk 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 can be used by the collectors to manage regular objects allocated in the heap.

All collectors we evaluate in this paper are generational collectors. We implement the collectors with a *bounded nursery:* the nursery is bounded by a maximum and minimum size. When the size of the nursery reaches the upper bound, even if there is free space available, we trigger a nursery collection, and if the nursery shrinks to the minimum size, we trigger a full heap collection. The unusual aspect is the upper bound, because it triggers collection earlier than necessary on grounds of available space. This early triggering is important in bounding pause time. It is important to realize that we consider here *only* the bounded nursery variants of each of the collectors we compare.

5.1.1 Generational mark-sweep

The JMTk MS collector 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 [7]: the 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.

The mark-sweep collector uses segregated free lists to manage the heap memory. The 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, the collector returns it to the pool of free blocks.

5.1.2 Generational mark-compact

We implemented a threaded mark-compact generational collector (MSC), based on the algorithm described by Martin [19]. 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, which 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 3%) in Jikes RVM, because the object layout (scalars and arrays in opposite directions) does not allow a sequential heap scan. MSC divides the heap into nursery, old generation, and bit map regions. It uses the same write barrier as MS. 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. For internal pointers, we use extra space to store an additional offset into the target object.

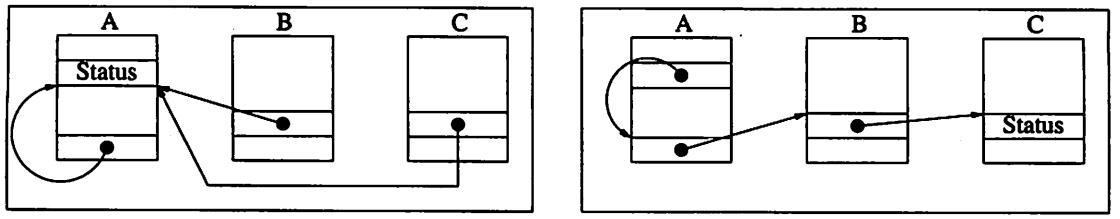
Figure 6 shows an example illustrating how MSC performs threading during the first phase. A, B, and C are three objects in the heap. A contains a self-referential pointer (considered a backward pointer), and B and C contain one pointer each to A. The collector creates a linked list starting from the status word for A. The status (which is distinguished by having its low bits non-zero) 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. Also, it chains 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.

5.1.3 MC^2

MC^2 divides the heap into a nursery and an old generation. It further 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. MC^2 uses a frame size of 8MB. Each frame accommodates one old generation window. 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 window number, as previously described. Each window has an associated remembered set, implemented using a sequential store buffer.

The write barrier is partially inlined. The inlined portion checks



(a) Before threading

(b) After threading

Figure 6: An example of pointer threading performed by the Mark-Compact collector

Benchmark	Description	Maximum Live size (MB)	Total Allocation (MB)
_202_jess	a Java expert system shell	6	319
_209_db	a small data management program	12	93
_213_javac	a Java compiler	13	280
_227_mtrt	a dual-threaded ray tracer	12	163
_228_jack	a parser generator	6	279
pseudojbb	SPECjbb2000 with a fixed number of transactions	30	384

Table 1: Description of the benchmarks used in the experiments

if the source of a pointer store is outside the nursery, and the target lies beyond the immortal region (this subsumes a null test, since the immortal region begins at address greater than 0). If so, the out-of-line code is invoked, which performs two tests. First, if the target is a nursery object, it stores the source slot in the nursery remembered set. Second, it checks a bit in the source object's header, to see if the object has been mutated since the last time it was scanned. If the object has not been mutated, it sets the bit and adds the object to a mutated-object remembered set.

During a nursery collection, MC^2 processes the nursery remembered set first. It then scans mutated objects, and records pointers from the immortal region into the old generation window remembered sets. It also calculates the survival rate of objects in the nursery, and uses it to maintain an average nursery survival rate. After a nursery collection, if the space occupied by the old generation exceeds a predefined *mark threshold*, MC^2 triggers a full collection and the mark phase begins. MC^2 maintains a per-window bit map. This is useful for more than marking objects during the mark phase: the collector uses it during the copy phase to locate objects in cards and to locate live objects in high occupancy windows. As described earlier, MC^2 performs marking in small increments during allocation. The increment size depends on the available space and nursery survival rate.

MC^2 initially creates remembered sets using SSBs. It 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 5% of the heap, MC^2 converts the largest remembered sets to card tables. In our current implementation, we do not coarsen boot image and immortal slots since the number of entries is small and limited. When the mark phase completes, the copy phase commences. MC^2 performs increments of old generation copying along with nursery collection. The size of the nursery grows and shrinks based on the nursery survival rate and the amount of space reclaimed so far in the copy phase. The goal is to finish the copy phase before the heap grows larger than the mark threshold (i.e., it is time to start the next full collection).

5.2 Experimental Setup

Jikes RVM compiles all bytecode to native code before execution. It 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. It optimizes methods at three different levels depending on the execution frequency. However, the adaptive system does not produce reproducible results, because it uses timers and may optimize different methods in different runs.

We used a *pseudo-adaptive* configuration to run our experiments with reproducible results. We first ran each benchmark 7 times with the adaptive run-time system, logging the names of methods that were optimized and their optimization levels. We then determined the methods that were optimized in a majority of the runs, and the highest level to which each of these methods was optimized in a majority of runs. We ran our experiments with exactly these methods optimized (to that optimization level) and all other methods baseline compiled. The resulting system behavior is repeatable, and does very nearly the same total allocation as a typical adaptive system run.

Jikes RVM is itself written in Java, and some system classes can be compiled either at run time or at system build time. We compiled all the system classes at build time to avoid any non-application compilation at run time. The system classes appear in a region called the *boot image*, that is separate from the program heap.

6. RESULTS

We compare GC times, total execution times, space overheads, and pause times for the three collectors (MC^2, MS, and MSC) across six benchmarks. Five of these come from from the SPECjvm98 suite [25], and the sixth is pseudojbb, a modified version of the SPECjbb2000 benchmark [26]. pseudojbb executes a fixed number of transactions (70000), which allows better comparison of the performance of the different collectors. We ran all SPEC benchmarks using the default parameters (size 100), and ignoring explicit GC requests. Table 1 describes each of the benchmarks we used.

We ran our experiments on 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 2.4.7-10. We performed our experiments with the machine in single user mode to maximize repeatability.

6.1 Execution times and pause times

Figure 7 shows GC times for the three collectors relative to the best GC time. Figure 8 shows total execution times for the collectors relative to the best execution 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. Table 2 shows the maximum pause times for the three collectors and the execution times for MS and MSC in a heap that is 1.8 times the program's maximum live size. All results are for configurations using a nursery with a maximum size of 1MB and a minimum size of 64KB. MC^2 uses 40 windows in the old generation.

We look first at the relative execution times of the collectors. The general trend we see in the graphs is that the copying collectors outperform MS in small heaps. Apart from javac, MSC usually performs better than MS in heaps that are smaller than twice the program live size. MC^2 generally performs better than MS in small heaps for all benchmarks other than javac and db.

The GC plots for jess, jack, and pseudojbb show that MC^2 and MSC have lower GC costs in small heaps. Apart from the bitmap, MC^2 has a space overhead bounded by 7.5% (2.5% for copying with 40 windows and 5% for remembered sets), and it is usually under 5%. The fragmentation for MS often exceeds that, and this has a larger effect in the small heaps. The copying collectors also compact data which can yield better locality and improve mutator time. In moderate and large heaps, the copying collectors usually perform better than MS for mtrt, jack, and pseudojbb, while MS performs better for jess and javac.

When we compare MC^2 and MSC, we see that their performance for jess, mtrt, and pseudojbb is very close. MC^2 outperforms MSC for jack and javac, while MSC performs better for db.

Both db and pseudojbb contain significant amounts of permanent data, which is advantageous to MC^2. Since it uses a mostly-copying technique, it does not have to copy large portions of the heap. It compacts a smaller portion of the heap, which contains transient data. In spite of not copying much data, MC^2 has only slightly lower GC times than MSC, because mark-compact collectors have the same property. They also do not repeatedly copy permanent data, since permanent data flows toward the lower end of the heap, and the compactor does not move any live data at the start of the heap. db is a benchmark that is especially sensitive to cache locality effects. Our experience shows that small changes in GC triggering points and object locations tend to have significant effects on the overall performance of db. MSC outperforms both MC^2 and MS for this benchmark. It is possible that the order-preserving property of MSC helps it obtain better locality.

Table 2 shows that, for all benchmarks, the maximum pause time for MC^2 is significantly lower than for the other two collectors, even in a tight heap. When compared with MSC, the maximum pause time for MC^2 is 10–17 times lower. When compared with MS, the pause times for MC^2 are about 7–13 times lower. The throughput for MC^2 is at least as good as that of the other two collectors for most benchmarks. For javac, execution time of MC^2 is about 6% higher than MS, and for db, MC^2 has a 13% higher execution time than MSC.

Summary: The results show that MC^2 can obtain low pause times and good throughput in constrained heaps. The pause times for MC^2 are 10–17 times lower than MSC and 7–13 times lower than MS in a heap that is 1.8 times the program live size. Although we do not show maximum pause times for every heap size, these results hold for larger heap sizes in which MSC and MS perform at least one full heap collection. Importantly, the execution times for MC^2 are comparable to those of both non-incremental collectors.

6.2 Pause time distribution

Figure 9 shows the distribution of MC^2 pause times for the six benchmarks in a heap that is 1.8 times the benchmark's maximum live size. The figure contains three plots for each benchmark: the first contains all pauses, the second only mark phase pauses, and the third only copy phase (nursery and old generation window copying) pauses. The graphs also shows 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. 76% of all pauses for javac are 10ms or less, and 89% of pauses for pseudojbb are in the 0–10ms range. (For jess, db, mtrt and jack, pauses that are 10ms or less account for 96%, 96%, 91% and 92% 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 7ms, and this occurs for javac. jess, db, and pseudojbb have mark pauses that are 3ms or less. All mark pauses for mtrt and jack are at most 6ms long.

The less frequent, longer pauses that are in the 10–50ms range typically result from nursery collections and collection of old generation windows. Of these, purely nursery collections are usually at the lower end of the pause time range. The pauses at the higher end usually result from collections during 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 20ms, and the average copy phase pause time for pseudojbb is 15.33ms. The longest copy pause time for the benchmarks is 47ms, and 99% of all copy pauses are shorter than 30ms in duration.

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, 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 yet implemented and evaluated this technique for MC^2.

javac occasionally causes the creation of a large remembered set (it does not happen at this particular heap size). This happens when a few highly referenced types and identifiers appear in the same window. The remembered set corresponding to the window containing these objects can grow to approximately 600KB. When this happens, MC^2 coarsens the remembered set and converts it to a card table. The referring objects in this case do not all lie close together, and are spread over a significant portion of the heap. When the collector scans the card table to copy objects out of this window, the pause is approximately 60ms, which is somewhat longer than any other pause for the benchmark. However, MC^2 isolates the popular objects residing in the window, and the long pause does not recur. MC^2 did not have to convert remembered set representation for any of the other benchmarks since the remembered set overhead was low.

6.3 Bounded 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

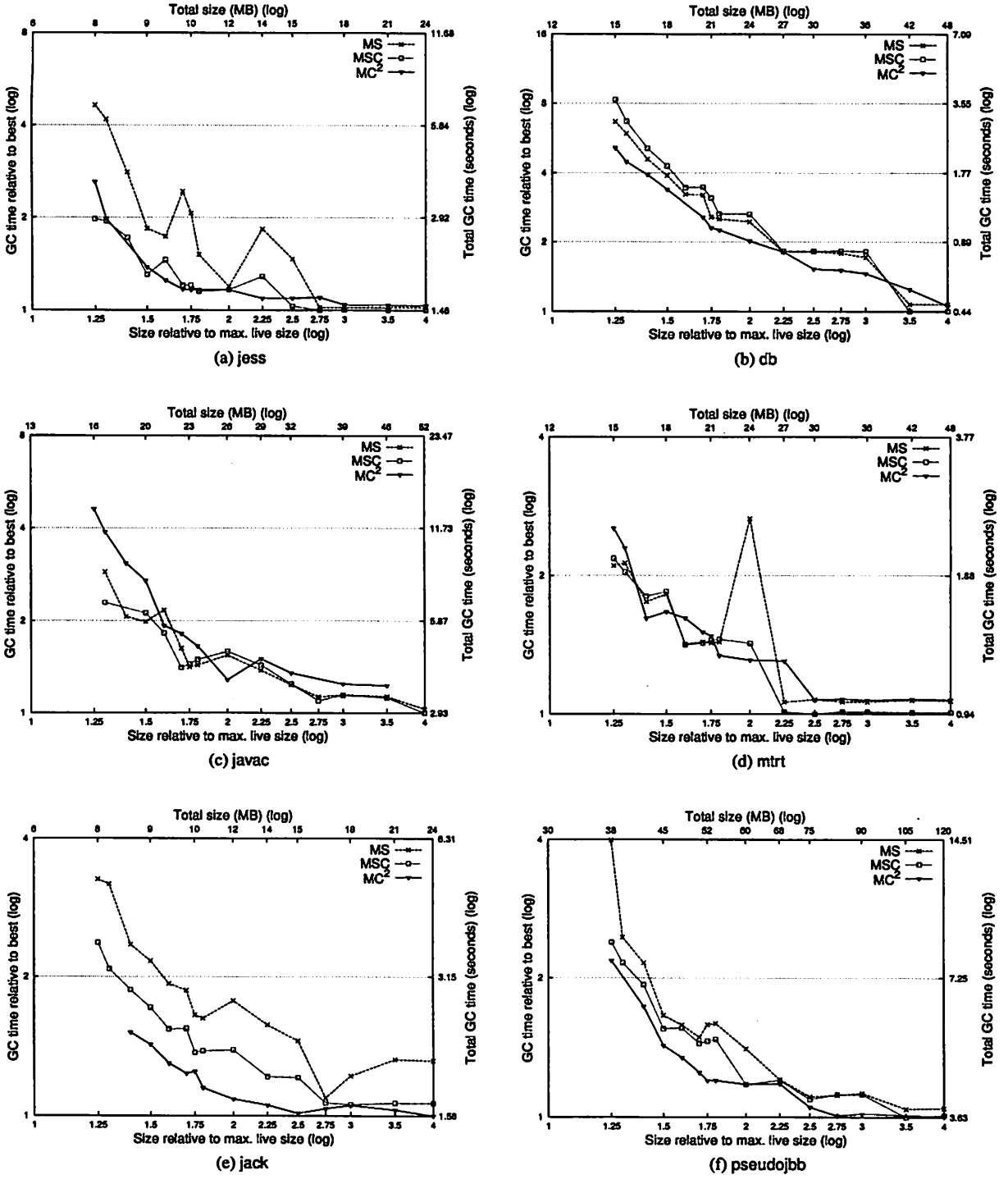


Figure 7: GC time relative to best GC time for MS, MSC, and MC^2

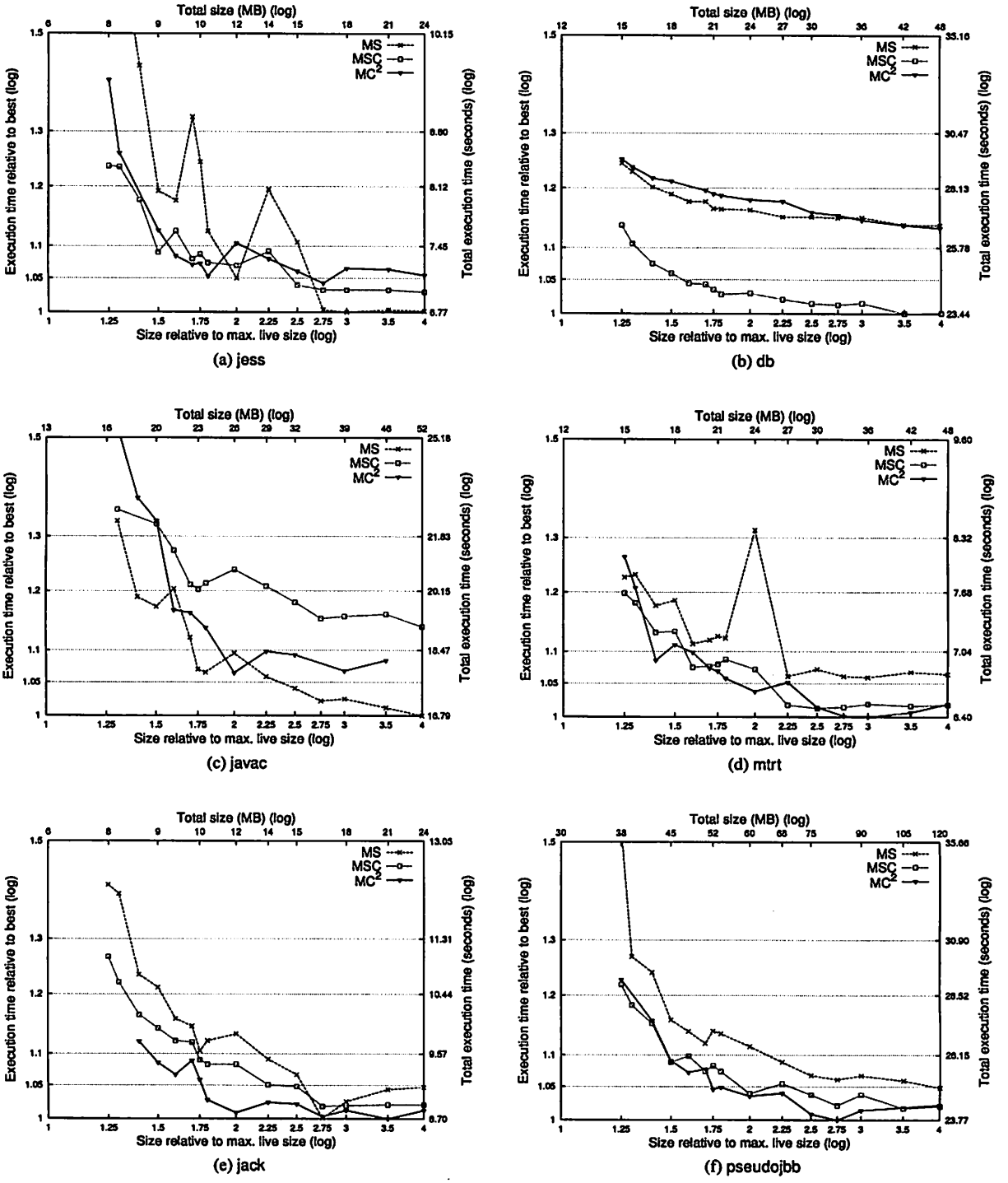


Figure 8: Total execution time relative to best total execution time for MS, MSC, and MC^2

Benchmark	MC^2 MPT (ms)	MS MPT (ms)	MS/MC^2 ET	MSC MPT (ms)	MSC/MC^2 ET
jess	15.91	197.69	1.07	220.87	1.02
db	20.44	276.89	0.98	362.19	0.87
javac	38.85	318.07	0.94	471.54	1.07
mtrt	24.68	315.53	1.04	422.47	1.01
jack	18.13	211.96	1.09	252.56	1.05
pseudojbb	46.49	323.01	1.08	474.32	1.02

Table 2: Maximum Pause Times (MPT) and relative Execution Times (ET) for MC^2, MS, and MSC, in a heap that is 1.8 times the maximum live size. All collectors use a 1MB nursery and MC^2 uses 40 windows.

the running of the program. For example, 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 present *mutator utilization* curves for the collectors, following the methodology of Cheng and Blelloch [10]. 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 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 10 shows 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).

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. For instance, for db, MC^2 and MS have lower throughput. However, since this is caused by higher mutator times (possibly because of locality effects), and not because of higher GC times, the BMU curves do not reflect the consequences. However, db is the only benchmark for which these graphs hide any loss in throughput for MC^2.

The three curves in each graph are for MC^2, MS, and MSC. The curve with the smallest x-intercept is for MC^2. MSC has the curve with the largest x-intercept; the MS curve has an x-intercept in between the other two.

For all benchmarks, MC^2 allows some mutator utilization in the worst case 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.

MC^2 can provide significantly higher utilization than the other two collectors in windows of time that are one second or smaller. Beyond that point, the utilization provided by MC^2 is usually higher or about the same, and the asymptotic y-values for the curves are very close. Only for javac does MC^2 provide (slightly) lower overall utilization. For db, as we mentioned earlier, the locality effects are not evident in the graph. MC^2 and MS provide lower overall utilization than MSC for the benchmark.

Summary: The mutator utilization curves for the collectors show that MC^2 not only provides shorter pause times, it also provides higher mutator utilization for small windows of time, even in the worst case, i.e., it spreads its pauses out well. This holds even for windows of time that are significantly larger than the maximum pause times for the non-incremental collectors. In windows of time that are larger than one second, the utilization provided by all collectors tends to be the same. The overall throughput of the collectors is close. The performance of MC^2 is a bit lower than MS for large windows of time for javac. MC^2 and MS provide lower utilization than MSC for db.

7. CONCLUSIONS

We have presented an incremental copying garbage collector, MC^2 (Memory-Constrained Copying), that runs in constrained memory and provides both good throughput and short pause times. These properties make the collector 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 MC^2 with a non-incremental generational mark-sweep (MS) collector and a generational mark-compact (MSC) collector, and showed that MC^2 provides throughput comparable to that of both of those collectors. We also showed that the pause times of MC^2 are 10–17 times lower than those for MSC, and 7–13 times lower than those of MS in constrained memory.

8. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under grant number CCR-0085792. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF. We are also grateful to IBM Research for making the Jikes RVM system available under open source terms. The JMTk component of Jikes RVM was particularly helpful in this work.

In addition, we thank Rick Hudson for the original idea that led to the MC and MC^2 collectors, and Kathryn McKinley and Csaba Andras Moritz for discussions that helped shape the paper.

9. REFERENCES

[1] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeño in Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 314–324, Denver, CO, October 1999. ACM Press.

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

[3] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collecor with low overhead and consistent utilization. In *Conference*

Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages, ACM SIGPLAN Notices, New Orleans, LA, January 2003. ACM Press.

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

[5] Ori Ben-Yitzhak, Irit Goft, Elliot Kolodner, Kean Kuiper, and Victor Leikehman. An algorithm for parallel incremental compaction. In ISMM '02 [15], pages 100–105.

[6] S. M. Blackburn, P. Cheng, and K. S. McKinley. A garbage collection design and bakeoff in JMTk: An effi cient extensible Java memory management toolkit. Technical Report TR-CS-03-02, Australian National University, March 2003.

[7] Stephen M. Blackburn and Kathryn S. McKinley. In or out? Putting write barriers in their place. In ISMM '02 [15], pages 175–184.

[8] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In Guy L. Steele, editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 256–262, Austin, TX, August 1984. ACM Press.

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

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

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

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

[13] B. K. Haddon and W. M. Waite. A compaction procedure for variable length storage elements. *Computer Journal*, 10:162–165, August 1967.

[14] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, University of Massachusetts, USA, 16–18 September 1992. Springer-Verlag.

[15] *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, Berlin, June 2002. ACM Press.

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

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

[18] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[19] Johannes J. Martin. An effi cient garbage compaction algorithm. *Communications of the ACM*, 25(8):571–581, August 1982.

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

[21] NewMonics Inc., PERC virtual machine. http://www.newmonics.com/perc/info.shtml.

[22] Tony Printezis and Alex Garthwaite. Visualising the Train garbage collector. In ISMM '02 [15], pages 100–105.

[23] Narendran Sachindran and J. Eliot B. Moss. Mark-Copy: Fast copying GC with less space overhead. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.

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

[25] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

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

[27] Sun Microsystems, The CLDC HotSpot Implementation Virtual Machine, Java 2 Platform, Micro Edition J2ME Technology, March 2004. http://java.sun.com/products/cldc/wp/CLDC-HI_whitepaper-March_2004.pdf.

[28] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) —Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

[29] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.

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

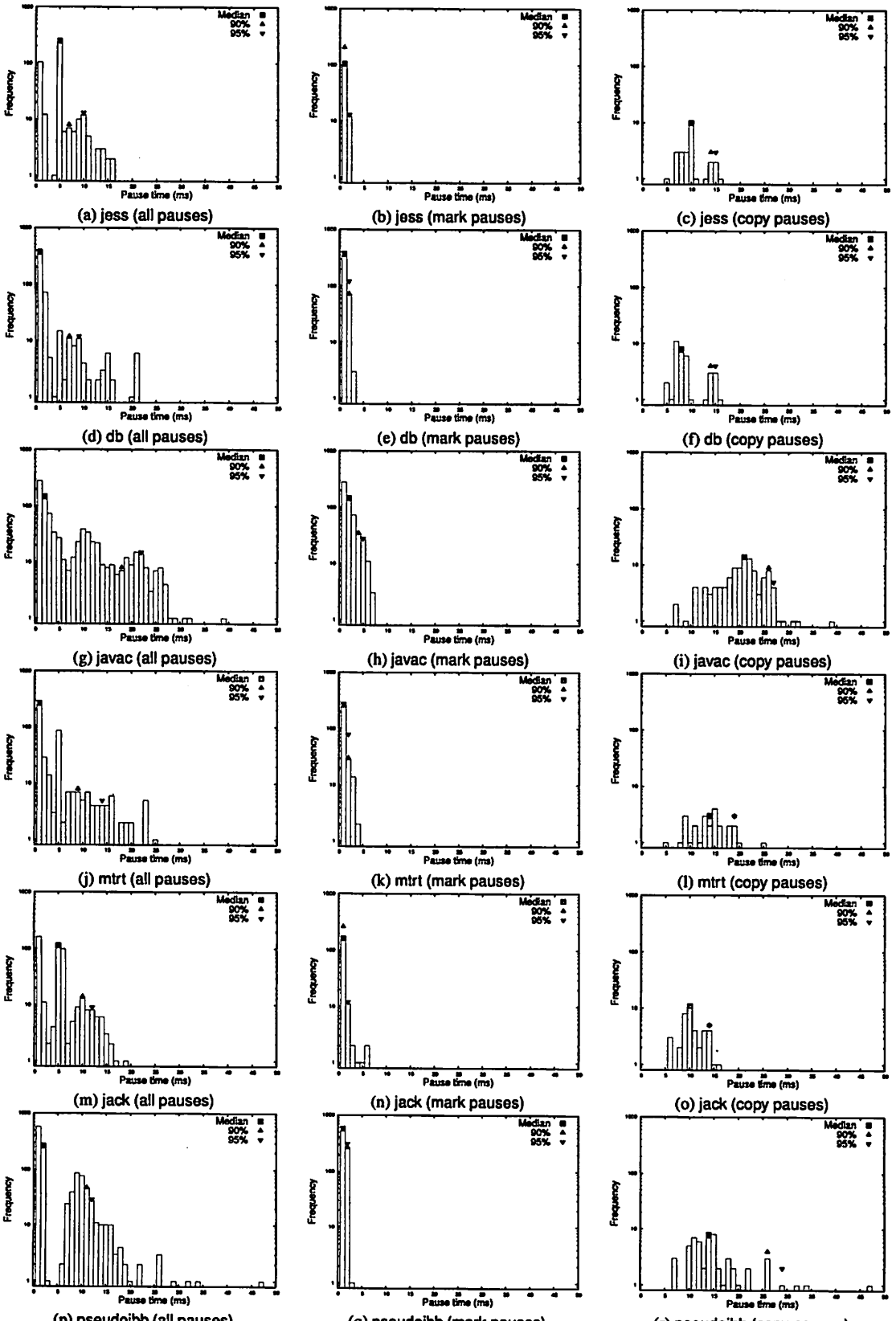


Figure 9: MC^2 pause time distributions, in a heap that is 1.8 times the program live size. The first column shows all pauses. Mark pauses (second column) are always 7ms or less. Copy phase pauses (third column) are longer (5–47ms).

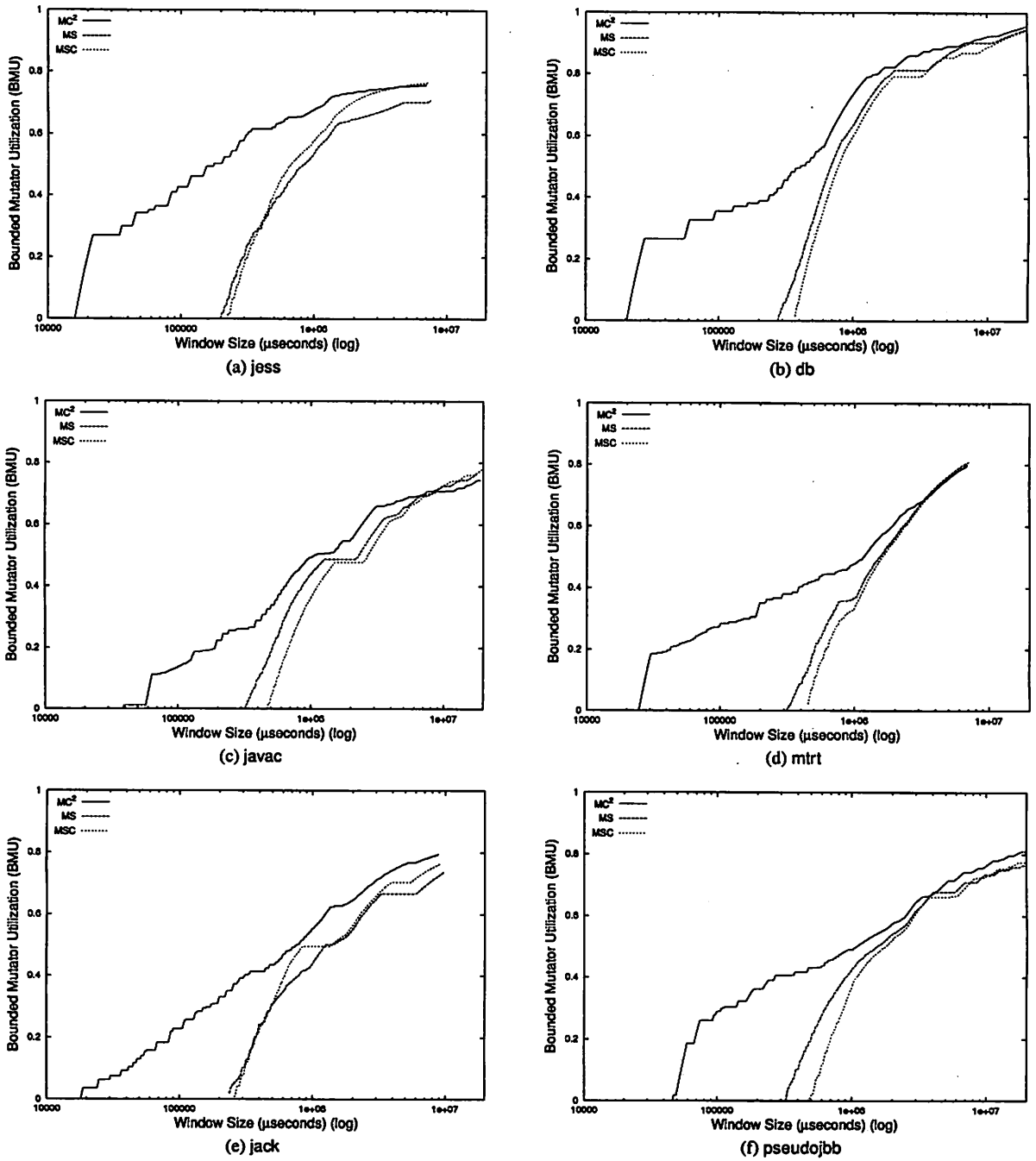


Figure 10: Bounded mutator utilization for MC^2, MS, and MSC in a heap that is 1.8 times the maximum live size