# 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 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 needs 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$^2$ attractive for other environments, including desktops and servers.

## 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-sweep, mark-compact, mark-copy, memory-constrained copying

## 1. INTRODUCTION

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 on 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 [23, 29] typically use mark-sweep (MS) [22, 25], mark-(sweep)-compact (MSC) [13], or generational variants of these collectors [20, 30] 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 [24]. 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 ba-


*OOPSLA'04,* Oct. 24-28, 2004, Vancouver, British Columbia, Canada.

sic 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 classify 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 (which in fact run concurrently with the application) are the collector by Dijkstra et al. [14] and Yuasa's collector [32]. The problem with mark-sweep collectors is that they suffer from fragmentation. Johnstone and Wilson [18] 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 systems, and our experience indicates that this property is not necessarily true with Java programs. Fragmentation 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. [7] 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 implemented by Bacon et al. [5]. 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.

## 2.2 Mark-Compact

Mark-(sweep)-compact (MSC) collectors [13, 15, 19, 21] 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 tend to have long pauses.

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 [6], Brooks's collector [10], and the Train collector [16]. 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 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.

# 3. BACKGROUND

We introduced the basic mark-copy algorithm, MC, in a previous paper [24]. 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.

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

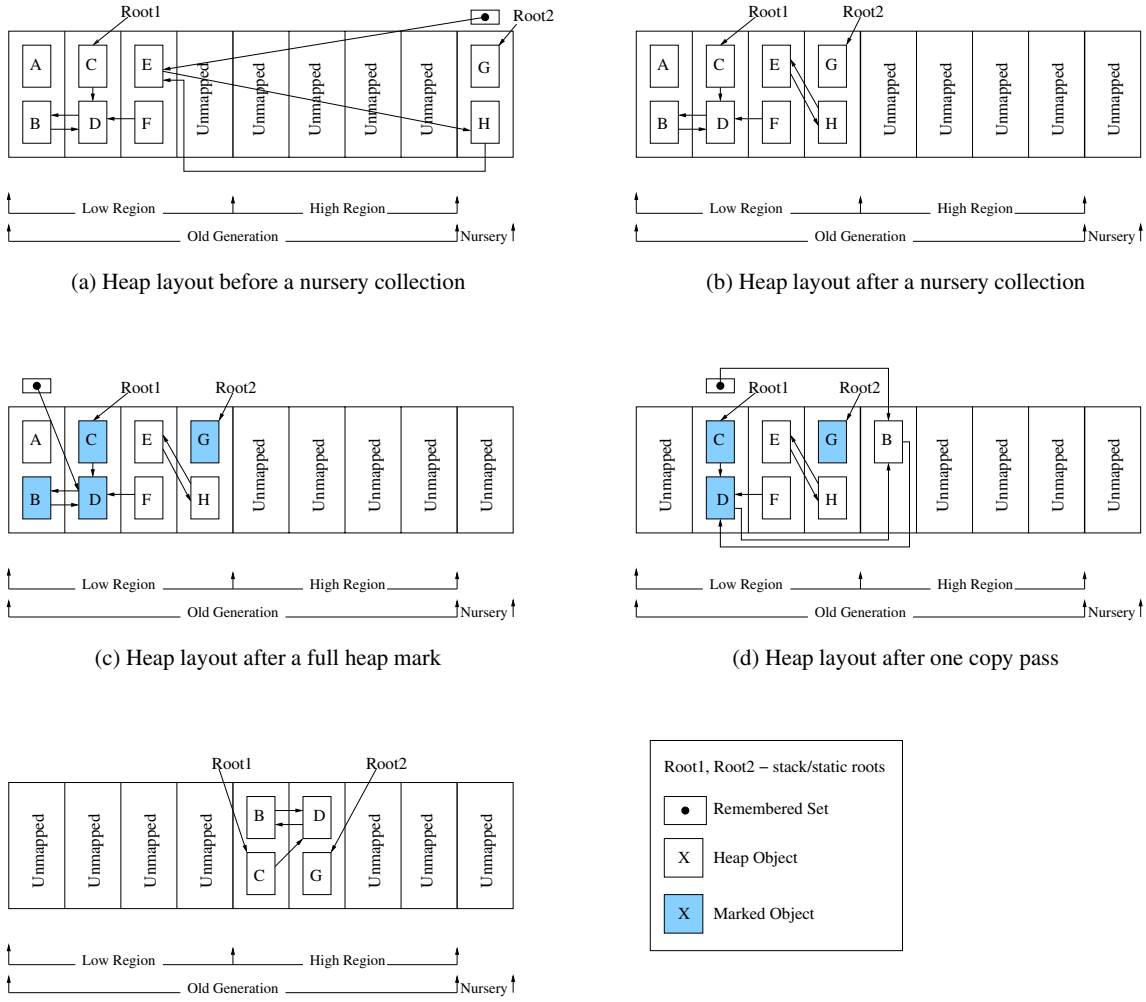



**Figure 1: Mark-Copy: An example of a nursery collection followed by a full collection**

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 the *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 nursery collection followed by a full collection using MC. For this example, we assume that all objects allocated in the heap are the same size, and that the heap can accommodate at most 10 objects. The heap consists of an old generation with 5 windows. Each of these windows can hold exactly 2 objects. Root1 and Root2 are root (stack/static) 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 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. 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 [24] 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 below a series of 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 across multiple collections. We describe the details of these features in Section 4.3.

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 occasionally copy smaller amounts of data (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.

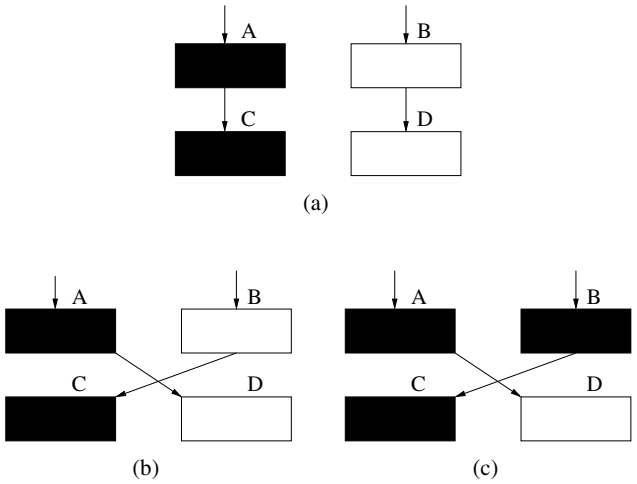



**Figure 2: Example of an error during incremental marking**

## 4.2 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 on 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 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:

[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.

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

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 [14], 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 [31] as *snapshot-at-the-beginning* and *incremental update*. Snapshot collectors [32] 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 [14, 28] 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 3 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(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 3: 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 [24] 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$^2$.)

### *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.

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.

### *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 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.

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 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

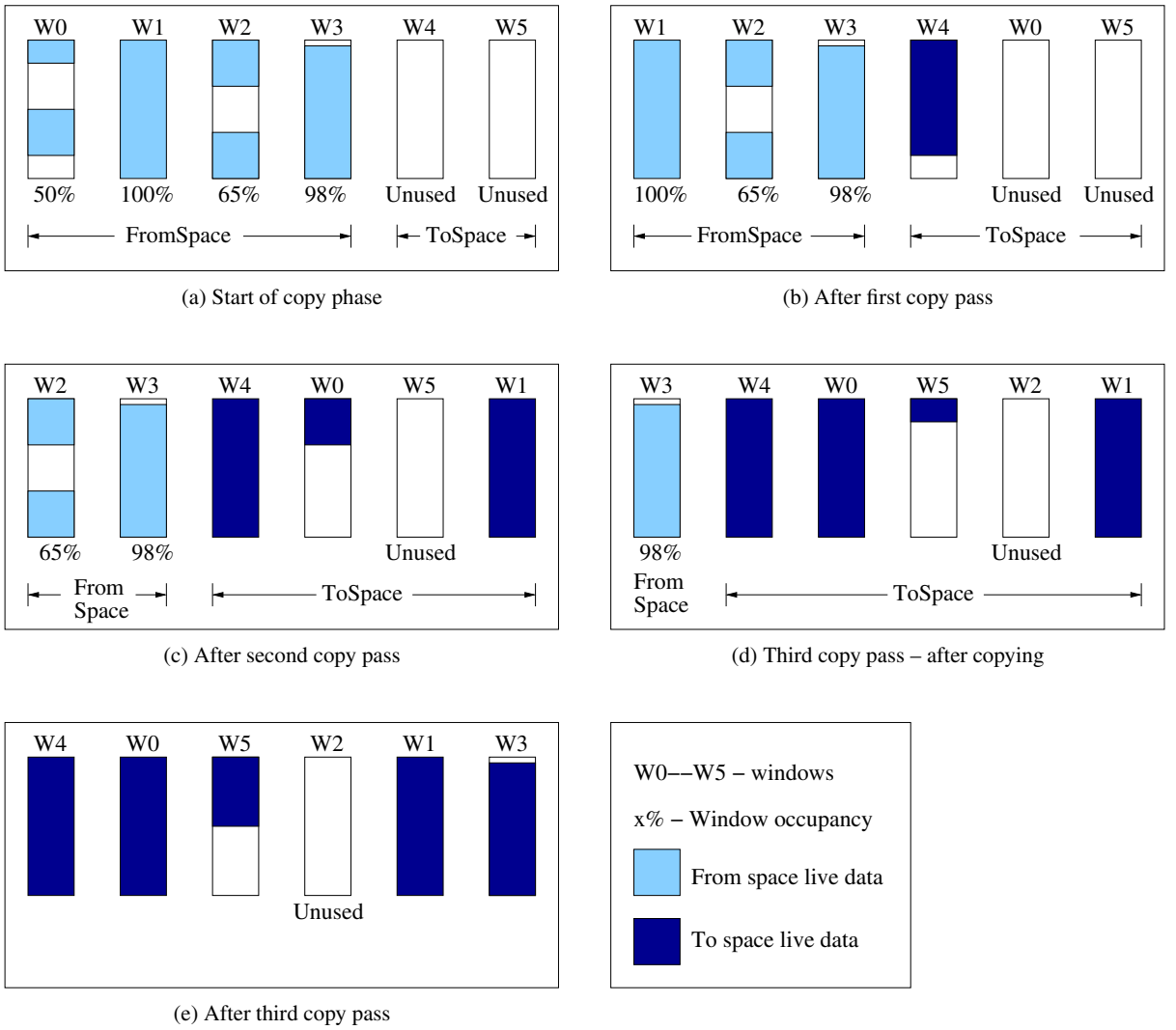



**Figure 4: Stages in the copy phase for MC$^2$**

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 [11]. While scanning to-space objects, 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 4 shows an example of the copy phase of the collector. 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 4(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 4(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 4(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 4(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:

*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.

### *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 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. [3] to manage card tables. While they describe the scheme for the Train collector, it works well with MC$^2$. 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.[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 (about 4.5% of the live data size). 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 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$^2$ 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.

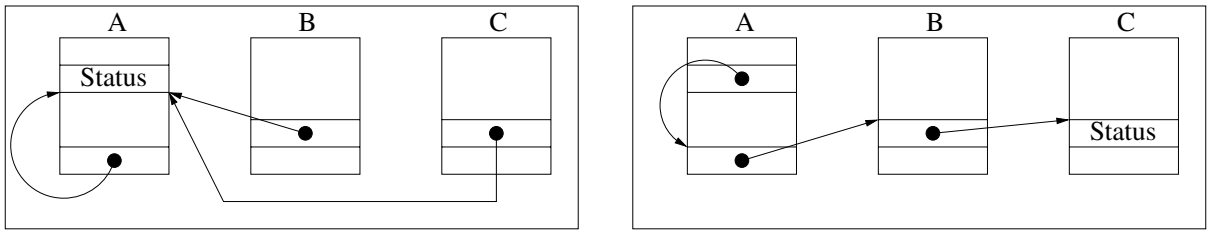
### *Large reference arrays*

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 large objects. When MC$^2$ allocates a large object, 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.

## 4.5 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.

[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.




(a) Before threading

(b) After threading

**Figure 5: An example of pointer threading performed by the Mark-Compact collector**

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.

# 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 (MMTk [8]), standard with Jikes RVM 2.2.3, as the base collector framework. MMTk 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 MMTk 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).

MMTk 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. MMTk 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. 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 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 MMTk 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 [9]: 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 mark-compact generational collector (MSC), based on the threaded algorithm described by Martin [21]. 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 about 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 (Jikes RVM uses a status word in every object that stores lock, hash and

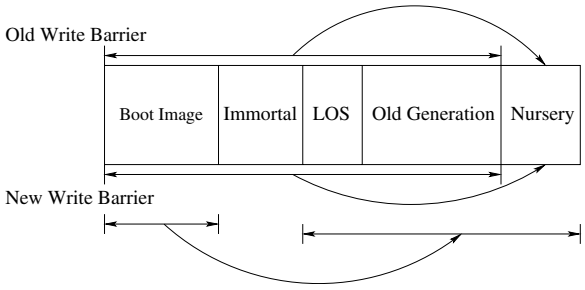



**Figure 6: Heap layout in a MMTk generational collector. The old write barrier records pointers into the nursery from objects outside the nursery. The new write barrier additionally records mutations to boot image objects.**

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

Figure 5 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 and copies the original status word into the header of the new copy. 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 Non-Incremental Collector Improvements*

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

*Boot image remembered sets:* During a full collection, the MMTk MS 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 this is simple to implement and makes the write barrier 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

```
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 7: Original MMTk generational write barrier**

```
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 8: New generational write barrier**

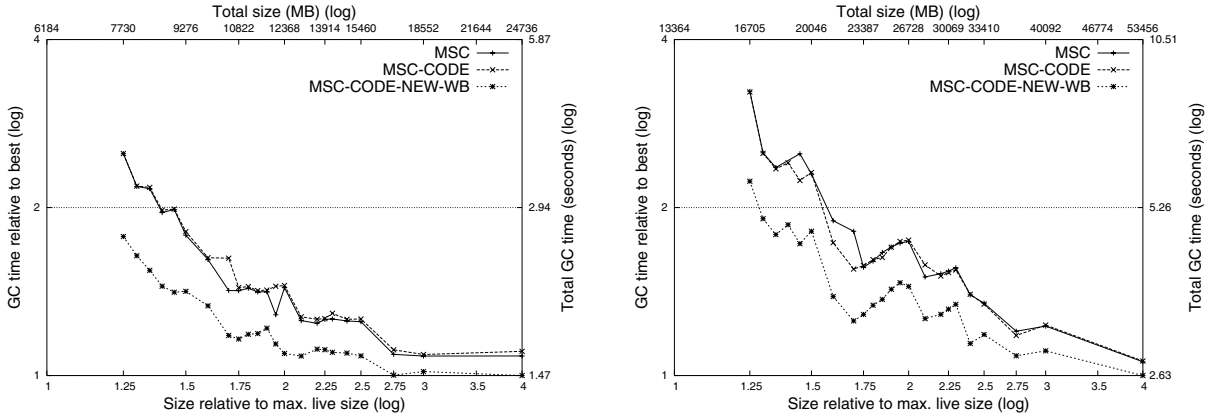
objects, and the collector spends a good portion of the collection time following these pointers. Our measurements show that only about 0.4% of all boot image pointers reference objects in the heap. This issue with boot image scanning has also been discussed by Bacon et al. [4].

We modified the MMTk MS collector to avoid having to scan the boot image during full collections. The modified MS collector uses a new write barrier. Figure 6 shows the layout of the heap in MMTk and the pointers that are recorded by the old and new write barriers. Figure 7 shows the old MMTk MS write barrier and Figure 8 shows the new write barrier (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 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).

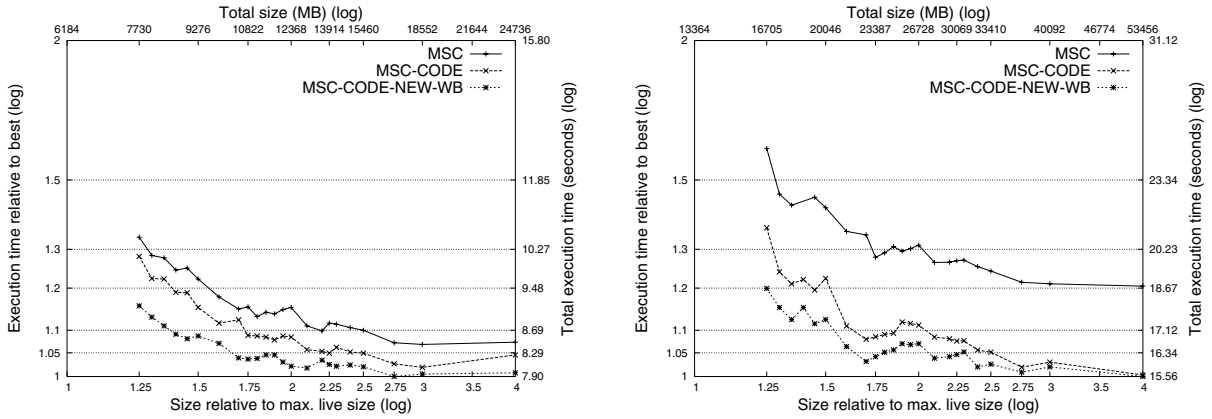
We also used this technique in a modified version of the generational mark-compact collector, and found improvements in execution and pause times. We present in the results section the performance of both versions of MS and MSC.

*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 and MC$^2$ due to poor code locality. We modified the collectors to allocate code and data in different spaces.




(a) jack (b) javac

**Figure 9: 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 GC time significantly.**




(a) jack (b) javac

**Figure 10: 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.**

For MSC, we achieved this by maintaining a separate code space to store code objects which we compact in place. For MC$^2$, we use separate *code windows* to hold code objects. We do not use this technique for MS, since the use of a separate MS 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 the copying collectors due to improved code locality.

Figure 9 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 10 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.

### *5.1.4 MC$^2$*

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

| Benchmark | Description | Maximum Live size (KB) | Total Allocation (MB) |
|---|---|---|---|
| _202_jess | a Java expert system shell | 5872 | 319 |
| _209_db | a small data management program | 12800 | 93 |
| _213_javac | a Java compiler | 13364 | 280 |
| _227_mtrt | a dual-threaded ray tracer | 12788 | 163 |
| _228_jack | a parser generator | 6184 | 279 |
| pseudojbb | SPECjbb2000 with a fixed number of transactions | 30488 | 384 |

**Table 1: Description of the benchmarks used in the experiments**

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.

## 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 space overheads, GC times, total execution times, and pause times for five collectors (MC$^2$, MS, MS with the new write barrier, MSC, and MSC with the new write barrier) across six benchmarks. The MSC collectors and MC$^2$ use separate regions for code and data. Five benchmarks come from the SPECjvm98 suite [26], and the sixth is pseudojbb, a modified version of the SPECjbb2000 benchmark [27]. 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 compute the live size by running each benchmark in the smallest heap for MSC, and recording the largest amount of live data in the old generation after a full collection. (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 value considerably for small benchmarks.)

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 2.4.20-31.9 (with the perfctr patch applied). We performed our experiments with the machine in single user mode to maximize repeatability.

## 6.1 MC$^2$ Space Overheads

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

- 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.
- 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. MC$^2$ coarsens SSB remembered sets when their total size reaches 4.2% of the total space. Table 2 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. The overheads are low, and jess, jack, javac (and mtrt at 1.5 times the live size) 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 3 shows the maximum queue overheads for our benchmark suite.

## 6.2 Execution times and pause times

Figure 11 shows GC times for the collectors relative to the best GC time. Figure 12 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 4 shows the maximum pause times for the collectors and relative execution times for the MS and MSC collectors in a heap that is 1.8 times the program's maximum live size. This is the smallest heap in which 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. Table 5 shows minimum, maximum and geometric means of the pause times across all benchmarks for MC$^2$ in heaps ranging from 1.5–2.5 times the live size. It also shows geometric means of pause times and average relative execution times for MS and MSC. All results are for configurations using a nursery with a maximum size of 1MB and a minimum size of 256KB. MC$^2$ uses 100 physical windows and 30 collection windows in the old generation.

| Benchmark | Heap Size relative to maximum live size | | | | | |
|---|---|---|---|---|---|---|
| | 1.5 | 1.75 | 1.8 | 2 | 2.25 | 2.5 |
| _202_jess | 7.95 | 4.86 | 6.63 | 6.91 | 3.88 | 3.32 |
| _209_db | 1.64 | 1.37 | 1.26 | 1.14 | 1.01 | 0.90 |
| _213_javac | 8.52 | 7.06 | 6.91 | 5.90 | 5.91 | 5.49 |
| _227_mtrt | 4.51 | 3.07 | 2.74 | 3.14 | 2.08 | 0.51 |
| _228_jack | 6.84 | 5.03 | 5.00 | 4.54 | 3.18 | 2.20 |
| pseudojbb | 2.24 | 1.81 | 1.78 | 1.54 | 1.38 | 1.23 |

**Table 2: Remembered set size (without coarsening) as percent of heap size (%), for MC$^2$ using 100 physical windows and 30 collection windows**

| Benchmark | Heap Size relative to maximum live size | | | | | |
|---|---|---|---|---|---|---|
| | 1.5 | 1.75 | 1.8 | 2 | 2.25 | 2.5 |
| _202_jess | 0.61 | 0.53 | 0.51 | 0.48 | 0.41 | 0.37 |
| _209_db | 0.46 | 0.40 | 0.38 | 0.34 | 0.31 | 0.27 |
| _213_javac | 0.70 | 0.57 | 0.54 | 0.32 | 0.47 | 0.43 |
| _227_mtrt | 0.17 | 0.15 | 0.15 | 0.13 | 0.12 | – |
| _228_jack | 0.46 | 0.39 | 0.38 | 0.34 | 0.31 | 0.28 |
| pseudojbb | 0.13 | 0.11 | 0.11 | 0.10 | 0.08 | 0.08 |

**Table 3: Maximum mark queue size as percent of heap size (%), for MC$^2$ using 100 physical windows and 30 collection windows**

| Benchmark | MC$^2$ | MS | | MS (New WB) | | MSC | | MSC (New WB) | |
|---|---|---|---|---|---|---|---|---|---|
| | MPT | MPT | ET | MPT | ET | MPT | ET | MPT | ET |
| _202_jess | 17.15 | 202.20 | 1.16 | 53.15 | 1.11 | 244.53 | 1.06 | 65.69 | 1.04 |
| _209_db | 19.89 | 278.00 | 1.10 | 123.02 | 1.11 | 353.55 | 0.96 | 198.51 | 0.96 |
| _213_javac | 40.39 | 317.04 | 0.96 | 171.88 | 0.92 | 458.09 | 0.92 | 308.95 | 0.89 |
| _227_mtrt | 29.57 | 284.40 | 1.06 | 138.05 | 1.02 | 379.55 | 1.01 | 225.24 | 0.96 |
| _228_jack | 23.89 | 210.04 | 1.07 | 59.69 | 1.03 | 234.04 | 1.03 | 92.89 | 0.99 |
| pseudojbb | 41.47 | 322.97 | 1.12 | 168.20 | 1.08 | 445.99 | 1.01 | 314.53 | 0.98 |
| Geo. Mean | 27.18 | 264.68 | 1.08 | 107.67 | 1.04 | 340.88 | 1.00 | 172.68 | 0.97 |

**Table 4: Maximum Pause Times (MPT, all in milliseconds) and execution times relative to MC$^2$ (ET) for MC$^2$, MS, and MSC, in a heap that is 1.8 times the maximum live size. MSC and MC$^2$ use separate data and code regions. All collectors use a 1MB nursery and MC$^2$ uses 100 physical windows and 30 collection windows.**

| Rel. Heap. | MC$^2$ | | | MS | | MS (New WB) | | MSC | | MSC (New WB) | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Min. MPT | Max. MPT | Mean MPT | Mean MPT | Mean ET | Mean MPT | Mean ET | Mean MPT | Mean ET | Mean MPT | Mean ET |
| 1.50 | 15.97 | 119.47 | 37.92 | 263.43 | 1.06 | 104.65 | 1.00 | 338.04 | 0.97 | 166.85 | 0.92 |
| 1.75 | 16.18 | 48.40 | 26.98 | 261.56 | 1.06 | 102.85 | 1.03 | 338.23 | 0.98 | 165.01 | 0.95 |
| 2.00 | 18.01 | 40.70 | 26.93 | 254.45 | 1.07 | 95.62 | 1.04 | 328.20 | 0.99 | 149.68 | 0.97 |
| 2.25 | 17.50 | 47.46 | 30.11 | 254.96 | 1.06 | 94.24 | 1.04 | 321.15 | 0.99 | 153.59 | 0.97 |
| 2.50 | 18.26 | 48.64 | 27.61 | 255.71 | 1.08 | 99.25 | 1.05 | 334.05 | 0.99 | 163.36 | 0.99 |

**Table 5: Min., max., and geometric mean of max. pause times (milliseconds) across all benchmarks for MC$^2$. Geometric mean of max. pause times (milliseconds) and geometric mean of execution times relative to MC$^2$ across all benchmarks for MS and MSC. For each heap size, we consider only benchmarks that cause invocation of at least one full collection for all collectors. MSC and MC$^2$ use separate data and code regions. All collectors use a 1MB nursery and MC$^2$ uses 100 physical windows and 30 collection windows.**

The execution times for MSC with the new write barrier are almost always lower than MSC with the boot image scan. The GC plots show that a significant reduction in GC time causes the performance improvement. Only for db is performance with the new barrier slightly worse in moderate and large heaps. This is because db mutates a large number of pointers into the old generation (90% of all stores), causing invocation of the slow path barrier each time. The old barrier ignores these writes since it records only pointers into the nursery.

Similarly, for MS the performance with the new barrier is usually better in small and moderate size heaps. As with MSC, the performance with the new barrier is slightly worse for db in moderate and large heaps. In the following discussion, we consider MS and MSC only with the new barrier.

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

MC$^2$ has lower GC times than MS for jess and jack. For all other benchmarks, GC times for MC$^2$ are slightly worse than those for MS. However, the overall performance of MC$^2$ is usually better due to improved locality.

For all benchmarks, the performance for MC$^2$ is usually within 5% of of MSC in heaps that are 1.5–1.8 times the program live size or larger. javac again is an exception and MC$^2$ is within 5-6% of MSC only in heaps that are twice the live size or larger.

MSC performs better than the other collectors for a couple of reasons. First, the GC cost for MSC is almost always the lowest. Second, the collector preserves allocation order, which yields better locality and thus lower mutator times.

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 copy large portions of data. It compacts a smaller portion of the heap, which contains transient data. In spite of copying little data, MC$^2$ does not have lower GC times than MSC, because 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 MSC does not move any live data at the start of the heap.

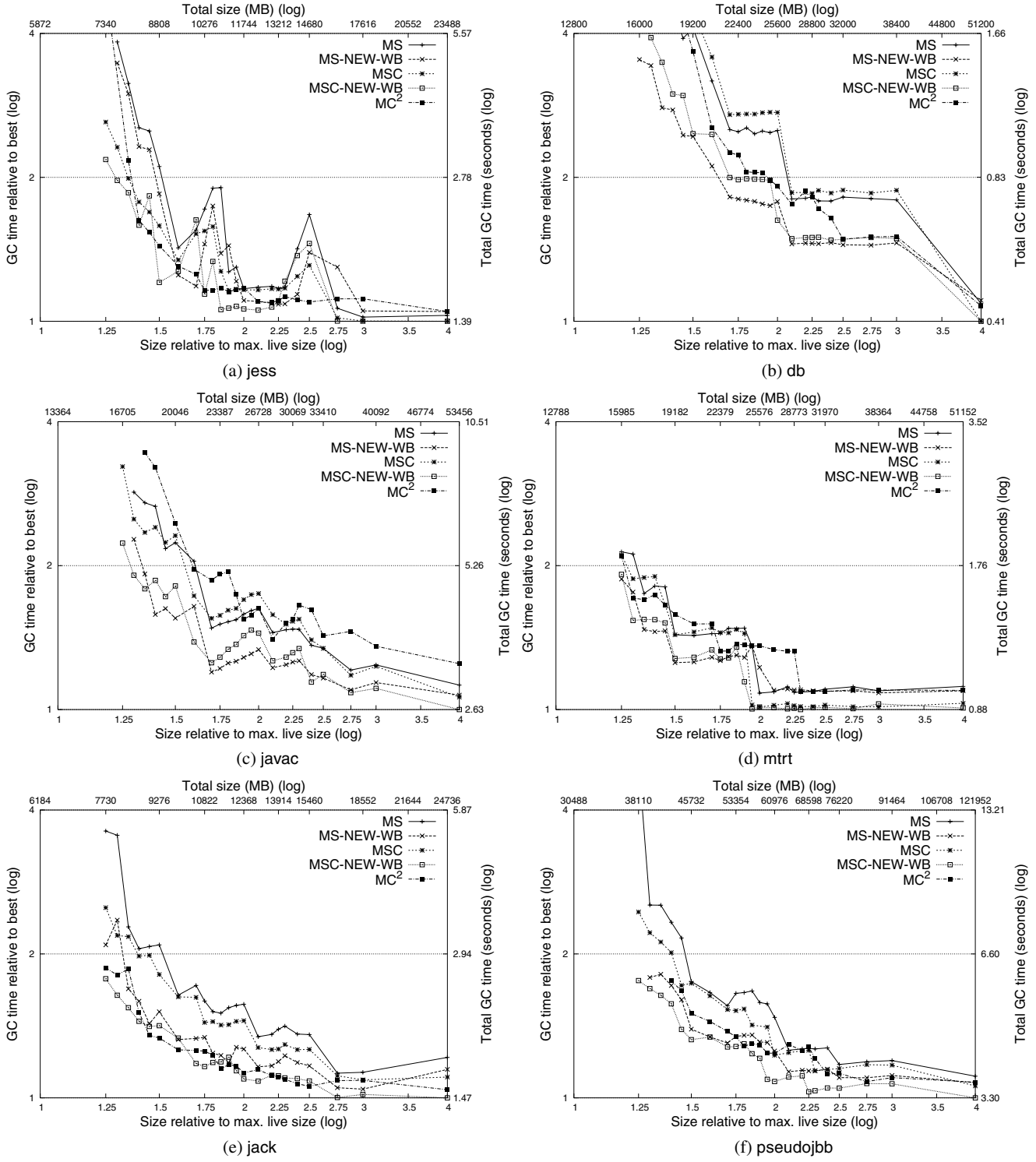



**Figure 11: GC time relative to best GC time for MS, MSC, and MC$^2$**

The data in table 4 show that for all benchmarks, the maximum pause times for MC$^2$ are significantly lower than those for MS and MSC, even in a tight heap. The geometric mean of the pause times for MC$^2$ is 27.18ms. The mean pause time for the MMTk MS collector is 264.68, almost a factor of 10 higher, with a mean performance degradation of 8%. MS with the new write barrier performs much better, but has a mean pause time that is a factor of 4 higher, with a mean performance degradation of 4%.

MSC with the boot image scan performs as well as MC$^2$ on average, with an average pause time that is almost a factor of 12 higher. With the new write barrier, MC$^2$ is 3% slower on average but its pause time is up to an order magnitude lower than MSC, with an average

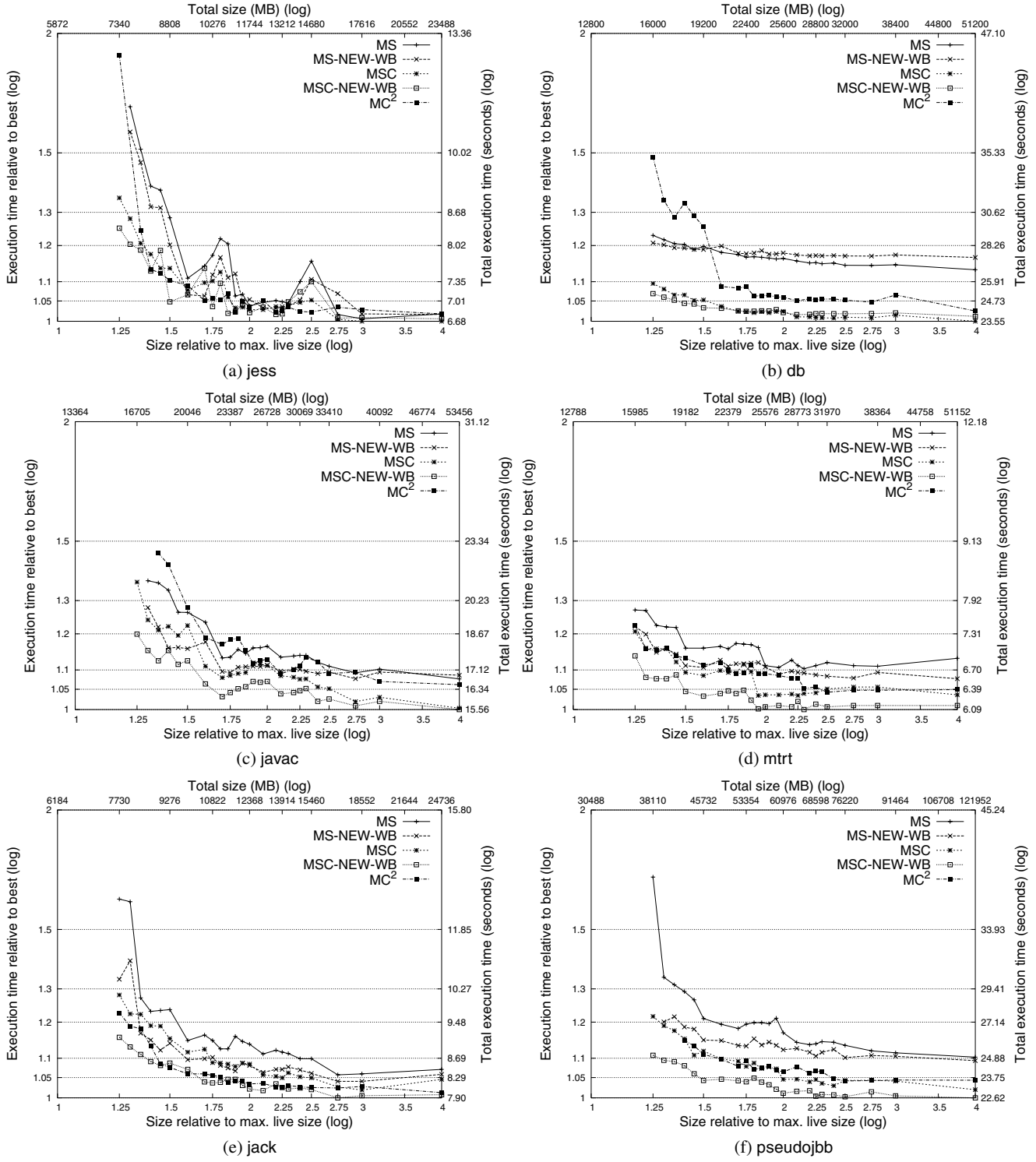



**Figure 12: Total execution time relative to best total execution time for MS, MSC, and MC$^2$**

pause time that is a factor of 6 lower. The only benchmark for which the performance of MC$^2$ is significantly worse is javac. MC$^2$ requires a slightly larger heap for javac and is usually about 5% worse than MSC in heaps that are twice the live size or larger.

Table 5 summarizes results for the collectors in heaps from 1.5–2.5 times the live size. In a heap that is 1.5 times the live size, MC$^2$ is about 8% slower than MSC on average and has a maximum pause time of 119.47ms (this long pause occurs for javac because a few copying passes are clustered together). In heaps between 1.75–2.5 times the live size, the maximum pause time for MC$^2$ is under 50ms for all benchmarks. The average pause time varies from 27–30ms. MC$^2$ is 3–5% faster than MS and 1–5% slower than MSC on average.

We also compared the performance of MC$^2$ (using SSBs with coarsening) with a version of 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 in all cards that contain an object into the window being collected, which can be more expensive.

**Summary:** The results show that MC$^2$ can obtain low pause times and good throughput in constrained heaps. The average pause time for MC$^2$ is 27.18ms in a heap that is 1.8 times the program live size. Importantly, the execution times for 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.

## 6.3 Pause time distribution

Figure 13 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 (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. 80% of all pauses for javac are 10ms or less, and 92% of pauses for pseudojbb are in the 0–10ms range. (For jess, db, mtrt and jack, pauses that are 10ms or less account for 97%, 93%, 87% 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 41ms 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 19ms. The longest copy pause time (41ms) is for pseudojbb, and 88% 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$^2$.

## 6.4 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, 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 [12]. 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 14 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.

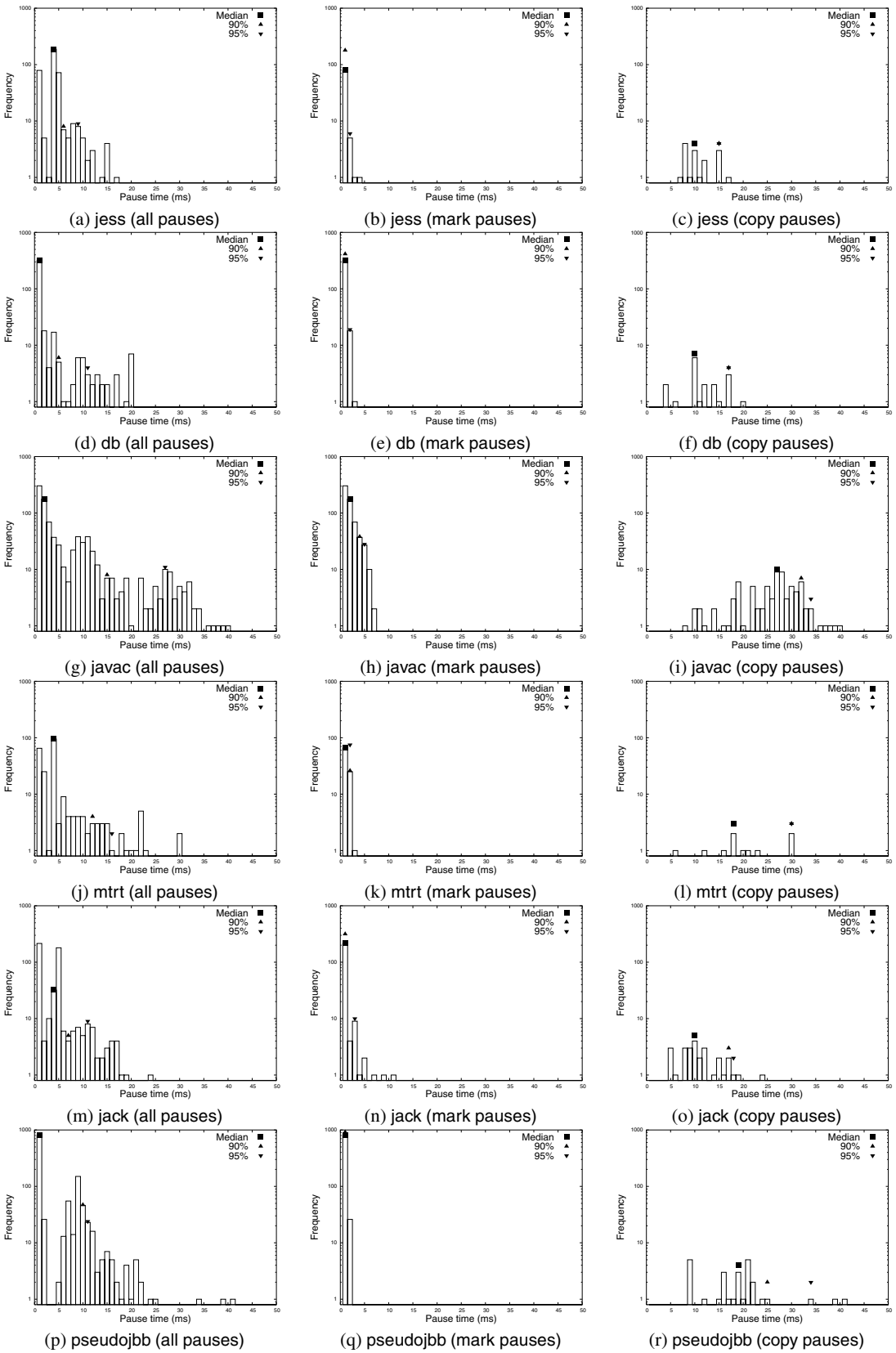
The three curves in each graph are for MC$^2$, MS, and MSC (all for versions using the new write barrier). The curve with the smallest x-intercept is for MC$^2$. MSC has the curve with the largest x-intercept; the MS curve has an x-intercept in between the other two.

For all benchmarks, MC$^2$ allows some mutator utilization 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 higher utilization than 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 MSC, utilization is higher for windows up to 7s, 3s, 800ms, 500ms, 400ms and 300ms for jess, javac, pseudojbb, mtrt, db, and jack respectively.

Beyond that, the utilization provided by 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 MC$^2$ is lower than MSC for windows larger than 300ms but overall utilization is close. Only for javac does MC$^2$ provide significantly lower overall 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 show that 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 utilization provided by all collectors tends to be about the same.




(a) jess (all pauses) (b) jess (mark pauses) (c) jess (copy pauses)

(d) db (all pauses) (e) db (mark pauses) (f) db (copy pauses)

(g) javac (all pauses) (h) javac (mark pauses) (i) javac (copy pauses)

(j) mtrt (all pauses) (k) mtrt (mark pauses) (l) mtrt (copy pauses)

(m) jack (all pauses) (n) jack (mark pauses) (o) jack (copy pauses)

(p) pseudojbb (all pauses) (q) pseudojbb (mark pauses) (r) pseudojbb (copy pauses)

**Figure 13: 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 9ms or less. Copy phase pauses (third column) are longer (4–41ms).**

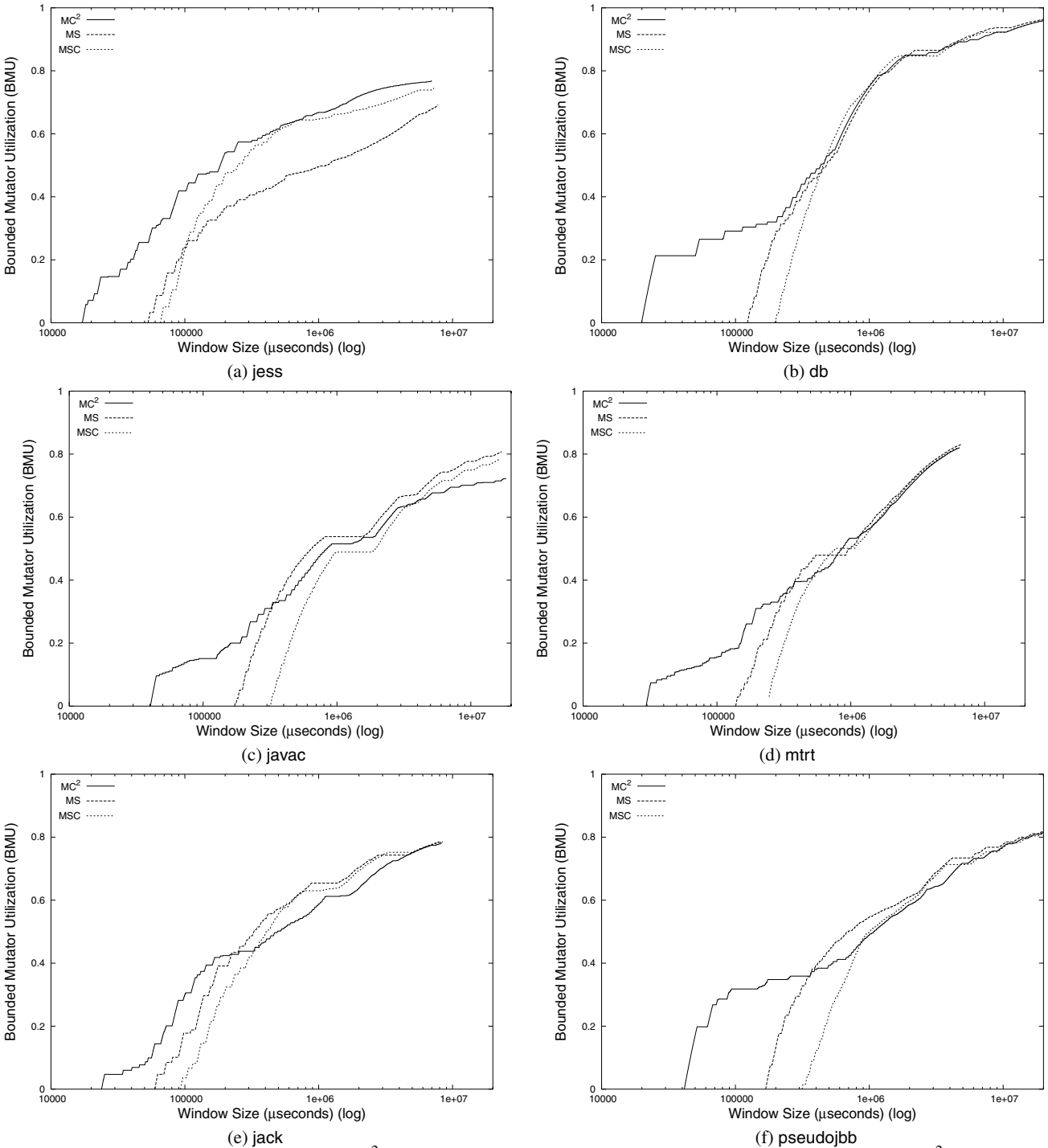



**Figure 14: Bounded mutator utilization for MC$^2$, MS, and MSC in a heap that is 1.8 times the maximum live size. MC$^2$ provides better utilization in small windows (usually up to 300ms) due to lower pause times and good pause distribution.**

# 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 significantly lower than those for MSC and MS in constrained memory.

## 8. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under grant number CCR-0085792. Additionally, Emery Berger was supported by NSF CAREER award CNS-0347339. 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 MMTk 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):211–238, February 2000.

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

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

[5] 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, pages 285–298, New Orleans, LA, January 2003. ACM Press.

[6] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978. Also AI Laboratory Working Paper 139, 1977.

[7] Ori Ben-Yitzhak, Irit Goft, Elliot Kolodner, Kean Kuiper, and Victor Leikehman. An algorithm for parallel incremental compaction. In ISMM '02 [17], pages 100–105.

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

[9] Stephen M. Blackburn and Kathryn S. McKinley. In or out? Putting write barriers in their place. In ISMM '02 [17], pages 175–184.

[10] 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.

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

[12] 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.

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

[14] 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.

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

[16] 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*, pages 388–403, University of Massachusetts, USA, 16–18 September 1992. Springer-Verlag.

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

[18] 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*, pages 26–36, October 1997.

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

[20] 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.

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

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

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

[24] 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, pages 326–343, Anaheim, CA, November 2003. ACM Press.

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

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

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

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

[29] 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.

[30] 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.

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

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