Garbage Collection Without Paging

Matthew Hertz Yi Feng Emery D. Berger

Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
{hertz, yifeng, emery}@cs.umass.edu

Abstract

Garbage collection offers numerous software engineering advantages, but interacts poorly with virtual memory managers. Existing garbage collectors require far more pages than the application's working set and touch pages without regard to which ones are in memory, especially during full-heap garbage collection. The resulting paging can cause throughput to plummet and pause times to spike up to seconds or even minutes. We present a garbage collector that avoids paging. This *bookmarking collector* cooperates with the virtual memory manager to guide its eviction decisions. Using summary information ("bookmarks") recorded from evicted pages, the collector can perform in-memory full-heap collections. In the absence of memory pressure, the bookmarking collector matches the throughput of the best collector we tested while running in smaller heaps. In the face of memory pressure, it improves throughput by up to a factor of five and reduces pause times by up to a factor of 45 over the next best collector. Compared to a collector that consistently provides high throughput (generational mark-sweep), the bookmarking collector reduces pause times by up to 218x and improves throughput by up to 41x. Bookmarking collection thus provides greater utilization of available physical memory than other collectors while matching or exceeding their throughput.

Categories and Subject Descriptors D.3.4 [*Processors*]: Memory management (garbage collection); D.4.2 [*Storage Management*]: Virtual memory

General Terms Algorithms, Languages, Performance

Keywords bookmarking collection, garbage collection, generational collection, memory pressure, paging, virtual memory

1. Introduction

Garbage collection is a primary reason for the popularity of languages like Java and C# [45, 52]. However, garbage collection requires considerably more space than explicit memory management [56, 31]. The result is that fewer garbage-collected applications fit in a given amount of RAM. If even one garbage-collected application does not fit in available physical memory, the garbage collector will induce *paging*, or traffic between main memory and the disk. Because disk accesses are approximately six orders of magnitude more expensive than main memory accesses, paging significantly degrades performance. Paging can also lead to pause times lasting for tens of seconds or minutes.

Even when an application's working set fits in main memory, collecting the heap may induce paging. During full-heap collections, most existing garbage collectors touch pages without regard to which pages are resident in memory and visit many more pages than those in the application's working set. Garbage collection also disrupts information about the reference history tracked by the virtual memory manager.

While this phenomenon is widely known, previous work has attacked it only indirectly. For example, generational garbage collectors concentrate their collection efforts on short-lived objects [37, 49]. Because these objects have a low survival rate, generational collection reduces the frequency of full-heap garbage collections. However, when a generational collector eventually performs a full-heap collection, it triggers paging.

This problem has led to a number of workarounds. One standard way to avoid paging is to size the heap so that it never exceeds the size of available physical memory. However, choosing an appropriate size statically is impossible on a multiprogrammed system, where the amount of available memory changes. Another possible approach is overprovisioning systems with memory, but high-speed, high-density RAM remains expensive. It is also generally impractical to require that users purchase more memory in order to run garbage-collected applications. Furthermore, even in an overprovisioned system, just one unanticipated workload exceeding available memory can render a system unresponsive. These problems have led some to recommend that garbage collection only be used for small applications with minimal memory footprints [46].

Contributions: This paper introduces *bookmarking collection* (BC), a garbage collection algorithm that virtually eliminates garbage collector-induced paging. Bookmarking collection records summary information ("bookmarks") about outgoing pointers from pages that have been evicted to disk. Once a page is evicted from memory, BC does not touch it unless the application itself makes it resident. Instead of visiting evicted pages, BC uses bookmarks to assist garbage collection. These bookmarks, together with BC's heap organization and cooperation with the virtual memory manager, allow BC to perform full-heap, compacting garbage collection without paging, even when large portions of the heap have been evicted. Because bookmarking necessarily sacrifices some connectivity information and could thus prevent garbage from being reclaimed (see Section 3.5), BC includes a fail-safe mechanism that preserves completeness by discarding bookmarks in the unlikely event of heap exhaustion.

PLDI'05, 12–15 June 2005, Chicago, Illinois, USA

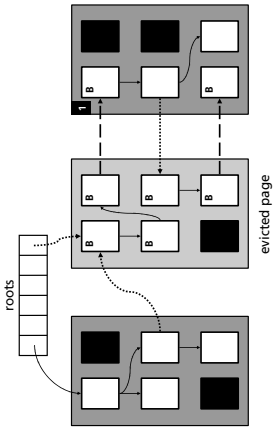


Figure 1. An example of bookmarking collection.

We have implemented BC in Jikes RVM [7, 8] using the MMTk toolkit [19]. The bookmarking collector relies on some additional operating system support, which consists of a modest extension to the Linux virtual memory manager (approximately six hundred lines of code). Without memory pressure, BC's performance matches that of generational mark-sweep (GenMS), a collector that consistently provides high throughput [18]. Under memory pressure, bookmarking collection outperforms the next best garbage collector we tested by a factor of five, and reduces pause times by a factor of 45. Compared to GenMS, bookmarking collection yields up to a 218-fold reduction in pause times and improves throughput by up to 41x. Bookmarking collection thus provides greater utilization of available physical memory than other collectors.

The paper is organized as follows: Section 2 provides an overview of bookmarking collection, and Section 3 describes the bookmarking garbage collector in detail. Section 4 presents some key implementation details, including our extensions to the Linux virtual memory manager. Section 5 presents empirical results comparing the performance of bookmarking collection to a range of existing garbage collectors, both with and without memory pressure. Section 6 discusses related work, Section 7 presents directions for future work, and Section 8 concludes.

2. Overview

The bookmarking collector was designed to achieve three goals: low space consumption, high throughput, and the elimination of GC-induced page faults. While BC normally uses mark-sweep collection, it minimizes space consumption by performing compaction when under memory pressure. BC provides high throughput by using a nursery generation to manage short-lived objects. Finally, and most importantly, it organizes the heap into groups of pages and reacts to signals from the virtual memory manager whenever pages are scheduled for eviction to disk or made resident in main memory.

Unlike existing garbage collectors, BC avoids paging caused by processing objects on evicted pages. BC scans pages prior to eviction and remembers outgoing pointers by "bookmarking" targeted objects. When BC is notified of an impending eviction and cannot shrink the heap or discard an empty page, BC ensures that only appropriate pages are evicted. BC scans each object on the victim page, bookmarks the targets of any references, and increments counters in the target superpages' headers. After processing all of the objects, BC informs the virtual memory manager that the page can be evicted. Because BC uses these bookmarks to recreate the evicted references, it can continue to collect the heap without paging.

Figure 1 presents a snapshot of the heap after a collection with bookmarks. Objects are the small rectangles inside larger rectangles, which are pages. Live objects are shown in white, and unreachable objects (garbage) are shown in black. The middle page (in light gray) has been evicted to disk. The lines represent pointers. The dotted lines are pointers into non-resident pages; BC ignores these during collection. The dashed lines denote outgoing pointers from a non-resident page: these induce bookmarks, represented by the letter "B". BC also conservatively bookmarks all objects on a page before it is evicted (see Section 3.4). The number "1" in the upper-left hand corner of the last page indicates that the bookmarks on this page are induced by exactly one evicted page. During collection, BC considers every bookmarked object to be live, even if it is not reachable from the roots.

3. Bookmarking Collection

The bookmarking collector is a generational collector with a bump-pointer nursery, a compacting mature space, and a page-based large object space. BC divides the mature space into *superpages*, page-aligned groups of four contiguous pages (16K). BC manages mature objects using *segregated size classes* [12, 13, 16, 36, 38, 42, 53]: objects of different sizes are allocated onto different superpages. Completely empty superpages can be reassigned to any size class [16, 38].

BC uses size classes designed to minimize both internal and external fragmentation (which we bound at 25%). Each allocation size up to 64 bytes has its own size class. Larger object sizes fall into a range of 37 size classes; for all but the largest five, these have a worst-case internal fragmentation of 15%. The five largest classes have between 16% and 33% worst-case internal fragmentation; BC could only do better by violating the bound on page-internal or external fragmentation. BC allocates objects larger than 8180 bytes (half the size of a superpage minus metadata) into the large object space.

When the heap fills, BC typically performs mark-sweep garbage collection. We use mark-sweep for two reasons. First, it provides good program throughput and short GC pauses. More importantly, mark-sweep does not increase memory pressure by needing a copy reserve of pages.[1]

3.1 Managing Remembered Sets

Like all generational collectors, BC must remember pointers from the older to the younger generation. It normally stores these pointers in page-sized write buffers that provide fast storage and processing but may demand unbounded amounts of space. To limit space overhead, BC processes buffers when they fill. During this processing, it removes entries for pointers from the mature space and instead marks the card for the source object in the card table used during the marking phase of garbage collection. BC begins each nursery collection by reviewing these cards and scanning only those objects whose cards are marked. After pruning all possible pointers and compacting the entries remaining in the buffer, the filtered slots are available for future storage. This filtering allows the bookmarking collector to use the fast processing of write buffers, but often consumes just a single page.

3.2 Compacting Collection

Because mark-sweep collection does not compact the heap, fragmentation can cause it to increase memory pressure. Using mark-sweep, the bookmarking collector cannot reclaim a superpage if it contains just one reachable object. BC avoids increased memory pressure by performing a two-pass compacting collection whenever

[1] Usually this copy reserve is half the heap size, but Sachindran et al. present copying collectors that allow the use of much smaller copy reserves [43, 44].

a full garbage collection does not free enough pages to satisfy the current allocation request.

BC begins this compacting collection with a marking phase. Each time it marks an object, it also increments a counter for the object's size class. After marking, BC computes the minimum number of superpages needed to hold the marked objects for each size class. It then selects the minimum set of "target" superpages that contain enough space to hold all of the marked objects. A second pass uses a Cheney scan to compact the reachable objects. Upon visiting an object in this second pass, BC checks if the object is on a target superpage. BC forwards those objects not on target superpages, while preserving objects already on a target. When this pass completes, reachable objects are only on target superpages and the remaining garbage (non-target) superpages are freed.

3.3 Cooperation with the Virtual Memory Manager

The approach described so far allows BC to avoid increasing memory pressure during garbage collection. In the face of paging, BC cooperates with the virtual memory manager to shrink the heap and reduce memory pressure. The bookmarking collector uses its knowledge of the heap to make good paging decisions. For this cooperation, we use an extended virtual memory manager; we describe these extensions in Section 4.1.

3.3.1 Reducing System Call Overhead

To limit overhead due to communication with the virtual memory manager, BC tracks page residency internally. Whenever BC allocates a new superpage, it checks in a bit array if the pages are already memory resident. When they are not, it increases the estimate of the current footprint and marks the pages as resident. During garbage collection, the collector uses this bit array to avoid following pointers into pages that are not resident.

3.3.2 Discarding Empty Pages

The virtual memory manager initiates communication by sending a signal whenever a page is scheduled for eviction or loaded back into memory. Upon receiving this signal, BC scans the bit array for an empty, memory-resident page and directs the virtual memory manager to reclaim this *discardable* page. When a discardable page cannot be found, BC triggers a collection and then directs the virtual memory manager to discard a newly-emptied page (if one exists). Most operating systems (e.g., Linux and Solaris) already include this functionality by using the `madvise` system call with the `MADV_DONTNEED` flag.

3.3.3 Keeping the Heap Memory-Resident

The notification of a pending eviction also alerts BC that the current heap footprint is slightly larger than the memory available to the application. Unlike previous collectors, BC tries not to grow at the expense of paging, but instead limits the heap to the current footprint. If memory pressure continues to increase, BC continues discarding empty pages and uses the new estimate as the target footprint. BC shrinks the heap to keep it entirely in memory and thus avoid incurring the cost of a page fault. While BC expands the heap and causes pages to be evicted when this is necessary for program completion, it ordinarily limits the heap to what can fit into available memory.

3.4 Bookmarking

When a non-discardable page must be evicted, BC selects a victim page. The page scheduled for eviction is usually an appropriate choice, since the virtual memory manager approximates LRU order and will therefore evict pages that are unlikely to be used again soon. However, BC will not select pages that it knows will soon be used, such as nursery pages or superpage headers (which we discuss below). In this case, upon processing the signal, BC touches the page that has been scheduled in order to prevent its eviction. This touching causes a different victim page to be scheduled for eviction.

If collecting the heap does not yield any discardable pages, BC scans the victim page for outgoing pointers and *bookmarks* the target objects of these outgoing references. These bookmarks (a single bit stored in the status word in the object header) act as a secondary set of root references, allowing full memory-resident collections without accessing evicted pages and thus without causing any page faults.

In addition to setting the bookmark bit of all of the target objects, BC increments the *incoming bookmark counter* for the target objects' superpages. BC uses this counter (the number of evicted pages pointing to objects on a given superpage) to release bookmarks when incoming pages become resident (we discuss clearing bookmarks in Section 3.4.2).

BC stores superpage metadata in each superpage header. This placement permits constant-time access by bit-masking, which is important because the metadata is used both for allocation and collection. While this metadata could instead be stored off to the side, that would create a large pool of unevictable pages, including information for superpages that do not exist. While storing the metadata in the superpage header prevents BC from evicting one-fourth of the pages, it reduces memory overhead and simplifies the memory layout, along with corresponding eviction/reloading code.

Since superpage headers are always resident, BC can increment the incoming bookmark counters without triggering a page fault. BC cannot bookmark every target, however, since these may reside on evicted pages. To prevent this from causing errors, BC conservatively bookmarks all objects on a page before it is evicted.

Once bookmarking completes, the victim page can be evicted. Having just been touched, however, it would not now be scheduled for eviction. BC thus communicates with the virtual memory manager one more time, informing it that the page should be evicted. While the stock Linux virtual memory manager does not provide support for this operation, BC uses a new system call provided by our extended kernel, `vm_relinquish`. This call allows user processes to voluntarily surrender a list of pages. The virtual memory manager places these relinquished pages at the end of the inactive queue from which they are quickly swapped out.

A race condition could arise if the relinquished pages are touched before the virtual memory manager can evict them. While BC would still have processed these pages for bookmarks, they would never be evicted and BC would never be informed that they are memory-resident. To eliminate this possibility, BC prevents pages from being accessed immediately after scanning it by disabling access to them (via the `mprotect` system call). When a protected page is next touched, the virtual memory manager notifies BC. In addition to re-enabling page access, BC can clear bookmarks (see Section 3.4.2).

3.4.1 Collection After Bookmarking

When the heap is not entirely resident in memory, BC starts the marking phase of each full-heap collection by scanning the heap for memory-resident bookmarked objects. While this scan is expensive, scanning every object is often much smaller than the cost of even a single page fault. BC further reduces this cost by scanning only those superpages with a nonzero incoming bookmark count. During this scan, BC marks and processes bookmarked objects as if they were root-referenced. Once BC completes this scan, it has reconstructed all the references in evicted objects and does not need to touch evicted pages. BC now follows its usual marking phase, but ignores references to evicted objects. After marking completes,

all reachable objects are either marked or evicted. A sweep of the memory-resident pages completes the collection.

When BC must compact the heap, slightly more processing is required. During marking, BC updates the object counts for each size class to reserve space for every possible object on the evicted pages. BC first selects all superpages containing bookmarked objects or evicted pages as compaction targets, selecting other superpages as needed.

BC scans the heap to process bookmarked objects once more at the start of compaction. Because these bookmarked objects reside on target superpages, BC does not move them. BC thus does not need to update (evicted) pointers to bookmarked objects.

3.4.2 Clearing Bookmarks

BC largely eliminates page faults caused by the garbage collector, but cannot prevent mutator page faults. As described in Section 3.4, BC is notified whenever the mutator accesses an evicted page because this triggers a protection fault. BC then tries to clear the bookmarks it set when the page was evicted.

BC scans the reloaded page's objects and decrements the incoming bookmark counter of any referenced objects' superpages. When a superpage's counter drops to zero, its objects are only referenced by objects in main memory. BC then clears the now-unnecessary bookmarks from that superpage. If the reloaded page's superpage also has an incoming bookmark count of zero, then BC clears the bookmarks that it set conservatively when the page was evicted.

3.4.3 Complications

So far, we have described page eviction as if the kernel schedules evictions on a page-by-page basis and maintains a constant number of pages in memory. In fact, the virtual memory manager schedules page evictions in large batches to hide disk latency. As a result, the size of available memory can fluctuate wildly. The virtual memory manager also operates asynchronously from the collector, meaning that it can run ahead of the collector and evict a page before BC can even be scheduled to run and process the page.

BC uses two techniques to avoid this problem. First, it maintains a store of empty pages and begins a collection when these are the only discardable pages remaining. If pages are scheduled for eviction during a collection, BC discards the pages held in reserve. When a collection does not free enough pages to replenish its empty page cache, BC examines the pages that had been scheduled for eviction and, processes and evicts any non-empty pages. This preventive bookmarking ensures BC always maintains some pages in memory in which it can allocate objects and ensures BC can process pages before their eviction.

BC employs a second technique to ensure it can process pages before they are evicted. Rather than discarding pages individually, BC discards all contiguous empty pages recorded on the same word in its bit array as the first discardable page it finds. By tracking the number of extra pages it returns to the virtual memory manager, BC prevents this aggressive discarding from decreasing its target memory footprint. This aggressiveness has other benefits. Providing more empty pages for reuse limits the number of pages the virtual memory manager must schedule for eviction and provides better program throughput by reducing the time BC spends handling these notifications.

3.5 Preserving Completeness

The key principle behind bookmarking collection is that the garbage collector must avoid touching evicted pages. A natural question is whether *complete* garbage collection (reclaiming all garbage objects) is possible without touching *any* evicted pages.

In general, it is impossible to perform complete garbage collection without traversing non-resident objects. Consider the case when we must evict a page full of one-word objects that all point to objects on other pages. A lossless summary of reachability information for this page requires as much memory as the objects themselves, for which we no longer have room.

This limitation means that, in the absence of other information about reachability, we must rely on conservative summaries of connectivity information. Possible summaries include summarizing pointers on a page-by-page basis, compressing pointers in memory, and maintaining reference counts. To minimize the space and processing required for summaries, we use the smallest summarization heuristic: a single bookmark bit already available in the object's header. When this bit is set, BC treats the object as the target of at least one pointer from an evicted page.

While this summary information is "free", bookmarking has a potential space cost. Because BC must select all superpages containing bookmarked objects and evicted pages as targets, compacting collection cannot always minimize the size of the heap. By treating all bookmarked objects as reachable (even ones which may be garbage), BC is further limited in how far it can shrink the heap.

Despite these apparent costs, bookmarking does not substantially increase the minimum heap size that BC requires. Section 5.3 shows that even in the face of megabytes of evicted pages, BC continues to run in very tight heap sizes. Such tight heaps mean that BC will perform more frequent garbage collections, but the time needed for these collections is still far less than the cost of the page faults that would otherwise occur.

In the event that the heap is exhausted, BC preserves completeness by performing a full heap garbage collection (touching evicted pages). Note that this worst-case situation for bookmarking collection (which we have yet to observe) is the common case for existing garbage collectors. We show in Section 5 that BC's approach effectively eliminates collector-induced paging.

4. Implementation Details

We implemented the bookmarking collector using MMTk [19] and Jikes RVM version 2.3.2 [7, 8]. When implementing the BC algorithm within MMTk and Jikes, we needed to make two minor modifications. In Jikes, object headers for scalars are found at the end of the object while object headers for arrays are placed at the start of an object.[2] This placement is useful for optimizing NULL checks [7], but makes it difficult for BC to find object headers when scanning pages. We solve this problem by further segmenting our allocation to allow superpages to hold either only scalars or only arrays. BC stores the type of objects contained in the superpage in each superpage header. Using this type and size class information in the superpage header (accessed by bit-masking) allows BC to locate all objects on a page.

BC does not employ two of the object layout optimizations included in Jikes RVM. Jikes normally aligns objects along word (4-byte) boundaries, but allocates objects or arrays containing longs or doubles on an 8-byte boundary. Aligning data this way improves performance, but makes it impossible for BC to locate object headers. BC would need to know an object's type to find its header, but the type is itself stored in the header. Another optimization allows Jikes RVM to compute an object's hash code based upon its address. While this provides many benefits, address-based hashing also requires that copied objects grow by one word, disrupting the size-class object counts that BC maintains.

While we needed to remove these optimizations from BC builds, we did not want to bias our results by removing them from builds that would benefit from their presence. Therefore, all builds except BC include these optimizations.

[2] In a recently-released version of Jikes RVM, all headers start at the beginning of the object.

4.1 Kernel Support

The bookmarking collector improves garbage collection paging performance primarily by cooperating with the virtual memory manager. We extended the Linux kernel to enable this cooperative garbage collection. This extension consists of changes or additions to approximately six hundred lines of code (excluding comments), as measured by SLOCcount [51].

The modifications are on top of the 2.4.20 Linux kernel. This kernel uses an approximate global LRU replacement algorithm. User pages are either kept in the active list (managed by the clock algorithm) or the inactive list (a FIFO queue).

When the application begins, it registers itself with the operating system so that it will receive notification of paging events. The kernel then notifies the runtime system just before any page is scheduled for eviction from the inactive list (specifically, whenever its corresponding page table entry is unmapped). The signal from the kernel includes the address of the relevant page.

To maintain information about process ownership of pages, we applied Scott Kaplan's lightweight version of Rik van Riel's reverse mapping patch [34, 50]. This patch allows the kernel to determine the owning process of pages currently in physical memory.

BC needs timely memory residency information from the virtual memory manager. To ensure the timeliness of this communication, our notifications use Linux real-time signals. Real-time signals in the Linux kernel are queueable. Unlike other notification methods, these signals cannot be lost due to other process activity.

5. Results

We evaluated the performance of BC by comparing it to five garbage collectors included with Jikes RVM: MarkSweep, SemiSpace, GenCopy, GenMS (Appel-style generational collectors using bump-pointer and mark-sweep mature spaces, respectively), and CopyMS (a variant of GenMS which performs only whole heap garbage collections).

Table 1 describes our benchmarks, which include the SPECjvm98 benchmark suite [27], pseudoJBB (a fixed-workload variant of SPECjbb [26]), and two applications from the DaCapo benchmark suite, ipsixql and jython. We compare execution and pause times of the pseudoJBB benchmark on our extended Linux kernel. This benchmark is widely considered to be the most representative of a server workload and is the only one of our benchmarks with a significant memory footprint.

5.1 Methodology

We performed all measurements on a 1.6GHz Pentium M Linux machine with 1GB of RAM and 2GB of local swap space. This processor includes a 32KB L1 data cache and a 1MB L2 cache. We report the mean of five runs with the system in single-user mode and the network disabled.

Benchmark statistics		
Benchmark	Total Bytes Alloc	Min. Heap
SPECjvm98		
201_compress	109,190,172	16,777,216
202_jess	267,602,628	12,582,912
205_raytrace	92,381,448	14,680,064
209_db	61,216,580	19,922,944
213_javac	181,468,984	19,922,944
228_jack	250,486,124	11,534,336
DaCapo		
ipsixql	350,889,840	11,534,336
jython	770,632,824	11,534,336
SPECjbb2000		
pseudoJBB	233,172,290	35,651,584

Table 1. Memory usage statistics for our benchmark suite.

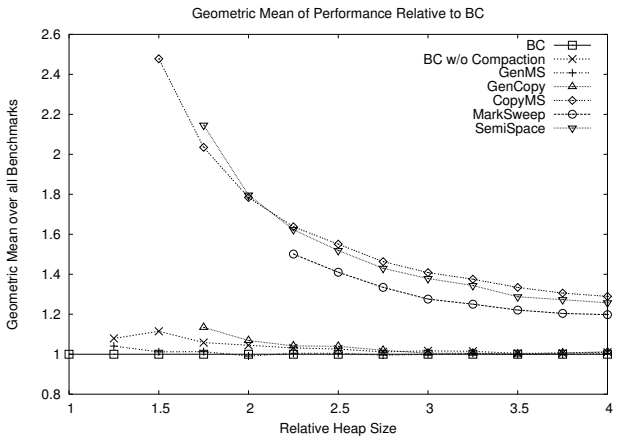


Figure 2. Geometric mean of execution time relative to BC absent memory pressure and across all benchmarks. At the smaller sizes, heap compaction allows BC to require less space while providing the best performance. Compaction is not needed at larger heap sizes, but BC typically continues to provide high performance.

To duplicate the environment found in live servers, all code is compiled by the optimizing compiler. We did not want compilation to affect our end results, however, as servers rarely spend time compiling code. Following Bacon et al. [13], we run two iterations of each benchmark, but report results only from the second iteration. The first iteration optimizes all of the code. Because compilation allocates onto the heap, the heap size is allowed to change during this iteration. After the first iteration is complete, Jikes performs a full heap collection to remove compilation objects and data remaining from the first run. We then measure the second iteration of the benchmark. Using this "compile-and-reset" methodology, we can compare the performance of collectors in an environment similar to that found in a large server.

To examine the effects of memory pressure on garbage collection paging behavior, we simulate increasing memory pressure caused by another application starting up or increasing its working set size. We begin these experiments by making available only enough memory for Jikes to complete the compilation phase without any memory pressure. We then use an external process we call signalmem. The Jikes RVM notifies signalmem when it completes the first iteration of a benchmark. Once alerted, signalmem uses `mmap` to allocate a large array, touches these pages, and then pins them in memory with `mlock`. The initial amount of memory, total amount of memory, and rate at which this memory is pinned are specified via command-line parameters. Using signalmem, we perform repeatable measurements under memory pressure while reducing disruption due to CPU load. This approach also allows us to compare the effects of different levels of memory pressure and a variety of page evictions rates.

5.2 Performance Without Memory Pressure

While the key goal of bookmarking collection is to avoid paging, it would not be practical if it did not provide competitive throughput in the absence of memory pressure. Figure 2 summarizes the results for the case when there is sufficient memory to run the benchmarks without any paging. We present the geometric mean increase in execution time relative to BC at each relative heap size for each collector.

As expected, BC is closest in performance to GenMS, although BC will occasionally run in a smaller heap size. Both collectors perform nursery collection and have a segregated-fit mark-sweep mature space, and behave similarly at large heap sizes and without memory pressure. At the largest heap size (where heap compaction is not needed), the two collectors are virtually tied (BC runs 0.3% faster). However, at smaller sizes, BC's compaction allows it to run in smaller heaps. For example, BC runs 4% faster at the 1.25x heap size.

The next best collector is GenCopy, which runs as fast as BC at the largest heap sizes but averages 7% slower at heaps as large as twice the minimum. The fact that GenCopy generally does not exceed BC's performance suggests that BC's segregated size classes do not significantly impact locality. Unsurprisingly, BC's performance is much greater than the single-generation collectors. At the largest heap size, MarkSweep averages a 20% and CopyMS a 29% slowdown. No collector at any heap size performs better on average than BC, demonstrating that the BC provides high performance when memory pressure is low.

5.3 Performance Under Memory Pressure

We evaluate the impact of memory pressure using three sets of experiments. We first measure the effect of *steady memory pressure*. Next, we measure the effect of *dynamically growing memory pressure*, as would be caused by the creation of another process or a rapid increase in demand (e.g., the "Slashdot effect"). Finally, we run *multiple JVMs simultaneously*. We use the pseudoJBB benchmark for all these experiments.

5.3.1 Steady Memory Pressure

To examine the effects of running under steady memory pressure, we measure the available memory needed so every collector can run without evicting a page. Starting with that amount of available memory, we begin the second iteration by having signalmem remove memory equal to 60% of the heap size. Results of these experiments are shown in Figure 3. Note that we do not show results for MarkSweep in these graphs, because runs with this collector can take hours to complete.

Figure 3 shows that under steady memory pressure, BC outperforms most of the other collectors (and all of the generational collectors). Although SemiSpace outperforms BC at the 80-95MB heap sizes, its execution time goes off the chart soon after. CopyMS also outperforms BC in the same range of heap sizes but runs nearly twice as slow as BC at the 130MB heap size. At this size, GenMS's average pause time is around 3 seconds, 30 times greater than BC's average pause time. To test CopyMS's behavior under greater memory pressure, we measured the effect of removing memory equal to 70% of the heap size. Under these conditions, CopyMS takes over an hour to execute pseudoJBB, while BC's execution time remains largely unchanged.

5.3.2 Dynamic Memory Pressure

To simulate a spike in memory pressure, we invoke signalmem so that it initially allocates 30MB and then allocates additional memory at a rate of 1MB every 100ms until it reaches the desired level of available memory. Figure 4 shows the average pause times and Figure 5 shows the average execution time for pseudoJBB as memory pressure increases (i.e., as available memory shrinks).

Figures 4 and 5 show that under memory pressure, BC significantly outperforms all of the other collectors both in total execution time and pause times. Note that, as with the previous paging experiments, we do not present MarkSweep here because of the dramatic performance degradation it suffers.

Because the mark-sweep based collectors do not perform compaction, objects become spread out over a range of pages. Once heap pages are evicted, visiting these pages during a collection triggers a cascade of page faults and orders-of-magnitude increases in execution time. For instance, at the largest heap size, GenMS's average garbage collection pause takes nearly 10 seconds — longer than it needed to execute pseudoJBB without memory pressure. Even when collections are relatively rare, spreading objects across a large number of pages can degrade performance by increasing mutator faults.

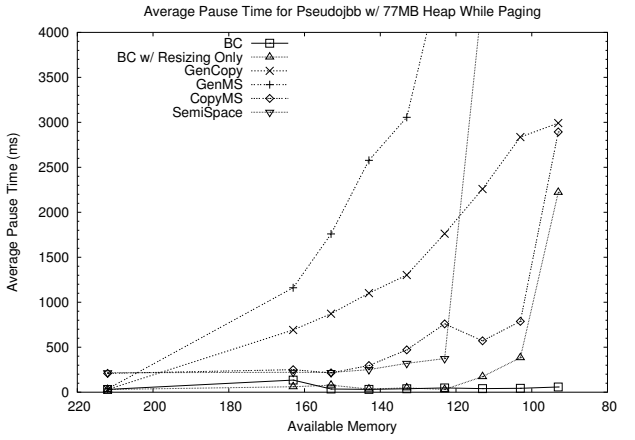


Figure 4. Dynamic memory pressure (increasing from left to right): average GC pause time running pseudoJBB. BC's average pause times remain unaffected by increasing memory pressure.

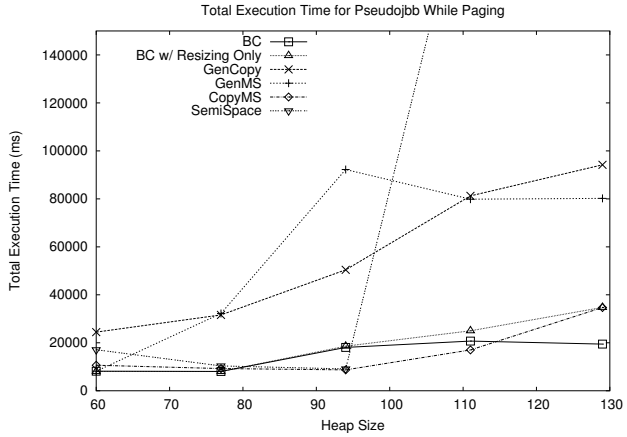
Compacting objects onto fewer pages can reduce faults, but the compacting collectors also suffer from paging effects. For example, the execution times for GenCopy shown in Figure 5(a) are an order of magnitude larger than its times when not paging. Paging also increases the average GenCopy pause to several seconds, while BC's pause times remain largely unchanged.

The collectors that come closest in execution time to BC in Figure 5 are SemiSpace and CopyMS. While these collectors perform well at low to moderate memory pressure, they perform far worse both under no memory pressure and under severe memory pressure. This effect is due to pseudoJBB's allocation behavior. pseudoJBB initially allocates a few immortal objects and then allocates only short-lived objects. While these collectors reserve heap space to copy the survivors of a collection, little of this space is used. LRU ordering causes nursery pages filled with dead objects to be evicted. While SemiSpace ultimately reuses these pages in subsequent collections, CopyMS's mark-sweep mature object space allows better heap utilization and fewer collections. This heap organization delays paging, but does not prevent it.

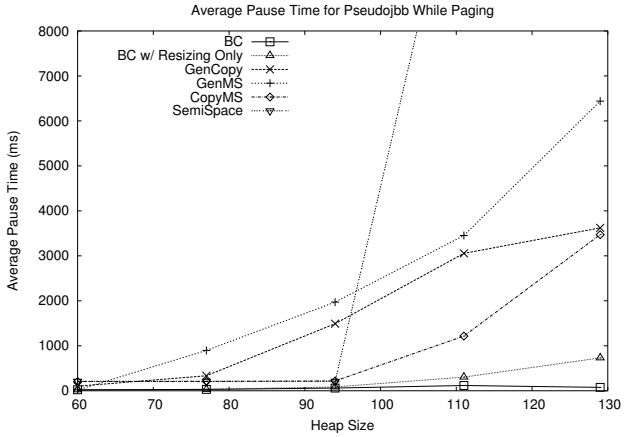
We next present *mutator utilization* curves [23]. Mutator utilization is the fraction of time that the mutator runs during a given time window. In other words, it is the amount of time spent by the application rather than by the garbage collector. We adopt the methodology of Sachindran et al. and present *bounded mutator utilization*, or BMU [44]. The BMU for a given window size is the minimum mutator utilization for all windows of that size or greater.

Figure 6 shows the BMU curves for the dynamic memory pressure experiments. With moderate memory pressure (143MB available RAM), both variants of BC and MarkSweep do very well, but all of the other collectors exhibit poorer utilization. In particular, GenMS requires window sizes orders of magnitude larger than BC's running time before the mutator makes any progress.

Under severe memory pressure (93MB available RAM), the full bookmarking collector outperforms the other collectors, achieving almost 0.9 utilization over a 10-second window. At this window

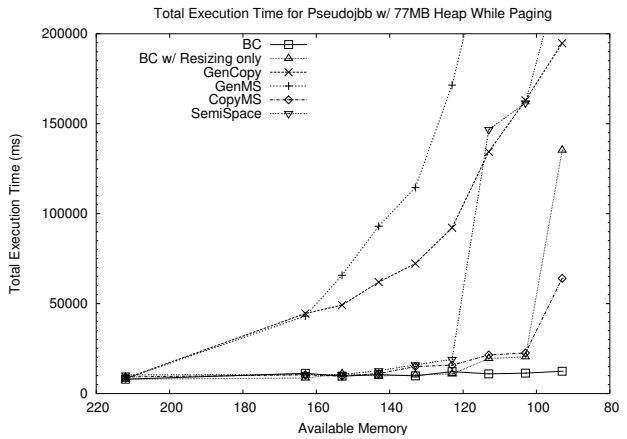


(a) Execution time running pseudoJBB.

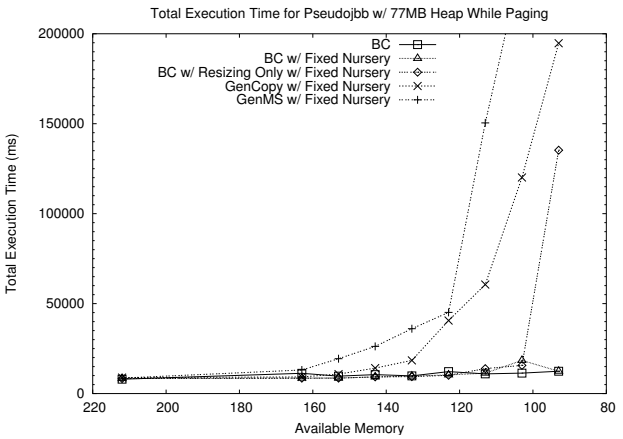


(b) Average GC pause time running pseudoJBB.

Figure 3. Steady memory pressure (increasing from left to right), where available memory is sufficient to hold only 40% of the heap. As the heap becomes larger, BC runs 7 to 8 times faster than GenMS and in less than half the time needed by CopyMS. Bookmarking is faster and yields shorter pause times than just resizing the heap.



(a) Execution time running pseudoJBB



(b) Execution time running pseudoJBB, *fixed-size nursery* collectors

Figure 5. Dynamic memory pressure (increasing from left to right). BC runs up to 4x faster than the next best collector and up to 41x faster than GenMS. While shrinking the heap can help, BC runs up to 10x faster when also using bookmarks.

size, all of the other collectors have zero utilization. The non-bookmarking variant of BC (shown as "BC w/Resizing only") and CopyMS are the next best collectors, but peak at around 0.5 mutator utilization over a 40-second window. Interestingly, MarkSweep has the worst utilization here, requiring a window size of almost 10 minutes before achieving 0.25 mutator utilization.

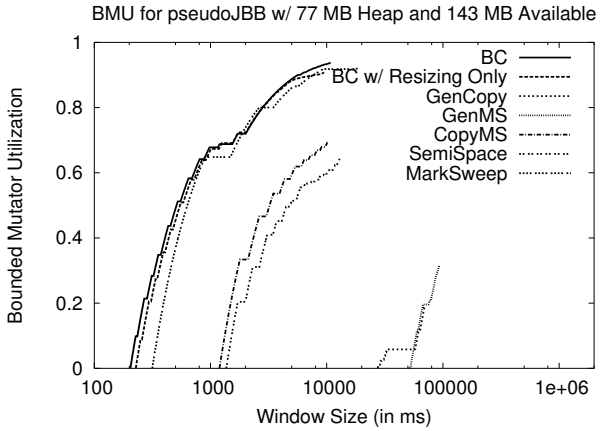
We also compared different variants of the bookmarking collector and the generational collectors to tease apart the impact of various strategies on paging.

Fixed-size nurseries: Fixed-size nurseries can achieve the same throughput as variable-sized nurseries but are generally believed to incur lower paging costs. Figure 5(b) presents execution times for variants of the generational collectors with fixed-size nurseries (4MB). The graph shows that the fixed-nursery variants of the other collectors do reduce paging, but perform as poorly as the variable-sized generational collectors once their footprint exceeds available memory.

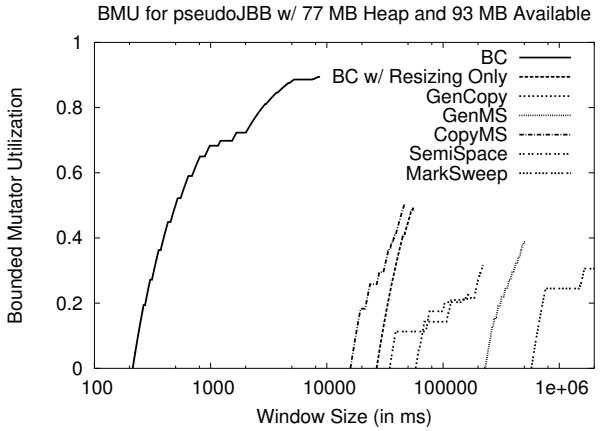
Bookmarking vs. resizing the heap: When memory pressure is relatively low, BC's strategy of discarding empty pages is sufficient to avoid paging. As memory pressure increases, however, a variant of BC that only discards pages (shown as "BC w/Resizing only") requires up to 10 times as long to execute as the full bookmarking collector. The results shown in Figures 3 and 4 demonstrate that bookmarking is vital to achieve high throughput and low pause times under moderate or severe memory pressure.

5.3.3 Multiple JVMs

Finally, we examine a scenario with two JVMs executing simultaneously. For this experiment, we start two instances of Jikes running the pseudoJBB benchmark and measure total elapsed time and garbage collection pause times. We cannot employ the

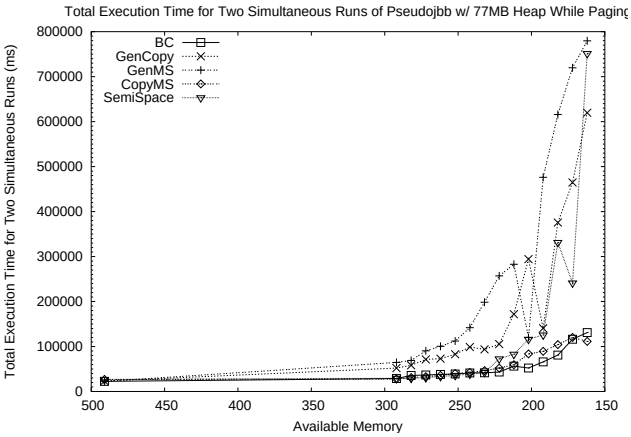


(a) 143MB available: BC is largely unaffected by moderate levels of memory pressure.

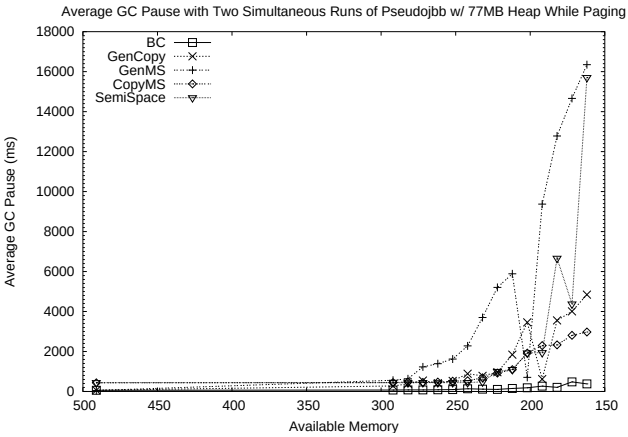


(b) 93MB available: under heavy memory pressure, bookmarks become increasingly important.

Figure 6. Dynamically increasing memory pressure: bounded mutator utilization curves (curves to the left and higher are better). BC consistently provides high mutator utilization over time. As memory pressure increases (available memory shrinks), the importance of bookmarks to limit garbage collection pauses becomes more apparent.



(a) Execution time running two instances of pseudoJBB



(b) Average GC pause time running two instances of pseudoJBB

Figure 7. Execution time and pause times when running **two** instances of pseudoJBB simultaneously. Note that the execution times are somewhat misleading, because for the other collectors, paging effectively deactivates one of the instances. Under heavy memory pressure, BC exhibits pause times around 7.5x lower than the next best collector.

"compile-and-reset" experimental methodology here because compiling the entire application generates too much paging traffic and we cannot "reset" the virtual memory manager's history. Instead, we employ the *pseudoadaptive* methodology [32, 44], which optimizes only the "hottest" methods (as determined from the mean of 5 runs). This methodology minimizes the impact of the compiler as much as possible, and does so deterministically rather than relying on the standard sampling-based approach. Eeckhout et al. report that all of the virtual machines they examined exhibit similar behavior for this benchmark, suggesting that the performance impact of the pseudoadaptive compiler here should be minimal [29].

Figure 7 shows the results of executing two instances of pseudoJBB, each with a 77MB heap. While BC performs the best, total elapsed time can be misleading: for all of the other collectors, paging effectively serializes the benchmark runs. This serialization means that first one instance of pseudoJBB runs to completion, and then the next. The average pause time numbers are thus more revealing. As available memory shrinks, bookmarking shows a gradual increase in average pause times. At the lowest amount of available memory, BC exhibits pause times averaging around 380ms, while average pause times for CopyMS (the next best collector) reach 3 seconds, nearly an eightfold increase.

6. Related Work

Most previous garbage collectors ignore virtual memory altogether, making them **VM-oblivious**. The literature on such collectors is extensive; Wilson's survey and Jones and Lins' text provide excellent overviews [33, 53].

VM-sensitive garbage collectors are designed to limit paging in virtual memory environments. Such collectors either compact memory to reduce the number of pages needed to hold an application's working set [14, 15, 20, 21, 22, 30], or divide the heap into generations to reduce the need for full-heap collections [9, 17, 37, 39, 41, 49]. Bookmarking collection builds on both: it employs compaction to reduce working set size and uses a nursery generation to improve throughput, but uses bookmarking to avoid collector-induced paging. Bookmarking collection may be thought of as a generalization of generational collection, where the evicted pages comprise a distinct (and dynamically changing) mature-space generation, and bookmarks act as remembered sets.

VM-cooperative garbage collectors receive information from or send information to the virtual memory manager (or both). Moon's *ephemeral collector* receives signals from the virtual memory manager whenever it evicts a page from main memory [39]. The collector then scans the page and records which generations it contains references to. During GC, Moon's collector then scans only those evicted pages containing pointers to the generation being collected. Unlike Moon's collector, bookmarking collection does not revisit any evicted pages during nursery or full-heap collections.

Cooper et al.'s collector informs the VM of empty memory pages that can be removed from main memory without being written back to disk [25]. BC also identifies and discards empty pages, but as we show in Section 5.3, BC's ability to evict non-empty pages gives it a significant advantage under memory pressure.

Another class of VM-cooperative collectors respond to memory pressure by adjusting their heap size. Alonso and Appel present a collector that consults with the virtual memory manager indirectly (through an "advisor") to shrink the heap after each garbage collection based upon the current level of available memory [6]. MMTk [19] and BEA JRockit [3] can, in response to the live data ratio or pause time, change their heap size using a set of pre-defined ratios. HotSpot [1] can adjust heap size with respect to pause time, throughput, and footprint limits specified as command line arguments. Novell Netware Server 6 [2] polls the virtual memory manager every 10 seconds, and shortens its GC invocation interval to collect more frequently when memory pressure is high. None of these approaches eliminate paging. Yang et al. report on an automatic heap sizing algorithm that uses information from a simulated virtual memory manager and a model of GC behavior to choose a good heap size [55]. Like these systems, BC can shrink its heap (by giving up discardable pages), but it does not need to wait until a full heap garbage collection to do so. BC also eliminates paging even when the size of live data exceeds available physical memory.

Researchers have leveraged virtual memory hardware to support garbage collection in other ways. Appel, Ellis and Li use virtual memory protection to improve the speed and concurrency of Baker's algorithm [10]. Appel and Li present a variety of ways to use virtual memory to assist garbage collection, including providing an inexpensive means of synchronization, remembering interesting pointers, or saving long-lived data to disk [11]. These uses of virtual memory are orthogonal to this work and to the VM-cooperative collectors described above, which use virtual memory cooperation primarily to reduce paging.

Like bookmarking collection, distributed garbage collection algorithms also treat certain references (those from remote systems) as *secondary roots*.[3] Distributed GC algorithms typically employ either reference counting or listing. Unlike bookmarking, both of these require substantial additional memory and updates on every pointer store, while BC only updates bookmarks when pages are evicted or made resident (relatively rare events).

[3] See Abdullahi and Ringwood [4] and Plainfossé and Shapiro [40] for excellent recent surveys of this area.

A number of researchers have focused on the problem of improving *application-level* locality of reference by using the garbage collector to modify object layouts [5, 24, 28, 32, 35, 47, 48, 54]. These studies do not address the problem of paging caused by garbage collection itself, which we identify as the primary culprit. These approaches are orthogonal and complementary to the work presented here.

Finally, bookmarking collection's use of segregated size classes for compaction is similar to the organization used by Bacon et al.'s Metronome collector [12, 13]. Unlike the Metronome, BC uses segregated size classes for mature objects only. BC's copying pass is also quite different. The Metronome sorts pages by occupancy, forwards objects by marching linearly through the pages and continues until reaching a pre-determined size, forwarding pointers later. BC instead copies objects by first marking target pages and then forwarding objects as they are discovered in a Cheney scan. This approach obviates the need for further passes and immediately brings memory consumption to the minimum level.

7. Future Work

While our results show that BC already yields significant performance improvements and robustness under memory pressure, there are several directions in which we would like to advance this work. First, because our modifications to the operating system kernel are straightforward, we would like to incorporate these in other operating systems to verify its generality and expand our range of benchmark applications.

Because memory pressure can be transient, BC may discard empty pages only to have memory again become available later. It is important that a brief spike in memory pressure not limit throughput by restricting the size of the heap. We are evaluating strategies to allow BC to grow its heap when additional physical memory becomes available.

We are also considering alternate strategies for selecting victim pages. First, we can prefer to evict pages with no pointers, because these pages cannot create false garbage. For some types of objects, e.g., arrays of doubles, we do not even need to scan the pages to determine that they are free of pointers. We could also prefer to evict pages with as few non-NULL pointers as possible. We have not yet explored these possibilities because we cannot predict the effect on the application of evicting a page that is not the last on the LRU queue (i.e., the one chosen by the virtual memory manager). Choosing to evict such a victim page may lead to more page faults in the application. We are currently developing a more advanced virtual memory manager that will enable us to predict the effect of selecting different victim pages and thus explore the tradeoffs of using more sophisticated eviction strategies.

Finally, the bookmarking collector presented here is a "stop-the-world" garbage collector, where all mutator work stops for garbage collection. However, server-based JVMs typically collect the heap concurrently while running the application. The current BC algorithm is not suitable for concurrent garbage collection: for example, a race could arise from the application modifying a page that the collector is scanning. We are developing a concurrent variant of the bookmarking collection algorithm.

8. Conclusion

The increasing latency gap between disk and main memory means that paging is now intolerable. Garbage collection's reference behavior can cause catastrophic paging. We present bookmarking collection, an algorithm that leverages cooperation between the garbage collector and virtual memory manager to eliminate nearly all paging caused by the garbage collector. When memory pressure is low, the bookmarking collector provides performance that gen-

erally matches or slightly exceeds that of the highest throughput collector we tested (GenMS). In the face of memory pressure, BC improves program performance by up to 5x over the next best garbage collector and reduces pause times by 45x, improving even more dramatically over GenMS. BC thus provides greater memory utilization and more robust performance than previous garbage collectors.

Acknowledgements

Thanks to Sam Guyer, Kathryn McKinley, Eliot Moss, Pritesh Sharma, Yannis Smaragdakis, and Ben Zorn for their comments on drafts of this paper. We are grateful to Scott Kaplan for his assistance in the implementation of our modified Linux memory manager. We are also grateful to IBM Research for making the Jikes RVM system available under open source terms. The MMTk memory management toolkit was particularly helpful.

This material is based upon work supported by the National Science Foundation under Award CNS-0347339. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] J2SE 1.5.0 Documentation - Garbage Collector Ergonomics. Available at `http://java.sun.com/j2se/1.5.0/docs/guide/vm/gc-ergonomics.html`.

[2] Novell Documentation: NetWare 6 - Optimizing Garbage Collection. Available at `http://www.novell.com/documentation/index.html`.

[3] Technical white paper - BEA Weblogic JRockit: Java for the enterprise. Available at `http://www.bea.com/content/news_events/white_papers/BEA_JRockit_wp.pdf`.

[4] S. E. Abdullahi and G. A. Ringwood. Garbage collecting the Internet: a survey of distributed garbage collection. *ACM Computing Surveys*, 30(3):330–373, Sept. 1998.

[5] A.-R. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney. Prefetch injection based on hardware monitoring and object metadata. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, June 2004.

[6] R. Alonso and A. W. Appel. Advisor for flexible working sets. In *Proceedings of the 1990 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 153–162, Boulder, CO, May 1990.

[7] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Shepherd, and M. Mergen. Implementing Jalapeño in Java. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10), pages 314–324, Denver, CO, Oct. 1999.

[8] B. Alpern, D. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), Feb. 2000.

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

[10] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

[11] A. W. Appel and K. Li. Virtual memory primitives for user programs. *ACM SIGPLAN Notices*, 26(4):96–107, 1991.

[12] D. F. Bacon, P. Cheng, and V. 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.

[13] D. F. Bacon, P. Cheng, and V. 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, Jan. 2003. ACM Press.

[14] H. D. Baecker. Garbage collection for virtual memory computer systems. *Communications of the ACM*, 15(11):981–986, Nov. 1972.

[15] H. G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978.

[16] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, Cambridge, MA, Nov. 2000.

[17] P. B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, MIT Laboratory for Computer Science, May 1977.

[18] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and reality: The performance impact of garbage collection. In *Sigmetrics - Performance 2004, Joint International Conference on Measurement and Modeling of Computer Systems*, New York, NY, June 2004.

[19] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *ICSE 2004, 26th International Conference on Software Engineering*, Edinburgh, May 2004.

[20] D. G. Bobrow and D. L. Murphy. Structure of a LISP system using two-level storage. *Communications of the ACM*, 10(3):155–159, Mar. 1967.

[21] D. G. Bobrow and D. L. Murphy. A note on the efficiency of a LISP computation in a paged machine. *Communications of the ACM*, 11(8):558–560, Aug. 1968.

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

[23] P. Cheng and G. Belloch. 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.

[24] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the First International Symposium on Memory Management*, volume 34(3), pages 37–48, Vancouver, BC, Canada, Oct. 1998.

[25] E. Cooper, S. Nettles, and I. Subramanian. Improving the performance of SML garbage collection using application-specific virtual memory management. In *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming*, pages 43–52, San Francisco, CA, June 1992. ACM Press.

[26] S. P. E. Corporation. Specjbb2000. Available at `http://www.spec.org/jbb2000/docs/userguide.html`.

[27] S. P. E. Corporation. Specjvm98 documentation, Mar. 1999.

[28] R. Courts. Improving locality of reference in a garbage-collecting memory management-system. *Communications of the ACM*, 31(9):1128–1138, 1988.

[29] L. Eeckhout, A. Georges, and K. D. Bosschere. How Java programs interact with virtual machines at the microarchitectural level. pages 169–186.

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

[31] M. Hertz and E. D. Berger. Automatic vs. explicit memory management: Settling the performance debate. Technical report, University of Massachusetts, Mar. 2004.

[32] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: Improving program locality. In *Proceeding of the ACM Conference on Object-Oriented Systems, Languages and Applications*, Vancouver, BC, Canada, Oct. 2004.

[33] R. E. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July

1996.

[34] S. F. Kaplan. In-kernel RIG: Downloads. Available at `http://www.cs.amherst.edu/~sfkaplan/research/rig/download`.

[35] M. S. Lam, P. R. Wilson, and T. G. Moher. Object type directed garbage collection to improve locality. In *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St Malo, France, 16–18 Sept. 1992. Springer-Verlag.

[36] D. Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html, 1997. Available at `http://gee.cs.oswego.edu/dl/html/malloc.html`.

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

[38] T. F. Lim, P. Pardyak, and B. N. Bershad. A memory-efficient real-time non-copying garbage collector. In *Proceedings of the First International Symposium on Memory Management*, volume 34(3), pages 118–129, Vancouver, BC, Canada, Oct. 1998.

[39] D. A. Moon. Garbage collection in a large LISP system. In G. L. Steele, editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–245, Austin, TX, Aug. 1984. ACM Press.

[40] D. Plainfossé and M. Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, Sept. 1995. Springer-Verlag.

[41] J. H. Reppy. A high-performance garbage collector for Standard ML. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ, Dec. 1993.

[42] D. T. Ross. The AED free storage package. *Communications of the ACM*, 10(8):481–492, Aug. 1967.

[43] N. Sachindran and J. E. B. Moss. MarkCopy: Fast copying GC with less space overhead. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Applications*, Anaheim, CA, Nov. 2003.

[44] N. Sachindran, J. E. B. Moss, and E. D. Berger. MC^2: High-performance garbage collection for memory-constrained environments. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Applications*, Vancouver, BC, Canada, Oct. 2004.

[45] P. Savola. LBNL traceroute heap corruption vulnerability. `http://www.securityfocus.com/bid/1739`.

[46] Software Verification, Ltd. Memory Validator - Garbage Collectors. Available at `http://softwareverify.com/memoryValidator/garbageCollectors.html`.

[47] D. Stefanović, K. S. McKinley, and J. E. B. Moss. Age-based garbage collection. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10), pages 370–381, Denver, CO, Oct. 1999.

[48] G. Tong and M. J. O'Donnell. Leveled garbage collection. *Journal of Functional and Logic Programming*, 2001(5):1–22, May 2001.

[49] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, volume 19(5), pages 157–167, Apr. 1984.

[50] R. van Riel. rmap VM patch for the Linux kernel. Available at `http://www.surriel.com/patches/`.

[51] D. A. Wheeler. SLOCcount. Available at `http://www.dwheeler.com/sloccount`.

[52] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St Malo, France, 16–18 Sept. 1992. Springer-Verlag.

[53] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, pages 1–116, Kinross, Scotland, Sept. 1995. Springer-Verlag.

[54] P. R. Wilson, M. S. Lam, and T. G. Moher. Effective static-graph reorganization to improve locality in garbage collected systems. *ACM SIGPLAN Notices*, 26(6):177–191, 1991.

[55] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Automatic heap sizing: Taking real memory into account. In *Proceedings of the 2004 ACM SIGPLAN International Symposium on Memory Management*, Nov. 2004.

[56] B. Zorn. The measured cost of conservative garbage collection. *Software Practice and Experience*, 23:733–756, 1993.