Page-Level Cooperative Garbage Collection

Matthew Hertz, Yi Feng and Emery D. Berger
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003
{hertz, yifeng, emery}@cs.umass.edu

ABSTRACT

Programs written in garbage-collected languages like Java often have large working sets and poor locality. Worse, a full garbage collection must visit all live data and is likely to touch pages that are no longer resident, triggering paging. The result is a pronounced drop in throughput and a spike in latency. We show that just a slight reduction in available memory causes the throughput of the SPECjbb benchmark to drop by 66% and results in garbage collection pauses lasting for several seconds.

This paper introduces *page-level cooperative garbage collection*, an approach in which the garbage collector works with the virtual memory manager to limit paging. We present a novel, cooperative garbage collection algorithm called *Hippocratic* collection. By communicating with the virtual memory manager and "bookmarking" objects, the Hippocratic collector *eliminates* paging caused by garbage collection. Our approach requires only modest extensions to existing virtual memory management algorithms. We present empirical results using our modified Linux kernel. We show that the Hippocratic collector runs in smaller footprints than traditional collectors while providing competitive throughput. Under memory pressure, Hippocratic collection improves the throughput of SPECjbb by a factor of two over the next best garbage collector.

1. Introduction

As to diseases, make a habit of two things – to help, or at least, to do no harm. —*Hippocrates* [18]

Garbage collection, and the numerous software engineering advantages it provides over explicit memory management [20, 26], is one primary reason for the popularity of languages like Java and C#. However, garbage collection suffers from large working set sizes and poor page-level locality [12]. The result is that fewer garbage-collected applications can fit in a given amount of RAM and that individual garbage-collected applications need more space than their explicitly-managed counterparts. When a garbage-collected application does not fit in RAM, its behavior interacts especially poorly with virtual memory management.

Figure 1 illustrates this problem with the SPECjbb Java Beans benchmark [21] executing on the HotSpot JVM [1] on a Linux box with 512MB of RAM. Exceeding available memory by just 20 MB triggers paging. Garbage collection while paging causes thrashing and reduces throughput by 66%.

The root cause of this problem is the lack of cooperation between the garbage-collector and the virtual memory manager. Virtual memory managers do not communicate memory pressure to the garbage collector and garbage collectors lack any mechanism to respond. The collector's activity disrupts the reference behavior tracked by the virtual memory manager and marches over heap pages with no regard to which are resident in RAM. Full heap collections may trigger massive paging and have been known to cause systems to become unresponsive for minutes.

This paper introduces the *Hippocratic collector* (HC), a *page-level cooperative garbage collector*. HC and the virtual memory manager work together to avoid paging. When needed, HC compacts the heap with a novel copying algorithm that does not need a copy reserve, allowing it to need up to 50% less heap space while delivering performance competitive with existing collectors. As memory pressure rises, HC guides virtual memory eviction decisions. Whenever possible, the Hippocratic collector returns garbage pages to the operating system because these do not require I/O. When these cannot be found, the collector *bookmarks* the targets of outgoing pointers from a victim page. Using these as roots, HC can perform full memory-resident garbage collections without paging. We present empirical results demonstrating a factor of two performance improvement for the SPECjbb benchmark over the next best garbage collector.

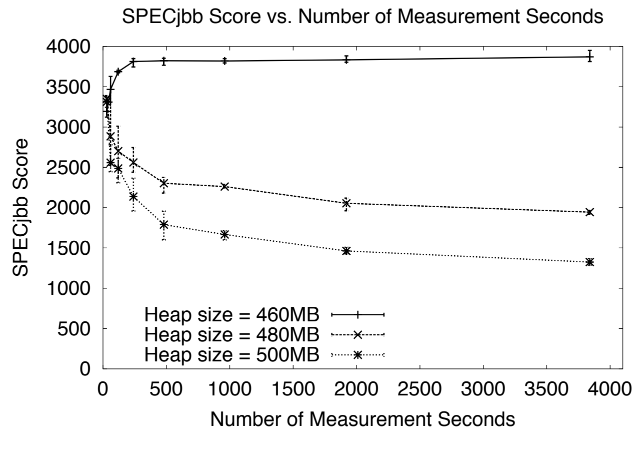


Figure 1: SPECjbb transactions per second (higher is better). A lack of available RAM triggers paging, reducing throughput by 66% and causing pauses lasting for seconds.

The remainder of this paper is organized as follows. We discuss related work in Section 2. We describe our Hippocratic garbage collector in detail in Section 3, and discuss our modest extensions to the Linux virtual memory manager in Section 4. We present our experimental methodology in Section 5. Section 6 presents detailed simulation results and empirical results. We discuss future work and conclude in Section 7.

2. Related Work

In this section, we discuss related work. We first classify garbage collection algorithms into four categories, based on their degree of involvement with the virtual memory manager. Those collectors that ignore virtual memory altogether we call *VM-oblivious*. Wilson's survey and Jones and Lins' text contain numerous references for traditional garbage collectors which almost all fit into this category [13, 27]. A garbage collector designed for use in virtual memory environments, especially one intended to limit paging, is *VM-sensitive*. A *VM-aware* garbage collector either receives information from or sends information to the virtual memory manager. We know of no previous garbage collector that is *VM-cooperative*, that is, where there is two-way communication between the garbage collector and the virtual memory manager. We therefore address only VM-sensitive and VM-aware garbage collectors here.

2.1 VM-Sensitive Garbage Collection

Fenichel and Yochelson first introduced *semispace collection* as a means of reducing the impact of paging by compacting live data [11]. Moon developed *ephemeral garbage collection* [17] to improve the paging behavior of garbage collection. Ephemeral collection relies on hardware-supported write barriers to track pointers into the ephemeral area, which holds short-lived objects. *Generational garbage collection* achieves the same result without hardware support [5, 15, 24], allowing short-lived objects to be reclaimed without the need for full-heap garbage collections. Reducing the frequency of full-heap garbage collections reduces paging. One variant of the Hippocratic collector that we present here also uses a nursery generation to exploit this "weak generational hypothesis" (that most objects die young), but goes further to avoid paging by direct interaction with the virtual memory manager.

A number of researchers have focused on the problem of improving page-level locality of reference *in the application* by using the garbage collector to modify object layout [9, 22, 23, 28]. These studies demonstrate reductions in the total number of page faults, but do not address the problem of paging caused by garbage collection itself, which we identify as the primary culprit.

Kim and Hsu examined garbage collection performance using the SpecJVM98 suite of benchmarks [14]. They found that paging causes garbage collection performance to suffer significantly, but that optimal heap sizes could be found. These optimal heap sizes require that applications maintain a consistent amount of live data and be running on a dedicated machine. Our system dynamically adapts to memory pressure and is thus suitable for multiprogrammed environments and applications with fluctuating workloads.

2.2 VM-Aware Garbage Collection

We know of only two garbage collection algorithms that attempt to incorporate information from or communicate with the virtual memory manager.

The closest work to our own is by Cooper, Nettles, and Subramanian, who used external pagers in the Mach operating system [16] to allow the garbage collector to influence virtual memory paging behavior [8]. Their garbage collector informed the pager of *discardable* (garbage) memory pages that can be removed from main memory without being written back to disk. Their system suffered from a number of limitations, some of which are related to their use of the external pager, which cannot dictate policy decisions to the virtual memory manager. First, their garbage collector may provide discardable pages to the virtual memory manager, but the VM may nonetheless choose to evict a page that must be copied back to disk. Their collector may also be over or under-aggressive at flushing discardable pages. Either approach degrades performance. Our system acts only under memory pressure signals from the virtual memory manager and directly guides the VM's eviction decisions. HC also identifies discardable pages, but additionally performs compaction, gives up non-discardable pages (with live data), and continues to perform full memory-resident garbage collections even when the heap does not fit in memory.

Alonso and Appel present a collector that can shrink the heap after each garbage collection based upon the current level of available memory, which they obtain from an "advisor" that calls `vmstat` [2]. Our work can also be viewed as shrinking the size of the heap by giving up discardable pages under memory pressure, but does not need to wait until the next collection suffers the effects of paging. More importantly, however, HC cooperates closely with the virtual memory manager on eviction decisions.

3. Hippocratic Garbage Collection

In this section, we first present an overview of our new Hippocratic collector and then discuss in detail the new algorithm's design and how it improves garbage collection paging performance. We then discuss aspects of HC that ensure high performance even when there is no memory pressure.

3.1 Hippocratic Algorithm Overview

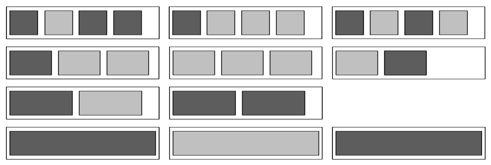
The Hippocratic collector divides the heap into *superpages*, page-aligned groups of contiguous pages. Our superpages consist of four pages (16K total). HC allocates objects using segregated size classes: objects of different sizes are allocated onto different superpages. When the heap fills, HC usually uses mark-sweep garbage collection. While mark-sweep collection provides good throughput, it is unable to compact the heap. HC's use of fixed-size superpages and segregated size classes permits compacting the heap through copying collection without needing a copy reserve, as we describe in Section 3.3.

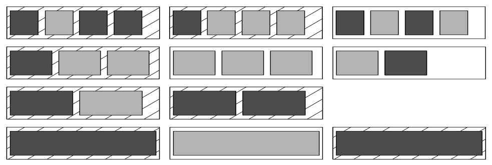
We augment both our collector and the virtual memory manager with simple means of communicating important information. Communication with the virtual memory manager only occurs under memory pressure. When notified by the virtual memory manager that increased memory pressure will soon cause paging, the Hippocratic collector works to keep heap memory resident and avoid mutator page faults. When heap pages must be evicted, a "bookmarking" algorithm allows HC to collect only those objects in main memory and *eliminates* page faults caused by the garbage collector.

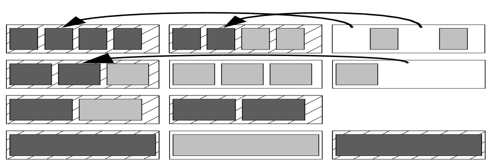
3.2 Superpages and Size Classes

In order to be able to manage page evictions, we need a heap organization that is page-oriented. HC uses groups of pages that we call superpages, which it manages using segregated size classes [7]. This organization allows HC to manage objects in superpages without the need for additional per-object metadata.

To maximize the utilization of our superpages, we chose size classes which minimize both internal and external fragmentation. We use exact (word-aligned) size classes for allocations up to 64 bytes. At larger object sizes, we considered only those sizes that minimize the external fragmentation in our superpages. From this large set of potential size classes, we selected sizes that came closest to a target worst-case fragmentation of 12.5%. Because we


(a) During the marking phase, HC finds and marks all of reachable objects. We show marked objects using a darker grey.


(b) HC selects the minimum needed superpages to serve as targets. Diagonal hashing marks the target superpages selected.


(c) HC copies reachable objects not already on a target page. Our collector does this in one pass using a Cheney scan. The arrows show where HC copies objects in this compacting collection.

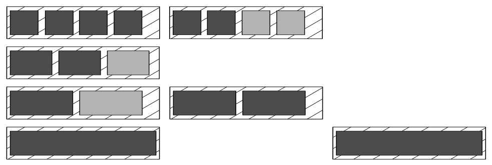

(d) Once copying is complete, non-target superpages only contain garbage objects. HC frees these superpages.

Figure 2: An illustration of how the Hippocratic collector performs copying collection without any copy reserve. After marking the reachable objects, HC compacts the heap into the minimum set of superpages. HC then frees the garbage superpages.

could not minimize the fragmentation for objects larger than half the superpage size, we keep them in a separate large object space.

3.3 Mark-Sweep + Mark-Compact Collection

HC typically collects the heap with mark-sweep collection. We use mark-sweep for two reasons. First, it usually provides good throughput. More importantly, it does not increase memory pressure by needing the "copy reserve" of pages required by most copying collectors [1]. However, the inability of mark-sweep to compact the heap can itself increase memory pressure. Using mark-sweep, HC cannot reclaim a superpage even if it contains just one reachable object.

We therefore extend our design with a novel copying collection algorithm that does not need a copy reserve. When a full garbage collection does not free enough pages to fulfill the current allocation request or sufficiently reduce memory pressure, HC compacts the heap. Because our compacting collection does not the additional copy reserve it will not add to memory pressure during collection. In fact, it often reduces memory pressure after collection by reducing the number of superpages the heap uses.

Figure 2 illustrates this copying collection. HC begins with a marking phase (Figure 2(a)). During this phase, HC counts the number of objects of each size class it marks. Once marking is complete, HC uses these counts to compute the minimum number of superpages needed for each size class and selects a minimum set of superpages as "targets." HC then sweeps through the superpages (Figure 2(b)). Afterwards, HC begins the copying phase. During copying, HC tests if each visited object is on a target superpage. If not, HC copies it onto an empty location on a target superpage (Figure 2(c)). HC does not move objects that are already on a target superpage. After this pass, all reachable objects are on target superpages, and HC can therefore free all non-target superpages (Figure 2(d)).

Our compacting algorithm copies objects without needing a copy reserve only because HC organizes the heap through superpages and segregated size classes. HC knows the exact number of objects each superpage can hold because segregated size classes fill the superpages without external fragmentation. After counting the number of reachable objects, HC computes exactly how many superpages are needed to hold each size class. Superpages avoid the need for a memory reserve because by checking if an object lies on a target superpage, HC knows if an object will move.

3.4 VM-Cooperative Approach

HC cooperates with the virtual memory manager to perform well when under memory pressure. This cooperation allows the HC to adjust and adapt to changing memory pressure while the virtual memory manager uses HC's knowledge of the heap to make good paging decisions.

Keeping the Heap Memory Resident

HC tracks superpages it has used previously and superpages currently being used while it runs. Communication between the virtual memory manager and HC begins when the virtual memory manager notifies the collector of the pending eviction of one of its pages. When this happens, HC looks for previously used superpages that it is not currently using. Rather than evict a useful page, HC directs the virtual memory manager to reclaim one of these *discardable* pages whose contents do not need to be saved. If a discardable page cannot be found, HC collects the heap. After collection, HC can direct the virtual memory manager to discard one of the newly-emptied pages. This approach allows HC to keep the entire heap memory resident whenever possible.

The signal also notifies HC of increasing memory pressure and the current amount of available memory. Unlike VM-oblivious collectors, HC will not grow at the expense of paging. Instead, our collector shrinks the heap to stay within the limits of physical memory. If memory pressure continues to increase, HC will discard additional pages and shrink the heap further. So long as the program continues running, HC shrinks the heap if this will avoid paging. By resizing the heap in response to this fine-grained information provided by the virtual memory manager, HC is often able to avoid the poor performance resulting from collecting the heap while paging and slow mutator response times caused by page faults.

Bookmarking

If memory pressure is sufficiently high, it may be impossible for HC to either shrink the heap to fit in main memory or fulfill an al-

[1]Usually this copy reserve is half of the size of the heap, but Sachindran et al. present a copying collector that allows the use of much smaller copy reserves [19].

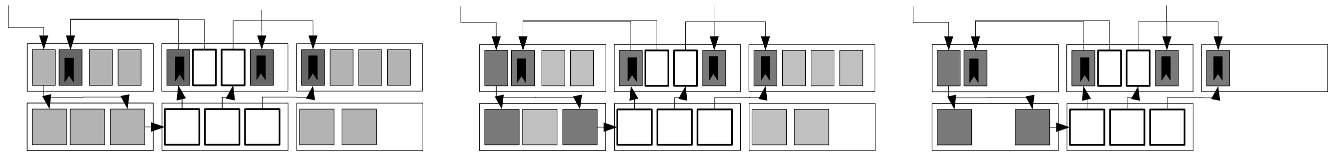

(a) The heap after HC scans for bookmarked objects, shown with bookmarks. Objects on evicted pages are outlined. (b) After completing the marking phase, reachable objects are either marked or evicted. Marked objects are dark gray. (c) HC uses bookmarks to sweep only memory-resident pages during mark-sweep collections.

Figure 3: The Hippocratic collector performing in-core mark-sweep collection using bookmarks (HC also uses bookmarks during copying compaction).

location within the available physical memory. In these situations, even our Hippocratic collector cannot prevent heap pages from being evicted. However, using an approach we call *bookmarking*, HC is able to limit the impact of these evictions. Despite parts of the heap not being resident in memory, bookmarking allows HC to perform full memory-resident garbage collection without paging. HC thus greatly improves garbage collection performance even when it has pages evicted to disk by *eliminating* page faults caused by allocation and garbage collection.

We now describe how the Hippocratic collector uses bookmarks to perform mark-sweep and copying collection without causing page faults and then discuss how HC processes pages to set and remove these bookmarks.

Garbage Collection with Bookmarks

Before a garbage collector can reclaim objects in the heap, it first determines which objects are reachable. While paging, VM-oblivious garbage collection performance suffers because the garbage collector visits all reachable objects, including those on evicted pages, to find the objects to which they refer. HC instead bookmarks objects that are the target of references on evicted pages. These bookmarks act as roots that allow full memory-resident collections without accessing the evicted pages and thus without triggering any page faults.

Figure 3 shows HC performing the marking phase of collection using bookmarks. While heap pages are evicted, HC begins each garbage collection by scanning the superpages for memory resident, bookmarked objects. During this scan, HC marks the bookmarked objects and treats them as if they were root-referenced (Figure 3(a)). After completing this scan, HC knows all the references on evicted objects and does not need to fault pages back in. HC now follows its usual marking phase except to ignore references to evicted objects. When the marking phase completes, all the reachable objects will be either marked or evicted (Figure 3(b)). During mark-sweep collections, HC sweeps the memory-resident pages and the collection completes (Figure 3(c)).

For copying collection, only slightly more processing is needed. After marking the heap, HC updates the marked counts to reserve space for every possible object on an evicted page. HC then selects all superpages containing bookmarked objects or evicted pages as targets. Since their superpages must be targets, this ensures HC never moves evicted objects. Similarly, HC never copies bookmarked objects and will not update (evicted) pointers to the object. HC can then select any other needed targets and is now able to perform its usual copying collection.

Page Eviction

HC eliminates page faults during garbage collection because it can bookmark objects before the virtual memory manager evicts pages. If HC is notified of an upcoming eviction and cannot shrink the heap or find a discardable page, it selects a victim page. Our current strategy chooses victim pages that have not recently been evicted and prefers those that contain few pointers.

Having selected the victim page, HC now scans each of the page's objects. During this scan, HC looks for references, bookmarks the target, and increments a counter in the target superpage's header. Once all HC processes all of the objects on the page, it can be evicted. Because the page has just been touched, however, the virtual memory manager would normally not evict it soon. HC communicates with the virtual memory manager one more time, specifying that the page should be evicted.

The mutator could change the references on the page after its eviction, but the page would first have to be brought back into main memory. Therefore, HC can rely on the bookmarks to remember the evicted objects' references until the evicted page is faulted back into memory.

Page Reloading

HC eliminates page faults within the garbage collector, but cannot prevent mutator page faults. Our extended virtual memory manager notifies the collector whenever a heap page is faulted in. Because the notification arrives before the page becomes memory resident, HC can remove the bookmarks created when the page was evicted.

By determining the size class of the objects on the page, HC can find each of the objects on the page. The collector scans each object and, analogous to HC's processing during eviction, HC decrements the bookmark counter for referenced objects' superpages.

Superpages whose bookmark counters are reduced to zero no longer contain any objects referred to from evicted objects. Since all references to the object can be found during collection, HC clears all the bookmarks on the superpage.

Limitations of Bookmarking

Bookmarks guarantee that reachable objects are never reclaimed and eliminate page faults caused by the garbage collector. However, there is a space cost to bookmarking. Because HC must target all superpages containing bookmarked objects and evicted pages, compacting collection cannot always minimize the size of the heap. Additionally, HC must treat all bookmarked objects as reachable. We believe that avoiding paging during garbage collection is worth this potential cost in excess space retention.

3.5 Collector Throughput

Collector performance when there is no memory pressure is just as important as performance under memory pressure. We have designed the Hippocratic collector to maintain good throughput both while paging and when not paging. While the mutator primarily accesses newly-allocated objects, HC's segregated superpages cannot take advantage of this. HC instead allocates into a bump pointer-based nursery. Allocation into this nursery is faster than into segregated superpages, and, because most objects die young, collecting this nursery is also very cheap. While this nursery cannot support bookmarking, the frequency of mutator accesses would make these pages poor eviction choices. Including this nursery within the Hippocratic collector improves collector throughput without adding significantly to memory pressure.

Collecting this nursery requires a copy reserve, which could add memory pressure. However, HC's space management quickly shrinks the size of both the nursery and the copy reserve when memory pressure increases. As a result of this careful management, adding the nursery does not change HC's memory demands.

This nursery also allows us to consider generational collection. Generation collection often improves garbage collector performance. It also improves page-level locality by focusing both garbage collector and mutator activity on the nursery [17]. We therefore also implemented a generational Hippocratic collector (GenHC).

GenHC needs a method of remembering pointers from the older to the younger generation. GenHC normally stores these pointers in page-sized write buffers, which allow fast storage but could demand unbounded amounts of space. GenHC limits this space overhead by processing buffers when they fill. During this processing, GenHC removes pointers from the older generation and replaces them with an object-based card marking scheme. This filtering provides the fast storage of the write buffers but need only a single page. Given the increased throughput of generational collection, using an extra page from the heap is not a significant problem.

3.6 HC Implementation

We have implemented the Hippocratic collector using JMTk within the Jikes RVM version 2.2.2 [3, 4]. When implementing the HC algorithm within JMTk and Jikes, we needed to make two small 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. This placement is useful for optimizing bounds checks [4], but makes it difficult for HC to find object headers when scanning pages. We solve this problem by further segmenting our allocation and allowing superpages to hold either only scalars or only arrays. Within each superpage header, HC stores the type of objects contained in the superpage. With the type and size class information from the superpage header, HC quickly locates the objects and their headers.

4. Kernel Support

The Hippocratic collector improves garbage collection paging performance primarily by cooperating with the virtual memory manager. In this section, we describe our extensions to the Linux kernel that enable cooperative garbage collection.

We implemented these modifications on the 2.4.20 Linux kernel. This kernel uses the 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). Evictions are made from the pages at the end of the inactive list.

We added notification for processes at three separate events: when a page is initially scheduled for eviction, when a page is actually evicted, and when a page is swapped back into memory. In addition to notifying the application of the specific event, the kernel includes the address of the relevant page. To keep track of the owning process(es) of a page, we applied Rik van Riel's lightweight reverse mapping patch [25]. This patch allows determination of the owning process of only those pages in physical memory. We extended this reverse mapping to include pages on the swap partition.

HC needs memory residency information with sufficient time to work with the virtual memory manager on the eventual decision. To ensure the timeliness of this communication, we implemented our notifications using Linux real-time signals. Real-time signals in the Linux kernel are queueable. Unlike other notification methods, we can use these signals without worrying about signals lost due to other process activity.

In addition to these signals, we needed one new system call, `vm_relinquish()`. This call allows user processes to voluntarily surrender an arbitrarily large number of pages, which can optionally be labeled as discardable. Our kernel can immediately reuse discardable pages. Nondiscardable pages passed to the kernel are placed at the end of the inactive queue and quickly swapped out. Our cooperative garbage collector can keep heap pages resident by first providing discardable pages to be evicted. When paging is inevitable, HC uses this system call to evict a different page after it has been bookmarked.

5. Experimental Methodology

In this section, we present our experimental infrastructure and discuss our experimental methodology, which includes both empirical evaluation and simulation studies.

5.1 Empirical Results Methodology

We have recently completed the implementation of our extended Linux virtual memory manager. We include experiments using the SPECjbb benchmark on our implemented kernel and plan to include an extensive suite of experiments in the final version of this paper. We performed these experiments on a 2.4GHz Pentium 4 Linux machine with 892MB of RAM and 2GB of local swap space. This processor includes an 8KB L1 data cache and a 512KB L2 cache. We ran each experiment five times with the system in single-user mode and the network disabled. We report the average performance of the five experiments.

5.2 Simulation Methodology

We performed our remaining experiments on a 1.7GHz Pentium 4 Linux machine with 512 MB RAM. This processor has a 8KB L1 data cache, a 12KB L1 instruction cache and a 256KB L2 cache. For the experiments investigating paging behavior, we use a simulated virtual memory manager. Running in a simulator allows us to compare program performance independent of scheduler and I/O effects. Simulations also make it easier to tease apart differences in collectors' paging behavior.

Our simulation infrastructure includes a simulated virtual memory manager based on a segmented queue [6]. Our LRU queue algorithm traps accesses to all but a small number of the most recently used pages. The protected pages are kept in strict LRU order. Since most accesses are to these first few pages, leaving these pages unprotected provides a high-quality approximation to LRU order at reasonable cost. When Jikes touches a protected page, our simulator moves this page into the upper region of the queue and removes its protections. During start-up and until the unprotected region is full, this completes the process. Otherwise, our simulator selects an unprotected page at random, protects its contents, and moves it to the head of the protected LRU queue.

For our simulation, all heap pages are initially mapped by Jikes without read, write, or execute access. Any access to a page therefore generates a segmentation fault. We implement our simulator within the Jikes segmentation fault handler using page protection as a means of trapping accesses. While this simulation slows overall program execution, nearly all of the extra time is spent within the kernel. We estimate running times by combining user mode execution times with an aggressive 5ms penalty for each simulated page fault.

Benchmark statistics				
Benchmark	Total Bytes Alloc	Min. Heap Size	Alloc/Min.	Description
_201_compress	314,289,252	11,534,336	27.25	Compression/decompression
_202_jess	668,792,828	7,340,032	91.11	Java Expert System Shell problem solver
_205_raytrace	384,138,556	8,388,608	45.79	Raytrace generator
_209_db	311,160,772	13,631,488	22.83	Database-like query program
_213_javac	991,272,712	14,680,064	67.86	Java compiler
_228_jack	764,949,296	8,388,608	91.19	Code parser
pseudoJBB	757,154,424	30,408,704	24.90	Java beans benchmark

Table 1: Memory usage statistics for our benchmark suite.

6. Results

In this section, we present our experimental analysis of the Hippocratic garbage collection algorithm. We first analyze the performance of HC and GenHC when there is no memory pressure and then examine how well they respond to memory pressure.

We compare the performance of our HC and GenHC collectors with four of the collectors included with Jikes: SemiSpace, CopyMS (a whole heap collector with a bump-pointer nursery and mark-sweep mature space), GenCopy, and GenMS (Appel-style generational collectors using bump-pointer and mark-sweep mature spaces, respectively)[2]. We evaluated the benchmarks listed in Table 1 over a range of heap sizes. For these experiments, we use "OptOptFast" configurations. These configurations use the optimizing compiler to build as much code as possible into the boot image. Additionally, the optimizing compiler is used on the running program. While most virtual machines use an adaptive system to optimize only "hot" methods, research has found that this skews results for short-running programs [10]. We thus use the optimizing compiler to ensure that each garbage collection algorithm runs with an identical load.

6.1 Throughput without memory pressure

While we designed HC and GenHC to perform well while paging, these algorithms would not be useful if they did not provide good throughput when not paging. Figure 4 presents the results of experiments for the case when there is adequate memory to run the benchmarks without any paging. Limiting fragmentation and eliminating the copy reserve has additional benefits of allows HC to run in very small heaps. HC runs in heaps smaller than CopyMS requires for every benchmark we examine, while SemiSpace runs in this minimum heap size only once. As Figure 4(f) shows, HC runs in under 40% of the heap space SemiSpace needs and less than 20% of the heap CopyMS needs. Similarly, Figure 4(d) shows HC needing less than 44% of the heap space needed by Copy MS and 36% of that needed by SemiSpace.

While this improved space utilization is helpful when paging, it also enables HC to outperform other collectors at the smallest heap sizes. As Figure 4(a) shows, using HC to run jess is at least 15% faster than CopyMS and from 1% - 28% faster than SemiSpace. Figure 4(c) and Figure 4(e) show the two benchmarks in which SemiSpace significantly outperforms HC. Cache locality strongly influences performance on db as Figure 4(c) reflects. Figure 4(e) shows HC's performance on jack. HC suffers at the smallest heap sizes because a large number of superpages are tied up by a small number of objects.

Figure 4(g) shows the average increase in execution time relative to HC at each relative heap size. Except for jack, HC can run in heaps at least 25% smaller than that needed by SemiSpace; these two smallest heap sizes include only results from jack. Just as SemiSpace's 50% runtime increase at 1.5 times the minimum heap size is due to a single outlier (in that case, a 550% increase in the time needed to run raytrace), so are the leftmost data. On average, HC outperforms CopyMS by at least 4%. While SemiSpace performs slightly better on heaps almost four times HC's minimum size, the average improvement is at most 4%.

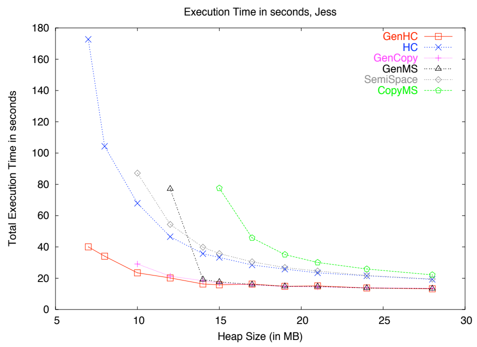
Few systems use whole heap collection, however. Most existing applications rely on a generational collector because they provide good throughput and limited garbage collection pauses. We now discuss GenHC's performance to two generational collectors distributed with JMTk: GenCopy (using copying collection in its older generation) and GenMS (which uses mark-sweep collection for the older objects).

We find that, like HC, GenHC runs in significantly smaller heaps than either of the other two generational collectors. We found that our filtering remsets were so successful that they enabled GenHC to run in heaps just as small as those needed by HC. As is the case for whole-heap collectors, GenMS never completes within the minimum heap size needed by GenHC and SemiSpace can only match GenHC's heap utility for jack. Figure 4(d) shows that both GenMS and GenCopy need heaps 50% larger than that needed by GenHC to run javac. Similarly, in Figure 4(f) GenCopy needs a heap 50% larger that that needed by GenHC. While the improvements to the needed heap size are less dramatic for other collectors, this improved space utilization provides a nice validation of fragmentation approach.

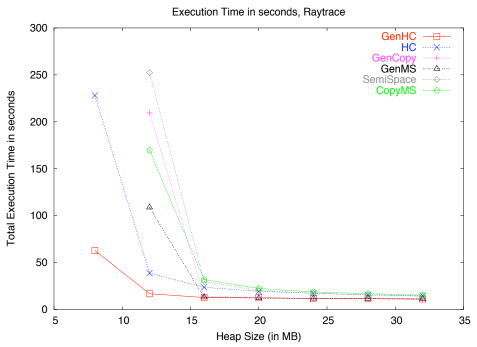
GenHC's performance on the locality-driven db is an outlier in our experiments. As can be seen in Figure 4(c), while GenHC is consistently within 4% of GenMS, it performs up to 14% worse than GenCopy. Running javac, GenHC is at least 5% faster than GenCopy at all but the largest heap sizes and, even at these large sizes, remains slightly faster than the copying collector. It also compares favorably with GenMS. Once GenMS catches up with GenHC's performance at 2.5 times the minimum heap size, execution times of the two collectors stay within 1% of each other.

Figure 4(h) shows that GenHC substantially outperforms the other generational collectors at the smallest heap sizes. As the heap grows, differences between the collectors largely averages out. At the largest heap size, GenCopy runs a mere 3% faster. This small runtime improvement comes at a significant cost, however. Running compress at four times the smallest heap size any collector needs for this benchmark, GenMS is 6% faster than GenHC. While this is the only instance where GenMS improves upon GenHC by more than 4%, there is at least one heap size per benchmark where GenMS runs at least 7% slower. Similarly, the time savings offered by GenCopy must be weighed against its needing 3.5 times

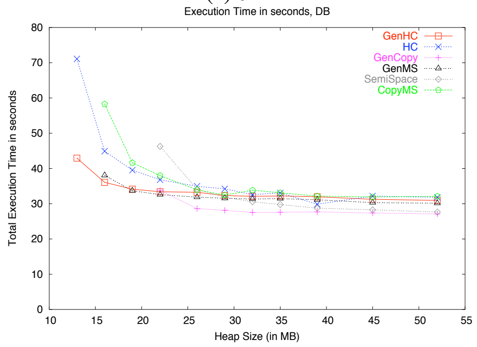
[2]We are working on properly accounting for the page usage of the MarkSweep collector in JMTk, and plan to include this collector in the final version of the paper.



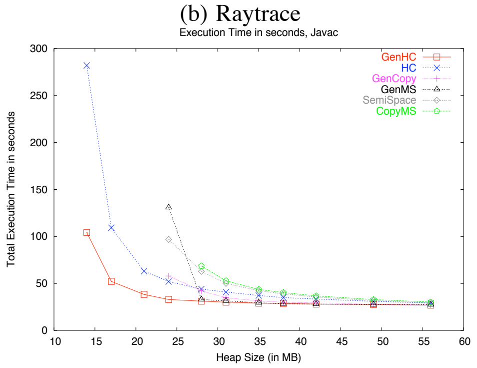
(a) Jess



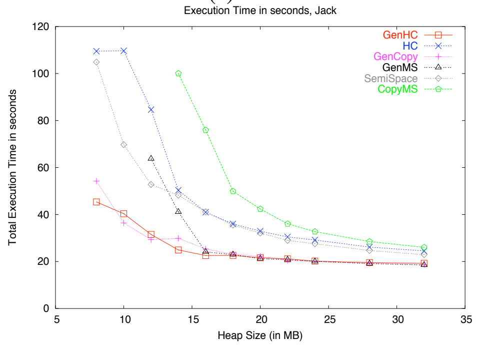
(b) Raytrace



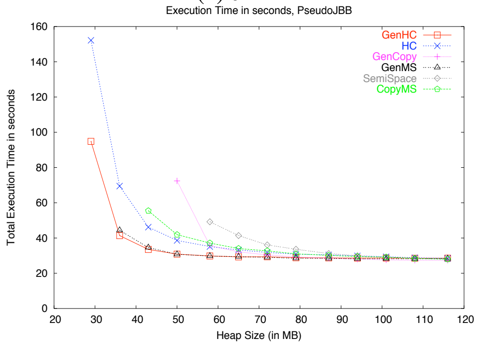
(c) DB



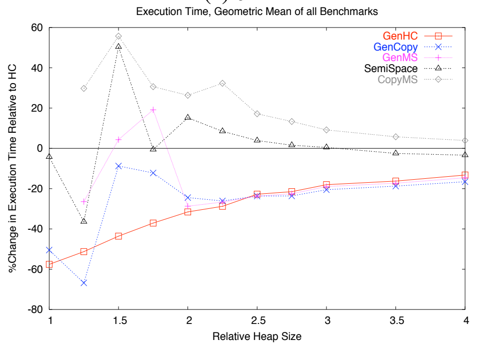
(d) Javac



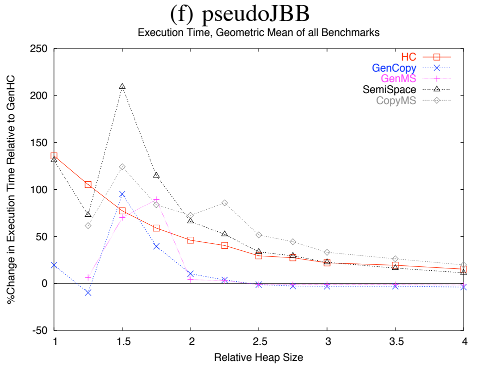
(e) Jack



(f) pseudoJBB



(g) Geo. Mean Relative to HC



(h) Geo. Mean Relative to GenHC

Figure 4: Relative performance of the collectors across a range of benchmarks when memory is plentiful. The generational Hippocratic collector (GenHC) generally provides the best performance and is often able to run in smaller heaps than the other collectors.

as much time and 50% more space to run raytrace than GenHC (at 1.5 times the smallest needed heap, GenHC completes in 16.86 seconds versus 228.01 seconds for GenCopy).

6.2 Fixed Memory Pressure

While HC and GenHC perform well when not paging, their throughput while paging is even more impressive. Figure 5 shows the simulated execution times of runs using a single heap size and simulating available memory from 45% to 100% of the heap size. Figure 5(a) examines runs of "extended" pseudoJBB using an 87MB heap. To increase the running time of the application and ensure that compilation does not dominate behavior, these runs double the usual number of transactions pseudoJBB performs. Figure 5(b) shows results for runs of javac using a 49MB heap.

In both of these figures, there is little performance difference between the whole-heap collectors when the heap fits entirely in memory. HC prevents running with less physical memory from having any substantial impact. HC runs 21% slower in 45% less available memory. This contrasts with SemiSpace. Both graphs in Figure 5 show that even the least number of evicted pages cause SemiSpace runtime to soar. The effectively random access patterns exhibited during garbage collection for SemiSpace leads paging to dominate its performance and execute up to eight times slower than HC. CopyMS focuses allocations onto a small number of nursery pages and, as implemented within JMTk, rarely reallocates heap pages once freed from the older space. While leading to a large memory footprint, this minimizes paging while the nursery stays resident. As can be seen in both benchmarks, however, CopyMS's performance resembles SemiSpace's once nursery pages are evicted. The Hippocratic collector offers significant savings when paging.

Even at the rightmost point on the graph, when the available memory equals the heap size, GenMS triggers nearly 20,000 faults and GenCopy causes over 30,000 faults when running the extended pseduoJBB. Even with what should be the "correct" heap size, paging causes these collectors run at least 45% slower than GenHC in Figure 5(a) and to take at least 3 times as long in Figure 5(b).

As predicted by Moon [17], the collector and mutator's focus on accessing the nursery limits either collector from initially suffering any further significant paging penalties until available memory becomes so limited that the VM starts evicting pages from the nursery. The left side of Figure 5(a) shows that, once nursery pages are evicted, the collectors suffer significant performance penalties because of paging. By contrast, once notified by the VM of increasing memory pressure, GenHC limits allocations to prevent heap pages from being evicted. By cooperating with the VM, our generational collector can execute this workload faster with only 38MB of available memory than either of the other generational collectors execute with the full 87MB of memory.

6.3 Dynamic Memory Pressure

We now examine the effects of dynamic memory pressure on garbage collection paging behavior. In particular, we consider the performance impact of increasing memory pressure caused by an application starting up or another application requesting memory. We model this by rapidly reducing the number of available physical pages until reaching our target memory size. Our simulated VM reclaims the last page on the LRU every 20ms of program execution.

We run each three iterations of each benchmark. Figure 6 shows the simulated time required for each collector. These graphs show the final size of available memory along the X-axis. These target memory sizes vary from 95% - 50% of the heap size.

We expected SemiSpace to perform poorly while paging, but were surprised at the 7 to 14-fold increase in execution time relative to HC seen in Figure 6(a) and the 9 to 15-fold increases in Figure 6(a). We discovered that SemiSpace suffers from poor paging behavior because it effectively loops through memory. Looping over more memory than available triggers LRU's well-known worst-case behavior. The remaining VM-oblivious collectors behave much as they did with fixed memory pressure.

Figure 6 show that the Hippocratic collectors adjust to changing memory pressure while maintaining the high throughput they have in all our experiments. Bookmarking is especially helpful here. At the smallest memory sizes of Figure 6(a), GenHC spends over half of its execution with an average of 101 pages evicted while HC averages 167 pages evicted. Bookmarking allows both collectors to proceed without suffering a single page fault.

6.4 Empirical Results

We have recently implemented our extended Linux virtual memory manager, and Figure 7 shows our experiments with the different collectors running pseudoJBB on this kernel. To generate equal memory pressure for each run, we used a program that allocated and locked memory at application start-up and then touched these pages every 5 seconds. While two of these jobs ran in the background, we ran pseduoJBB on Jikes using each collector.

Figure 7(a) shows the average runtime for each collector at a range of heap sizes over five executions. Not only do these results validate our simulations, they also show additional performance penalties caused by a large number of page faults in a system running multiple processes. Besides the collectors we previously discussed, this graph includes data from two variants of GenHC: *w/o cooperation* and *w/o communication*. The first receives signals from the virtual memory manager, but does not cooperate to give up discardable pages or suggest eviction targets. The second runs GenHC in a purely VM-oblivious manner. When unable to communicate or cooperate, GenHC performs about as well as any other algorithm. These final results make it clear that cooperation with the virtual memory manager is vital in order to perform well while paging.

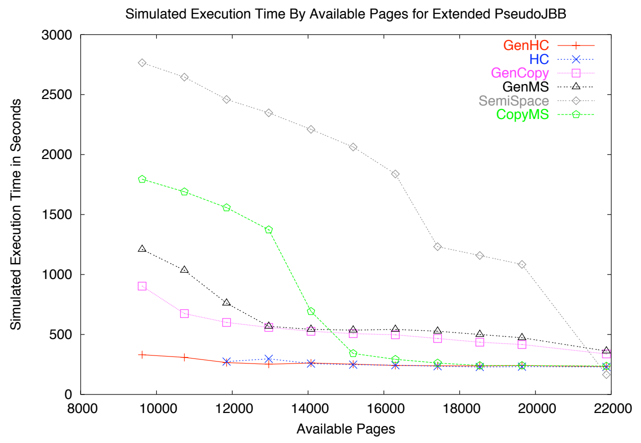
7. Conclusion and Future Work

This paper introduces page-level cooperative garbage collection, in which the garbage collector and virtual memory manager cooperate to improve application performance while paging. We present our Hippocratic collector, which cooperates with the VM to limit mutator page faults and uses a novel "bookmarking" approach that eliminates paging caused by the garbage collector. When memory pressure is low, the generational variant of the Hippocratic collector requires up to 50% smaller heaps while performing competitively with some of the best existing collectors. With our extended Linux kernel and under memory pressure, the Hippocratic collector executes pseudoJBB in almost half the time required by the next best garbage collector.

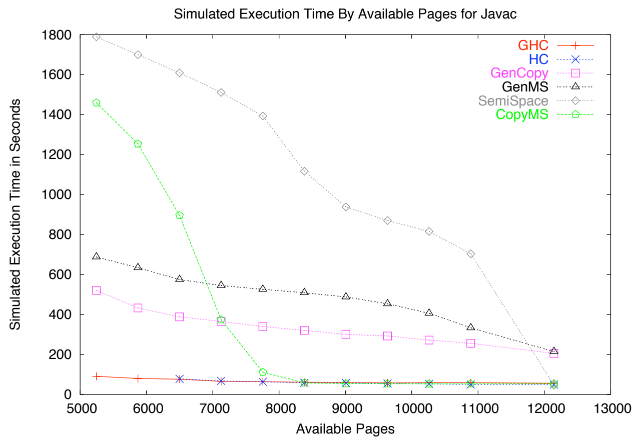
While our results show that this approach already yields signficant performance improvements and robustness under memory pressure, there are several directions we would like to advance this work. Our Hippocratic collector currently focuses on finding a heap size in which it can run that does not significantly increase memory pressure. We are exploring extensions to the virtual memory manager which will allow HC to cheaply determine when it is appropriate to grow the resident heap. We are also planning to incorporate recency information from the kernel to decide which non-discardable pages to evict.

Acknowledgements

Thanks to Eliot Moss, Pritesh Sharma, and Ben Zorn for their insightful comments and feedback. We are especially grateful to

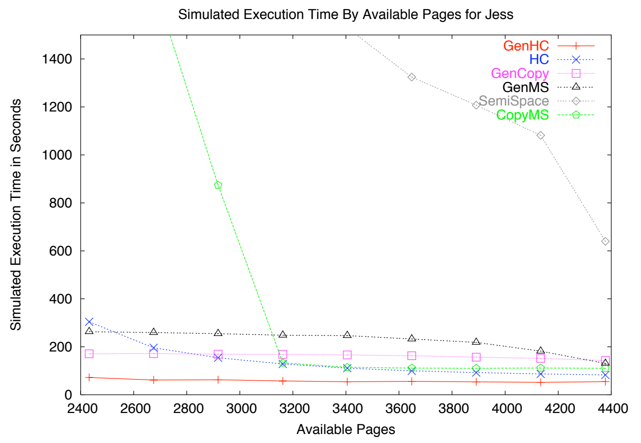


(a) Simulated execution time running extended pseudoJBB with an 87 MB heap

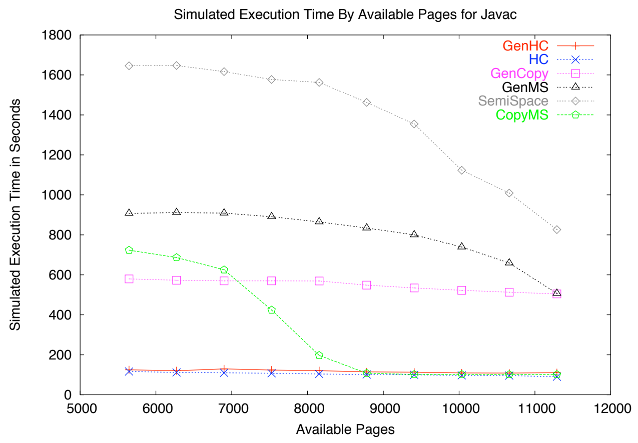


(b) Simulated execution time running javac with a 49MB heap

Figure 5: Simulated execution time (lower is better) for all collectors when available memory is fixed. Both Hippocratic collector variants outperform other collectors at almost every memory size.



(a) jess with a 20MB heap



(b) javac with a 49MB heap

Figure 6: Simulated execution time when available memory is reduced at runtime.

Scott Kaplan for his invaluable contributions to our understanding and assistance in the implementation of our modified Linux virtual memory manager.

8. References

[1] Sun JDK 1.4.0_01. Available at http://java.sun.com/j2se.

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

[3] B. Alpern, C. R. 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. F. Mergen, T. Ngo, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalepeño virtual machine. *IBM Systems Journal*, 39(1), Feb. 2000.

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

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

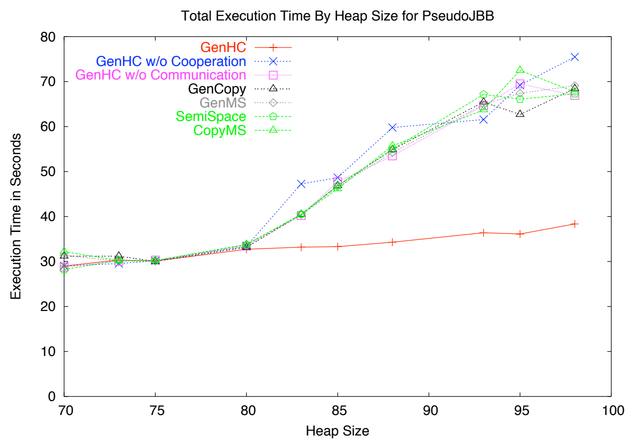
[6] O. Babaoglu and D. Ferrari. Two-level replacement decisions in paging stores. *IEEE Transactions on Computers*, C-32(12):1151–1159, Dec. 1983.

[7] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 117–128, Cambridge, MA, November 2000.

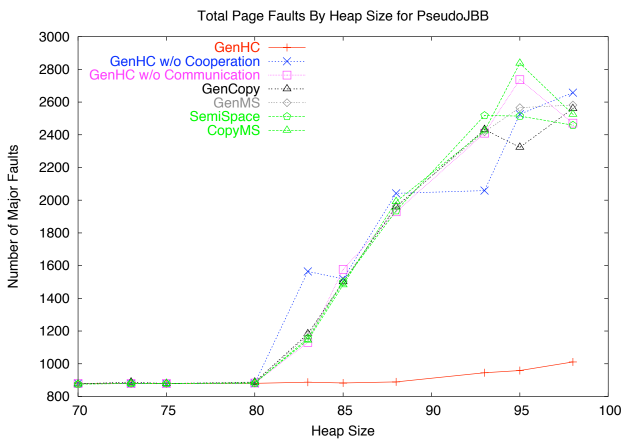
[8] E. Cooper, S. Nettles, and I. Subramanian. Improving the performance of SML garbage collection using application-specific virtual memory management. In *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming*, pages 43–52, San Francisco, CA, June 1992.

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

[10] L. Eeckhout, A. Georges, and K. D. Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the ACM SIGPLAN 2003 Conference on*



(a) Actual execution time running pseudoJBB



(b) Number of major faults running pseudoJBB

Figure 7: Actual execution time and page fault count for all collectors on our modified Linux kernel. Our cooperative, communicating collector performs well while paging, requiring as little as half the time as other algorithms.

Object-Oriented Programming Systems, Languages and Applications, volume 38(11), pages 169–186, Oct. 2003.

[11] 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.

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

[13] R. Jones and R. Lins. *Garbage collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, New York, NY, 1996.

[14] K.-S. Kim and Y. Hsu. Memory system behavior of java programs: Methodology and analysis. In *Proceedings of the ACM SIGMETRICS 2002 International Conference on Measurement and Modeling of Computer Systems*, volume 28(1), pages 264–274, Santa Clara, CA, June 2000.

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

[16] D. McNamee and K. Armstrong. Extending the Mach external pager interface to accomodate user-level page replacement policies. In *Proceedings of the USENIX Association Mach Workshop*, pages 17–29, 1990.

[17] D. Moon. Garbage collection in a large Lisp system. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 235–246, Austin, TX, Aug. 1984.

[18] S. Platt, editor. *Respectfully Quoted: A Dictionary of Quotations Requested from the Congressional Research Service*. Government Printing Office, Washington, D.C., 1989.

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

[20] P. Savola. Lbnl traceroute heap corruption vulnerability. http://www.securityfocus.com/bid/1739.

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

[22] D. Stefanovic. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. PhD thesis, University of Massachusetts, 1999.

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

[24] D. 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*, pages 157–167, Pittsburgh, Pennsylvania, Apr. 1984. *ACM SIGPLAN Notices 19*, 5 (May 1984).

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

[26] P. R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, St. Malo, France, Sept. 1992. Springer-Verlag.

[27] 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.

[28] P. R. Wilson, M. S. Lam, and T. G. Moher. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 26(6), June 1991.