The Mapping Collector: Virtual Memory Support for Generational, Parallel, and Concurrent Compaction

Michal Wegiel
Computer Science Department
University of California, Santa Barbara
mwegiel@cs.ucsb.edu

Chandra Krintz
Computer Science Department
University of California, Santa Barbara
ckrintz@cs.ucsb.edu

Abstract

Parallel and concurrent garbage collectors are increasingly employed by managed runtime environments (MREs) to maintain scalability, as multi-core architectures and multi-threaded applications become pervasive. Moreover, state-of-the-art MREs commonly implement compaction to eliminate heap fragmentation and enable fast linear object allocation.

Our empirical analysis of object demographics reveals that unreachable objects in the heap tend to form clusters large enough to be effectively managed at the granularity of virtual memory pages. Even though processes can manipulate the mapping of the virtual address space through the standard operating system (OS) interface on most platforms, extant parallel/concurrent compactors do not do so to exploit this clustering behavior and instead achieve compaction by performing, relatively expensive, object moving and pointer adjustment.

We introduce the Mapping Collector (MC), which leverages virtual memory operations to reclaim and consolidate free space without moving objects and updating pointers. MC is a nearly-single-phase compactor that is simpler and more efficient than previously reported compactors that comprise two to four phases. Through effective MRE-OS coordination, MC maintains the simplicity of a non-moving collector while providing efficient parallel and concurrent compaction.

We implement both stop-the-world and concurrent MC in a generational garbage collection framework within the open-source HotSpot Java Virtual Machine. Our experimental evaluation using a multiprocessor indicates that MC significantly increases throughput and scalability as well as reduces pause times, relative to state-of-the-art, parallel and concurrent compactors.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory Management (Garbage Collection); D.4.2 [*Operating Systems*]: Storage Management—Virtual Memory

General Terms Algorithms, Languages, Performance

Keywords Virtual Memory, Compaction, Parallel, Concurrent

ASPLOS'08, March 1–5, 2008, Seattle, Washington, USA.
Copyright © 2008 ACM 978-1-59593-958-6/08/03…$5.00

1. Introduction

Modern systems are increasingly complex implementing multi-layered software stacks and employing more and more processing cores. These systems support a vast diversity of applications ranging from multi-media and software development to web-services and distributed gaming (among others). To extract high performance from these systems, it is vital that the layers of the software stack cooperate efficiently to make the most of the underlying hardware resources. Two key layers common to most extant systems, in support of these applications, are the operating system (OS) and the managed runtime environment (MRE) – the popular execution environment for portable, type-safe applications (e.g., those written in Java or the .Net languages).

An MRE component that can significantly impact the performance of applications and that has the potential for more intelligent coordination between the MRE, operating system, and underlying architecture, is memory management. Garbage collection (GC) in MREs typically is multi-threaded (parallel) and/or concurrent [20] to exploit increasing numbers of available processing cores, and employs compaction to eliminate heap fragmentation to enable fast object allocation [19]. Extant compacting GCs achieve compaction by moving live (reachable) objects. This involves copying, which is increasingly expensive because of the growing processor-memory performance gap, and can adversely impact application performance [6].

To address this limitation and to improve the performance of MRE systems, we investigate a new approach to parallel/concurrent, compacting, garbage collection in which the MRE coordinates its memory management efforts with the virtual memory system of the underlying machine – to eliminate object copying altogether. Our GC, to which we refer as the Mapping Collector (MC), exploits the widely-known phenomenon that objects with similar lifetimes tend to exhibit spatial locality in the heap [33]. In particular, we find that dead objects often occur in large clusters. MC exploits this behavior to consolidate free space (compact the heap) through virtual page mapping operations available via standard, cross-platform, unprivileged, OS system calls. Such reclamation is simple to implement and does not require OS modification. Moreover, MC avoids object copying and its associated costly memory operations, and facilitates a very efficient GC implementation. To enable this, MC trades off a small heap space overhead for fast, inexpensive compaction. In practice, this space overhead is below 6% on average and MC can additionally bound it by an infrequent fall-back to state-of-the-art, perfect compaction based on object moving.

The contributions that we make herein include:

- The design and implementation of MC, a generational [19] GC that supports both stop-the-world (STW) and concurrent compaction. We implement MC in the open-source HotSpot Java

Virtual Machine [22]. MC is applicable to both server systems (which typically employ concurrent GC to reduce pause times at the cost of resource over-provisioning [18]) and deskside systems (which tend to use STW GC because of its simplicity, higher throughput, and more efficient use of the underlying resources [18]).

- Virtual memory support for copy avoidance. Unlike previously reported systems, MC employs virtual memory unmapping as a primary and sole technique to implement STW/concurrent compaction in a modern MRE. MC achieves the same effect as object moving but avoids object copying and thus improves GC performance while imposing a small space overhead.
- Reduction in the number of GC phases. MC is a nearly-single-phase compactor while extant compacting GCs require at least two phases. MC enables this by replacing object moving with virtual page unmapping (whose cost is proportional to a liveness bitmap size that is approximately 3% of the heap size).
- An experimental evaluation of MC against both STW and concurrent state-of-the-art compacting GC systems. We empirically evaluate the throughput, pause times, and scalability of MC and compare MC against the STW and concurrent versions of the Compressor [20], and the production-quality, highly-optimized parallel STW compacting GC currently available in the HotSpot JVM. We evaluate MC on a multiprocessor system using a set of standard community benchmarks and find that MC enables significant performance gains for these metrics.

Prior work on compaction has focused on both partial elimination of object moving [13, 18] and reducing the number of GC phases [20, 1, 14, 28]. MC leverages MRE-OS interaction to improve over these approaches by eliminating copying altogether. Virtual memory support for GC has been shown to be effective in other contexts including preventing collector-induced paging [15, 35, 36, 16, 34] and reducing the space overhead of copying collection via page unmapping [20, 27].

In the next section, we overview the background and related work. We then detail the design and implementation of MC (Section 3), present the results of our empirical evaluation (Section 4), and conclude in Section 5.

2. Background and Related Work

While most prior work on parallel/concurrent compaction has focused on virtual-memory-oblivious compactors, the interaction between the collector and virtual memory has recently gained interest [20, 35, 16]. Previously reported compactors achieve compaction by moving all (or some [13, 18]) live objects and need at least two phases. MC attempts to achieve compaction without any object moving and is a nearly-single-phase compactor. Following the methodology used in [20], we define a GC phase as an operation with cost proportional to the heap or live data size. The first phase in state-of-the-art compactors is marking [19] which identifies live objects through parallel/concurrent tracing.

2.1 The Compressor

The Compressor [20] is a parallel compacting GC that requires two phases: marking and compaction. It supports both STW and concurrent collection. The Compressor (herein referred to as CP) uses virtual memory operations (i.e., page mapping and unmapping) but accomplishes compaction by moving live objects and adjusting the pointers. The compaction is perfect (i.e., heap fragmentation is fully eliminated). The compactor employs two virtual spaces and copies objects page by page from one space to the other. CP, akin to a copying collector, always moves all objects. It updates pointers after moving using information it has recorded in auxiliary data structures (which include the block-offset array). This process is accompanied by freeing pages in the source space and allocating pages in the destination space. CP imposes a small constant space overhead (1.5% for 256-byte blocks) for auxiliary data structures. By contrast, MC performs compaction in nearly one phase and eliminates object moving and pointer adjustment. MC imposes a variable space overhead (on average $<6\%$, which can be bounded). Both compactors preserve object order. CP unmaps and maps the entire heap each time the compaction is invoked. MC limits the number of virtual memory operations to the number of dead-object clusters.

2.2 The HotSpot Compactor

The parallel compactor currently available in the HotSpot JVM [18] (herein referred to as HS) is similar in spirit to the Compressor. HS updates pointers in the same way but moves objects only when it is necessary. HS is a STW virtual-memory-oblivious collector with two phases: marking and compaction. HS divides the heap into fixed-size regions (chunks) and uses a liveness bitmap to record the locations of live objects. During marking, HS computes additional per-chunk data needed for pointer adjustment. The compaction phase is parallel. Threads claim available regions atomically and fill them with live objects. A region becomes available when all its objects have been evacuated (it is empty) or it has been compacted onto itself. HS updates interior object pointers as it fills regions. Filling a region does not require synchronization and involves identifying source objects destined for the region and copying them until the region is full or no more objects are left. HS computes a new location of a live object as the start of its destination region plus the size of live objects that precede the object in that region. HS performs perfect, sliding compaction and preserves the object order. HS imposes a constant space overhead of 3% (needed for per-region data that includes the current compaction state for each region). The advantages of MC over HS are similar to those over the Compressor: nearly one GC phase (instead of two) and avoidance of object moving and pointer manipulation.

2.3 The IBM Compactor

The IBM collector [1] is a parallel STW compactor that comprises three phases: marking, object moving, and pointer fix-up. This collector does not manipulate virtual memory mapping. It does not guarantee perfect compaction and, therefore, imposes an application-specific space overhead, similarly to MC. The system divides the heap into fixed-size blocks. Initially, GC threads perform intra-block compaction and proceed to inter-block compaction as free contiguous areas begin to appear in the already-compacted blocks. In the moving phase, the system collects information needed for pointer adjustment. In the final phase, the system divides the heap into as many areas as there are GC threads, and each thread redirects pointers in its own area. Pointer adjustment is performed in a similar way as in the Compressor. In contrast, MC neither moves objects nor updates pointers and is a nearly-single-phase collector.

2.4 The Flood Compactor

The compactor presented by Flood et al. [14] is a parallel version of the Lisp2 [19, 11] collector. This STW GC requires four phases: marking, forwarding pointer installation, pointer adjustment, and object moving. The heap is divided into p contiguous regions where p is the number of parallel GC threads. The sliding direction alternates between left and right for even and odd regions and as a result $\frac{p}{2}$ groups of objects are formed in the heap. Thus, free space is consolidated only partially. This compactor uses forwarding pointers instead of the block-offset array and the mark-bit vector. Thus, pointer updates are more efficient but one additional phase is necessary. MC achieves higher-quality compaction (the free space is mostly consolidated, no object groups are formed) in nearly one phase and without object moving and pointer adjustment.

2.5 Memory Management with Virtual Memory Support

Recently proposed collectors that leverage virtual memory either focus on copying, not on compaction (like MC), or aim at reducing heap space usage, not at avoiding object moving (like MC). For example, MarkCopy [27] leverages virtual memory mapping to reduce the space overhead of a copying collector. The collector does not require a copy reserve since it maps and unmaps consecutive pages as copying progresses (in a way similar to that of the Compressor [20]). Unlike MC, these approaches involve object moving.

Collectors that cooperate with the virtual memory manager to reduce the collector-induced paging [15, 35, 16, 36, 34] are orthogonal and complementary to MC. The Bookmarking collector [16] records summary information about outgoing pointers from evicted pages to avoid accessing non-resident pages during full-heap compacting collections. CRAMM [35] and IV heap sizing [15] use VM paging behavior to predict and set dynamically an appropriate, application-specific, heap size that adapts to changing memory pressure.

The Boehm-Demers-Weiser [8] garbage collector is a mark-sweep (non-compacting) collector for C/C++ which uses page unmapping as an optional and supplementary mechanism to reduce fragmentation. This collector is conservative (i.e., not all garbage can be identified). In contrast, MC employs unmapping as a primary technique to achieve compaction and is the first to do so among non-conservative (precise) collectors. Doug Lee's malloc library [21] uses mmap/munmap primitives for memory allocation/reclamation. This system, however, does not support or provide garbage collection.

An alternative to STW collection is concurrent GC, commonly employed for server systems, which interleaves application (mutator) and GC execution via additional synchronization and resource (memory and processor) over-provisioning, to reduce GC pause times. The concurrent version of the Compressor [20], Garbage-First collector [13], and mostly-concurrent mark-sweep [24] are recent examples of concurrent GCs. Concurrent collectors commonly protect virtual pages in order to detect conflicts with mutators and to exploit cache locality [20]. Extant systems supporting concurrent/parallel collection either do not attempt compaction [25, 24, 23, 3, 4, 7] or move/copy live objects [20, 17, 10, 2, 13]. In contrast, MC achieves compaction without object moving.

3. The Mapping Collector (MC)

MC exploits the widely-observed statistical property that unreachable objects tend to cluster together [33] and form contiguous dead regions in the heap. Our experimental analysis of modern Java programs (which we present in Section 4) confirms this property and reveals that clusters of dead objects are often sufficiently large to make their reclamation via virtual page unmapping practical.

Extant garbage collectors do not take advantage of the level of indirection offered by virtual memory and compact the heap by moving objects and updating pointers. MC remaps the free space into a contiguous region in a newly allocated area in virtual memory. This approach is simpler and more efficient than object copying and pointer adjustment. It enables nearly-single-phase compaction, while state-of-the-art compactors comprise at least two phases. In addition to marking, MC requires only a single traversal over the liveness bitmap (whose size is 3% of the heap).

To achieve portability, MC relies only on standard virtual memory operations [26], such as page mapping and unmapping, that are available for (unprivileged) processes as part of an operating system interface (system calls) on most modern platforms. We note that it is not sufficient to rely on the OS paging mechanism to swap out unreachable, never-accessed pages, and completely avoid garbage collection. Periodic page unmapping is necessary to free the associated OS resources (e.g., the swap space) – otherwise they are not freed until program termination.

Since virtual page granularity is larger than the unit of allocation (most objects are small) and because of the page alignment requirements of modern systems (e.g., 4KB in Linux), MC incurs a certain heap space overhead, which we evaluate in detail herein. We find that the size of the uncollected free space is modest in most cases and can be bounded via an infrequent fall-back to perfect compaction (Section 3.4).

By remapping free space into a new area in virtual memory, MC consumes increasingly more address space as subsequent compactions occur. This phenomenon, however, is not a problem on modern 64-bit architectures that have practically inexhaustible virtual address space at their disposal.

Like most state-of-the-art compactors, MC is designed for a tenured generation in a generational [31, 19] garbage collection system. In the young generation, normally a copying collection is used as it is more efficient than compaction if the expected percentage of live objects is low. The cost of collecting the tenured generation typically dominates GC performance.

The tenured generation contains objects with relatively long lifetimes and the allocation rate in the tenured generation is relatively low (compared to the young generation). Thus, the expected rate at which new dead clusters appear is low and address space usage remains tolerable even on 32-bit architectures (which we have verified experimentally).

MC consists of a single parallel marking phase (which imposes the dominant cost of the collector) and a series of operations for unmapping and updating auxiliary data structures. Unmapping occurs immediately following marking and has a cost proportional to the size of the liveness bitmap (which is approximately 3% of the mapped heap size). Thus, MC is a nearly-single-phase compactor.

MC can be implemented as both STW and concurrent compactor. During unmapping, MC does not access live objects at all, and therefore can execute concurrently with the application without the need for any synchronization. This significantly simplifies the design – note that moving compactors require OS support to handle concurrent mutations to the moved objects.

While STW compaction is triggered only upon heap space exhaustion, concurrent compaction is initiated early, when a certain heap occupancy is reached (typically around 70%). This is necessary to guarantee space for allocation while the compaction progresses in the background.

3.1 STW/Concurrent Marking

The marking phase identifies all reachable objects in the heap and records the starting and ending words for each live object in the liveness bitmap. Both STW and concurrent marking can be used with MC.

State-of-the-art STW parallel marking [14, 18] uses work stealing for dynamic load balancing. The root set is assigned to the marking GC threads in a round-robin fashion. Whenever a thread becomes idle, it steals a group of references from another (randomly-selected) thread. Each thread maintains a local marking stack (for depth-first search). To ensure that each live object is processed exactly once, marking GC threads claim objects atomically. GC threads coordinate marking termination via barrier synchronization.

State-of-the-art concurrent parallel marking [25, 18] consists of three sub-phases: STW initial marking, concurrent marking, and STW final marking. Initial marking suspends mutators to record all objects directly reachable from the roots. Concurrent marking re-

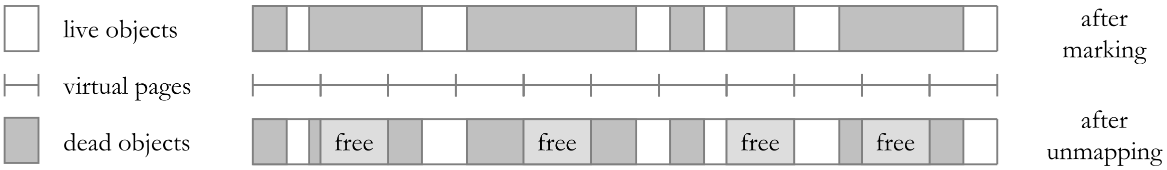


Figure 1. Page-based free space reclamation in MC. Virtual pages fully contained in dead clusters are returned to the OS.

sumes mutators and marks a transitive closure of reachable objects. Due to concurrent pointer updates some live objects might be left unmarked. Therefore, the algorithm keeps track of all pointer updates by leveraging a card table mechanism of a generational GC system. Final marking suspends the mutators and repeats marking from the roots treating modified pointers as additional roots. Final marking is typically short as it skips the already-marked objects. Each sub-phase can be executed by multiple parallel GC threads.

3.2 STW Unmapping

STW MC performs unmapping when the mutators are suspended. The goal of the unmapping scan (which amounts to a traversal over the liveness bitmap) is to return reclaimable pages to the OS and to compute the total size of free space available in dead clusters.

MC performs the unmapping scan in parallel. Since the size of the liveness bitmap is relatively small, we do not employ dynamic load balancing. MC statically partitions the bitmap into nearly-equal-sized chunks (as many as the number of GC threads). A boundary between two adjacent chunks is the first word of a live object. Thus, the subdivision does not hinder our ability to detect regions suitable for unmapping. No synchronization is necessary between the parallel threads since we divide the marking bitmap between threads at live object boundaries and, as a result, no conflicts can occur.

MC invokes the unmapping system calls in parallel which is more scalable than serialized unmapping, especially given that pages returned to the OS by different GC threads belong to disjoint virtual memory areas. OS kernels that support fine-grain locking in the memory management subsystem can likely handle such concurrency with little contention.

Figure 1 illustrates how MC reclaims free space on a virtual page basis. The unmapping scan identifies unreachable regions and unmaps their fragments that fully cover the underlying virtual pages. Since MC does not move objects, the freed areas never contract, and unmapped pages remain unused. The space overhead tends to improve over time as small dead fragments scattered across the heap assemble into larger clusters that MC can later unmap.

MC maintains a page bitmap to track heap pages that are currently unmapped. Its size is approximately 0.003% of the used address space (1 bit per 4KB). Without this additional data structure, the performance of long-running applications that exhibit high object turnover in the tenured generation may degrade. The unmapping scan traverses over the liveness bitmap which has a size of approximately 3% of the address space currently used by the heap. This includes the unmapped areas. Therefore, to keep the cost of the unmapping scan proportional to 3% of the heap size (not the used address space), MC must distinguish between mapped and unmapped regions. With this enhancement, MC can traverse (and clear) the liveness bitmap only partially (skipping the unmapped regions). In addition, this reduces the number of unmapping system calls (as we do not unmap the same clusters multiple times).

Once the unmapping scan is complete, MC expands the heap by the total size of the newly-discovered free space (not the total size of the newly-unmapped pages) in the heap (to enable identical behavior as and a fair comparison to perfect compacting collectors). The space overhead of MC then, is the size of this expansion minus the total size of the pages that MC has unmapped in the current collection cycle.

3.3 Concurrent Unmapping

In concurrent MC, unmapping takes places after resuming the mutator threads. MC first traverses over the liveness bitmap, finds dead clusters (their addresses and sizes are stored in the cluster array), and clears the bitmap. During the bitmap traversal, MC also computes a new object-start array, necessary in a generational GC system to locate the first object on any 512-byte card during the young generation collection [29]. Since these activities are performed concurrently to mutators, a young-generation GC might take place in the background (two collectors may execute at the same time). Therefore, MC must compute the object-start array using a separate (shadow) array. This translates to 0.2% space overhead (1 byte per 512 bytes). Next, MC suspends the mutators, and finishes the computation of the shadow array. Note that during the concurrent pass over the bitmap, new allocations might have taken place in the old generation. These new objects need to be taken into account when generating the shadow array. While the mutators are stopped, MC switches to the new shadow array and inserts filler objects into dead clusters. Card table entries (dirty/clean cards) are left intact (as no object moves). In addition, MC computes the new size of free space and resizes the heap accordingly (by the total size of the newly-discovered free space). Finally, the mutators are resumed, and free clusters are unmapped concurrently. Thus, there is one STW sub-phase and two concurrent sub-phases. Auxiliary data structures used by concurrent MC (the cluster array and the shadow array) impose additional space overhead. However, this overhead is small in practice, and, as we discuss later, is not an issue given that concurrent GC needs significantly over-provisioned heaps.

3.4 Bounding Space Overhead

STW MC supports space-bounded collection by falling back to perfect compaction in cases when unmapping fails to reclaim a sufficient amount of free space. In case of concurrent MC, there is no need for bounding the space overhead as concurrent MC requires significantly more heap space than STW MC (much more than the imposed space overhead). This is because concurrent GC trades pause times for space and throughput (Section 4.6).

STW MC evaluates whether to perform a fall-back after STW parallel unmapping. In most state-of-the-art parallel compactors, (including MC, HS, and CP), a liveness bitmap is the interface that bridges marking and the subsequent phases. Therefore, MC can directly proceed to the second phase of a conventional moving compactor without any additional processing, once it determines that a fall-back is needed.

Our current MC fall-back is the STW Compressor. The compaction phase of the Compressor is described in Section 2.1. An alternative solution is a fall-back to the HotSpot compactor, but STW CP imposes a smaller space overhead and is simpler. The space-bounded MC uses two mutually-distant areas in the address space, one of which is active (and mapped) at any given point in time. The non-moving unmapping-based compaction always takes

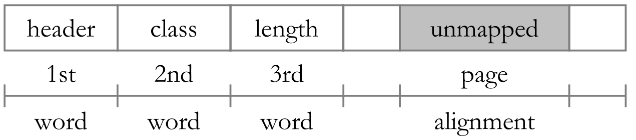


Figure 2. The format of a filler object. First three words form the header of an array object. The page-aligned part of the rest of the cluster is subject to unmapping.

place in the currently active space. If a fall-back is needed, then all objects from the active space are moved to the other space and the roles of the two spaces are flipped (as in the Compressor). The time overhead imposed by a fall-back is the unmapping scan (the moving compaction does not benefit from this scan) and includes bitmap traversal, unmapping, filler object insertion, and object-start array computation.

3.5 Implementation Details

We have implemented STW MC (the unbounded and the space-bounded variant), concurrent MC, and the STW/concurrent Compressor in HotSpot [22], an open-source (GPL) high-performance Java Virtual Machine available from Sun Microsystems and written in C/C++ (source code released on 3/21/2007). The HotSpot JVM uses a generational [31] heap layout that comprises the permanent, tenured (old), and young generation. The young generation is further subdivided into eden and two equal-sized survivor spaces (called from-space and to-space). The permanent generation contains run-time meta-data for the loaded classes. The system allocates objects initially in the eden (if their size precludes eden allocation, it allocates them directly in the tenured generation). Upon space exhaustion in the eden, a copying collector [9, 14] (called the scavenger) performs a minor collection. The scavenger evacuates live objects from the eden-space and from-space to the to-space, and promotes objects that survive several minor collections (or those that do not fit into the to-space) to the tenured generation. The roles of the survivor spaces exchange after each minor collection. When space in the tenured generation is exhausted, a major collection (compaction) takes place. The parallel STW compactor currently available in HotSpot is described in Section 2.2. GC threads in HS are schedulable kernel threads. HotSpot assigns each generation a contiguous region in the virtual address space and maps only the currently used portion.

We implement STW/concurrent MC as a parallel compactor in the tenured generation. Both STW and concurrent MC use STW parallel marking. We reuse and simplify (MC does not require per-chunk summary data) the marking phase of the STW parallel HotSpot compactor. We increase the distance between generations in virtual memory to reserve address space for page remapping.

MC compacts the young generation (which is much smaller than the tenured generation) by object moving and pointer adjustment. This compaction, however, is not part of the major collection. It takes place as an epilogue of a failed minor collection. Consequently, MC does not need to update any pointers during major collections (unlike HS and CP).

Since the scavenger uses a card table to find roots during minor collections, the unmapping scan in MC must compute an offset of the first live object for each 512-byte card (the object-start array). This additional processing is concomitant to the dead-cluster unmapping and does not require a separate pass.

Free regions cannot be entirely unmapped as the scavenger must be able to traverse (object by object) an arbitrary subspace of the tenured generation (in search for roots) during minor collections. Therefore, we insert a filler object into every free area during each unmapping scan. Figure 2 depicts the format of the filler object. The type of a filler object is an integer array (`int[]`), to ensure that there are no interior reference fields for the scavenger to follow. Thus, each free region is reclaimable except for three words that are necessary for the header of a filler object. The minor GC treats filler objects as if there are live, however, since they are unreachable, the next major collection considers them to be garbage. Following the HotSpot convention, we use a single system call (`mmap`) to perform both mapping and unmapping (for the latter we employ the `MAP_NORESERVE` flag).

Concurrent MC requires a STW phase in order to atomically update the object-start array, insert filler objects, and resize the heap. We piggyback on the STW young generation collection to avoid introducing additional expensive safepoints [18]. Young generation GC is relatively frequent and a slightly-delayed STW phase is not a problem in practice.

3.5.1 Generational Compressor

We extend the Compressor to support generational compaction, and implement it in the tenured generation. The Compressor moves objects, therefore it needs to update the pointers in the young and permanent generations upon each compaction. We use 256-byte blocks, as we have found them to be the best tradeoff between space overhead and performance. The concurrent Compressor has two concurrent sub-phases, separated by a single STW sub-phase. In the first sub-phase, the Compressor computes the block-offset array (used for pointer forwarding) and the shadow object-start array. In the STW sub-phase, the system updates the shadow object-start array (to include new allocations) and sets it as the current object-start array, invalidates card tables (because objects are moved), forwards pointers in the young generation and permanent generation, protects heap pages and switches to the other semi-space. In the third sub-phase, a concurrent thread reads subsequent pages (one word per page to generate SEGV traps) to ensure that all the pages are eventually moved, and clears the liveness bitmap.

4. Evaluation

We empirically evaluate 6 compactors: STW HotSpot, STW unbounded MC, STW space-bounded MC, concurrent unbounded MC, STW Compressor, and concurrent Compressor. We compare these GCs in two groups, one comprising 4 STW compactors and the other comprising 2 concurrent compactors. In addition, we compare STW MC with concurrent MC to investigate the STW/concurrent tradeoffs.

Our experimental platform is an SMP with 4 processors each of which is a 2-way SMT (the machine has 8 logical CPUs). Each physical processor is a 32-bit Intel Xeon with 1MB of cache, clocked at 1.6GHz. The machine is equipped with 7GB of main memory and is running Linux Red Hat 3.4.6 with the 2.6.9 kernel. The virtual page size is 4KB. We run HotSpot 7-ea-b10 compiled with GCC 3.2.3 in the optimized client-compiler (C1) mode.

4.1 Benchmarks

We employ a diverse set of benchmarks with a wide range of behaviors. These benchmarks include three multi-threaded server benchmarks: VolanoMark 2.5 [32], PseudoSPECjbb 2000 [30], and Hsqldb from the DaCapo 2006 suite [12], and three deskside utilities (from DaCapo 2006): Xalan, Chart, and Pmd. We list the basic statistics for these benchmarks (i.e., the minimum heap size, total execution time, total GC time, and the number of GCs), that we obtain using the HotSpot compactor, in Table 1.

VolanoMark is a standard server benchmark derived from a commercial chat server (VolanoChat), which simulates a multi-user environment with multiple chat rooms. The benchmark exchanges a given number of messages and reports execution time and communication throughput. PseudoSPECjbb is a variant of SPECjbb

Benchmark	Heap[MB]	Time[s]	GC[%]	#GCs
Chart	27	25.79	13.21	16
Xalan	31	20.39	43.48	68
Pmd	31	29.54	28.16	26
Hsqldb	100	18.62	36.48	4
Volano	33	80.25	24.41	112
JBB	174	95.62	43.02	84

Table 1. GC statistics for the HotSpot compactor: the minimum heap size, execution time, percentage of GC time relative to execution time, and the number of GCs. The measurements have been obtained for the minimum heap size for each benchmark.

that executes a given number of transactions and reports execution time. The benchmark emulates a three-tier client-server system (with emphasis on the middle tier) where clients are replaced by driver threads and database storage by binary trees of objects. Hsqldb is a relational SQL database management system that supports in-memory and disk-based data storage. DaCapo employs Hsqldb to execute an in-memory benchmark that comprises a number of transactions against a model of a banking application. Xalan transforms XML documents into HTML. Pmd analyzes a set of Java classes for a range of source code problems. Chart plots a number of complex line graphs and renders them into a PDF file.

4.2 Methodology

Each of our experiments uses a fixed-size heap. We report total heap size, which includes the young, old, and permanent generation. Total heap size does not include auxiliary data structures as they are located outside of the heap. The young generation size is 25% of the old generation. The permanent generation is 12MB (HotSpot default). Explicit GC invocation and adaptive generation resizing are disabled. We employ 4 parallel GC threads (except for the scalability experiments where we use 1–8 threads). Survivor spaces (from-space and to-space) occupy 33% of the young generation (the remaining space is used by the eden). For concurrent MC/Compressor we start compaction when 65% of the old generation is used. Concurrent compaction uses a single concurrent GC thread.

We repeat each measurement three times and report the average result along with the standard deviation (error bars in the plots), wherever appropriate. We employ the default input size for all DaCapo benchmarks. VolanoMark is run with 44 chat rooms and performs 100 iterations in the networked mode. The server and the client are on the same machine. PseudoJBB is configured to execute 10^5 iterations against 8 (for STW GC) and 4 (for concurrent GC) warehouses.

4.3 Clustering

Figure 3 shows CDFs for the sizes of clusters of dead objects for the deskside benchmarks (a), server benchmarks (b), and across the benchmarks (c). We report data obtained for the minimum heap sizes using STW unbounded MC. Percentage of clusters greater than 4KB (virtual page size) is 24% for Chart, 52% for Xalan, 38% for Pmd, 1% for Hsqldb, 5% for Volano, and 9% for JBB. Fragmentation is higher in server benchmarks. MC achieves low space overhead for these benchmarks by reclaiming relatively few big clusters rather than many smaller ones. Average cluster size is 26KB, minimum cluster size is 28B, and maximum cluster size is 184MB.

4.4 STW Compactors

We compare STW unbounded MC (UN) and STW space-bounded MC (SP) with STW Compressor (CP) and STW HotSpot (HS) in terms of memory footprint, throughput, pause times, and scalability. For SP, we employ the 10% space overhead bound in all experiments. We also investigate the impact of other bounds on the fall-back frequency and average pause times.

4.4.1 Space Overhead

HS and CP impose a constant space overhead of 3% (for 2KB chunks) and 1.5% (for 256B blocks), respectively. In MC, the space overhead is variable and application-specific (but can be bounded) and depends on the degree of dead-object clustering in the heap.

The bar graph in Figure 4(a) shows space overhead imposed by STW unbounded MC and STW space-bounded MC. For each benchmark, we report the average value across the heap sizes. The overhead is shown as a percentage of the heap size. On average, the unbounded MC imposes 5.8% overhead while the space-bounded MC (with the 10% bound) imposes 3.5% overhead.

4.4.2 Throughput

In Figure 5, we present per-benchmark graphs, each with four performance curves for a range of heap sizes. Each graph shows execution time as a function of heap size (starting from the minimum heap size).

For the minimum heap sizes and relatively to HS, UN improves throughput by up to 23.5% (Hsqldb) and by 13.3% on average. For the minimum heap sizes and relatively to CP, UN improves throughput by up to 42.1% (PseudoJBB) and by 23.3% on average.

For the minimum heap sizes and relatively to HS, SP improves throughput by up to 22.7% (Hsqldb) and by 10.9% on average. For the minimum heap sizes and relatively to CP, SP improves throughput by up to 40.1% (PseudoJBB) and by 21.1% on average.

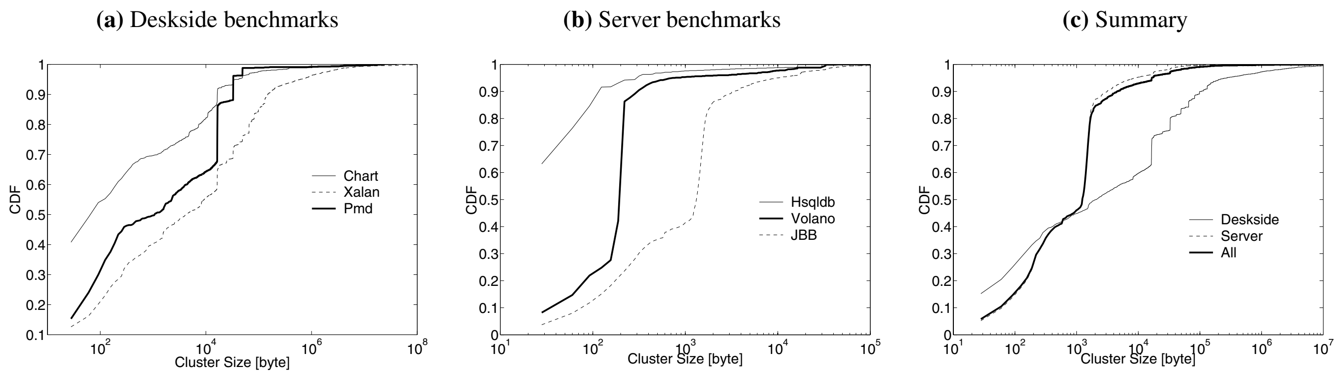
4.4.3 Pause Times

Figures 4(b) and 4(c) present average and maximum pause times for UN, SP, HS, and CP. For each benchmark, we report the average value across the heap sizes.

Compared to HS, UN reduces average (maximum) pause times by up to 69.7% (78.7%) and on average by 63.4% (68.4%). Compared to CP, UN reduces average (maximum) pause times by up to 73.8% (74.4%) and on average by 66.8% (67.5%). Compared to HS, SP reduces average (maximum) pause times by up to 67.8% (76.4%) and on average by 49.3% (31.4%). Compared to CP, SP reduces average (maximum) pause times by up to 72.2% (71.7%) and on average by 53.9% (31.5%).

A commonly-employed metric for the evaluation of collector-imposed pauses are minimal mutator utilization (MMU) curves [10] that lend insight into the distribution of GC pauses across program execution. Mutator utilization for a given time window w is defined as the fraction of the time that the mutator (as opposed to the collector) executes within the window w. Minimum mutator utilization for a window of a specific size s is the lowest mutator utilization for all time windows of size s across the program execution. Thus, the x-intercept of a MMU curve is the maximum pause time and the asymptotic y-value corresponds to the application throughput. As shown in Figure 6, UN achieves the highest MMU for all window sizes across all benchmarks and attains non-zero utilization for windows shorter than SP, HS, and CP. SP achieves better or the same utilization as HS and CP for all benchmarks. Since SP falls back to CP, its maximum pause time is often similar to CP. HS achieves better or comparable utilization as CP.

Figure 7 compares average (data points) and maximum (error bars) pause times for UN, SP, HS, and CP. For these experiments, we vary the number of parallel GC threads for a fixed heap size (we use the minimum heap sizes). Both UN and SP consistently decrease pause times relative to HS and CP, independent of the number of parallel GC threads. For 1 GC thread, UN reduces

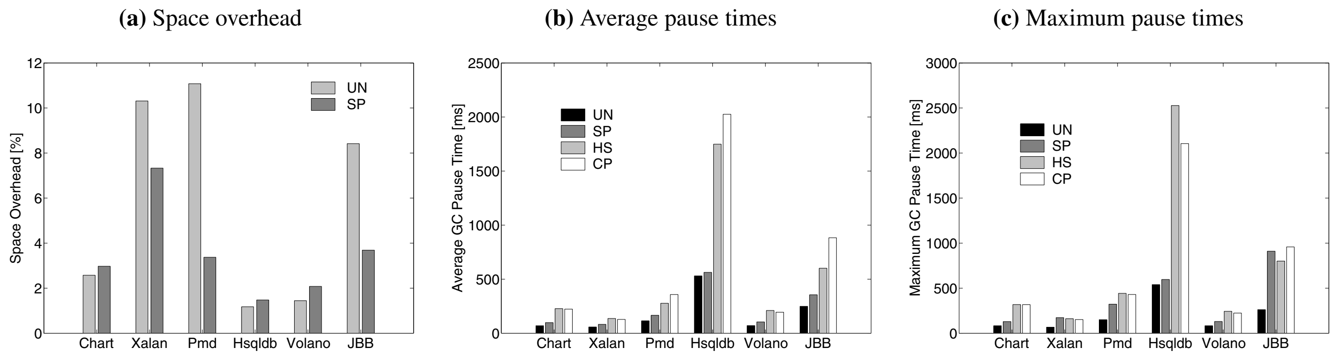


(a) Deskside benchmarks

(b) Server benchmarks

(c) Summary

Figure 3. Distribution of cluster sizes for the deskside benchmarks (a), server benchmarks (b) and across the benchmarks (c). We report a CDF for each individual benchmark as well as summary CDFs for the deskside, server, and all benchmarks.



(a) Space overhead

(b) Average pause times

(c) Maximum pause times

Figure 4. GC statistics across the heap sizes for STW unbounded MC (UN), STW space-bounded MC (SP) with the 10% bound, STW HotSpot compactor (HS), and STW Compressor (CP): heap space overhead (a), average pause times (b), and maximum pause times (c).

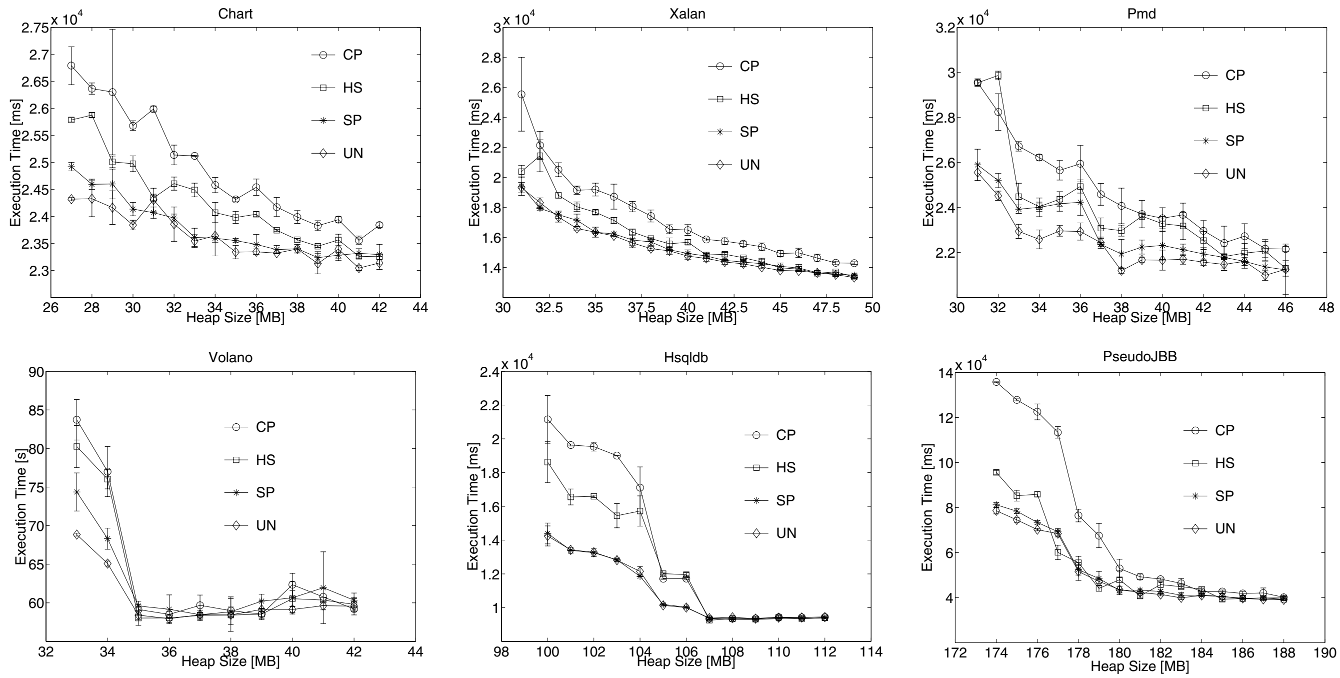


Figure 5. Benchmark performance (execution time) across the heap sizes for STW unbounded MC (UN), STW space-bounded MC (SP) with the 10% bound, STW HotSpot compactor (HS), and STW Compressor (CP). Error bars indicate the standard deviation across 3 runs.

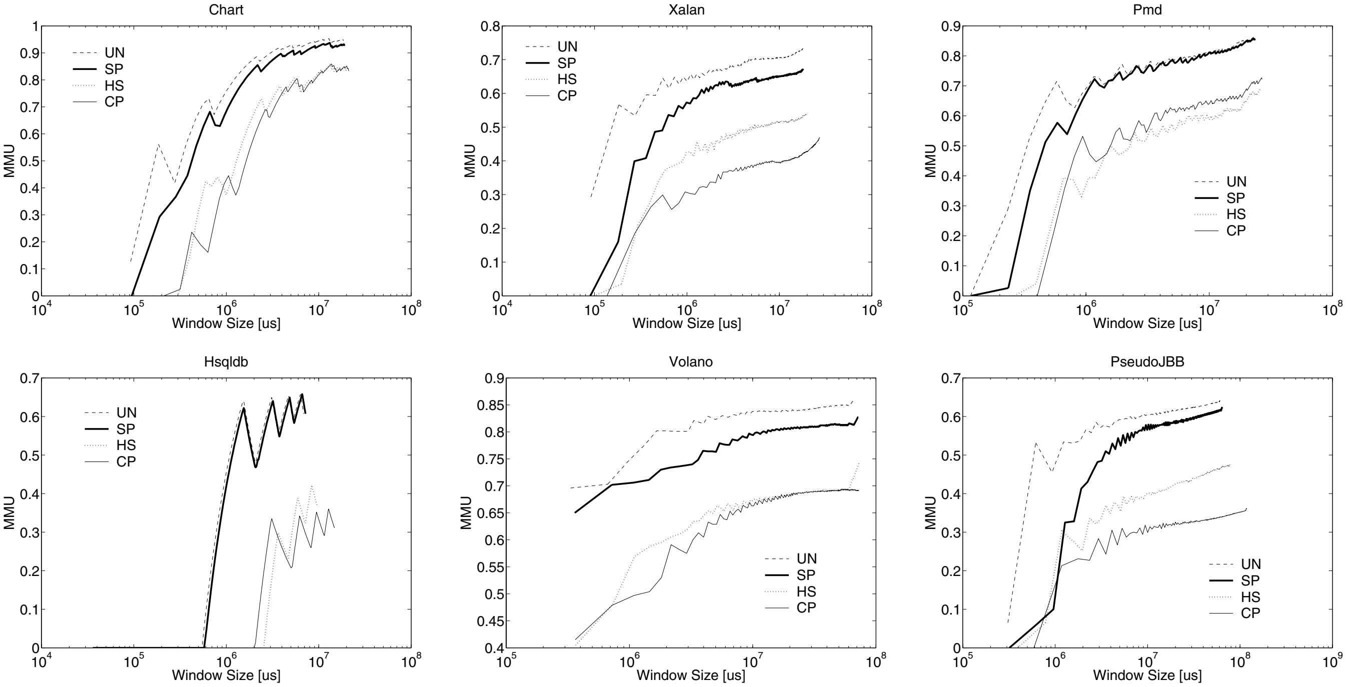


Figure 6. Minimum mutator utilization (MMU) curves for the minimum heap sizes for STW unbounded MC (UN), STW space-bounded MC (SP) with the 10% bound, STW HotSpot compactor (HS), and STW Compressor (CP). Window size is in microseconds.

pauses on average by 49% relative to HS and by 61% relative to CP, while SP reduces pauses on average by 44% relative to HS and by 56% relative to CP. For 4 GC threads, UN reduces pauses on average by 61% relative to HS and by 66% relative to CP, while SP reduces pauses on average by 51% relative to HS and by 57% relative to CP. Finally, for 8 GC threads, UN reduces pauses on average by 65% relative to HS and by 66% relative to CP, while SP reduces pauses on average by 54% relative to HS and by 54% relative to CP.

4.4.4 Scalability

Our experimental platform has four 2-way SMT processors virtualized by the operating system as 8 logical CPUs. We investigate the scalability (speedup) of UN, SP, HS, and CP in the context of both multi-processing and multi-threading parallelism. We measure the unscaled speedup – we apply an increasing number of GC threads (from 1 through 8) to a fixed-size workload and the minimum heap for each benchmark. We compute the speedup for p threads as a ratio of the average GC pause time for 1 thread and for p threads.

As shown in Figure 8, server benchmarks scale better (e.g., Hsqldb/UN achieves 5.9 speedup while the maximum for deskside benchmarks is 3.4 for Chart/UN). HS has the worst scalability because it computes per-chunk statistics during marking, which entails more synchronization. The plots in Figure 7 provide absolute average GC pause times from which the speedup graphs have been derived.

When considering only multi-processing parallelism (4 GC threads), the speedup averages at 2.86 for UN, 2.6 for SP, 2.22 for HS, and 2.49 for CP. Thus, UN improves speedup by 30% relative to HS and by 15% relative to CP, while SP improves speedup by 17% relative to HS and by 4% relative to CP.

When multi-threading is taken into account (8 GC threads), the speedup averages at 3.75 for UN, 3.19 for SP, 2.56 for HS, and 3.03 for CP. Thus, UN improves speedup by 47% relative to HS and by 23% relative to CP, while SP improves speedup by 23% relative to HS and by 3% relative to CP.

4.4.5 Fall-Back Rate

SP falls back to CP if excessive fragmentation in the heap makes it impossible to reclaim a satisfactory fraction of free space. Table 2 shows the rate of fall-back to perfect compaction that is necessary to guarantee a specific space overhead bound (2% to 20%). We express this rate as the percentage of GCs that need to fall back to conventional moving compaction to keep the space overhead below a given threshold.

The fall-back statistics for the minimum heap sizes indicate that even for tight bounds, relatively infrequent fall-back is necessary. For instance, in order to achieve 5% bound, on average, 6.5% collections need to trigger a fall-back (for 7% bound it is 3.7% and for 10% bound it is 1.8%). In addition, we have measured average GC pause times for different space bounds. The results for UN and SP, reported in Table 2, indicate that the space-bounded MC reduces pauses significantly relative to HS and CP (for all bounds that we investigate), and increases average pause times by around 20% compared to the unbounded MC.

4.5 Concurrent Compactors

Next, we compare concurrent unbounded MC (UN) and concurrent Compressor (CP) in terms of memory footprint, throughput, and pause times.

4.5.1 Space Overhead

Concurrent collection requires heap space over-provisioning to avoid the situation when allocators exhaust the heap before the ongoing background collection is complete. Therefore, space overhead is less of a problem in concurrent MC than in STW MC. Figure 9(a) shows space overhead (as a heap percentage) averaged across the heap sizes. As explained earlier, CP has a constant space

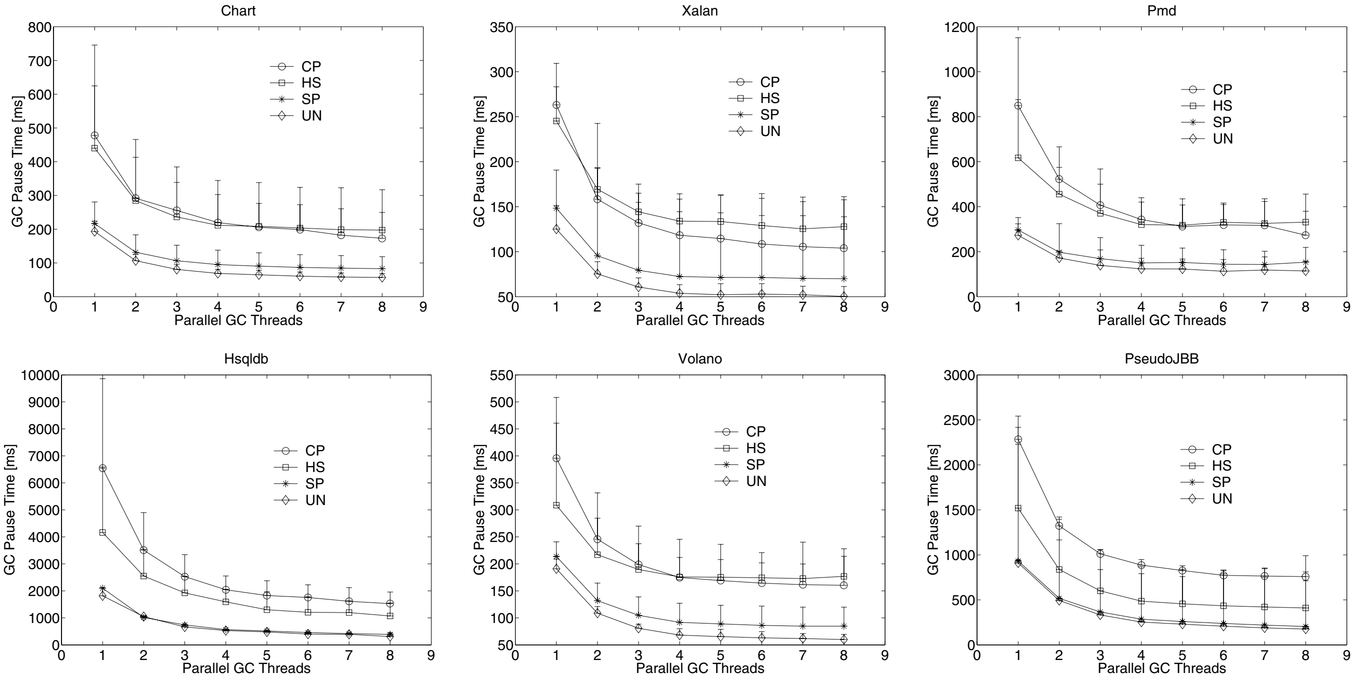


Figure 7. Average (data points) and maximum (error bars) GC pause times for 1–8 parallel GC threads and the minimum heap sizes for STW unbounded MC (UN), STW space-bounded MC (SP) with the 10% bound, STW HotSpot compactor (HS), and STW Compressor (CP). We report average values across 3 runs.

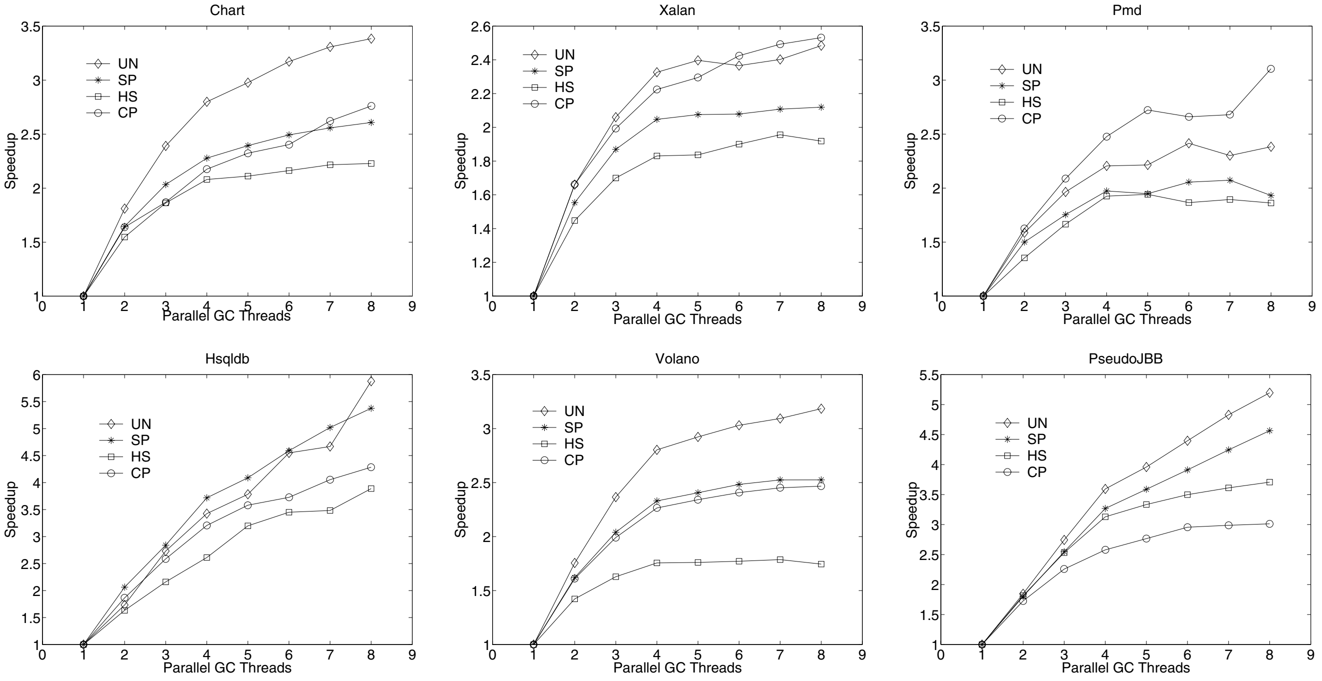


Figure 8. Scalability (unscaled speedup) for 1–8 parallel GC threads, fixed workload, and the minimum heap sizes for STW unbounded MC (UN), STW space-bounded MC (SP) with the 10% bound, STW HotSpot compactor (HS), and STW Compressor (CP). Speedup is computed for average GC pause times. Absolute average GC pause times are reported in Figure 7.

Bound [%]	2	5	7	10	15	20
Benchmark	Fall-back frequency [%]					
Chart	12.8	0.0	0.0	0.0	0.0	0.0
Xalan	99.6	32.6	16.4	5.7	1.5	0.4
Pmd	6.6	3.9	4.2	4.1	4.1	0.0
Hsqldb	0.0	0.0	0.0	0.0	0.0	0.0
Volano	1.0	0.0	0.0	0.0	0.0	0.0
JBB	4.3	2.3	1.6	1.2	0.0	0.0
Compactor	Avg. Pause Decrease [%]					
STW CP	62.6	64.7	65.2	65.1	65.2	66.1
STW HS	53.2	55.8	56.4	56.3	56.4	57.5
Compactor	Avg. Pause Increase [%]					
STW UN	26.8	22.5	21.4	21.5	21.4	19.4

Table 2. GC statistics for STW space-bounded MC for different space bounds obtained using the minimum heap sizes. The first part shows fall-back frequency (percentage of GCs with a fall-back). The second part shows percentage decrease in average GC pause times relative to STW Compressor (CP) and STW HotSpot (HS). The third part shows percentage increase in average GC pause times relative to STW unbounded MC (UN).

overhead of 1.5%. Across the benchmarks, the space overhead of concurrent UN averages at 4.1%. Note that concurrent UN requires about 28% more heap space than STW UN (Section 4.6). Thus, bounding space overhead in concurrent MC does not seem necessary/practical.

4.5.2 Throughput

In Figure 10, we present per-benchmark graphs, each of which shows execution time as a function of heap size (starting from the minimum heap size). For the minimum heap sizes, concurrent UN improves throughput by up to 52% (Xalan) and by 29% on average (relative to the concurrent Compressor).

4.5.3 Pause Times

Figures 9(b) and 9(c) present average and maximum pause times for concurrent UN and concurrent CP. For each benchmark, we report the average value across the heap sizes. Note that we do not consider pauses imposed by concurrent marking here, only those imposed by concurrent compaction. Compared to concurrent CP, concurrent UN reduces average pause times by up to 96% (Volano) and on average by 90%, while reducing maximum pause times by up to 94% (Volano) and on average by 88%. Since concurrent CP moves objects, it needs to update the pointers in the young and permanent generations as part of its STW phase. Concurrent MC does not need to do that and thus its STW pause is much shorter.

4.6 STW/Concurrent Tradeoffs

To lend insight into the tradeoffs associated with STW and concurrent compaction [5, 13, 19, 25], we compare STW UN with concurrent UN, in terms of throughput, pause times, and memory footprint. Both compactors use the same STW parallel marking algorithm.

Table 3 shows experimental results for our benchmarks. We report the minimum heap size in MB (columns 2 and 3), maximum pause time in ms (columns 4 and 5), and execution time in seconds (columns 6 and 7). Execution time and pause times are measured for the minimum heap size of concurrent UN (shown in column 3). This heap size is often much larger than the minimum heap size of STW UN – in some cases big enough to prevent STW UN from any GC activity (we then report pause times as n/a).

Concurrent GC trades pause times for throughput and heap space. On average, relative to STW UN, concurrent UN requires 28% more heap space and degrades throughput by 28%. Maximum pause times (needed for compaction, not marking), however, are shorter for concurrent UN by 89% on average.

Bench-mark	Min.Heap[MB]		Max.Pause[ms]		Exec.Time[s]	
	SMC	CMC	SMC	CMC	SMC	CMC
Chart	27	39	81.4	7.9	23.13	23.85
Xalan	31	37	66.3	6.2	15.64	24.62
Pmd	31	37	160.4	11.2	22.44	37.36
Hsqldb	100	108	n/a	39.7	9.43	12.03
Volano	33	48	n/a	2.1	60.25	61.07
JBB	92	122	147.0	24.1	23.63	26.49

Table 3. Comparison of STW unbounded MC (SMC) and concurrent unbounded MC (CMC). We report the minimum heap sizes, maximum GC pause times, and execution times. Execution times and pause times are obtained for the minimum heap size of CMC (as it is bigger than the minimum heap size of SMC). The reported pause times correspond to compaction only (marking is excluded).

Bench-mark	# System Calls	GC Time [ms]	System Time [ms]	System Time [%]
Chart	4358	3405	13.5	0.4
Xalan	92796	8869	287.7	3.2
Pmd	26585	8319	82.4	1.0
Hsqldb	911	6818	2.8	0.0
Volano	12514	19592	38.8	0.2
JBB	299953	41134	929.9	2.3

Table 4. The cost of unmapping system calls in STW unbounded MC. We report the total number of the `mmap` calls, total GC time, total time spent in the system calls, and percentage of GC time spent in the system calls. System time has been conservatively estimated using a serial micro-benchmark.

4.7 Unmapping Overhead

We have evaluated the cost of the `mmap` system calls relative to GC time in STW UN. Table 4 presents per-benchmark data obtained for the minimum heap sizes. We report total number of system calls, total GC time, total system call time, and percentage of GC time spent in system calls. We estimate the cost of a single unmapping system call using a separate micro-benchmark. Our platform needs 3.1s to perform 10^6 unmapping calls. The length of the unmapped region does not impact this cost. On average, STW UN spends 1.2% of GC time in system calls. Note that this result is an upper bound as our micro-benchmark is not parallel.

4.8 Other Benchmarks

Thus far, we have only presented detailed experimental data for a subset of benchmarks that we have studied. For our in-depth analysis we have selected standard, modern benchmarks whose performance is considerably affected by GC. Table 5 summarizes the experimental data obtained for the remaining deskside utility benchmarks that we have investigated: Db (memory-resident database) and Javac (Java compiler) from the SPECjvm (1998) suite [30] as well as Bloat (bytecode analyzer/optimizer), Fop (XSL parser and formatter), and Lusearch (text search engine) from the DaCapo (2006) suite [12].

In Table 5 we report results for both STW and concurrent UN (slash-separated) in comparison to STW HS and STW/concurrent CP. We report the minimum heap size for STW/concurrent UN (column 2), space overhead for STW/concurrent UN (column 3), and average pause time reduction in comparison to HS and CP (columns 4–5). Column 4 compares STW UN and STW HS. Col-

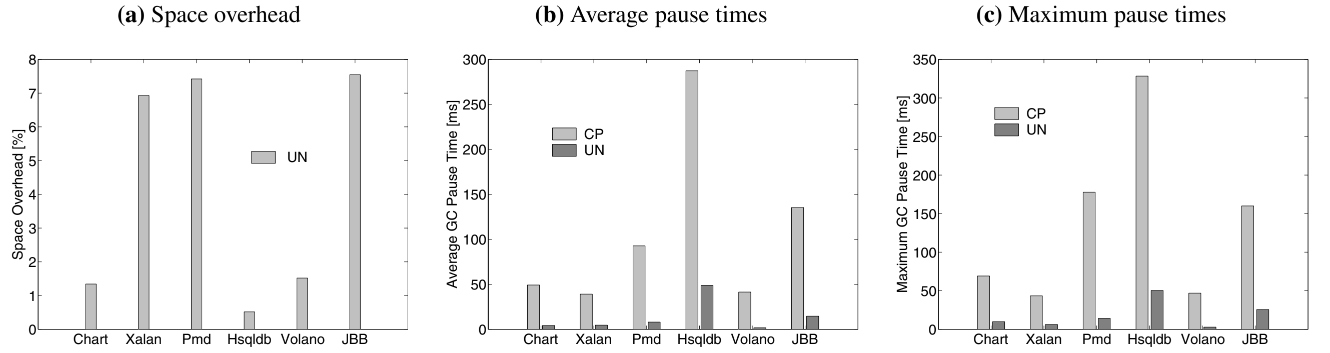


Figure 9. GC statistics across the heap sizes for concurrent unbounded MC (UN), and concurrent Compressor (CP): heap space overhead (a), average pause times (b), and maximum pause times (c). We use the same heap size ranges as in Figure 10.

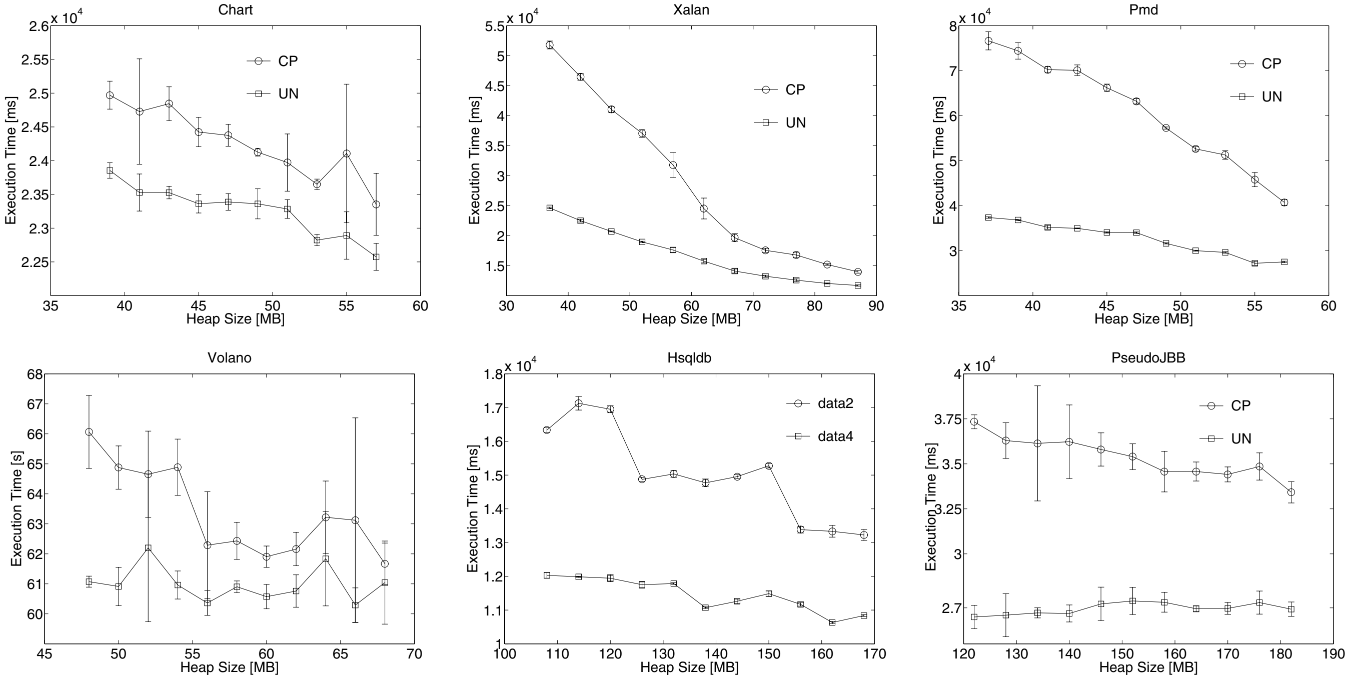


Figure 10. Benchmark performance (execution time) across the heap sizes for concurrent unbounded MC (UN) and concurrent Compressor (CP). Error bars indicate the standard deviation across 3 runs.

umn 5 compares STW UN with STW CP as well as concurrent UN with concurrent CP.

For our additional benchmarks, on average, concurrent UN requires 36.5% more heap space than STW UN. Space overhead, across these benchmarks, is 6% for STW UN and 3% for concurrent UN. Concurrent UN reduces average pause times by 92% compared to concurrent CP. STW UN reduces average pause times by 59% relative to STW HS and by 70% relative to STW CP.

5. Conclusions

We introduce the Mapping Collector (MC) which is a generational, parallel GC that supports both stop-the-world and concurrent compaction. MC coordinates with the underlying virtual memory system of the operating system and performs compaction in nearly one phase. Thus, MC is simpler and more efficient than state-of-the-art compactors which require at least two phases. Unlike previously

Bench-mark	Heap Size [MB]	% Space Overhead	% Avg. Pause	
			vs. HS	vs. CP
Bloat	16 / 20	5.9 / 3.1	69.6	73.0 / 96.9
Fop	20 / 24	3.1 / 1.0	57.3	68.8 / 91.6
Lusearch	16 / 24	4.4 / 3.1	62.6	73.7 / 97.2
Db	24 / 32	0.8 / 0.5	47.6	66.7 / 94.2
Javac	24 / 37	15.7 / 8.5	56.1	68.8 / 80.8

Table 5. GC statistics for additional benchmarks using the minimum heap sizes. Slash delimits data for STW and concurrent MC. In subsequent columns, we report the minimum heap size for STW/concurrent unbounded MC (2), space overhead for STW/concurrent unbounded MC (3), average pause time reduction for STW unbounded MC relative to STW HS (4) and STW Compressor (5), and average pause time reduction for concurrent unbounded MC relative to concurrent Compressor (5).

reported compactors, MC is a non-moving collector that leverages the level of indirection provided by virtual memory to consolidate free space into a single contiguous region. By doing so, MC is able to avoid costly object copying and pointer adjustment. The motivation for MC is the observation that unreachable objects in the heap tend to form clusters that can be effectively reclaimed at the granularity of virtual pages. Space overhead imposed by MC is variable but modest in practice and can be bounded by relatively infrequent fall-back to conventional, perfect compaction. MC is particularly attractive for concurrent compaction as it does not require synchronization with the mutators and its space overhead is not a problem in the light of heap over-provisioning.

We implement MC in the open-source HotSpot JVM and evaluate it experimentally on a multiprocessor using a range of different benchmarks and metrics, including throughput, pause times, and scalability. We show that MC significantly outperforms state-of-the-art, stop-the-world parallel compactors (the Compressor and the HotSpot compactor), as well as the concurrent Compressor, for the metrics and benchmarks that we investigate.

Acknowledgments

We thank the anonymous reviewers for providing insightful comments on this paper. This work was funded in part by NSF grants CCF-0444412, CNS-0546737, and CNS-0627183.

References

[1] D. Abuaiadh, Y. Ossia, E. Petrank, and U. Silbershtein. An efficient parallel heap compaction algorithm. In *the ACM Conference on Object-Oriented Systems, Languages and Applications*, 2004.

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

[3] K. Barabash, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, Y. Ossia, A. Owshanko, and E. Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Transactions on Programming Languages and Systems*, 27(6):1097–1146, 2005.

[4] K. Barabash, Y. Ossia, and E. Petrank. Mostly concurrent garbage collection revisited. In *the ACM Conference on Object-Oriented Systems, Languages and Applications*, 2003.

[5] S. M. Blackburn and K. S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *the ACM Conference on Object-Oriented Systems, Languages and Applications*, 2003.

[6] H.-J. Boehm. Reducing garbage collector cache misses. In *Second International Symposium on Memory Management*, ACM SIGPLAN Notices. ACM Press, 2000.

[7] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6), 1991.

[8] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

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

[10] P. Cheng and G. Blelloch. A parallel, real-time garbage collector. In *the ACM Conference on Programming Languages Design and Implementation*, 2001.

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

[12] The DaCapo Benchmark Suite. `http://dacapobench.org`.

[13] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *the ACM International Symposium on Memory Management*, 2004.

[14] C. Flood, D. Detlefs, N. Shavit, and C. Zhang. Parallel garbage collection for shared memory multiprocessors. In *the USENIX Java Virtual Machine Symposium*, 2001.

[15] C. Grzegorczyk, S. Soman, C. Krintz, and R. Wolski. Isla Vista heap sizing: Using feedback to avoid paging. In *the ACM Conference on Code Generation and Optimization*, 2007.

[16] M. Hertz, Y. Feng, and E. D. Berger. Garbage collection without paging. In *the ACM Conference on Programming Languages Design and Implementation*, 2005.

[17] A. L. Hosking. Portable, mostly-concurrent and mostly-copying garbage collection for multi-processors. In *the ACM International Symposium on Memory Management*, 2004.

[18] HotSpot Virtual Machine Garbage Collection. `http://java.sun.com/javase/technologies/hotspot/gc/index.jsp`.

[19] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996.

[20] H. Kermany and E. Petrank. The Compressor: Concurrent, incremental and parallel compaction. In *the ACM Conference on Programming Languages Design and Implementation*, 2006.

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

[22] Open Source J2SE Implementation. `http://openjdk.java.net`.

[23] Y. Ossia, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, and A. Owshanko. A parallel, incremental and concurrent GC for servers. In *the ACM Conference on Programming Languages Design and Implementation*, 2002.

[24] Y. Ossia, O. Ben-Yitzhak, and M. Segal. Mostly concurrent compaction for mark-sweep GC. In *the ACM International Symposium on Memory Management*, 2004.

[25] T. Printezis and D. Detlefs. A generational mostly-concurrent garbage collector. In *the ACM International Symposium on Memory Management*, 2000.

[26] R. Rashid, A. Tevanian, M. Young, et al. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *the ACM Conference on Architectural Support for Programming Languages and Operating Systems*, 1987.

[27] N. Sachindran and E. Moss. MarkCopy: Fast copying GC with less space overhead. In *the ACM Conference on Object-Oriented Systems, Languages and Applications*, 2003.

[28] K. Sagonas and J. Wilhelmsson. Mark and split. In *the ACM International Symposium on Memory Management*, 2006.

[29] P. Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. Technical Report AITR-1417, MIT AI Lab, Feb. 1988.

[30] The SPEC Benchmarks. `http://www.spec.org`.

[31] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, Apr. 1984.

[32] The VolanoMark Benchmark. `http://www.volano.com/benchmarks.html`.

[33] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *the International Workshop on Memory Management*, 1995.

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

[35] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *the ACM Conference on Operating Systems Design and Implementation*, 2006.

[36] C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, and M. Ogihara. Program-level adaptive memory management. In *the ACM International Symposium on Memory Management*, 2006.