Low Pause-Time Garbage Collection
Written Preliminary Examination II

Christian DeLozier

Abstract

Garbage collection prevents security vulnerabilities that can otherwise be caused by incorrect manual memory management at the cost of increased runtime and space overheads. In some domains, the runtime overhead and unpredictable, lengthy pauses for garbage collection could be deemed unacceptable. Decades of research have proposed incremental, concurrent, and generational garbage collection as possible solutions for decreasing the runtime performance overheads and unpredictable application behavior caused by garbage collection. In this report, we survey three modern garbage collectors that attempt to limit application pauses and provide acceptable runtime performance in their respective domains. Schism [8] targets real-time applications, C4 [10] targets applications with large heap sizes and fast application response-time requirements, and HAMM [6] provides hardware support for a wide range of software garbage collectors for general-purpose applications.

1 Introduction

Automatic memory management, or *garbage collection*, prevents security vulnerabilities (*e.g.* use-after-free errors) that can otherwise occur in languages with manual memory management, such as C. By automatically preventing this class of security vulnerabilities, garbage-collected languages offer increased productivity; programmers no longer have to debug errors due to incorrect manual memory management. Unfortunately, the enhanced security and productivity provided by garbage collection comes at the cost of possible increases in runtime and space overheads. Although garbage collection has become increasingly common in modern programming languages, manual memory management is still favored in some domains, such as real-time systems, due to the common perception that garbage collection causes application behavior to become unpredictable.

Garbage collection *pauses*, which occur when the garbage collector stops application threads from executing, cause unpredictable changes in the *mutator's* (the user program being collected) execution. Garbage collection time (or pause time) measures the total amount of time that the application is paused for garbage collection. Many factors influence the frequency and duration of garbage collector pauses, including the size of the heap and the garbage collection algorithm. For example, generational garbage collectors reduce the average pause time by only scanning a small portion of the heap in the common case. Unfortunately, collectors that aim to limit garbage collection time generally do so at the expense of increasing the amount of work related to garbage collection that must be performed by the mutator.

Garbage collectors often trade a small amount of mutator runtime performance for a large decrease in garbage collection time. Compared to a collection, which takes linear time with regard to the amount of heap scanned, most garbage collection work that is performed by the mutator requires only constant time. For example, generational collection requires a *write barrier* before every store. However, this write barrier only requires constant time, and the overall reduction in garbage collection time due to generational collection often outweighs the slowdown caused by write barriers. Reference counting and concurrent collection are two additional collection algorithms that trade a small amount of work by the mutator for a large reduction in garbage collection time. Although the amount of work performed by the mutator is generally small, it can still unpredictably change the applications behavior.

Garbage collection and allocation are closely coupled. When the allocator runs out of space, the garbage collector will attempt to recycle previously allocated memory that is no longer in use. Garbage collection generally requires twice as much space as manual memory management for reasonably efficient runtime performance [5]. Without enough space, garbage collection must be performed too frequently, leading to an increase in garbage collection time. An application can run out of space either because the total amount of available memory has been exhausted or because the available memory has become fragmented such that no contiguous blocks of memory exist that fit the size of a requested allocation. Memory fragmentation can limit the amount of free space that the allocator is able to actually use, leading to increases in garbage collection time.

Existing garbage collection algorithms, including semi-space and mark-compact collection, attempt to recover from or limit memory fragmentation. However, as noted by Blackburn and McKinley [1], prior approaches that recover from memory fragmentation require either a large amount of additional heap space (semi-space) or increased garbage collection time (mark-compact). In general, recovering from memory fragmentation is difficult because it requires moving objects in memory and correcting any existing references to the relocated objects. Other existing collectors attempt to limit the amount of memory fragmentation by using strategies such as two-level allocation [8]. However, these approaches do not completely eliminate memory fragmentation.

Modern garbage collection algorithms attempt to manage memory fragmentation [11, 9], limit or eliminate garbage collection time [7, 9, 11], and limit the amount of performance perturbation in the mutator [7, 9, 11]. Although years of research in garbage collection have produced a large number of garbage collection algorithms that attempt to solve all three of these problems, this paper limits itself to the following collection algorithms:

- **HAMM** [7]: A hardware extension that augments a software-based garbage collector with hardware-based reference-counting and allocation primitives to limit the number of software garbage collections with little effect on mutator performance.

- **Schism** [9]: A real-time garbage collector that builds upon the Immix [1] *mark-region* garbage collection algorithm to handle memory fragmentation without requiring significant garbage collection pauses.

- **C4** [11]: A garbage collector from Azul Systems that extends the Pauseless GC Algorithm [4] with generational garbage collection to provide efficient garbage collection on large heaps (96 GB and up) with no pauses.

2 Overview

This section provides a brief overview of the context, target applications, and goals of each garbage collector.

2.1 HAMM

Hardware Assisted Memory Management (HAMM) attempts to limit the frequency of software garbage collections for general-purpose Java applications [7]. HAMM does not entirely replace but rather supports a software allocator and garbage collector. Using hardware-supported reference counting, HAMM identifies blocks of memory that are no longer reachable and provides an interface for the software allocator to reuse these free memory blocks. By providing free blocks of memory to the software allocator, HAMM limits how often the software garbage collector must be invoked to find free memory for the allocator.

2.2 Schism

Schism provides fragmentation-tolerant garbage collection for real-time Java applications [9]. Specifically, Schism targets applications that require two guarantees: critical application threads cannot be preempted, and the application cannot crash from running out of memory. To avoid preempting critical threads, Schism runs concurrently and at a lower priority than all critical threads. Schism avoids stop-the-world pauses for garbage collection and limits the amount of work that must be performed by application threads. To avoid out-of-memory crashes, Schism relies on a fragmentation-tolerant allocation scheme to provide a hard bound on the maximum memory space used by an application. Unlike other approaches that periodically attempt to recover from memory fragmentation, Schism assumes that the heap is fragmented and allocates objects in fixed-size chunks into the fragmented heap.

2.3 C4

The Continuously Concurrent Compacting Collector (C4) provides efficient garbage collection for enterprise Java applications with large heaps (*e.g.* 670 GB) and fast response-time requirements. To keep up with the extremely high allocation rates of enterprise Java applications, C4 constantly performs concurrent garbage collection to identify free memory. Every C4 collection compacts the heap to recover from memory fragmentation, which undoubtedly occurs under such high rates of allocation and collection. Unlike its predecessor (The Pauseless GC Algorithm), C4 does not require custom Azul Systems hardware and can run on X86 hardware.

3 Memory Fragmentation

As a program allocates and deallocates memory over time, available heap memory can become fragmented into blocks that may not match the size of requested allocations. If memory fragmentation becomes too severe, an allocation may fail, possibly causing the program to crash. Even if memory fragmentation does not lead to a program crash, it can still lead to increases in garbage collection time because the allocator is unable to effectively use the available memory. To avoid program crashes and increases in garbage collection time, many garbage collection algorithms attempt to prevent or recover from memory fragmentation.

3.1 Copying Collection

Copying garbage collectors handle memory fragmentation by compacting the heap after every collection. These collectors generally use *bump allocation*. Under bump allocation, the allocator maintains a pointer to the start of available memory and simply increments the pointer by the allocation size requested by the program. A *semi-space* copying collector maintains two memory spaces, a *from-space* and a *to-space*. The semi-space collector bump allocates new objects into the from-space until the total amount of available memory in the from-space has been exhausted. At this time, the semi-space collector marks the objects in the from-space and copies all live objects from the from-space into the to-space. Once copying has completed, the to-space becomes the new from-space (and vice-versa).

Semi-space collectors do not suffer from memory fragmentation because copying collection always compacts the surviving memory. However, semi-space collectors require more space than other collection algorithms, such as mark-sweep and mark-compact, because the from-space and to-space must be large enough that copying does not occur too frequently [1]. The Schism garbage collector uses a semi-space collector to manage its array allocation metadata [9].

Mark-compact collectors offer an alternative to semi-space collection with lower space overheads but longer garbage collection times [1]. Rather than copying live objects to a separate memory space, mark-compact collectors relocate objects within the same memory space. To compact objects without the use of a separate memory space, mark-compact collectors require at least two passes over the heap. The first pass marks live objects and identifies free regions of memory. The second pass copies live objects into the free regions, forming a contiguous address space. A mark-compact collector cannot copy objects during the first pass because the free regions of memory have not yet been identified.

3.2 Mark-Region Collection

The Schism [9] garbage collector is based on the mark-region collection algorithm introduced by Immix [1]. Immix avoids memory fragmentation by dividing allocation into fixed size blocks and lines and applying opportunistic evacuation (compaction). Immix allocates fixed-size blocks as local allocation buffers to each thread. The thread then bump allocates into the block in increments of the fixed line size. When collection occurs, the collector marks lines as free, and if a block becomes completely free, it is returned to the global allocator. Immix tracks the level of fragmentation in the heap and occasionally defragments the heap. To defragment the heap, Immix identifies possible blocks as candidates for object evacuation and copies the objects to free lines.

Immix is able to achieve better runtime performance on average than all of the standard garbage collection algorithms that it compares itself to (mark-sweep, mark-compact, semi-space). In terms of space efficiency, only mark-compact is slightly more space efficient than Immix, due to the fact that Immix only opportunistically compacts the heap.

CMR, which is the base collector for Schism [9], extends the basic mark-region algorithm for concurrent collection. Allocation in CMR mixes first-fit and best-fit strategies by finding the first available block that the allocation will fit in and then finding the set of lines within that block that best fits the allocation. Once a location has been found, future allocations will attempt to use bump-pointer allocation to the same block and set of lines. Although this approach limits the effects of fragmentation, CMR cannot recover from memory fragmentation once it has occurred because it does not use the opportunistic evacuation scheme from Immix. The authors of Schism [9] evaluate CMR on a synthetic benchmark that allocates a large number of small arrays (that become garbage) and then attempts to allocate a large number of larger arrays. CMR is unable to allocate any large arrays, and this behavior causes an out-of-memory program crash.

Immix and CMR highlight an important choice that many collection algorithms make between copying collection and concurrent collection. Immix performs copying collection but requires stop-the-world pauses. Immix can safely relocate objects because it can update any references to relocated objects while the mutator is paused. On the other hand, CMR does not require stop-the-world pauses, but it does not relocate objects to compact the heap. To perform concurrent, copying collection, the collector must be able to prevent the mutator from accessing the old location of an object, usually through the use of a read barrier. The use and behavior of read barriers for concurrent copying collection will be further discussed in Section 5.1.2.

3.3 Fragmentation-Tolerant Allocation

Schism improves upon the CMR collector to prevent memory fragmentation for real-time environments. For real-time systems, space efficiency is key, and hard bounds on space usage are often required. If memory becomes fragmented, the amount of address-space used by the program may exceed what is actually required, possibly leading to an out-of-memory crash. Schism avoids such crashes using *fragmented allocation.* All user allocations are allocated by Schism in fixed-size fragments of 32 bytes. Objects are allocated as linked lists of these fixed-size fragments, and arrays are allocated as *arraylets.* In its least optimized form, an arraylet consists of a *sentinel*, which the user program holds a reference to, that points to an arraylet *spine.* The arraylet spine contains references to the rest of the fixed-size fragments that form the array. Arraylet spines must contain enough references to point to all of the fixed-length fragments required to hold the entire array, meaning that each arraylet spine has a non-fixed size. To avoid memory fragmentation from the allocation and deallocation of arraylet spines, Schism uses a separate semi-space collector for the *spine space.* The use of a semi-space collector for the spine space requires additional memory but fixes any fragmentation that occurs in the spine space.

By allocating all memory in fixed-size chunks, Schism is able to continue allocating into a fragmented heap as long as enough chunks are free to match the size of the requested allocation. The fixed-size, fragmentation-tolerant allocation scheme allows for mathematical bounds on the space used by each object and array. For each object, the space used by Schism is bounded by the following equations where b_i is the total size of the object (up to the ith field) and f_i is the size of the ith field.

$$\begin{aligned} b_0 &= 12 \\ b_i &= align(b_{i-1}, f_i) + f_i \end{aligned}$$

The initial size (b_0) of 12 consists of a 4 byte pointer to the next linked fragment, a 4 byte field for garbage collector metadata, and a 4 byte type identifier. The *align* function pads each 32 byte fragment with the 4 bytes required for the reference to the next linked fragment. Otherwise, align increments the current position by 1 until a correct alignment for the field is found. *align* is only guaranteed to reach a fixpoint for field sizes less than 16. Luckily, Java fields are always less than 8 bytes in size because object fields are simply references and all primitive types have size less than 8 bytes. In a language like C, where fields can have an arbitrary size, the allocator for Schism would need to be modified to handle fields with size larger than 16.[1] Given the fields of a class, the above formulas yield a guaranteed amount of space that Schism will use to allocate an instance of the object in memory, regardless of how fragmented the heap is.

Schism also provides a hard bound on the space used by arrays, which are allocated as arraylets. This space bound accounts for both the space used to store the actual array and the space used to store the arraylet spine. The following equations describe the space used to allocate an array where e is the size (in bytes) of each array element and l is the length of the array ($l \times e$ is the total size of the array).

$$B = \begin{cases} 32 & \text{if } l \times e \leq 16 \\ 96 & \text{if } l \times e \leq 96 \\ 32 + 32\left\lceil \frac{l \times e}{32} \right\rceil & \text{if } l \times e > 96 \end{cases}$$

Schism optimizes the layout of arrays of size less than 16 bytes by inlining the array in the sentinel. Schism also optimizes the array layout for arrays of size less than 96 bytes by inlining the spine in the last

[1]For the purpose of formalizing a hard space bound, it may suffice to simply flatten sub-object fields into the fields that the sub-object contains.

12 bytes of the sentinel (three 4 byte fields). For all other arrays, the array requires 32 bytes for the sentinel and enough space to layout the rest of the array elements in 32 byte blocks. The first 16 bytes of the 32 byte sentinel fragment consists of a 4 byte pointer to the spine, a 4 byte field for garbage collection metadata, a 4 byte type identifier, and a 4 byte array length field. Again, this allocation strategy works for Java programs because each array element can be no larger than 8 bytes.

Schism must also bound the amount of space required for the semi-space that manages the arraylet spines. The amount of spine space required for an array of total size $l \times e$ is bounded by the following equation.

$$s = 8 + 4 \left\lceil \frac{l \times e}{32} \right\rceil$$

The first 8 bytes are required for the spine header, which contains a 4 byte forwarding pointer (used when the semi-space collector relocates the spine) and a 4 byte array length field. The spine also requires a 4 byte pointer field for each 32 byte fragment used to store the array. A spine is only required for arrays larger than 96 bytes. To determine the maximum amount of heap space required for the spine space, we need to calculate s/B for arrays larger than 96 bytes (the last case of B).

$$s/B = \frac{8 + 4 \left\lceil \frac{l \times e}{32} \right\rceil}{32 + 32 \left\lceil \frac{l \times e}{32} \right\rceil}$$

For large values of $l \times e$, the constant factor will become irrelevant, and s/B will converge to 1/8. For the smallest possible value of $l \times e$ that requires spine allocation (97 bytes), $s/B = 0.15$. Thus, the semi-space collector that manages the spine space requires 0.3 bytes for every array byte that is allocated by the program (0.15 for the from-space and 0.15 for the to-space).

Given the above equations, Schism provides hard bounds on the amount of space used by its fragmented allocation scheme. In a real-time system, the ability to derive a maximum space bound is important to ensure that the system will not run out of memory during execution. Deriving such a bound under Schism requires knowledge of the number and type of objects and arrays that are in use when the real-time system is using the maximum amount of memory that it will ever require. Due to the additional fields required for linked objects and arraylets, Schism requires a larger minimum heap size than CMR on some benchmarks. However, Schism is able to deal with fragmented allocation, and CMR is not. Thus, on longer running workloads that suffer from memory fragmentation, Schism will be able to continue running where CMR will fail. The runtime performance effects of Schism's fragmented allocation scheme, which are considerable, will be discussed in Section 5.2.1.

3.4 Hand-Over-Hand Compaction

The Pauseless GC Algorithm [4], which is the base algorithm for C4 [11], handles memory fragmentation in a similar manner to a semi-space collector in its *Relocate* and *Remap* phases. One of the main benefits of this algorithm is its distinction between free physical memory and free virtual memory. The *Relocate* phase moves objects from pages with mostly dead objects to pages with mostly live objects. When a page no longer contains any live objects, the physical memory backing the page is freed and can be immediately reused by the operating system. In most relocation schemes, physical memory cannot be immediately freed because forwarding pointers must be left to point to the relocated object from the old object location. In the Pauseless GC Algorithm, side arrays are used to hold forwarding pointers to the relocated objects. The *Remap* phase then visits all references in the heap and updates them to point to the relocated objects.

For concurrent access, the Pauseless GC Algorithm uses a hardware supported read-barrier that traps and updates references to relocated objects as the mutator loads them from main memory. At the end of the Remap phase, which corrects any references to relocated objects that have not already been corrected by the read barrier, the virtual memory pages that are no longer backed by physical memory are freed.

C4 improves upon the Pauseless GC Algorithms compaction scheme using hand-over-hand compaction. In C4, the physical page (that would have been freed and returned to the operating system) is used as the next target for relocating objects. Using hand-over-hand compaction, C4 is able to compact the entire heap, even if it only has a single free page available when compaction starts. Each free page allows C4 to relocate all of the objects from at least one additional page, which then becomes the next target for compaction. Thus, C4 does not require free space equal to the current size of the heap, as is required for semi-space copying collection.

C4 manages the heap using separate spaces for small, medium, and large objects. C4 is configured to handle large heaps and therefore manages the heap using 2 MB physical pages. Each thread allocates using bump pointer allocation in its own local buffer. The small object space contains objects of size less than 256 KB. The medium object space contains objects of size less than 16 MB and is managed as an array of 32 MB blocks (backed by 2 MB physical pages). Large objects are allocated on 2 MB boundaries, and the allocation of physical and virtual pages is done in increments of 2 MB to fit the size of the large object. Large objects are compacted by simply remapping the backing virtual pages to a contiguous address space. Medium objects are copied in increments of 4 KB to improve mutator runtime performance, rather than forcing the mutator to possibly wait for an entire 16 MB object to be copied to a new location. If the mutator attempts to access some part of a medium object, the 4 KB chunk is immediately copied to the new location, allowing the mutator to proceed once the 4 KB copy is complete. The collector then continues to copy the other 4 KB chunks in the background.

C4's allocation scheme suffers from some fragmentation due to space wasted in the fixed-size blocks used for allocation. At worst, the small object space can waste up to 256 KB out of every 2 MB (12.5%). For the medium and large object spaces, the worst-case space wasted is a single 2 MB physical page (11.1%).

Unfortunately, the evaluations of the Pauseless GC Algorithm [4] and C4 [11] do not provide data on the runtime space usage of either collector. Presumably, the authors are not overly concerned with the space usage of the collector because the target server environments contain large amounts of physical memory (*e.g.* 96 GB). However, the allocation and compaction schemes roughly mirror those used by Immix [1] but with much larger block and line sizes, and we would expect to see similar space usage characteristics for C4. The authors note that any fragmentation due to the separate allocation spaces used by C4 (small, medium, large) can be arbitrarily reduced by adding more allocation tiers, which may be important for applying C4 to other application domains (*i.e.* not enterprise Java applications).

3.5 Summary and Discussion

Of the collectors examined, Schism and C4 prevent memory fragmentation through fragmentation-tolerant allocation and compaction, respectively. Although Schism's fragmentation-tolerant allocator provides a hard bound on space usage, the loss of runtime performance due to fragmentation-tolerant allocation is significant. For both latency and throughput, CMR and other Java garbage collectors provide better performance on average than Schism. Schism does provide more predictable response times than other collectors, which is important for real-time applications. Also, in a real-time environment, the hard bound on space usage and low worst-case response times provided by Schism may be worth the decreased latency and throughput.

C4 prevents memory fragmentation by constantly compacting the heap. For maximum efficiency, C4 requires both hardware and operating system support. The read barrier used by C4, which is necessary for concurrent compaction, can perturb the performance of the mutator. This effect will be discussed in Section 5.1.2. C4 also requires a modified operating system that can support frequent page remapping. The operating system changes required for C4 extend the already present page mapping functionality to function efficiently in a parallel environment and allow larger page mappings. Both of these changes seem to be generally useful and could simply be included in the standard Linux distribution. With the correct support, C4 is able to attain high throughput and low worst-case response times. Unfortunately, the C4 and Pauseless GC papers are light on data, and it is unclear how much space C4 requires compared to standard Java garbage collectors. Due to its use of hand-over-hand compaction, C4 should not suffer from the same high space usage overheads as other copying collectors (*e.g.* semi-space).

HAMM does not provide a solution for handling memory fragmentation. In fact, the allocation scheme used by HAMM can cause memory fragmentation. Each call to `REALLOCMEM` searches a per-core table of available blocks of memory. Ideally, this table contains a pointer to an available block of memory that matches each of the 64 size classes supported by HAMM. However, the software allocator may request a block of arbitrary size (*i.e.* does not match any of the 64 size classes). In this case, the closest matching larger sized block is returned to the allocator. Similarly, if HAMM identifies that an arbitrary sized block of memory is unreachable, the block is placed in the closest matching smaller size class. Due to these policies, HAMM will sometimes use more memory than is necessary for an allocation, which can cause future allocation requests to fail. However, HAMM does not require a specific software garbage collector, so the software garbage collector can recover from or prevent fragmentation through compacting collection or fragmentation-tolerant allocation.

4 Limiting the Frequency and Duration of Pauses

One of the key metrics for garbage collector performance is the frequency and duration of garbage collector pauses, often termed *garbage collection time.* In a stop-the-world garbage collector, a trade-off exists between how often collections occur and how long those collections take. If the mutator is given more time to execute in between collections, both the overall heap size and the amount of garbage are likely to increase. Garbage collection generally requires multiple passes over the entire heap and therefore requires linear execution time with regard to the size of the heap. Generational garbage collection attempts to reduce the average time for garbage collection by collecting only a small portion of the overall heap in the common case. Concurrent collection attempts to eliminate pauses completely by performing collection concurrently while the mutator continues to execute. All of the collectors examined in this paper attempt to reduce the frequency and duration of stop-the-world pauses to allow for more predictable application performance.

4.1 Generational Collection

Generational garbage collection reduces the average duration of garbage collection pauses by dividing the heap into two (or more) spaces: a *young generation*, which contains recently allocated objects, and an *old generation*, which contains objects that have survived at least one collection [8]. Most of the time, the generational collector will only collect garbage from the young generation, using pointers from the old generation to the young generation as roots. Although young generation collections may be frequent, these collections are faster than full-heap collections. Generational collectors rely on the *weak generational hypothesis* that most objects become garbage in a relatively short time after allocation [8]. Thus, the collector is likely to find some garbage by only scanning the young generation, allowing the allocator to continue filling requests using the recycled memory. Full-heap collections only need to occur when the garbage collector is unable to collect enough garbage from the young generation to proceed with an allocation. Additional background information on generational garbage collection can be found in the book by Jones and Lins [8]. C4 [11] uses generational collection to constantly recycle memory for reuse by the mutator.

C4 concurrently collects both the young and old generations. Unlike most generational collectors, C4's old generation collection only collects the old generation instead of collecting the full heap. By only collecting the old generation, C4 avoids interfering with the mark state of concurrent young generation collections. To enable old generation only collection, C4 starts a young generation collection at the beginning of every old generation collection that provides a set of young-to-old roots. Additionally, some synchronization is required between the young and old collections when the young generation collection needs to access old generation objects (*e.g.* Java class structure objects). By collecting both generations concurrently, C4 ideally never requires lengthy stop-the-world pauses for old generation collection. HAMM [7] provides a mechanism for tagging separate memory spaces (*i.e.* separate tags for young and old generation).

4.2 Reference Counting

HAMM provides hardware-based *reference counting* to reduce the total number and frequency of software garbage collections. Reference counting is a form of automatic memory management in which each allocation has an associated numerical reference count. The reference count for each allocation is incremented each time a new reference is created that refers to the allocation, and the reference count is decremented when that reference no longer refers to the allocation, either by going out of scope or being assigned to refer to a different location. In general, reference counting suffers from two main weaknesses. First, reference counting significantly degrades the performance of the mutator because instructions for adjusting the reference count must be executed after every reference assignment. The performance degradation is further magnified in parallel programs because reference count updates must be synchronized to avoid losing updates. Second, reference counting cannot collect data structures that form a cycle of references (*e.g.* a circular linked list). In most cases, reference counting schemes are augmented with an additional garbage collector that is capable of collecting data structures with cycles.

HAMM attempts to eliminate the runtime performance overheads of reference counting using hardware support. Similar to most reference counting schemes, HAMM is meant to work alongside a software garbage collector that can handle the collection of cyclical data. HAMM adds an additional byte for the reference count in the header of each object. If the reference count would overflow due to an increment, HAMM simply stops processing increment and decrement operations on that reference count and leaves the object for the software garbage collector to eventually collect.

As briefly noted above, reference counting can incur significant runtime performance overheads in parallel environments due to the need to synchronize updates to the reference count field. HAMM deals with this performance issue by locally tracking per-core reference count updates in Reference Count Coalescing Buffers (RCCB). The per-core (L1) reference counts are then combined in an L2 RCCB prior to being propagated to main memory. If the in-memory reference count reaches zero, the object can be considered garbage, and its memory can be reallocated using HAMM's `REALLOCMEM` instruction. By tracking reference counts on a per-core basis, HAMM avoids the synchronization that is otherwise necessary for reference counting in a parallel environment.

Using reference counting, HAMM is able to identify free blocks of memory for the software allocator. On average, HAMM's `REALLOCMEM` function is able to find a free block for the software allocator on 70% of all allocation requests. Garbage collection pauses generally occur when the allocator is unable to find enough memory to fill an allocation request. Thus, HAMM is able to limit the frequency of garbage collection pauses. On average, HAMM eliminates 52% of young generation collections and 50% of old generation collections. Essentially, the reference counting hardware acts as a concurrent full-heap collector that does not need to traverse the entire heap to confirm that an object is garbage. Although HAMM decreases the frequency of software garbage collection pauses, the length of these pauses often increases because more garbage is accumulated in between collections. Presumably, the heap also becomes fragmented due to HAMM's two-level reallocation mechanism, possibly causing compaction to occur on every full-heap collection (depending on the software garbage collector's policies).

4.3 Concurrent Collection

Concurrent garbage collection aims to avoid all stop-the-world garbage collection pauses by constantly performing garbage collection concurrently with the mutator. If the concurrent collector is able to find and recycle memory at a rate that is greater than or equal to the mutator's allocation rate, stop-the-world collection is never necessary (though many collectors will still use stop-the-world pauses to handle memory fragmentation). HAMM [7] implicitly uses concurrent collection through its hardware-supported reference counting scheme that is able to locate free blocks of memory while the mutator is executing. Schism [9] and C4 [11] are both concurrent garbage collectors (though both admit to needing small stop-the-world pauses for technical reasons).

Many concurrent collectors rely on the availability of extra cores for running concurrent garbage collection threads. C4 [11] was originally designed to run on Azul Systems hardware, which contains anywhere from 24 to 384 cores. Thus, finding spare cores is not a major issue for C4. On the other hand, Schism [9] is designed to run in real-time systems that likely do not have nearly as many spare cores as the Azul Systems servers. In the case that Schism does not have a dedicated core to run on, it uses slack-based scheduling with the collector thread having a lower priority than any critical real-time threads. Therefore, the Schism collector should never interfere with the execution of a real-time task with a hard real-time deadline.

The C4 garbage collection algorithm does not require any stop-the-world pauses. However, the C4 implementation does require minor stop-the-world pauses due to various technical considerations. The authors do not note any specific reasons for stop-the-world pauses in the C4 paper [11]. However, the reported pause times (less than a millisecond) are significantly lower than the pause times reported for the original Pauseless GC collector (up to 21 milliseconds) [4]. In an interview, one of the authors noted that the remaining pauses are due to engineering problems related to the Java programming language, such as cleaning out the compiled code cache and other runtime bookkeeping [12]. Although Schism and C4 do not require traditional stop-the-world pauses for garbage collection, both collectors require minor thread local pauses for garbage collection related work.

4.3.1 Checkpoints

Schism and C4 avoid stop-the-world pauses through the use of *checkpoints* (or *ragged safepoints* in Schism). The garbage collector uses checkpoints to request that all mutator threads perform some small amount of garbage collection related work at the next safe-point. Unlike a stop-the-world pause at a standard safe-point, a checkpoint allows mutator threads to continue executing concurrently once they have finished the work required by the garbage collector.

Schism uses checkpoints mainly for scanning the stack roots of each mutator thread. Relying on an earlier approach [10], high priority threads scan their own roots and report them to the collector. This approach limits the time that high priority threads are paused to the amount of time required for them to scan their own stack. Otherwise, a stop-the-world pause for stack scanning would stop all threads for at least the amount of time necessary for the collector to scan the largest thread stack. This approach relies on the assumption that higher priority threads will generally have small stacks and thus will not require a large amount of pause time to scan their own stacks. These checkpoints require priority-boosting for low priority threads to ensure that all threads complete the requested garbage collection related work in a reasonable amount of time.

Although checkpoints are less obtrusive than stop-the-world pauses, they still prevent the mutator from performing useful work for some amount of time. The main benefit of using checkpoints is that the GC time for a checkpoint is bound by the amount of work that each thread needs to perform locally. Thus, threads that need to perform a large amount of work do not block the progress of threads that only need to perform a small amount of work. Overall, checkpoints provide a useful mechanism for performing garbage collection without stopping all threads. Both Schism and C4 provide low worst-case response times because they do not require lengthy stop-the-world pauses.

4.4 Summary and Discussion

All three of the examined collectors avoid stop-the-world pauses by performing concurrent collection. Although HAMM does not explicitly call itself a concurrent collector, the reference counting performed by hardware occurs concurrently with the mutator's execution and does not require the mutator to execute any additional instructions (*i.e.* reference count increment and decrement). Rather, mutator instructions that would have already been executed are replaced by the compiler with new variants that allow reference counting to be performed concurrently by the hardware.

Schism and C4 are concurrent garbage collectors that use new techniques to avoid all stop-the-world pauses, at least in theory. Schism's slack-based scheduling policy is beneficial in real-time systems because it avoids preempting higher priority tasks. With additional cores, such a scheduling policy is unnecessary, as demonstrated by C4. Checkpoints for performing small amounts of garbage collection related work are beneficial for limiting worst-case response times because each thread is only bound by the amount of work that it needs to perform on itself. Thus, checkpoints offer some amount of control over response-times to the programmer, who could consciously attempt to limit the amount of garbage collection work required by response-time critical threads (*i.e.* limit the stack size).

5 Limiting Mutator Runtime Performance Overheads

As discussed in the previous section, garbage collectors attempt to limit the frequency and duration of stop-the-world pauses for collection. However, the mechanisms used to avoid garbage collection pauses often interfere with the runtime performance of the mutator. For example, software based reference counting requires no stop-the-world garbage collection pauses, but the increment and decrement operations on reference counts are, in essence, inserted into the mutator code, causing the total runtime of the mutator to increase. Changes in mutator runtime performance are generally measured in terms of *minimum mutator utilization*, which, given a time window of increasing length, measures the minimum amount of time that the mutator was able to execute.

Applying garbage collection to a program often affects the spatial locality of memory accesses by the mutator. Changes in spatial locality can be caused by 1) changes in object layout due to metadata used by the garbage collector and 2) changes in object layout with respect to other objects. These changes in locality and cache behavior can lead to increased mutator runtime. The effects of a garbage collector on spatial locality can be evaluated by measuring overall mutator performance and change in percentage of cache misses at each level of the cache.

5.1 Hardware Support for Garbage Collection

Hardware support can be used in multiple ways to limit the performance impact of garbage collection on the mutator. The authors of HAMM [7] outline many prior approaches to hardware-based garbage collection and conclude that these prior approaches overfit to specific garbage collection algorithms. In this section, we outline the hardware support used to enable efficient mutator performance in both HAMM [7] and C4 [11].

5.1.1 Reference Counting

HAMM performs hardware-supported reference counting by replacing instructions (during compilation) that create and destroy references with new instructions that increment and decrement a reference count for the referred-to object. For efficiency in parallel environments, HAMM coalesces reference count increment and decrement operations in per-core L1 RCCBs and then in L2 RCCBs before propagating the reference count to main memory. Unfortunately, HAMM's effect on the runtime performance of the mutator cannot be evaluated in a standard way (using minimum mutator utilization) because the hardware support for HAMM can only be simulated. Thus, the paper does not provide results for the overall increase in runtime of a program under the effects of hardware-based reference counting. To evaluate the increase in mutator runtime due to HAMM, the authors simulated slices of 200 million instructions. On average, the instructions-per-cycle (IPC) of the simulated slices decreased by 0.4%, indicating that the runtime performance of the mutator is nearly unchanged by HAMM. However, the paper does not indicate the characteristics (*i.e.* percent of instructions that manipulate reference counts) of the evaluated program slices, so it is difficult to infer how much the IPC might be affected on different program slices.

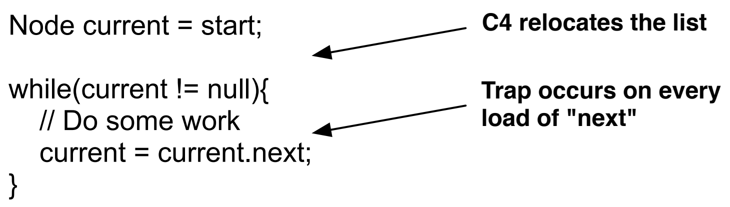
5.1.2 Read Barriers

Concurrent copying collectors[2], such as C4 [11], require *read barriers*. Read barriers prevent the mutator from attempting to access the old location of an object that has already been relocated by the collector. Traditional read barriers are implemented with either page protection [8] or an indirection pointer [2]. Both of these solutions introduce significant runtime performance overheads to the mutator's execution. C4 [11] largely eliminates these overheads through the use of a hardware-supported[3] *Loaded Value Barrier* (LVB).

C4's Loaded Value Barrier is issued after every load instruction. The LVB ensures that (1) all loaded references will be marked by the concurrent collector and (2) all loaded references point to the current location of the target object [11]. When the LVB encounters a reference that violates one of these conditions, a garbage collection trap is triggered that either (1) corrects the reference's marking metadata and adds the reference to the collector's list of objects to be scanned or (2) corrects the reference to point to the relocated object. When the LVB corrects a reference to a relocated object, it also corrects the memory location that the reference was loaded from. This *self-healing* behavior prevents C4 from repeatedly trapping when the

[2]Note that concurrent, non-copying collectors do not require read barriers because no objects are moved. Stop-the-world, copying collectors also do not require read barriers because the collector can update all references to the old object location to point to the new location while the mutator is paused.

[3]No hardware support is required for the X86 version of C4. The read barrier for the X86 implementation incurs a 20% slowdown. However, the X86 processors are 5x faster than Azul Systems custom processors[3].



```
Node current = start;

while(current != null){
   // Do some work
   current = current.next;
}
```

Figure 1: Code that may exhibit a trap storm in C4

same reference is loaded from memory multiple times — only the first load of a reference (by a single thread) will cause a trap.

Although the self-healing behavior of C4's LVB prevents repeated traps on the same reference, relocating a linked data structure can cause frequent traps until all of its references have been corrected. In their prior paper [4], the authors coin the term *trap storm* for this behavior. Figure 1 demonstrates how the traversal of a linked list may cause a trap storm if C4 has relocated the list. After the list has been relocated, every load of the "next" field will cause a trap for C4 to correct the reference. After all of the references have been corrected by C4's LVB, future list traversals (before the next relocation) will execute faster. This behavior causes some amount of unpredictability in the runtime execution of the mutator because it is difficult to predict when C4 will relocate an object. Unlike prior read barriers, the LVB avoids trapping on `null` references, which are quite common in Java.

5.1.3 Write Barriers

In order to avoid marking the old generation on every young generation collection, generational garbage collectors require *write barriers* that prevent unnoticed updates to reference fields in old generation objects that may point to young generation objects. C4 [11] provides a *Stored Value Barrier* (SVB) for efficient updates to the set of references from the old generation to the young generation. C4's SVB uses a card marking technique in which a bitmap specifies to the collector that some segment of the old generation may contain a reference to the young generation, forcing the collector to scan that heap segment for possible references. In general, the card marking technique provides reasonable performance (common case is two instructions) even without hardware support. However, the SVB used by C4 uses only a single instruction and is able to reduce performance by avoiding additional overheads for null references, similar to the LVB. No specific evaluation is done on the runtime performance impact of the SVB, but it is likely small compared to the performance impact of the LVB[4].

5.2 Spatial Locality

Along with the runtime performance effects of garbage collection related code that is executed by the mutator, garbage collection can also affect the spatial locality of memory accesses by the mutator. Most collectors require some metadata, often stored in the object header, that is used by the collector to denote important information such as which objects have already been marked. As the size of the object header increases, less objects will fit into each cache line, which affects the spatial locality of the program. Copying collectors alter the layout of the entire heap by relocating objects in memory, possibly improving the spatial locality of memory accesses.

[4]Read barriers generally incur more runtime overhead than write barriers because loads are more common than stores.

5.2.1 Object Layout

Both C4 [11] and HAMM [7] add additional bits of metadata to the object header for garbage collection. In Java, objects already have metadata including type information and garbage collection metadata for the default collector. Thus, the additional metadata for C4 and HAMM is likely not a source of significant runtime overhead. In a language like C++ in which objects do not have headers already, the runtime and space overheads due to garbage collection metadata would likely be more of a concern.

Schism [9] drastically alters the layout of objects and arrays in memory, allocating objects as linked lists of fixed-size fragments and allocating arrays as arraylets. Under Schism's fragmentation-tolerant allocation scheme, two object fields or two elements of an array that are logically neighbors may not actually be located next to each other in memory. Consider a program that sequentially iterates over an array of `ints`. Under standard Java, each block that is loaded into the cache will likely contain more than one `int`[5], allowing the loop to take advantage of spatial locality and execute faster. Similarly, a hardware prefetcher may be able to predict that future blocks will be loaded into the cache, further increasing the performance of the loop. Under Schism's allocation scheme, the actual array data may not be allocated in contiguous fragments. Thus, iterating over the array of `ints` will likely require that more blocks are loaded into the cache. This additional cache traffic can lead to additional cache conflicts and additional cache coherence invalidations. Further, the hardware prefetcher will be less effective at predicting which cache blocks to load next, possibly polluting the cache. The arraylet representation used by Schism also further decreases runtime performance due to the indirection from the mutator reference to the sentinel to the arraylet spine to the actual array data. Compared to the standard Java allocator and collector, Schism only manages to exhibit 65% of the original throughput. Compared to a C implementation of a real-time benchmark, Schism suffers from an overhead of 37% per loop iteration.

5.2.2 Heap Layout

Copying collectors are often able to improve the runtime performance of the mutator by compacting objects in memory for improved spatial locality. C4 [11] constantly concurrently compacts objects in the heap, possibly resulting in improved mutator performance. Unfortunately, the cache miss rates under C4 were not evaluated, so it is difficult to know what effect C4 has on spatial locality. Schism does not relocate objects, so it does not benefit from the possible increase in locality from copying collection. HAMM does not require copying or non-copying collection for the software garbage collector that will be used with it. However, HAMM's ability to quickly provide blocks of memory to the allocator that have recently become garbage may increase the spatial locality of the heap.

5.3 Cache Interference

Garbage collection may interfere with the execution of the mutator by touching memory that alters the cache behavior of the mutator. For example, a concurrent collector marking thread may load and scan memory that is not currently being used by the mutator (or if the memory is garbage, that the mutator does not even have a reference to). The cache blocks required by the marking thread may conflict with cache blocks that are required by the mutator. Further, the amount of cache space required by the mutator may fill the cache, leading to capacity cache misses when the marking thread executes.

HAMM [7] updates reference counts on the creation or destruction of a reference. Although the reference count updates are not immediately applied, updating the reference count in main memory requires a load

[5]Assuming a cache line size of at least 8 bytes.

and store to the reference count field. These updates pollute the cache with data that is possibly not being used by the mutator, resulting in a 1.2% increase in L2 cache misses. Due to its quick reuse of memory, L1 cache misses actually decrease by 1%. In either case, the change in cache behavior is not significant in HAMM and should not significantly affect the performance of the mutator. As noted earlier, Schism [9] and C4 [11] both likely cause some amount of cache interference with the mutator due to concurrent collection, but the cache behavior is not measured for either collector.

5.4 Discussion

Each of the examined collectors trade some amount of mutator runtime performance for a large decrease in garbage collection time. HAMM barely alters the performance of the mutator because its reference counting instructions simply replace instructions that were already present in the mutator. All of the work done by HAMM is off of the critical path, so the only remaining performance concern for HAMM is how much the extra memory accesses affect the cache behavior of the mutator. However, HAMM does not completely eliminate the need for software garbage collections. Thus, the mutator's runtime performance may also be affected by the software garbage collector in the event that a generational, incremental, or concurrent garbage collector is used.

Schism alters the mutator's runtime performance by drastically changing the layout of objects and arrays. Although both latency and throughput are affected by Schism's fragmentation-tolerant allocation scheme, the overheads are likely acceptable for real-time applications because of the increase in predictability with regard to response times and space usage.

C4 requires write barriers (for generational collection) and read barriers (for concurrent collection). However, C4 maintains reasonable throughput compared to other current collectors and offers much lower worst-case response times. Trap storms due to C4's LVB can cause the performance of the mutator to be unpredictable, but this unpredictability seems to only cause a small amount of perturbation in the experimental execution times.

6 Discussion and Future Directions

The following sections discuss additional comparison points for the featured garbage collectors and some possible avenues for future work.

6.1 Application Domains

Both Schism [9] and C4 [11] target specific application domains. Schism targets real-time applications, and C4 targets enterprise applications with large heaps. HAMM [7] does not target a particular environment, but it is nonetheless interesting to consider applying HAMM to the target domains of Schism and C4.

6.1.1 Real-Time HAMM

The main limiting factor to applying HAMM in a real-time environment is its allocation policy. Without using HAMM for reallocation of memory blocks, the reference counting scheme used by HAMM provides no benefits. Unfortunately, HAMM's reallocation mechanism can cause memory fragmentation, which is undesirable in a real-time environment. To fix this problem, a concurrent compacting collector could be applied to remove the effects of memory fragmentation. Using a stop-the-world copying collector is likely not feasible because the pause times required for stop-the-world collection are not acceptable for real-time applications. Applying a concurrent copying collector along with HAMM requires handling the significant issue of concurrently updating the reference count of a relocated object. When the collector relocates an object, HAMM specifies that the `FLUSHRC` instruction must be called on the address of the relocated object to remove any pending reference count updates from the RCCBs. The collector must then manually update the reference count upon completion of the collection. Instead, with a concurrent copying collector, `FLUSHRC` should not be called, and the read barrier should handle updates to the relocated object's reference count. A page-protection based read barrier would need to handle reference count updates manually, possibly increasing the runtime overhead. An indirection style read barrier could be used with HAMM without modification as long as the indirection pointer is never relocated. A hardware read barrier could efficiently support HAMM's reference counting by simply updating the address of the reference count in hardware. During a concurrent collection, HAMM could still reference count and reallocate blocks of free memory that were allocated after the beginning of the concurrent collection (after the ABT was reset).

Rather than using a compacting collector, Schism could be used as the software garbage collector for HAMM. Schism's allocation policy would likely work well with HAMM because all allocated blocks have a fixed-size. Thus, all of the blocks reallocated by HAMM would fit exactly the amount requested by Schism's allocator. One major concern might be how effective HAMM would be at reference counting the linked structures used by Schism for objects (linked lists) and arrays (arraylets). For example, once the last reference to an array sentinel is destroyed by the mutator, HAMM would be able to free the sentinel block. HAMM would then scan the sentinel block for references and would find a reference to the array spine. Decrementing the array spine reference count would cause its reference count to be zero, and HAMM would be able to scan the spine for references to each of the fragments used to store the actual array data. The process of collecting an entire arraylet via reference counting takes multiple steps, and it is unclear how quickly HAMM would be able to free all of the blocks used for the array. HAMM would add an additional byte-sized field (for the reference count) to each object and arraylet, but this would not significantly impact Schism's data representation, which already includes multiple bytes for metadata. HAMM provides support for multiple spaces, which could be used to separate the allocation policies for the spine space and the rest of the heap.

6.1.2 Enterprise HAMM

In enterprise applications, the primary concerns are keeping up with the application's allocation rate and minimizing mutator response times. C4 would not likely benefit from HAMM because C4 already constantly compacts the heap to find free memory in both the young and old generations. An interesting comparison point might be to evaluate HAMM with the standard Java garbage collector against C4. HAMM's reference counting and reallocation mechanisms act as a constant concurrent collector. However, HAMM's allocation policy does lead to memory fragmentation, which would eventually require concurrent compaction. In an enterprise environment, it may be the case that any concurrent collector capable of executing on top of HAMM would simply be more efficient on its own. The key question to answer would be to determine how often concurrent collection must run along with HAMM to avoid excessive memory fragmentation. If the concurrent collector must constantly run, then HAMM may not be suited for enterprise environments.

6.1.3 Real-Time C4

C4 was designed to be used in enterprise environments with large heaps, large amounts of physical memory, and large numbers of cores. Using C4 in a real-time environment may require a slack-based or time-based scheduling policy such as those used by Schism in the event that additional cores are not available. This would likely reduce the effectiveness of C4 because it relies on the ability to constantly concurrently execute with the mutator to stay ahead of the mutator's allocation rate. However, allocation rates in real-time programs are likely lower than allocation rates in enterprise applications. The read and write barriers used by C4 are not likely to affect the mutator's runtime performance more than Schism's object layout does because the indirection incurred by Schism's object layout requires more critical-path instructions on every load and store than the single non-critical-path instruction needed for a C4 read or write barrier. Thus, with some modification, C4 could likely be used for real-time applications.

6.1.4 Enterprise Schism

Due to its runtime performance characteristics, Schism is not well suited for enterprise applications. Experiments showed that Schism only achieved 65% of the throughput of the standard Java garbage collector. On the other hand, C4 maintains consistent throughput compared to the Hotspot collector and significantly improves worst-case response times. Schism also tends to require a larger minimum heap size for its fragmentation-tolerant object layouts (30% of the total heap size for array spines). Schism's array optimizations are also best suited for small arrays, and enterprise applications are more likely to be dominated by large arrays.

6.2 Applicability to Other Languages

HAMM, Schism, and C4 all provide garbage collection mainly for Java programs. By focusing on Java, the designers of these garbage collectors certainly made design and implementation choices that were affected by the characteristics of Java programs. The Java language exhibits a few main characteristics that affect the design of garbage collectors for it.

The majority of Java objects are small (less than 128 bytes). Schism [9] allocates objects and arrays over fixed-size blocks of 32 bytes. In any language with larger average object sizes, a larger fixed-size block might reduce how frequently objects and arrays are split in memory. HAMM [7] uses 64 classes of fixed-size blocks for reallocation but does not specify what the specific sizes are. However, it is likely that, with 64 size classes, HAMM's allocation sizes could be broadly applied to other languages. For sizes that do not fit HAMM's size classes, the garbage collector can be used to manage any allocations of non-matching size. C4 [11] is designed for enterprise Java applications and already handles large sized allocations. Thus, C4 could likely be applied to a language with larger objects without much modification.

By default, all Java objects are allocated on the heap, so Java programs typically exhibit high rates of heap allocation. In a language that gives the programmer more control over whether or not an object is allocated on the stack or the heap, the rate of heap allocation will likely be lower. In languages with this feature, it is unclear whether or not the weak generational hypothesis applies because objects that become garbage quickly are often allocated on the stack instead of the heap. However, there is little to no experimental evidence to prove or disprove the weak generational hypothesis for languages that feature stack allocation. A generational, concurrent collector like C4 is likely less necessary in a language that allows stack allocation because the overall rate of allocation is probably lower.

Java does not allow references to internal object members.[6] Although applying Schism or C4 to a language that allows internal references may require changes to the implementation of each collector's mark phase, the presence of internal references is not likely to be a major obstacle for either collector. On the other hand, applying HAMM to a language that allows internal references would require significant changes to HAMM's method of finding an object's reference count. Currently, HAMM uses a fixed offset from the

[6]In C, a reference to an internal object member is created using the address-of operator (*e.g.* &foo.bar).

object's address to find the reference count [6]. If internal references are allowed, the reference count will not be at a fixed offset from an object field's address. Further, at compile time, it is difficult to know whether or not a reference (*e.g.* a function parameter) refers to an internal object. Thus, HAMM would likely need to be augmented with a mechanism for doing a range lookup on the object address that locates the reference count field. Luckily, HAMM's reference count updates are off the critical path (at instruction commit), so such a change should not significantly increase the runtime of the mutator. However, supporting internal object references would increase the complexity of the hardware-support required for HAMM.

In Java, references in memory can be precisely identified using type identifiers in object headers. In languages like C and C++, references cannot be precisely identified from other values that look like references (*e.g.* `unsigned long`). The mechanisms used by both HAMM [7] and C4 [11] for identifying reference values are both insufficient for precisely identifying all references in a language that does not already precisely identify references. In HAMM, registers and stack locations are tagged with a single bit indicating that a reference is present. However, references in main memory are not individually tagged, but can be identified in Java using type identifiers and object layout descriptions. In a language like C++, further support would be required for a copying collector to be able to identify references in main memory. However, HAMM's register and stack reference tagging could assist in a precise collection scheme for C++. In C4, the LVB is used to correct references that point to old object locations as the references are loaded. However, C4's LVB does not correct references in registers or on the stack, and the LVB only corrects references that are loaded from memory. The Remap phase corrects any additional references in main memory using the type identifiers and object layout descriptions provided by Java.

7 Conclusions

In this paper, we examined three modern garbage collection algorithms that attempt to provide more predictable application behavior. By handling memory fragmentation, Schism and C4 eliminate the need for stop-the-world compaction. All three collectors reduce the amount of garbage collection time required during application execution while limiting the amount of performance perturbation in the mutator. Overall, all three collectors reduce the unpredictability of garbage collection and make garbage collection more viable in their respective domains. With these and future advances, garbage collection should become even more commonplace and further eliminate the security risks caused by use-after-free errors.

References

[1] Blackburn, S. M., and McKinley, K. S. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. *SIGPLAN Not. 43*, 6 (June 2008), 22–32.

[2] Brooks, R. A. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming* (1984), ACM.

[3] Click, C. *Azul and Open Source*, 2011. `http://www.azulsystems.com/blog/wp-content/uploads/2011/02/2011_FOSDEM_OpenSrc.pdf`.

[4] Click, C., Tene, G., and Wolf, M. The pauseless gc algorithm. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments* (2005), ACM.

[5] Hertz, M., and Berger, E. D. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (2005), ACM.

[6] Joao, J. A. *HAMM Technical Report*, 2013. Email Correspondence.

[7] Joao, J. A., Mutlu, O., and Patt, Y. N. Flexible reference-counting-based hardware acceleration for garbage collection. In *Proceedings of the 36th annual International Symposium on Computer Architecture* (2009), ACM.

[8] Jones, R., and Lins, R. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley, 1996.

[9] Pizlo, F., Ziarek, L., Maj, P., Hosking, A. L., Blanton, E., and Vitek, J. Schism: Fragmentation-tolerant real-time garbage collection. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming Language Design and Implementation* (2010), ACM.

[10] Puffitsch, W., and Schoeberl, M. Non-blocking root scanning for real-time garbage collection. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems* (2008), ACM.

[11] Tene, G., Iyengar, B., and Wolf, M. C4: The continuously concurrent compacting collector. *SIGPLAN Not. 46*, 11 (June 2011), 79–88.

[12] Venners, B. *Azul's Pauseless Garbage Collector.* Artima Developer, 2010. `http://www.artima.com/lejava/articles/azul_pauseless_gc.html`.