Taking Off the Gloves with Reference Counting Immix

Rifat Shahriyar, Stephen M. Blackburn, Xi Yang
Australian National University
<First.Last>@anu.edu.au

Kathryn S. McKinley
Microsoft Research
mckinley@microsoft.com

Abstract

Despite some clear advantages and recent advances, reference counting remains a poor cousin to high-performance tracing garbage collectors. The advantages of reference counting include a) immediacy of reclamation, b) incrementality, and c) local scope of its operations. After decades of languishing with hopelessly bad performance, recent work narrowed the gap between reference counting and the fastest tracing collectors to within 10%. Though a major advance, this gap remains a substantial barrier to adoption in performance-conscious application domains.

Our work identifies heap organization as the principal source of the remaining performance gap. We present the design, implementation, and analysis of a new collector, RC Immix, that replaces reference counting's traditional free-list heap organization with the line and block heap structure introduced by the Immix collector. The key innovations of RC Immix are 1) to combine traditional reference counts with per-line live object counts to identify reusable memory and 2) to eliminate fragmentation by integrating copying with reference counting of new objects and with backup tracing cycle collection. In RC Immix, reference counting offers efficient collection and the line and block heap organization delivers excellent mutator locality and efficient allocation. With these advances, RC Immix closes the 10% performance gap, outperforming a highly tuned production generational collector. By removing the performance barrier, this work transforms reference counting into a serious alternative for meeting high performance objectives for garbage collected languages.

Categories and Subject Descriptors Software, Virtual Machines, Memory management, Garbage collection

Keywords Reference Counting, Immix, Mark-Region, Defragmentation

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/2509136.2509527

1. Introduction

In 1960, researchers introduced the two main branches of automatic garbage collection: *tracing* and *reference counting* [14, 24]. Reference counting directly identifies dead objects by counting the number of incoming references. When the count goes to zero, the object is unreachable and the collector may reclaim it. Tracing takes the opposite tack. It identifies live objects by performing a transitive closure over the object graph, implicitly identifying dead objects. It then reclaims all untraced objects.

Reference counting has advantages. 1) It may reclaim objects as soon as they are no longer referenced. 2) It is inherently incremental. 3) Its operations are object-local, rather than global in scope. Its major disadvantage is that it cannot reclaim cycles and therefore it requires a backup tracing collector [2, 18]. This limitation has the practical consequence that any reference counter that guarantees completeness (i.e., it will eventually reclaim all garbage) essentially requires two collector implementations. Furthermore, the performance of reference counting implementations lagged high performance tracing collectors by 30% or more until recently [21, 22, 27]. In 2012, Shahriyar et al. solved two problems responsible for much of the performance overhead of reference counting. This paper identifies and solves the remaining problems, completely eliminating performance degradation as a barrier to adoption.

Shahriyar et al. identify the following characteristics of programs and use them to optimize reference counting. (We call their collector RC for simplicity.)

1. *The vast majority of reference counts are low, less than five.* The RC collector uses only a few bits for the reference count. It *sticks* counts at a maximum before they overflow and then corrects stuck counts when it traces the heap during cycle collection.
2. *Many reference count increments and decrements are to newly allocated objects.* RC elides reference counting of new objects and allocates them as dead, which eliminates a lot of useless work.

RC performs deferred reference counting and occasional backup cycle tracing. Deferral trades vastly fewer reference counting increments and decrements for less immediacy of reclamation. RC divides execution into three distinct phases:

mutation, reference counting collection, and cycle collection. The result is a reference counting collector with the same performance as a whole-heap tracing collector, and within 10% of the best high performance generational collector in MMTk [6, 8, 27].

This paper identifies that the major source of this 10% gap is that RC's free-list heap layout has poor cache locality and imposes instruction overhead. Poor locality occurs because free-list allocators typically disperse contemporaneously allocated objects in memory, which degrades locality compared to allocating them together in space [6, 8]. Instruction overheads are greater in free lists, particularly when programming languages require objects to be pre-initialized to zero. While a contiguous allocator can do bulk zeroing very efficiently, a free-list allocator must zero object-by-object, which is inefficient [33].

To solve these problems, we introduce Reference Counting Immix (RC Immix). RC Immix uses the allocation strategy and the line and block heap organization introduced by Immix mark-region garbage collection [6]. Immix places objects created consecutively in time consecutively in space in free lines within blocks. Immix allocates into partially free blocks by efficiently skipping over occupied lines. Objects may span lines, but not blocks. Immix reclaims memory at a line and block granularity.

The granularity of reclamation is the key mismatch between reference counting and Immix that RC Immix resolves. Reference counting reclaims objects, whereas Immix reclaims lines and blocks. The design contributions of RC Immix are as follows.

- RC Immix extends the reference counter to count live objects on a line. When the live object count of a line is zero, RC Immix reclaims the free line.
- RC Immix extends *opportunistic copying* [6], which mixes copying with leaving objects in place. RC Immix adds *proactive* copying, which combines reference counting and copying to compact newly allocated live objects. RC Immix on occasion *reactively* copies old objects during cycle detection to eliminate fragmentation.

Combining copying and reference counting is novel and surprising. Unlike tracing, reference counting is inherently local, and therefore in general the set of incoming references to a live object is not known. However, we observe two important opportunities. First, in a reference counter that coalesces increments and decrements [21, 22], since each new object starts with no references to it, the first collection must enumerate *all* references to that new object, presenting an opportunity to move that object proactively. We find that when new objects have a low survival rate, the remaining live objects are likely to cause fragmentation. We therefore copy new objects, which is very effective in small heaps. Second, since completeness requires a tracing cycle collection phase, RC Immix seizes upon this opportunity to incorporate reactive defragmentation of older objects. In both cases, we use opportunistic copying, which mixes copying and leaving objects in place, and thus can stop copying when it exhausts available memory.

Two engineering contributions of RC Immix are improved handling of roots and sharing the limited header bits to serve triple duty for reference counting, backup cycle collection with tracing, and opportunistic copying. The combination of these innovations results in a collector that attains great locality for the mutator and very low overhead for reference counting.

Measurements on a large set of Java benchmarks show that for all but the smallest of heap sizes RC Immix outperforms the best high performance collector in the literature. In some cases RC Immix can perform substantially better. In summary, we make the following contributions compared to the previous state of the art [27].

1. We identify heap organization as the remaining performance bottleneck for reference counting.
2. We merge reference counting with the heap structure of Immix by marrying per-line live object counts with object reference counts for reclamation.
3. We identify two opportunities for copying objects — one for young objects and one that leverages the required cycle collector — further improving locality and mitigating fragmentation both proactively and reactively.
4. RC Immix improves performance by 12% on average compared to RC and sometimes much more, outperforming the fastest production and eliminating the performance barrier to using reference counting.

Because the memory manager determines performance for managed languages and consequently application capabilities, these results open up new ways to meet the needs of applications that depend on performance and prompt reclamation.

2. Motivation and Related Work

This section motivates our approach and overviews the necessary garbage collection background on which we build. We start with a critical analysis of the performance of Shahriyar et al's reference counter [27], which we refer to simply as RC. This analysis shows that inefficiencies derive from 1) remaining reference counting overheads and 2) poor locality and instruction overhead due to the free-list heap structure. We then review existing high performance collectors, reference counting, and the Immix [6] garbage collector upon which we build.

2.1 Motivating Performance Analysis

All previous reference counting implementations in the literature use a free-list allocator because when the collector determines that an object's count is zero, it may then immediately place the freed memory on a free list. We start our analysis by understanding the performance impact of this

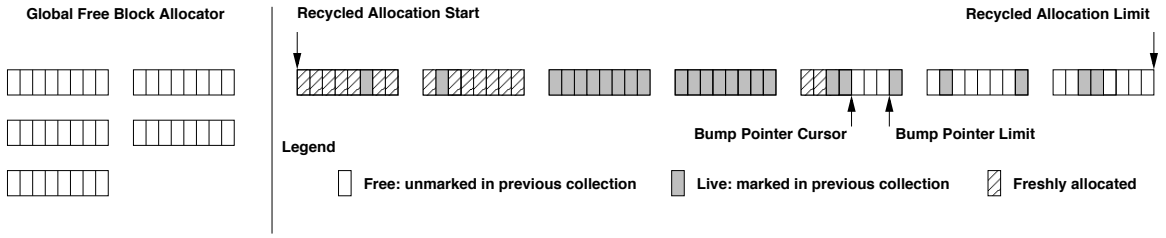


Figure 1. Immix Heap Organization

choice, using hardware performance counters. We then analyze RC further to establish its problems and opportunities for performance improvements.

Free-List and Contiguous Allocation The allocator plays a key role in mutator performance since it determines the placement and thus locality of objects. Contiguous memory allocation appends new objects by incrementing a bump pointer by the size of the new object [13]. On the other hand, modern free-list allocators organize memory into k size-segregated free lists [4, 8]. Each free list is unique to a size class and is composed from blocks of contiguous memory. It allocates an object into a free cell in the smallest size class that accommodates the object. Whereas a contiguous allocator places objects in memory based on allocation order, a free list places objects in memory based on their size and free memory availability.

Blackburn et al. [8] show that contiguous allocation in a copying collector delivers significantly better locality than free-list allocation in a mark-sweep collector. Feng and Berger [17] show similar locality benefits from initial contiguous allocation in a free list for C applications, but only when allocation to live ratios are very low since with high ratios the allocator reverts to free-list allocation. We confirm the locality benefits of contiguous on contemporary hardware, workloads, and allocator implementations below.

When contiguous allocation is coupled with copying collection, the collector must update all references to each moved object [13], a requirement that is at odds with reference counting's local scope of operation. Because reference counting does not perform a closure over the live objects, in general, a reference counting collector does not know of and therefore cannot update all pointers to an object it might otherwise move. Thus far, this prevented reference counting from copying and using a contiguous allocator.

On the other hand, the Immix mark-region heap layout offers largely contiguous heap layout, line and block reclamation, and copying [6]. Figure 1 shows how Immix allocates objects contiguously in empty lines and blocks (see Section 2.3 for more details). In partially full blocks, Immix skips over occupied lines.

First to explore the performance impact of free-list allocation, we compare the *mutator* time, which is the total time minus the collector time, in Table 1. We measure Immix, mark-sweep using a free list, and semi-space [13], across a suite of benchmarks. (See Section 4 for methodology details.) We compare mutator time of Immix to mark-sweep to cleanly isolate the performance impact of the free-list allocator versus the Immix allocator. Mark-sweep uses the same free-list implementation as RC, and neither Immix nor mark-sweep use barriers in the mutator. We also compare to semi-space. Semi-space is the canonical example of a contiguous allocator and thus an interesting limit point, but it is incompatible with reference counting. The semi-space data confirms that Immix is very close to the ideal for a contiguous allocator.

Mutator	Immix	Mark-Sweep	Semi-Space
Time	1.000	1.087	1.007
Instructions Retired	1.000	1.071	1.000
L1 Data Cache Misses	1.000	1.266	0.966

Table 1. The *mutator* characteristics of mark-sweep relative to Immix using the geometric mean of the benchmarks. GC time is excluded. Free-list allocation increases the number of instructions retired and L1 data cache misses. Semi-space serves as an additional point of comparison.

The contiguous bump allocator has two advantages over the free list, both of which are borne out in Table 1. The combined effect is almost a 9% performance advantage. The first advantage of a contiguous allocator is that it improves the cache locality of contemporaneously allocated objects by placing them on the same or nearby cache lines, and interacts well with modern memory systems. Our measurements in Table 1 confirm this intuition, showing that a free list adds 26% more L1 data cache misses to the mutator, compared to the Immix contiguous allocator. This degradation of locality has two related sources. 1) Contemporaneously allocated objects are much less likely to share a cache line when using a free list. 2) A contiguous allocator touches memory sequentially, priming the prefetcher to fetch lines before the allocator writes new objects to them. On the other hand, a free-list allocator disperses new objects, defeating hardware prefetching prediction mechanisms. Measurements by Yang et al. show these effects [33].

Mutator	Sticky Immix	RC	Immix
Time	1.000	1.093	0.975
Instructions Retired	1.000	1.092	0.972
L1 Data Cache Misses	1.000	1.329	1.018

Table 2. The *mutator* characteristics of RC and Sticky Immix, which except for heap layout have similar features. GC time is excluded. RC's free list allocator increases instructions retired and L1 cache misses. Immix serves as a point of comparison.

The second advantage of contiguous allocation is that it uses fewer instructions per allocation, principally because it zeros free memory in bulk using substantially more efficient code [33]. The allocation itself is also simpler because it only needs to check whether there is sufficient memory to accommodate the new object and increase the bump pointer, while the free-list allocator has to look up and update the metadata to decide where to allocate. However, we inspect generated code and confirm the result of Blackburn et al. [8] — that in the context of a Java optimizing compiler, where the size of most objects is statically known, the free-list allocation sequence is only slightly more complex than for the bump pointer. The overhead in additional instructions shown in Table 1 is therefore solely attributable to the substantially less efficient cell-by-cell zeroing required by a free-list allocator. We measure a 7% increase in the number of retired instructions due to the free list compared to Immix's contiguous allocator.

Analyzing RC Overheads We use a similar analysis to examine mutator overheads in RC [27] by comparing to Sticky Immix [6, 15], a generational variant of Immix. We choose Sticky Immix for its similarities to RC. Both collectors a) are mostly non-moving, b) have generational behavior, and c) use similar write barriers. This comparison holds as much as possible constant but varies the heap layout between free list and contiguous.

Table 2 compares mutator time, retired instructions, and L1 data cache misses of RC and Sticky Immix. The mutator time of RC is on average 9.3% slower than Sticky Immix, which is reflected by the two performance counters we report. 1) RC has on average 9.2% more mutator retired instructions than Sticky Immix. 2) RC has on average 33% more mutator L1 data cache misses than Sticky Immix. These results are consistent with the hypothesis that RC's use of a free list is the principal source of overhead compared to Sticky Immix, and motivates our design that combines reference counting with the Immix heap structure.

2.2 High Performance Reference Counting

The first account of reference counting was published by George Collins in 1960 [14], just months after John McCarthy first described tracing garbage collection [24]. The two approaches are duals. Reference counting directly identifies dead objects by keeping a count of the number of references to each object, freeing the object when its count reaches zero. Tracing algorithms, such as McCarthy's, do not directly identify dead objects, but rather, they identify live objects, and the remaining objects are implicitly dead. Most high performance tracing algorithms are *exact*, which means that they precisely identify all live objects in the heap. To identify live objects, they must enumerate all live references from the running program's stacks, which means that the runtime must maintain accurate *stack maps*. Maintaining stack maps is a formidable engineering burden, and is a reason why some language developers use reference counting rather than tracing [19]. To build stack maps, the compiler (or interpreter) must be able to determine for every register and stack location, at *every* point in the program's execution where a GC is legal, whether that location contains a valid heap reference or not.

Collins' first reference counting algorithm suffered from significant drawbacks including: a) an inability to collect cycles of garbage, b) overheads due to tracking very frequent pointer mutations, c) overheads due to storing the reference count, and d) overheads due to maintaining counts for short lived objects. The following paragraphs briefly outline five important optimizations developed over the past fifty years to improve over Collins' original paper. Shahriyar et al. show that together these optimizations deliver competitive performance [27].

Deferral To mitigate the high cost of maintaining counts for rapidly mutated references, Deutsch and Bobrow introduced deferred reference counting [16]. Deferred reference counting ignores mutations to frequently modified variables, such as those stored in registers and on the stack. Deferral requires a two phase approach, dividing execution into distinct mutation and collection phases. This tradeoff reduces reference counting work significantly, but delays reclamation.

Since deferred references are not accounted for during the mutator phase, the collector counts other references and places zero count objects in a zero count table (ZCT) deferring their reclamation. Periodically in a GC reference counting phase, the collector enumerates all deferred references into a root set and then reclaims any object in the ZCT that is not in the root set.

Bacon et al. [3] eliminate the zero count table by buffering decrements between collections. At collection time, the collector temporarily increments a reference count to each object in the root set and then processes all of the buffered decrements. Although much faster than naïve immediate reference counting, these schemes typically require stack maps to enumerate all live pointers from the stacks. Stack maps are an engineering impediment, which discourages many reference counting implementations from including deferral [19].

Coalescing Levanoni and Petrank observed that all but the first and last in any chain of mutations to a reference within a

given window can be *coalesced* [21, 22]. Only the initial and final states of the reference are necessary to calculate correct reference counts. Intervening mutations generate increments and decrements that cancel each other out. This observation is exploited by remembering (logging) only the initial value of a reference field when the program mutates it between periodic reference counting collections. At each collection, the collector need only apply a decrement to the initial value of any over-written reference (the value that was logged), and an increment to the latest value of the reference (the current value of the reference).

Levanoni and Petrank implemented coalescing using *object remembering*. The first time the program mutates an object reference after a collection phase a) a write barrier logs *all* of the outgoing references of the mutated object and marks the object as logged; b) all subsequent reference mutations in this mutator phase to the (now logged) object are ignored; and c) during the next collection, the collector scans the remembered object, increments *all* of its outgoing pointers, decrements all of its remembered outgoing references, and clears the logged flag. This optimization uses two buffers called the mod-buf and dec-buf. The allocator logs all new objects, ensuring that outgoing references are incremented at the next collection. The allocator does *not* record old values for new objects because all outgoing references start as *null*.

Limited Bit Counts Each object has a reference count. A dedicated word for the count guarantees that it will never overflow since a word is large enough to count a pointer from every address in the address space. However, reference counting may use fewer bits [20]. Shahriyar et al. show that using a full word adds non-negligible overhead. They instead use just four bits that are available in the object's header word in many systems [27]. The reference counter leaves any count that is about to overflow in a stuck state, protecting the integrity of the remainder of the header word, but introducing a potential garbage leak. Each time the cycle collector runs it resets each reference count, which has the effect of bounding the impact of stuck reference counts. Shahriyar et al. show that this strategy performs well.

Cycle Collection Reference counting suffers from the problem that cycles of objects will sustain non-zero reference counts, and therefore cannot be collected. To attain completeness, a separate backup tracing collector executes from time to time to eliminate cyclic garbage [31]. Backup tracing must enumerate live root references from the stack and registers, which requires stack maps. For this reason, naïve reference counting implementations usually do not perform cycle collection.

A backup tracing collector typically collects cycles by performing a mark-sweep trace of the entire heap. Researchers tried limiting tracing to mutated objects [2], but subsequently Frampton showed backup tracing, starting from the roots, performs better [18].

Both RC and RC Immix use backup tracing [18, 27]. Performing the trace requires a mark bit in each object header. During the trace, RC takes the opportunity to recompute all reference counts and thus may fix stuck counts. RC then sweeps all dead objects to the free list. RC Immix uses the same approach, except that it also recomputes object counts on lines and then reclaims free lines and blocks.

Young Objects As the weak generational hypothesis states, most objects die young [23, 30], and as a consequence, young objects are a very important optimization target. All high performance collectors today exploit this observation, typically via a copying generational nursery [30]. Prior work applies the weak generational hypothesis to reference counting by combining reference counting with a copying nursery [5] and by in-place mark-sweep tracing [25].

Shahriyar et al. applied two optimizations to deferred, coalescing reference counting to exploit short lived young objects: 1) *lazy mod-buf insertion* and 2) *allocate as dead*. Lazy mod-buf insertion avoids adding new objects to the mod-buf. Instead, it sets a *new* bit in object headers during allocation and the collector only adds new objects to the mod-buf lazily when it processes increments. During collection whenever the subject of an increment has its new bit set, the collector first clears the new bit and then pushes the object into the mod-buf. Because in a coalescing deferred reference counter, all references from roots and old objects will increment all objects they reach, this approach will only retain new objects directly reachable from old objects and the roots. For each object in the mod-buf, the collector will increment each of its children, which makes this scheme transitive. Thus new objects are effectively traced.

The allocate as dead optimization is a simple extension of the above strategy. Instead of allocating objects live with a reference count of one and enqueueing a compensating decrement, this strategy allocates new objects as dead and does not enqueue a decrement. This optimization inverts the presumption: the reference counter does not need to identify new objects that are dead, but it must rather identify live objects. This inversion means that the collector performs work in the infrequent case when a new object survives a collection, rather that in the common case when it dies. New objects become live when they receive their first increment when the collector processes the mod-buf. This strategy removes the need for creating compensating decrements and avoids explicitly freeing short lived objects.

Modern widely used implementations of reference counting employ few, if any, of the above optimizations. The likely explanation for this phenomena is two-fold. First, reference counting lagged the best generational garbage collectors by 40% until 2012 when Shahriyar et al. closed the gap to 10%. Consequently, reference counting is not currently used in performance critical settings. Second, one attraction of simple reference counting implementations is that they do not

require sophisticated runtime system support, such as precise stack maps. Many of the optimizations we describe here require the same runtime support as a tracing collector, undermining a principal advantage of a simple implementation. The reference counting implementations in widely used languages such as PHP and Objective-C are naïve. Because they lack these optimizations, they are inefficient. Shahriyar et al.'s collector implements all these optimizations and is our RC baseline.

We have outlined the state of the art in reference counting. RC Immix builds upon this foundation and then extends it by a) changing the underlying heap structure, and b) performing proactive and reactive copying to mitigate fragmentation and improve locality. The result is that RC Immix entirely eliminates the 10% performance overhead suffered by the fastest previous reference counting implementation.

2.3 Heap Organization and Immix

Blackburn and McKinley outline three heap organizations: a) free lists, b) contiguous, and c) regions [6]. Until now, reference counting used a free-list heap structure. In this paper, we adapt reference counting to use regions. In particular, we combine object reference counting with the line and block reclamation strategy used by Immix.

Free List A free-list allocator uses a heap structure that divides memory into cells of various fixed sizes [32]. When space is required for an object, the allocator searches a data structure called a free list to find a cell of sufficient size to accommodate the object. When an object becomes free, the allocator returns the cell containing the object to the free list for reuse. Free lists are used by explicit memory management systems and by mark-sweep and reference counting garbage collectors. Importantly, free-list allocators do not require copying of objects, which makes them particularly amenable to systems that use reference counting and to systems that require support for pinning of objects (i.e. objects that cannot be moved).

Free lists support immediate and fast reclamation of individual objects, which makes them particularly suitable for reference counting. Other systems, such as evacuation and compaction, must identify and move live objects before they may reclaim any memory. Also, free lists are a good fit to backup tracing used by many reference counters. Free lists are easy to *sweep* because they encode free and occupied memory in separate metadata. The sweep identifies and retains live objects and returns memory occupied by dead objects to the free list. Free lists suffer two notable shortcomings. First, they are vulnerable to fragmentation of two kinds. They suffer from internal fragmentation when objects are not perfectly matched to the size of their containing cell, and they suffer external fragmentation when free cells of particular sizes exist, but the allocator requires cells of another size. Second, they suffer from poor locality because they often position contemporaneously allocated objects in spatially disjoint memory, as discussed in Section 2.1.

Mark-Region Mark-region memory managers use a simple bump pointer to allocate objects into regions of contiguous memory [6]. A tracing collection marks each object and marks its containing region. Once all live objects have been traced, it reclaims unmarked regions. This design addresses the locality problem in free-list allocators. A mark-region memory manager can choose whether to move surviving objects or not. By contrast, evacuating and compacting collectors must copy, leading them to have expensive space or time collection overheads compared to mark-sweep collectors. Mark-region collectors are vulnerable to fragmentation because a single live object may keep an entire region alive and unavailable for reuse, and thus must copy some objects to attain good performance.

Immix: Lines, Blocks, and Opportunistic Copying Immix is a mark-region collector that uses a region hierarchy with two sizes: lines, which target cache line locality, and blocks, which target page level locality [6]. Each block is composed of lines, as shown in Figure 1. The allocator places new objects contiguously into empty lines and skips over occupied lines. Objects may span lines, but not blocks. Immix uses a bit in the header to indicate whether an object straddles lines, for efficient line marking. Immix recycles partially free blocks, allocating into them first.

Immix tackles fragmentation using *opportunistic* defragmentation, which mixes marking with copying. At the beginning of a collection, Immix identifies fragmentation as follows. Blocks with available memory indicate fragmentation because although available, the memory was not usable by the mutator. Furthermore, the live/free status for these blocks is up-to-date from the prior collection. In this case, Immix performs what we call here, a *reactive* defragmenting collection. To mix marking and copying, Immix uses two bits in the object header to differentiate between marked and forwarded objects. At the beginning of a defragmenting collection, Immix identifies source and target blocks. During the mark trace, when Immix first encounters an object that resides on a source block and there is still available memory for it on a target block, Immix copies the object to a target block, leaving a forwarding pointer. Otherwise Immix simply marks the object as usual. When Immix encounters forwarded objects while tracing, it updates the reference accordingly. This process is opportunistic, since it performs copying until it exhausts memory to defragment the heap. The result is a collector that combines the locality of a copying collector and the collection efficiency of a mark-sweep collector with resilience to fragmentation.

The best performing production collector in Jikes RVM is generational Immix (GenImmix) [6], which consists of a copying young space and an Immix old space.

3. Design of RC Immix

This section presents the design of RC Immix, which combines the RC and Immix collectors described in the previous section. This combination requires solving two problems. 1) We need to adapt the Immix line/block reclamation strategy to a reference counting context. 2) We need to share the limited number of bits in the object header to satisfy the demands of both Immix and reference counting.

In addition, RC Immix seizes two opportunities for defragmentation using *proactive* and *reactive* opportunistic copying. When identifying new objects for the first time, it opportunistically copies them, proactively defragmenting. When it, on occasion, performs cyclic garbage collection, RC Immix performs reactive defragmentation.

Similar to RC, RC Immix has frequent reference counting phases and occasional backup cycle tracing phases. This structure divides execution into discrete mutation, reference counting collection, and cycle collection phases.

3.1 RC and the Immix Heap

Until now, reference counting algorithms have always used free-list allocators. When the reference count for an object falls to zero, the reference counter frees the space occupied by the object, placing it on a free list for subsequent reuse by an allocator. Immix is a mark-region collector, which reclaims memory regions when they are completely free, rather than reclaiming memory on a per-object basis. Since Immix uses a line and block hierarchy, it reclaims free lines and if all the lines in a block are free, it reclaims the free block. Lines and block cannot be reclaimed until all objects within them are dead.

RC Immix Line and Block Reclamation RC Immix detects free lines by tracking the number of live objects on a line. RC Immix replaces Immix's line mark with a per-line live object count, which counts the number of live objects on the line. (It does not count incoming references to the line.)

As mentioned in Section 2.2, each object is born dead in RC, with a zero reference count to elide all reference counting work for short lived objects. In RC Immix, each line is also born dead with a zero live object count to similarly elide all line counting work when a newly allocated line only contains short lived objects. RC only increments an object's reference count when it encounters it during the first GC after the object is born, either directly from a root or due to an increment from a live mutated object. We propagate this laziness to per-line live object counts in RC Immix.

A newly allocated line will contain only newly born objects. During a reference counting collection, before RC Immix increments an object's reference count, it first checks the new bit. If the object is new, RC Immix clears the new object bit, indicating the object is now old. It then increments the object reference count *and* the live object count for the line. When all new objects on a line die before the collection, RC Immix will never encounter a reference to an object on the line, will never increment the live object count, and will trivially collect the line at the end of the first GC cycle. Because Immix's line marks are bytes (stored in the metadata for the block) and the number of objects on a line is limited by the 256 byte line size, live object counts do not incur any space penalty in RC Immix compared to the original Immix algorithm.

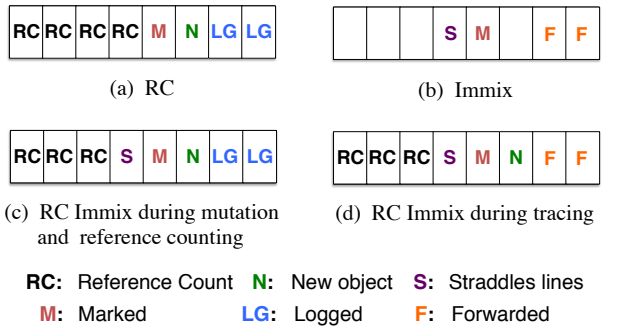


Figure 2. How RC, Immix, and the different phases of RC Immix use the eight header bits.

Limited Bit Count In Jikes RVM, one byte (eight bits) is available in the object header for use by the garbage collector. RC uses all eight bits. It uses two bits to log mutated objects for the purposes of coalescing increments and decrements, one bit for the mark state for backup cycle tracing, one bit for identifying new objects, and the remaining four bits to store the reference count. Figure 2(a) illustrates how RC fully uses all its eight header bits. Table 3 shows that four bits for the reference count is sufficient to correctly count references to more than 99.8% of objects.

To integrate RC and Immix, we need some header bits in objects for Immix-specific functionality as well. The base Immix implementation requires four header bits, fewer header bits than RC, but three bits store different information than RC. Both Immix and RC share the requirement for one mark bit during a tracing collection. Immix however requires one bit to identify objects that span multiple lines and two bits when it forwards objects during defragmentation. (Copying collectors, including Immix and RC Immix, first copy the object and then store a forwarding pointer in the original object's header.) Figure 2(b) shows the Immix header bits.

Immix and RC Immix both require a bit to identify objects that may span lines to ensure that all affected lines are kept live. Immix and RC Immix both use an optimization called conservative marking which means this bit is only set for objects that are larger than one line, which empirically is relatively uncommon [6]. Immix stores its line marks in per-block metadata and RC Immix does the same.

bits used	mean	compress	jess	db	javac	mpegaudio	mtrt	jack	avrora	bloat	chart	eclipse	fop	hsqldb	luindex	lusearch	pjbb	pmd	sunflow	xalan
1	**47.65**	33.93	57.31	9.37	66.77	7.29	20.23	44.76	34.74	49.54	47.08	86.36	49.68	96.89	66.31	75.83	38.23	59.62	13.54	47.83
2	**6.75**	0.08	0.16	0.80	4.96	0.16	0.95	0.95	15.74	2.54	1.83	9.38	4.73	47.38	20.75	5.57	4.67	5.15	0.01	2.47
3	**0.65**	0	0.08	0.68	0.59	0	0.16	0.01	6.69	0.10	0.02	1.15	0.31	0.21	0.16	0.01	0.14	1.53	0.01	0.59
4	**0.11**	0	0.06	0.68	0.28	0	0.08	0	0.06	0.05	0.01	0.24	0.10	0.01	0.01	0.01	0.02	0.26	0.01	0.17
5	**0.06**	0	0.03	0.49	0.12	0	0.03	0	0.06	0.03	0.01	0.07	0.05	0.01	0.01	0.01	0.01	0.14	0.01	0.06

Table 3. Percentage of objects that overflow for a given number of reference counting bits. RC Immix and RC use three and four bits, respectively. Data from Shahriyar et al. [27]. On average, 0.65% of objects overflow with three bits.

Immix and RC Immix both need to forward objects during defragmentation. Forwarding uses two bits during a collection to record the forwarding state (not forwarded, being forwarded, forwarded).

At first cut, it seems that there are not enough bits since adding Immix functionality to RC requires three bits and would thus reduce the bits for the reference count to just one. However, we observe that RC Immix only needs the logged bits for an object to coalesce increments and decrements during reference counting, and it only needs forwarding bits when tracing new objects and during backup cycle collection. These activities are mutually exclusive in time, so they are complementary requirements.

We therefore put the two bits to use as follows. 1) During mutation RC Immix follows RC, using the logged bits to mark modified objects that it has remembered for coalescing. 2) During a reference counting collection, RC Immix follows RC. For old objects, RC Immix performs increments and decrements as specified by coalescing and then clears the two bits. 3) For new objects and during cycle collection, RC Immix follows Immix. It sets the now cleared bits to indicate that it has forwarded an object and at the end of the collection, reclaims the memory. RC Immix thus overloads the two bits for coalescing and forwarding. Figure 2(c) shows how RC Immix uses the header bits during mutation and reference counting. Figure 2(d) shows how RC Immix repurposes the logged bits for forwarding during a collection. All the other bits remain the same in both phases.

Consequently, we reduce the number of reference counting bits to three. Three bits will lead to overflow in just 0.65% of objects on average, as shown in Table 3. When a reference count is about to overflow, it remains stuck until a cycle collection occurs, at which time it is reset to the correct value or left stuck if the correct count is higher.

Several optimizations and languages such as C# require pinning. Pinned objects are usually identified by a bit in the header. The simplest way to add pinning is to steal another bit from the reference count, reducing it to two bits. A slightly more complex design adds pinning to the logged and forwarded bits, since each of logged and forwarding only require three states. When we evaluated stealing a reference count bit for pinning, it worked well (see Section 5.3), so we did not explore the more complex implementation. Our default RC Immix configuration does *not* use pinning.

3.2 Cycle Collection and Defragmentation

Cycle Collection Reference counting suffers from the problem that cycles of objects will sustain non-zero reference counts and therefore cannot be collected. The same problem affects RC Immix, since line counts follow object liveness. RC Immix relies on a backup tracing cycle collector to correct incorrect line counts and stuck object counts. It uses a mark bit for each object and each line. It takes one bit from the line count for the mark bit and uses the remaining bits for the line count. The cycle collector starts by setting all the line marks and counts to zero. During cycle collection, the collector marks each live object, marks its corresponding line, and increments the live object count for the line when it first encounters the object. At the end of marking, the cycle collector reclaims all unmarked lines.

Whenever any reference counting implementation finds that an object is dead, it decrements the reference counts of all the children of the dead object, which may recursively result in more dead objects. This rule applies to reference counting in RC and RC Immix. RC and RC Immix's cycle collection is tasked with explicitly resetting all reference counts. In addition, RC Immix restores correct line counts. This feature eliminates the need to sweep dead objects altogether and RC Immix instead sweeps dead lines.

RC Immix performs cycle collection on occasion. How often to perform cycle collection is an empirical question that trades off responsiveness with immediacy of cycle reclamation that we explore below.

Defragmentation with Opportunistic Copying Reference counting is a local operation, meaning that the collector is only aware of the number of references to an object, not their origin. Therefore it is generally not possible to move objects during reference counting. However, RC Immix seizes upon two important opportunities to copy objects and thus mitigate fragmentation. First, we observe that when an object is subject to its first reference counting collection, *all* references to that object will be traversed, giving us a unique

opportunity to move the object during a reference counting collection. Because each object is unreferenced at birth, at its first GC, the set of all increments to a new object must be the set of *all* references to that object. Second, we exploit the fact that cycle collection involves a global trace, and thus presents another opportunity to copy objects. In both cases, we use *opportunistic* copying. Opportunistic copying mixes copying with in-place reference counting and marking such that it can stop copying when it exhausts the available space.

Proactive Defragmentation RC Immix's proactive defragmentation copies as many surviving new objects as possible given a particular *copy reserve*. During the mutator phase, the allocator dynamically sets aside a portion of memory as a copy reserve, which strictly bounds the amount of copying that may occur in the next collection phase. In a classic semi-space copying collector, the copy reserve must be large enough to accommodate all objects surviving because it is dictated by the worst case survival scenario. Therefore, every new block of allocation requires a block for the copy reserve. Because RC Immix is a mark-region collector, which can reuse partially occupied blocks, copying is optional. Copying is an optimization rather than required for correctness. Consequently, we size the copy reserve according to performance criteria.

Choosing the copy reserve size reflects a tradeoff. A large copy reserve eats into memory otherwise available for allocation and invites a large amount of copying. Although copying mitigates fragmentation, copying is considerably more expensive than marking and should be used judiciously. On the other hand, if the copy reserve is too small, it may not compact objects that will induce fragmentation later.

Our heuristic seeks to mimic the behavior of a generational collector, while making the copy reserve as small as possible. Ideally, an oracle would tell us the survival rate of the next collection (e.g., 10%) and the collector would size the copy reserve accordingly. We seek to emulate this policy by using past survival rate to predict the future. Computing fine-gain byte or object survival in production requires looking up every object's size, which is too expensive. Instead, we use line survival rate as an estimate of byte survival rate. We compute line survival rates of partially full blocks when we scan the line marks in a block to recycle its lines. This computation adds no measurable overhead.

Table 4 shows the average byte, object, line, and block percentage survival rates. Block survival rates significantly over predict actual byte survival rates. Line survival rates over predict as well, but much less. The difference between line and block survival rate is an indication of fragmentation. The larger the difference between the two, the more live objects are spread out over the blocks and the less likely a fresh allocation of a multi-line object will fit in the holes (contiguous free lines).

We experimented with a number of heuristics and choose two effective ones. We call our default copy reserve heuristic

Benchmark	Immix Alloc MB	Min Heap MB	Immix Survival Byte %	Object %	Line %	Block %
compress	0.3	21	6	5	7	11
jess	262	20	1	1	7	53
db	53	19	8	6	8	10
javac	174	30	17	19	32	66
mpegaudio	0.2	13	41	37	44	100
mtrt	97	18	3	3	6	11
jack	248	19	3	2	6	32
avrora	53	30	1	4	8	9
bloat	1091	40	1	1	5	32
chart	628	50	4	5	17	67
eclipse	2237	84	6	6	7	36
fop	47	35	14	13	29	69
hsqldb	112	115	23	23	26	56
jython	1349	90	0	0	0	0
luindex	9	30	8	11	11	15
lusearch	1009	30	3	2	4	22
lusearch-fix	997	30	1	1	2	8
pmd	364	55	9	11	14	26
sunflow	1820	30	1	2	5	99
xalan	507	40	12	5	24	51
pjbb2005	1955	355	11	12	24	87

Table 4. Benchmark characteristics. Bytes allocated into the Immix heap and minimum heap, in MB. The average survival rate as a percentage of bytes, objects, lines, and blocks measured in an instrumentation run at 1.5× the minimum heap size. Block survival rate is too coarse to predict byte survival rates. Line survival rate is fairly accurate and adds no measurable overhead.

Heuristic	Heap Size 1.2×	1.5×	2×
MAX	1.031	0.984	0.976
EXP	1.036	0.990	0.982

Table 5. Two proactive copying heuristics and their performance at 1.2, 1.5 and 2 times the minimum heap size, averaged over all benchmarks. Time is normalized relative to GenImmix. Lower is better.

MAX. MAX simply takes the maximum survival rate of the last N collections (4 in our experiments). Also good, but more complex, is a heuristic we call EXP. EXP computes a moving window of survival rates in buckets of N bytes of allocation (32 MB in our experiments) and then weights each bucket by an exponential decay function (1 for the current bucket, 1/2 for the next oldest, 1/4, and so on). Table 5 shows that the simple MAX heuristic performs well. We believe better heuristics are possible.

Reactive Defragmentation RC Immix also performs *reactive* defragmentation, during cycle collection. At the start of each cycle collection, the collector determines whether

Threshold		Heap Size		
Cycle	Defrag	1.2×	1.5×	2×
1%	**1%**	**1.030**	**0.983**	**0.975**
5%	5%	1.041	0.983	0.976
10%	10%	1.096	0.993	0.980

Table 6. Sensitivity to frequency of cycle detection and reactive defragmentation at 1.2, 1.5 and 2 times the minimum heap size, averaged over all benchmarks. Time is normalized relative to GenImmix. Lower is better.

to defragment based on fragmentation levels, any available free blocks, and any available partially filled blocks containing free lines, using statistics it gathers in the previous collection. RC Immix uses these statistics to select defragmentation sources and targets. If an object is unmovable when the collector first encounters it, the collector marks the object and line live, increments the object and line counts, and leaves the object in place. When the collector first encounters a movable live object on a source block, and there is still sufficient space for it on a target block, it opportunistically evacuates the object, copying it to the target block, and leaves a forwarding pointer that records the address of the new location. If the collector encounters subsequent references to a forwarded object, it replaces them with the value of the object's forwarding pointer.

A key empirical question for cycle detection and defragmentation is how often to perform them. If we perform them too often, the system loses its incrementality and pays both reference counting and tracing overheads. If we perform them too infrequently, it takes a long time to reclaim objects kept alive by dead cycles and the heap may suffer a lot of fragmentation. Both waste memory. This threshold is necessarily a heuristic. We explore thresholds as a function of heap size.

We use the following principle for our heuristic. If at the end of a collection, the amount of free memory available for allocation falls below a given threshold, then we mark the next collection for cycle collection. We can always include defragmentation with cycle detection, or we can perform it less frequently. Triggering cycle collection and defragmentation more often enables applications to execute in smaller minimum heap sizes, but will degrade performance. Depending on the scenario, this choice might be desirable. We focus on performance and use a free memory threshold which is a fraction of the total heap size. We experiment with a variety of thresholds to pick the best values for both and show the results for three heap sizes in Table 6. (See Section 4 for our methodology.) Based on the results in Table 6, we use 1% for both.

3.3 Optimized Root Scanning

The existing implementation of the RC algorithm treats Jikes RVM's boot image as part of the root set [27], enumerating each reference in the boot image at each collection. We identified this as a significant bottleneck in small heaps and instead treat the boot image as a non-collected part of the heap, rather than part of the root set. This very simple change delivers a significant performance boost to RC in modest heaps and is critical to RC Immix's performance in small heaps (Figure 4(a)).

4. Methodology

This section presents software, hardware, and measurement methodologies that we use to evaluate RC Immix.

Benchmarks. We draw 21 benchmarks from DaCapo [10], SPECjvm98 [28], and pjbb2005 [9]. The pjbb2005 benchmark is a fixed workload version of SPECjbb2005 [29] with 8 warehouses that executes 10,000 transactions per warehouse. We do not use SPECjvm2008 because that suite does not hold workload constant, so is unsuitable for GC evaluations unless modified. Since a few DaCapo 9.12 benchmarks do not execute on our virtual machine, we use benchmarks from both 2006-10-MR2 and 9.12 Bach releases of DaCapo to enlarge our suite.

We omit two outliers, mpegaudio and lusearch, from our figures and averages, but include them grayed-out in tables, for completeness. The mpegaudio benchmark is a very small benchmark that performs almost zero allocation. The lusearch benchmark allocates at three times the rate of any other. The lusearch benchmark derives from the 2.4.1 stable release of Apache Lucene. Yang et al. [33] found a performance bug in the method `QueryParser.getFieldQuery()`, which revision r803664 of Lucene fixes [26]. The heavily executed `getFieldQuery()` method unconditionally allocated a large data structure. The fixed version only allocates a large data structure if it is unable to reuse an existing one. This fix cuts total allocation by a factor of eight, speeds the benchmark up considerably, and reduces the allocation rate by over a factor of three. We patched the DaCapo lusearch benchmark with just this fix and we call the fixed benchmark lusearch-fix. The presence of this anomaly for over a year in public releases of a widely used package suggests that the behavior of lusearch is of some interest. Compared with GenImmix, RC Immix improves the performance of lusearch by 34% on i7-2600, but we use lusearch-fix in our results.

Jikes RVM & MMTk. We use Jikes RVM and MMTk for all of our experiments. Jikes RVM [1] is an open source high performance Java virtual machine (VM) written in a slightly extended version of Java. We use Jikes RVM release 3.1.2+hg r10475 to build RC Immix and compare it with different GCs. MMTk is Jikes RVM's memory management sub-system. It is a programmable memory management toolkit that implements a wide variety of collectors that reuse shared components [8].

All of the garbage collectors we evaluate are parallel [7]. They use thread local allocation for each application thread to minimize synchronization. During collection,

the collectors exploit available software and hardware parallelism [12]. To compare collectors, we vary the heap size to understand how well collectors respond to the time space tradeoff. In our experiments, no collector consistently ran in smaller heaps than the other collectors. Therefore we selected for our minimum heap size the smallest heap size in which *all* of the collectors execute, and thus have complete results at all heap sizes for all collectors.

Jikes RVM does not have a bytecode interpreter. Instead, a fast template-driven baseline compiler produces machine code when the VM first encounters each Java method. The adaptive compilation system then judiciously optimizes the most frequently executed methods. Using a timer-based approach, it schedules periodic interrupts. At each interrupt, the adaptive system records the currently executing method. Using a cost model, it then selects frequently executing methods it predicts will benefit from optimization. The optimizing compiler compiles these methods at increasing levels of optimizations.

To reduce perturbation due to dynamic optimization and to maximize the performance of the underlying system that we improve, we use a *warmup replay* methodology. Before executing any experiments, we gathered compiler optimization profiles from the 10th iteration of each benchmark. When we perform an experiment, we execute one complete iteration of each benchmark without any compiler optimizations, which loads all the classes and resolves methods. We next apply the benchmark-specific optimization profile and perform no subsequent compilation. We then measure and report the subsequent iteration. This methodology greatly reduces non-determinism due to the adaptive optimizing compiler and improves underlying performance by about 5% compared to the prior replay methodology [11]. We run each benchmark 20 times (20 invocations) and report the average. We also report 95% confidence intervals for the average using Student's t-distribution.

Operating System. We use Ubuntu 10.04.01 LTS server distribution and a 64-bit (x86_64) 2.6.32-24 Linux kernel.

Hardware Platform. We report performance, performance counter, and detailed results on a 32nm Core i7-2600 Sandy Bridge with 4 cores and 2-way SMT running at 3.4GHz. The two hardware threads on each core share a 32KB L1 instruction cache, 32KB L1 data cache, and 256KB L2 cache. All four cores share a single 8MB last level cache. A dual-channel memory controller is integrated into the CPU. The system has 4GB of DDR3-1066 memory installed.

5. Results

We first compare RC Immix with other collectors at a moderate 2× heap size, then consider sensitivity to available memory, and perform additional in depth analysis.

5.1 RC Immix Performance Overview

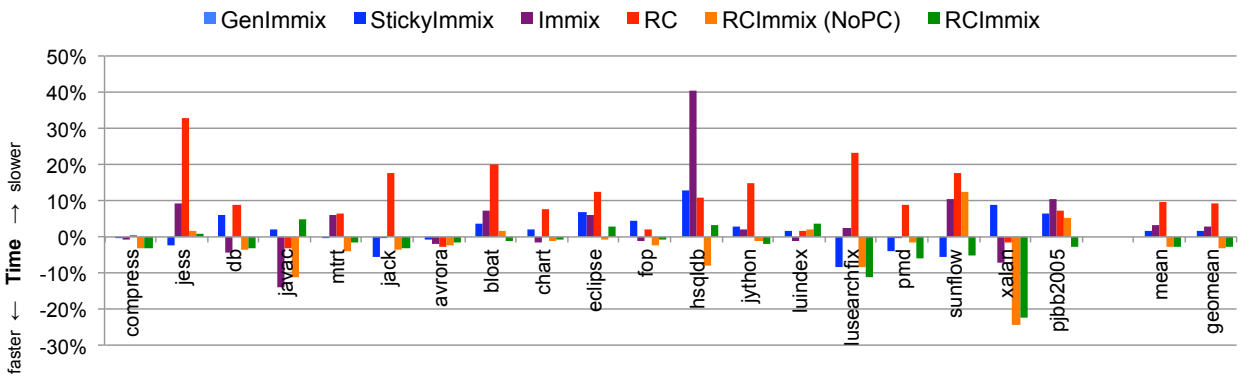
Table 7 and Figure 3 compare total time, mutator time, and garbage collection time of RC Immix and RC Immix without proactive copying ('no PC') against a number of collectors. The figure illustrates the data and the table includes raw performance as well as relative measurements of the same data. This analysis uses a moderate heap size of 2× the minimum in which all collectors can execute each benchmark. Production systems often use this heap size because it strikes a balance in the space-time tradeoff exposed by garbage collected languages between memory consumption and garbage collection overheads. We explore the space-time tradeoff in more detail in Section 5.2. In Figure 3(c) and 3(d), results are missing for some configurations on some benchmarks. In each of these cases, either the numerator or denominator or both performed no GC (see Table 7).

The table and figure compare six collectors.

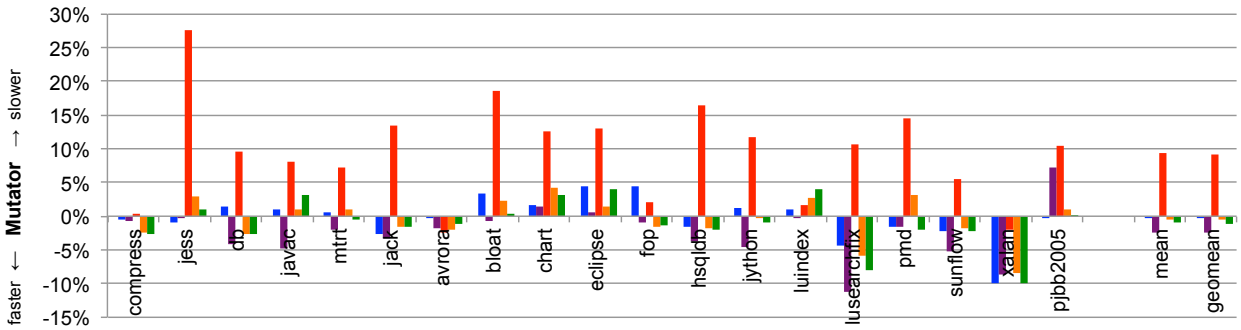
1. GenImmix, which uses a copying nursery and an Immix mature space.
2. Sticky Immix, which uses Immix with an in-place generational adaptation [6, 15].
3. Full heap Immix.
4. RC from Shahriyar et al.
5. RC Immix (no PC) which excludes proactive copying and performs well in moderate to large heaps due to very low collection times.
6. RC Immix as described in the previous section, which performs well at all heap sizes.

We normalize to GenImmix since it is the best performing in the literature [6] across all heap sizes and consequently is the default production collector in Jikes RVM. All of the collectors, except RC, defragment when there is an opportunity, i.e., there are partially filled blocks without fresh allocation and fragmentation is high, as described in Section 3.2.

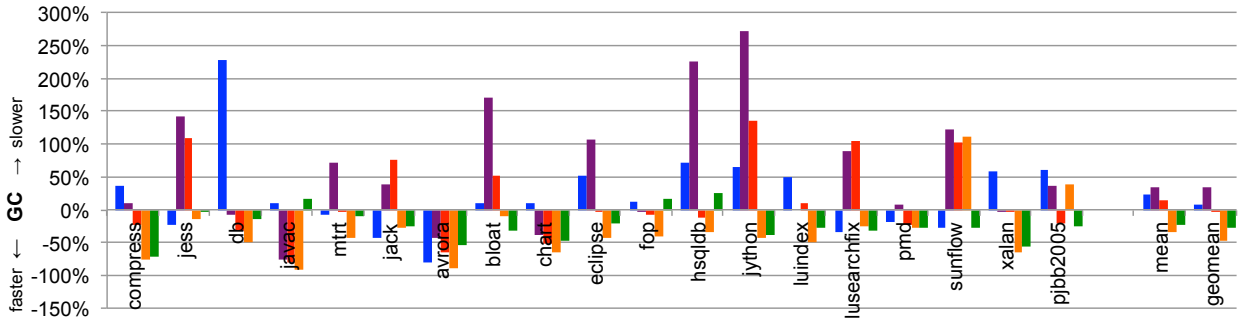
These results show that RC Immix outperforms the best performing garbage collector at this moderate heap size and completely eliminates the reference counting performance gap. The $\mathsf{time}_{\mathsf{gc}}$ show that, not surprisingly, Immix, the only full heap collector that does not exploit any generational behaviors, has the worst collector performance, degrading by on average 34%. Since garbage collection time is a relatively smaller influence on total time in a moderate heap, all but RC perform similarly on total time. At this heap size RC Immix performs the same as RC Immix (no PC), but its worse-case degradation is just 5% while its best case improvement is 22%. By comparison, RC Immix (no PC) has a worst case degradation of 12% and best case improvement of 24%. Table 7 and Figure 3(c) show that RC Immix (no PC) has the best garbage collection time, outperforming GenImmix by 48%. As we show later, RC Immix has an advantage over RC Immix (no PC) when memory is tight and fragmentation is a bigger issue.



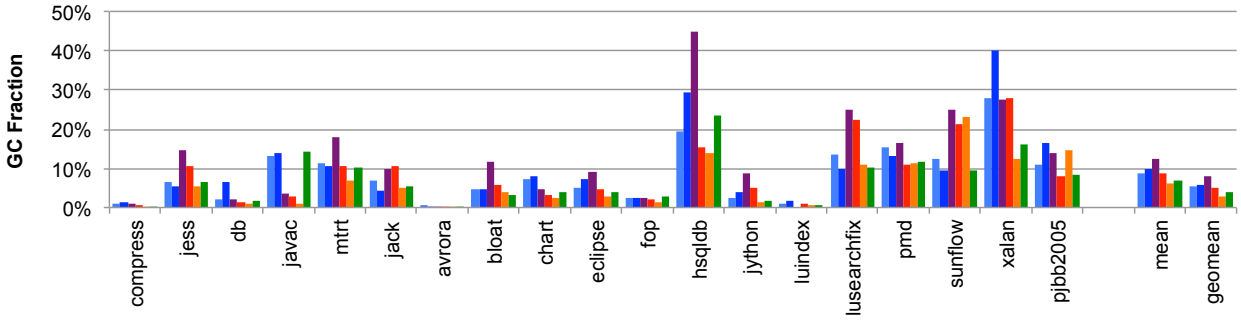
(a) Total slowdown compared to GenImmix



(b) Mutator slowdown compared to GenImmix



(c) GC slowdown compared to GenImmix



(d) Percentage of total execution time spent in GC

Figure 3. RC Immix performs 3% better than GenImmix, the highest performance generational collector, at a moderate heap size of 2 times the minimum. The first three graphs compare total, mutator, and GC slowdowns relative to GenImmix; lower is better. The fourth graph indicates the GC load seen by each configuration. RC Immix eliminates all the mutator time overheads of RC. Error bars are not shown, but 95% confidence intervals are given in Table 7.

Benchmark	GenImmix (milliseconds)			StickyImmix (Normalized to GenImmix)			Immix			RC			RC Immix (no PC)			RC Immix		
	time	$time_{mu}$	$time_{gc}$	time	$time_{mu}$	$time_{gc}$	time	$time_{mu}$	$time_{gc}$	time	$time_{mu}$	$time_{gc}$	time	$time_{mu}$	$time_{gc}$	time	$time_{mu}$	$time_{gc}$
compress	2256 ±0.2	2237 ±0.2	20 ±4.6	1.00 ±0.2	1.00 ±0.2	1.37 ±5.9	0.99 ±0.2	0.99 ±0.2	1.09 ±4.1	1.00 ±0.1	1.00 ±0.2	0.76 ±3.2	0.97 ±0.2	0.98 ±0.2	0.25 ±2.9	0.97 ±0.2	0.97 ±0.2	0.28 ±1.6
jess	485 ±0.7	453 ±0.7	32 ±4.3	0.98 ±0.6	0.99 ±0.6	0.77 ±3.2	1.09 ±0.8	1.00 ±0.8	2.42 ±8.0	1.33 ±1.1	1.28 ±1.2	2.08 ±7.4	1.02 ±0.9	1.03 ±1.0	0.85 ±6.7	1.01 ±0.7	1.01 ±0.6	0.98 ±6.8
db	1491 ±0.4	1460 ±0.4	31 ±7.1	1.06 ±0.5	1.01 ±0.5	3.29 ±17.8	0.96 ±1.0	0.96 ±1.0	0.92 ±5.6	1.09 ±0.8	1.10 ±0.8	0.68 ±4.0	0.96 ±0.9	0.97 ±0.8	0.51 ±7.0	0.97 ±0.7	0.97 ±0.8	0.85 ±8.0
javac	1048 ±0.7	911 ±0.4	137 ±4.5	1.02 ±0.6	1.01 ±0.3	1.10 ±4.6	0.86 ±0.4	0.95 ±0.3	0.25 ±0.9	0.97 ±0.5	1.08 ±0.3	0.20 ±0.7	0.89 ±0.5	1.01 ±0.3	0.08 ±1.5	1.05 ±2.3	1.03 ±0.5	1.17 ±15.2
mpegaudio	1406 ±0.1	1406 ±0.1	0 ±0.0	1.01 ±0.2	1.01 ±0.2	0.00 ±0.0	1.01 ±0.1	1.01 ±0.1	0.00 ±0.0	1.00 ±0.1	1.00 ±0.1	0.00 ±0.0	0.97 ±0.1	0.97 ±0.1	0.00 ±0.0	0.97 ±0.1	0.97 ±0.1	0.00 ±0.0
mtrt	340 ±3.5	302 ±3.8	38 ±2.8	1.00 ±3.7	1.01 ±4.2	0.92 ±7.1	1.06 ±2.6	0.98 ±2.7	1.72 ±5.7	1.06 ±2.6	1.07 ±2.9	1.00 ±3.9	0.96 ±3.6	1.01 ±4.2	0.58 ±3.0	0.98 ±3.3	0.99 ±3.4	0.89 ±7.0
jack	715 ±0.7	665 ±0.7	50 ±7.3	0.94 ±0.6	0.97 ±0.6	0.57 ±3.4	1.00 ±0.8	0.97 ±0.7	1.40 ±7.4	1.18 ±0.8	1.13 ±0.7	1.75 ±9.2	0.97 ±0.8	0.98 ±0.7	0.72 ±6.6	0.97 ±0.7	0.99 ±0.7	0.74 ±4.6
mean	**1056** ±0.9	**1005** ±0.9	**51** ±4.4															
geomean				**1.00**	**1.00**	**1.12**	**0.99**	**0.97**	**1.06**	**1.10**	**1.11**	**0.85**	**0.96**	**1.00**	**0.39**	**0.99**	**0.99**	**0.75**
avrora	3154 ±1.2	3134 ±1.2	20 ±9.6	0.99 ±1.3	1.00 ±1.3	0.21 ±1.7	0.98 ±1.1	0.98 ±1.1	0.58 ±5.1	0.97 ±1.3	0.98 ±1.3	0.35 ±2.6	0.98 ±1.2	0.98 ±1.2	0.12 ±1.2	0.98 ±1.1	0.99 ±1.1	0.46 ±16.4
bloat	3164 ±0.4	3018 ±0.5	145 ±1.7	1.04 ±0.5	1.03 ±0.6	1.09 ±2.1	1.07 ±0.8	0.99 ±0.8	2.71 ±6.4	1.20 ±0.5	1.19 ±0.6	1.51 ±2.6	1.02 ±0.8	1.02 ±0.7	0.90 ±4.0	0.99 ±0.5	1.00 ±0.5	0.68 ±3.1
chart	3750 ±0.2	3473 ±0.1	276 ±1.6	1.02 ±0.2	1.02 ±0.2	1.09 ±1.4	0.98 ±0.5	1.01 ±0.5	0.60 ±0.9	1.08 ±0.5	1.13 ±0.5	0.48 ±0.9	0.99 ±0.8	1.04 ±0.8	0.35 ±1.0	0.99 ±0.5	1.03 ±0.7	0.52 ±3.2
eclipse	16203 ±4.0	15382 ±4.2	821 ±1.1	1.07 ±5.7	1.04 ±5.9	1.51 ±1.5	1.06 ±13.1	1.01 ±8.5	2.06 ±170.2	1.12 ±5.7	1.13 ±6.1	0.99 ±1.0	0.99 ±4.9	1.02 ±5.2	0.57 ±0.7	1.03 ±5.2	1.04 ±5.5	0.79 ±4.5
fop	868 ±0.8	848 ±0.8	20 ±2.0	1.05 ±0.9	1.04 ±0.9	1.11 ±1.9	0.99 ±0.9	0.99 ±0.9	0.98 ±4.1	1.02 ±0.9	1.02 ±0.9	0.92 ±1.8	0.97 ±0.9	0.98 ±0.9	0.59 ±1.2	0.99 ±1.0	0.99 ±1.0	1.16 ±12.4
hsqldb	970 ±0.8	783 ±0.1	188 ±4.3	1.13 ±1.9	0.98 ±0.2	1.72 ±11.2	1.41 ±2.4	0.96 ±2.8	3.25 ±10.0	1.11 ±0.7	1.16 ±0.2	0.88 ±2.7	0.92 ±0.6	0.98 ±0.5	0.66 ±2.3	1.03 ±0.6	0.98 ±0.1	1.26 ±3.9
jython	3581 ±0.5	3493 ±0.5	88 ±1.8	1.03 ±0.5	1.01 ±0.5	1.66 ±2.8	1.02 ±0.4	0.95 ±0.4	3.71 ±9.1	1.15 ±0.5	1.12 ±0.5	2.36 ±3.4	0.99 ±0.4	1.00 ±0.4	0.58 ±4.2	0.98 ±0.6	0.99 ±0.6	0.61 ±1.5
luindex	626 ±0.3	620 ±0.3	7 ±4.4	1.02 ±0.3	1.01 ±0.3	1.50 ±6.8	0.99 ±0.3	1.00 ±0.3	0.00 ±0.0	1.02 ±0.3	1.02 ±0.3	1.10 ±4.7	1.02 ±0.3	1.03 ±0.3	0.50 ±2.9	1.04 ±0.4	1.04 ±0.4	0.73 ±5.1
lusearch	3154 ±0.3	2147 ±0.3	1007 ±0.9	1.01 ±0.7	0.77 ±0.5	1.52 ±2.4	0.91 ±0.4	0.72 ±0.5	1.30 ±1.1	1.12 ±0.7	0.89 ±0.6	1.61 ±1.4	0.66 ±0.4	0.75 ±0.6	0.46 ±0.5	0.66 ±0.5	0.75 ±0.8	0.46 ±0.5
lusearchfix	887 ±3.2	767 ±3.7	120 ±2.9	0.92 ±2.8	0.96 ±3.2	0.66 ±1.7	1.03 ±3.0	0.89 ±3.1	1.90 ±4.2	1.23 ±4.1	1.11 ±4.5	2.05 ±4.3	0.92 ±2.7	0.94 ±3.2	0.75 ±1.7	0.89 ±2.2	0.92 ±2.6	0.68 ±1.7
pmd	934 ±1.2	790 ±1.2	144 ±4.5	0.96 ±1.0	0.98 ±1.2	0.82 ±4.5	1.00 ±1.4	0.98 ±1.0	1.07 ±7.8	1.09 ±1.5	1.15 ±1.2	0.78 ±6.0	0.98 ±1.2	1.03 ±1.2	0.73 ±5.3	0.94 ±1.1	0.98 ±1.1	0.72 ±3.5
sunflow	2482 ±1.1	2175 ±1.3	307 ±1.5	0.95 ±1.1	0.98 ±1.2	0.72 ±1.4	1.11 ±1.1	0.95 ±1.1	2.23 ±3.8	1.18 ±1.1	1.06 ±1.1	2.03 ±2.9	1.12 ±1.3	0.98 ±1.1	2.12 ±5.5	0.95 ±1.0	0.98 ±1.2	0.73 ±2.5
xalan	1393 ±8.5	1008 ±11.6	385 ±1.1	1.09 ±6.7	0.90 ±7.5	1.58 ±3.6	0.93 ±11.1	0.91 ±10.5	0.97 ±22.9	0.98 ±6.0	0.98 ±8.1	0.99 ±1.9	0.76 ±5.8	0.92 ±9.1	0.34 ±0.6	0.78 ±4.8	0.90 ±7.7	0.45 ±1.2
mean	**3168** ±1.8	**2957** ±2.1	**210** ±2.9															
geomean				**1.02**	**1.00**	**1.01**	**1.04**	**0.97**	**0.00**	**1.09**	**1.08**	**1.05**	**0.97**	**0.99**	**0.56**	**0.96**	**0.99**	**0.70**
pjbb2005	3775 ±1.0	3363 ±1.1	412 ±2.1	1.07 ±1.0	1.00 ±1.1	1.61 ±3.7	1.11 ±20.4	1.07 ±23.7	1.37 ±8.1	1.07 ±1.1	1.11 ±1.2	0.80 ±4.7	1.05 ±1.2	1.01 ±1.3	1.40 ±4.5	0.97 ±1.3	1.00 ±1.3	0.74 ±6.5
min	340	302	7	0.92	0.90	0.21	0.86	0.89	0.00	0.97	0.98	0.20	0.76	0.92	0.08	0.78	0.90	0.28
max	16203	15382	821	1.13	1.04	3.29	1.41	1.07	3.71	1.33	1.28	2.36	1.12	1.04	2.12	1.05	1.04	1.26
mean	**2533** ±1.4	**2362** ±1.6	**171** ±3.4															
geomean				**1.02**	**1.00**	**1.07**	**1.03**	**0.98**	**1.34**	**1.09**	**1.09**	**0.97**	**0.97**	**1.00**	**0.52**	**0.97**	**0.99**	**0.72**

Table 7. RC Immix performs 3% better than GenImmix at a moderate heap size of 2× the minimum. We show at left total, mutator, and GC time for GenImmix in milliseconds and performance of RC, Immix, Sticky Immix, RC Immix (no PC), and RC Immix normalized to GenImmix. Lower is better. We grey-out and exclude from aggregates lusearch and mpegaudio because of their pathological behaviors, although both perform very well with our systems. The numbers in grey beneath the corresponding arithmetic mean report 95% confidence intervals, expressed as percentages.

Mutator	GenImmix	RC	RC Immix
Time	1.000	1.087	0.985
Instructions Retired	1.000	1.094	1.012
L1 Data Cache Misses	1.000	1.313	1.043

Table 8. Mutator performance counters show RC Immix solves the instruction overhead and poor locality problems in RC. Applications executing RC Immix compared with GenImmix in a moderate heap size of 2× the minimum execute the same number of retired instructions and see only a slightly higher L1 data cache miss rate. Comparing RC to RC Immix, RC Immix reduces miss rates by around 20%.

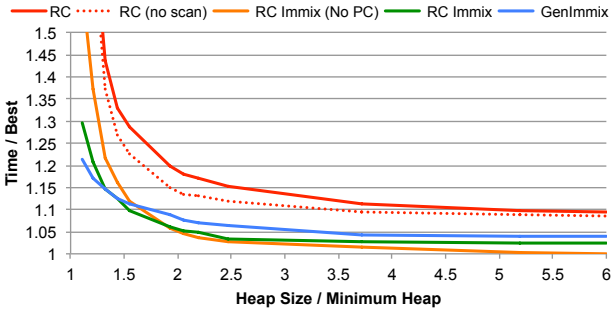
The time_{mu} columns of Table 7 and Figure 3(b) show that RC Immix matches or beats the Immix collectors with respect to mutator performance and improves significantly over RC in a moderate heap. The reasons that RC Immix improves over RC in total time stem directly from this improvement in mutator performance. RC mutator time is 9% worse than any other collector, as we reported in Table 2 and discussed in Section 2.1. RC Immix completely eliminates this gap in mutator performance.

Table 8 summarizes the reasons for RC Immix's improvement over RC by showing the number of mutator retired instructions and mutator L1 data cache misses for RC and RC Immix normalized to GenImmix. RC Immix solves the instruction overhead and poor locality problems in RC because by using a bump pointer, it wins twice.

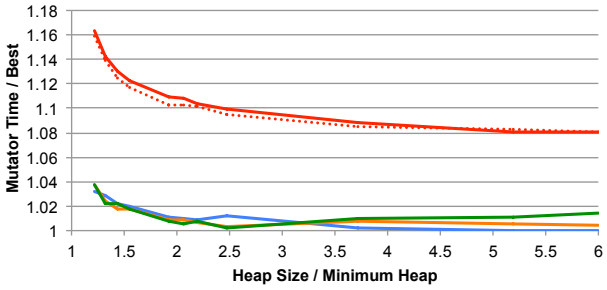
First, it gains the advantage of efficient zeroing of free memory in lines and blocks, rather than zeroing at the granularity of each object when it dies or is recycled in the free list (see Section 2.1 and Yang et al.'s measurements [33]). Second, it gains the advantage of contiguous allocation in memory of objects allocated together in time. This heap layout induces good cache behavior because objects allocated and used together occupy the same cache line, and because the bump pointer marches sequentially through memory, the hardware prefetcher correctly predicts the next line to fetch, so it is in cache when the program (via the memory allocator) accesses it. Yang et al.'s prefetching measurements quantify this effect [33]. Table 8 shows that compared to RC, RC Immix reduces cache misses by around 20% (1.043/1.313). GenImmix has slightly lower cache miss rates than RC Immix, which makes sense because it always allocates new objects contiguously (sequentially) whereas RC Immix sometimes allocates into partially full blocks and must skip over occupied lines.

5.2 Variable Heap Size Analysis

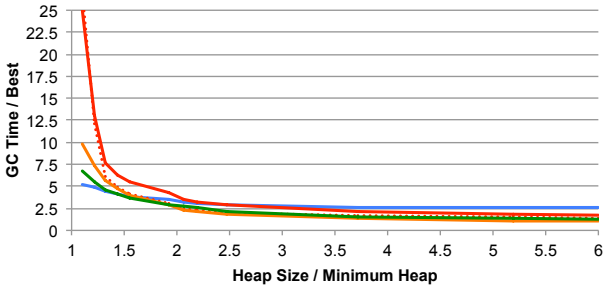
Figure 4 evaluates RC Immix performance as a function of available memory. Each of the three graphs varies heap size between 1× and 6× the minimum in which all collectors can execute the benchmark. The graphs plot total time, mutator time, and GC time as a geometric mean of the benchmarks, showing RC Immix, GenImmix, RC, RC with no boot image scanning (Section 3.3), and RC Immix with the no proactive copying. Figure 4(a) shows total time, and reveals that RC Immix dominates RC at all heap sizes, and consistently outperforms GenImmix at heap sizes above 1.4× the minimum. Figures 4(b) and 4(c) reveal the source of the behavior of RC Immix seen in Figure 4(a). Figure 4(b) reveals that the mutator performance of RC Immix is consistently good. This graph makes it clear that the underlying heap structure has a profound impact on mutator performance. Figure 4(c) shows that in GC time, RC Immix outperforms RC in tighter heaps, matches GenImmix at heap size 1.3× the minimum



(a) Total time



(b) Mutator time



(c) GC time

Figure 4. The performance of GenImmix, RC, RC Immix, and RC Immix with no proactive copying (No PC) as a function of heap size.

and outperforms GenImmix at heap sizes above 1.4× the minimum.

The faster degradation in RC Immix compared to GenImmix at the very smallest heap sizes is likely to be due to the more aggressive defragmenting effect of GenImmix's copying nursery. When a nursery collection occurs, those objects that survive are all copied into the mature space, which uses the Immix discipline, in the case of GenImmix. So while the surviving objects may be scattered throughout the nursery at the start of the collection, they will generally be contiguous after the nursery collection. GenImmix will tend to have a less fragmented mature space than RC Immix because the mature space is not intermingled with the fragmented young space, as it is in RC Immix and Sticky Immix. The result will be two-fold; both improved spatial locality and more importantly, reduced fragmentation which will substantially reduce GC load at small heap sizes, as seen in Figure 4(c). Furthermore, as hinted by the data cache miss rates for semispace in Table 1, the copying order of a copying collector will generally result in particularly good locality among the surviving objects.

5.3 Pinning

We conducted a simple experiment to explore the tradeoff associated with dedicating a header bit for pinning (see Section 3.1). While a pinning bit could be folded into the logged and forwarding bits, in this case we simply trade pinning functionality for reduced reference counting bits. In Jikes RVM, pinning can be utilized by the Java standard libraries to make IO more efficient, so although no Java application can exploit pinning directly, there is a potential performance benefit to providing pinning support.

Table 9 shows the result of a simple experiment with three configurations at three heap sizes. The performance numbers are normalized to GenImmix and represent the geometric mean of all benchmarks. In the first row, we have three reference counting bits and no pinning support, which is the default configuration used in this paper. Then we reduce the number of reference counting bits to two, *without* adding pinning. Finally we use two reference counting bits and add support for pinning. The results show that to the first approximation, the tradeoff is not significant, with the performance variations all being within 0.8% of each other. Although the variations are small, the numbers are intriguing. We see that at the 2× heap, the introduction of pinning improved total performance by around 0.5% when holding the reference counting bits constant. More interestingly, we see that the reduction in reference counting bits from three to two makes very little difference, perhaps even improving performance at 1.2× and 1.5×. This second result seems counter-intuitive. We surmise that the reason for this is that while the bulk of objects only need two bits to be correctly counted, many of the overflows may be attributable to objects that also overflow with three bits. The reduction in bits may thus be reducing the total number of increments and decrements performed without greatly reducing the efficacy of reclamation.

Bits Used		Heap Size		
count	pin	1.2×	1.5×	2×
3	0	1.030	0.998	0.978
2	0	1.022	0.991	0.979
2	1	1.023	0.991	0.974

Table 9. Performance sensitivity of RC Immix with pinning bit at 1.2, 1.5 and 2 times the minimum heap size, averaged over all benchmarks. Time is normalized relative to GenImmix. Lower is better.

5.4 Benchmark Analysis

Table 7 reveals that sunflow is significantly slower on RC Immix (no PC) than on GenImmix, whereas xalan and lusearch are significantly faster when using RC Immix. We now analyze these outlier results.

Sunflow Table 7 shows that sunflow is 12% slower in total time on RC Immix (no PC) than GenImmix, and that this slowdown is *entirely* due to a garbage collection slowdown of 2.12×. The source of this problem appears to be high fragmentation among surviving young objects in sunflow. It was this observation that encouraged us to explore proactive defragmentation, and this benchmark shows that the strategy is hugely effective, as RC Immix improves over GenImmix by 5%. sunflow has a high allocation rate [33], and our observation that GenImmix does a large number of nursery collections, but no mature space collections at 2× minimum heap size confirms this behavior. RC Immix (no PC) does a large number of collections, many of which are defragmenting cycle collections, and yet sunflow has few cycles [27]. Furthermore, Table 4 shows that although the line survival rate for sunflow is 5%, the block survival rate is a remarkable 99%. This indicates that surviving objects are scattered in the heap generating fragmentation, thus Immix blocks are being kept alive unnecessarily. We also established empirically that sunflow's performance degraded substantially if the standard defragmentation heuristic was made less aggressive.

Xalan Both RC Immix (no PC) and RC Immix perform very well on xalan, principally because they have lower GC time than GenImmix. RC Immix (no PC) has 66% lower GC time than GenImmix and RC Immix has 55% lower GC time than GenImmix. xalan has a large amount of medium lifetime objects, which can only be recovered by a full heap collection with GenImmix, but are recovered in a timely way in RC Immix.

Lusearch RC Immix performs much better on lusearch than GenImmix. In fact GenImmix has substantially worse mutator time than any other system. This result is due to the bug in lusearch that causes the allocation of a very large

number of medium sized objects (Section 4), leading GenImmix to perform over 800 nursery collections, destroying mutator locality. The allocation pathology of lusearch is established and is the reason why we use lusearch-fix in our results, exclude lusearch from all of our aggregate (mean and geomean) results, and leave it greyed out in Table 7. If we were to include lusearch in our aggregate results then both RC Immix (no PC) and RC Immix would be 5% faster in geomean than GenImmix.

5.5 Further Analysis and Opportunities

We have explored three further opportunities for improving the performance of RC Immix, namely reference level coalescing, conservative stack scan, and root coalescing.

Reference Level Coalescing When Levanoni and Petrank first described coalescing of reference counts, they described it in terms of remembering the address and value of each *reference* when it was first mutated [21]. However, in practice it is easier to remember the address and contents of each *object* when the first of its reference fields is mutated [22]. In the first case, the collector compares the GC-time value of the reference with the remembered value and decrements the count for the object referred to by the remembered reference and increments the count for the object referred to by the latest value of the reference. With *object* level coalescing, each reference within the object is remembered and compared. The implementation challenge is due to the need to only remember each reference once, and therefore efficiently record somewhere that a given reference had been remembered. Using a bit in the object's header makes it easy to do coalescing at an object granularity. Both RC and RC Immix use *object* level coalescing.

As part of this work, we implemented *reference* level coalescing. We did this by stealing a high order bit within each reference to record whether that reference had been remembered. We then map two versions of each page to a single physical page (each one corresponding to the two possible states of the high order bit). We must also modify the JVM's object equality tests to ensure that the stolen bit is ignored in any equality test. We were disappointed to find that despite the low overhead bit stealing approach we devised, we saw no performance advantage in using reference level coalescing. Indeed, we observed a small slowdown. We investigated and noticed that reference level coalescing places a small but uniform overhead on each pointer mutation, but the potential benefit for the optimization is dominated by the young object optimizations implemented in RC and RC Immix. As a result, we use object level coalescing in RC Immix.

Conservative Stack Scan One of the explanations for the continued use of naïve reference counting rather than deferred reference counting is that deferred reference counting requires an enumeration of roots [16], which is challenging to implement correctly. To precisely enumerate roots requires implementing stack maps. We note that a conservative stack scan could only introduce false positives, and therefore could never lead to an incorrect decrement, and thus reclamation of a live object. We therefore believe that RC Immix could be implemented with conservative stack scans, circumventing a major barrier to the use of high performance reference counting. We plan to explore this in future work.

Root Elision A key advantage of reference counting over generational collection is that it continuously collects mature objects. The benefits are borne out by the improvements we see in xalan, which has many medium lived objects. These objects are promptly reclaimed by RC and RC Immix, but are not reclaimed by a generational collector until a full heap collection occurs. However, this timely collection of mature objects does not come for free. Unlike a nursery collection in a generational collector, a reference counting collector must enumerate all roots, including all pointers from the stacks and all pointers from globals (statics). We realized that it may be possible to greatly reduce the workload of enumerating roots by selectively enumerating only those roots that have changed since the last GC. In the case of globals/statics this could be achieved either by a write barrier or by keeping a shadow set of globals. We note that the latter may be feasible because the amount of space consumed by global pointers is typically very low. In the case of the stack, we could utilize a return barrier [34] to only scan the parts of the stack that have changed since the last GC. We plan to explore this in future work.

6. Conclusion

In the garbage collection literature, two fundamental algorithms identify dead objects. Reference counting identifies them directly and tracing identifies them implicitly. Despite its intrinsic advantages, such as promptness of recovery and dependence only on local rather than global state, reference counting did not deliver high performance and it suffered from incompleteness due to cycles. Recent advances by Shahriyar et al. closed, but did not eliminate this performance gap.

This paper identified heap organization as the principal source of this gap. In the literature, allocators use three heap organizations to place objects in memory: free lists, contiguous, and regions. Until this paper, reference counting always used a free list because it offered a constant time operation to reclaim each dead object. Unfortunately, optimizing for reclamation time neglects the more essential performance requirement of cache locality on modern systems. We show that indeed RC in a free list heap suffers poor locality compared to contiguous and hierarchical memory organizations. Unfortunately, the contiguous heap organization and freeing at an object granularity are fundamentally incompatible. Fortunately, the region heap organization and reference counting are compatible.

We describe the design and implementation of a new hybrid RC Immix collector. The key design contributions of

our work are an algorithm for performing per-line live object counts and the integration of proactive and reactive opportunistic copying. We show how to copy new objects proactively to mitigate fragmentation and improve locality. We further show how to combine reactive defragmentation with backup cycle detection. The key engineering contribution of our work is how to use limited header bits efficiently, serving triple duties for reference counting, backup cycle collection with tracing, and opportunistic copying.

Looking forward, we believe that RC Immix offers a new direction for both high performance throughput collectors and soft real-time collectors because of its ability to provide incremental reclamation with high throughput.

Acknowledgements

We thank Daniel Frampton for his contributions to the RC collector and Bertrand Maher and James Bornholt for their comments on earlier drafts of this paper.

References

[1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. J. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes RVM Project: Building an open source research community. *IBM System Journal*, 44(2):399–418, 2005. doi: 10.1147/sj.442.0399.

[2] D. F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In *European Conference on Object-Oriented Programming, Budapest, Hungary, June 18 - 22, 2001*, pages 207–235. LNCS, 2001. ISBN 3-540-42206-4. doi: 10.1007/3-540-45337-7_12.

[3] D. F. Bacon, C. R. Attanasio, H. B. Lee, V. T. Rajan, and S. Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *ACM Conference on Programming Language Design and Implementation, PLDI'01, Snowbird, UT, USA, June 2001*, pages 92–103. ACM, 2001. doi: 10.1145/378795.378819.

[4] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In *ACM Conference on Programming Language Design and Implementation, PLDI'01, Snowbird, UT, USA, June 2001*, pages 114–124. ACM, 2001. doi: 10.1145/378795.378821.

[5] S. M. Blackburn and K. S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications, OOPSLA'03, Anaheim, CA, USA, Oct, 2003*, pages 344–358. ACM, 2003. doi: 10.1145/949305.949336.

[6] S. M. Blackburn and K. S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator locality. In *ACM Conference on Programming Language Design and Implementation, PLDI'08, Tucson, AZ, USA, June 2008*, pages 22–32. ACM, 2008. doi: 10.1145/1379022.1375586.

[7] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *The 26th International Conference on Software Engineering, ICSE'04, Edinburgh, Scotland, 2004*, pages 137–146. ACM/IEEE, 2004.

[8] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *SIGMETRICS – Performance 2004, Joint International Conference on Measurement and Modeling of Computer Systems, New York, NY, USA, June 12–16, 2004*, pages 25–36. ACM, 2004. doi: 10.1145/1005686.1005693.

[9] S. M. Blackburn, M. Hirzel, R. Garner, and D. Stefanović. pjbb2005: The pseudojbb benchmark, 2005. URL http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005.

[10] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications, OOPSLA'06, Portland, OR, USA, Oct, 2006*, pages 169–190. ACM, 2006. doi: 10.1145/1167473.1167488.

[11] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffman, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century. *Commun. ACM*, 51(8):83–89, Aug. 2008.

[12] T. Cao, , S. M. Blackburn, T. Gao, and K. S. McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *The 39th International Conference on Computer Architecture, ISCA'12, Portland, OR, June, 2012*, pages 225–236. ACM/IEEE, 2012. doi: 10.1145/2366231.2337185.

[13] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, Nov. 1970. doi: 10.1145/362790.362798.

[14] G. E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, December 1960. doi: 10.1145/367487.367501.

[15] A. Demmers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker. Combining generational and conservative garbage collection: framework and implementations. In *ACM Symposium on the Principles of Programming Languages, POPL'90, San Francisco, CA, USA*, pages 261–269. ACM, 1990. doi: 10.1145/96709.96735.

[16] L. P. Deutsch and D. G. Bobrow. An efficient, incremental, automatic garbage collector. *Commun. ACM*, 19(9):522–526, September 1976. doi: 10.1145/360336.360345.

[17] Y. Feng and E. D. Berger. A locality-improving dynamic memory allocator. In *Proceedings of the 2005 Workshop on Memory System Performance*, pages 68–77, 2005. doi: 10.1145/1111583.1111594.

[18] D. Frampton. *Garbage Collection and the Case for High-level Low-level Programming*. PhD thesis, Australian National University, June 2010. URL http://cs.anu.edu.au/~Daniel.Frampton/DanielFrampton_Thesis_Jun2010.pdf.

[19] I. Jibaja, S. M. Blackburn, M. R. Haghighat, and K. S. McKinley. Deferred gratification: Engineering for high performance garbage collection from the get go. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC 2011), San Jose, CA, June 5, 2011*. ACM, 2011. doi: 10.1145/1988915.1988930.

[20] R. E. Jones, A. Hosking, and J. E. B. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman and Hall/CRC Applied Algorithms and Data Structures Series, USA, 2011. URL http://gchandbook.org/.

[21] Y. Levanoni and E. Petrank. An on-the-fly reference counting garbage collector for Java. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications, OOPSLA'01, Tampa,*

FL, USA, Oct, 2001, pages 367–380. ACM, 2001. doi: 10.1145/504282.504309.

[22] Y. Levanoni and E. Petrank. An on-the-fly reference-counting garbage collector for Java. *ACM Trans. Prog. Lang. Syst.*, 28(1):1–69, January 2006. doi: 10.1145/1111596.1111597.

[23] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, June 1983. doi: 10.1145/358141.358147.

[24] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, April 1960. doi: 10.1145/367177.367199.

[25] H. Paz, E. Petrank, and S. M. Blackburn. Age-oriented concurrent garbage collection. In *Compiler Construction*, volume 3443 of *Lecture Notes in Computer Science*, pages 121–136. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-25411-9. doi: 10.1007/978-3-540-31985-6_9.

[26] Y. Seeley. JIRA issue LUCENE-1800: QueryParser should use reusable token streams, 2009. URL https://issues.apache.org/jira/browse/LUCENE-1800.

[27] R. Shahriyar, S. M. Blackburn, and D. Frampton. Down for the count? Getting reference counting back in the ring. In *Proceedings of the 11th International Symposium on Memory Management, ISMM 2012, Beijing, China, June 15 - 16, 2012*. ACM, 2012. doi: 10.1145/2258996.2259008.

[28] SPEC. *SPECjvm98, Release 1.03*. Standard Performance Evaluation Corporation, Mar. 1999. URL http://www.spec.org/jvm98.

[29] SPEC. *SPECjbb2005 (Java Server Benchmark), Release 1.07*. Standard Performance Evaluation Corporation, 2006. URL http://www.spec.org/jbb2005.

[30] D. Ungar. Generation Scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments, SDE 1, 1984*, pages 157–167. ACM, 1984. ISBN 0-89791-131-8. doi: 10.1145/800020.808261.

[31] J. Weizenbaum. Recovery of reentrant list structures in Lisp. *Commun. ACM*, 12(7):370–372, July 1969. doi: 10.1145/363156.363159.

[32] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *IWMM'95, Proceedings of the International Workshop on Memory Management, IWMM, Kinross, Scotland, UK, Sep 27-29, 1995*, volume 986 of *Lecture Notes in Computer Science*, pages 1–116. Springer Berlin Heidelberg, 2005. doi: 10.1007/3-540-60368-9_19.

[33] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley. Why nothing matters: The impact of zeroing. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications, OOPSLA'11, Portland, Oregon, USA, Oct, 2011*, pages 307–324. ACM, 2011. doi: 10.1145/2048066.2048092.

[34] T. Yuasa, Y. Nakagawa, T. Komiya, and M. Yasugi. Return barrier. In *Proceedings of the International Lisp Conference*, 2002.