Fast Conservative Garbage Collection

Rifat Shahriyar
Australian National University
Rifat.Shahriyar@anu.edu.au

Stephen M. Blackburn
Australian National University
Steve.Blackburn@anu.edu.au

Kathryn S. McKinley
Microsoft Research
mckinley@microsoft.com

Abstract

Garbage collectors are exact or conservative. An *exact* collector identifies all references precisely and may move referents and update references, whereas a *conservative* collector treats one or more of stack, register, and heap references as ambiguous. *Ambiguous references* constrain collectors in two ways. (1) Since they may be pointers, the collectors must retain referents. (2) Since they may be values, the collectors cannot modify them, pinning their referents.

We explore *conservative* collectors for managed languages, with ambiguous stacks and registers. We show that for Java benchmarks they retain and pin remarkably few heap objects: $<$0.01% are falsely retained and 0.03% are pinned. The larger effect is collector design. Prior conservative collectors (1) use mark-sweep and unnecessarily forgo moving all objects, or (2) use mostly copying and pin entire pages. Compared to generational collection, overheads are substantial: 12% and 45% respectively. We introduce high performance conservative Immix and reference counting (RC). Immix is a mark-region collector with fine *line*-grain pinning and *opportunistic* copying of unambiguous referents. Deferred RC simply needs an object map to deliver the first conservative RC. We implement six exact collectors and their conservative counterparts. Conservative Immix and RC come within 2 to 3% of their exact counterparts. In particular, conservative RC Immix is slightly *faster* than a well-tuned exact generational collector. These findings show that for managed languages, conservative collection is compatible with high performance.

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

Keywords Conservative, Reference Counting, Immix, Mark-Region

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
OOPSLA '14, October 20–24, 2014, Portland, OR, USA.
Copyright © 2014 ACM 978-1-4503-2585-1/14/10…$15.00.
http://dx.doi.org/10.1145/2660193.2660198

1. Introduction

Language semantics and compiler implementations determine whether memory managers may implement *exact* or *conservative* garbage collection. Exact collectors identify all references and may move objects and redirect references transparently to applications. Conservative collectors must reason about *ambiguous references*, constraining them in two ways. (1) Because ambiguous references may be pointers, *the collector must conservatively retain referents*. (2) Because ambiguous references may be values, *the collector must not change them* and cannot move (must pin) the referent.

Languages such as C and C++ are not memory safe: programs may store and manipulate pointers directly. Consequently, their compilers cannot prove whether any value is a pointer or not, which forces their collectors to be conservative and non-moving. Managed languages, such as Java, C#, Python, PHP, JavaScript, and safe C variants, have a choice between exact and conservative collection. In principle, a conservative collector for managed languages may treat stacks, registers, heap, and other references conservatively. In practice, the type system easily identifies heap references exactly. However, many systems for JavaScript, PHP, Objective C, and other languages treat ambiguous references in stacks and registers conservatively.

This paper explores conservative collectors with ambiguous stacks and registers. We first show that the direct consequences of these ambiguous references on *excess retention* and *pinning* are surprisingly low. Using a Java Virtual Machine and 18 Java benchmarks, conservative roots falsely retain less than 0.01% of objects and pin less than 0.03%. However, conservative constraints have had a large indirect cost by how they shaped garbage collection algorithms.

Many widely used managed systems implement collectors that are conservative with respect to stacks and registers. Microsoft's Chakra JavaScript VM implements a conservative mark-sweep Boehm, Demers, Weiser style (BDW) collector [15, 19]. This non-moving free-list collector was originally proposed for C, but some managed runtimes use it directly and many others have adapted it. Apple's WebKit JavaScript VM implements a Mostly Copying Conservative (MCC) collector, also called a Bartlett-style collec-

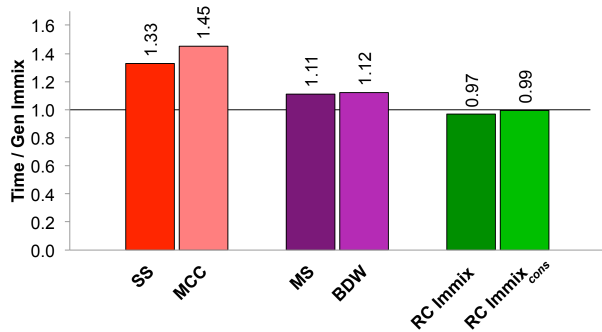


Figure 1. Performance of exact semi-space (SS), conservative MCC, exact mark-sweep (MS), conservative BDW, exact RC Immix, and conservative RC Immix$_{cons}$ normalized to exact Gen Immix at a moderate heap size. Lower is better. Prior conservative collectors sacrifice performance. RC Immix$_{cons}$ performs similarly to Gen Immix and RC Immix, the best exact collectors.

tor [5, 8, 32, 37]. MCC divides memory into pages, evacuates live objects onto empty pages, and pins entire pages that contain targets of ambiguous references. These systems are purposefully sacrificing proven performance benefits of exact generational collectors [9, 11]. To quantify this performance cost, we implement and compare them to a copying generational collection (Gen Immix), the production collector in Jikes RVM. Figure 1 summarizes our results, plotting geometric mean of total (mutator + collector) time on 18 Java Benchmarks. The total time penalty is 12% for BDW mark-sweep and 45% for MCC.

These systems purposefully chose conservative over exact collection to reduce their compiler burden. Exactly identifying root references requires a strict compiler discipline that constructs and maintains *stack maps* that precisely report every word on the stack and in registers that holds a live reference for every point in execution where a collection may occur. This process is a formidable implementation task that requires tracking every reference in all optimizations and compiler intermediate forms, and it restricts some optimizations, such as code motion [1, 21, 24].

An alternative tactic for limiting the compiler burden is naive reference counting, which is used by Objective-C, Perl, Delphi, PHP, and Swift. These collectors never examine the stack because the compiler or interpreter simply inserts increments and decrements when the program changes an object reference. Previous measurements quantify the penalty of deferred reference counting, which eliminates increments and decrements on local variables and is thus faster than naive reference counting, as 40% compared to a copying generational collector [4, 20, 30, 31]. All of these language implementations either forbid cycles, leak cycles, or perform costly trial deletion [22, 30]. Naive reference counting imposes an even larger performance sacrifice.

This paper shows how to combine high performance with the engineering advantages of conservative collection. We introduce conservative Immix, conservative deferred reference counting, and combine them in conservative RC Immix. The result is slightly faster than a well tuned generational collector. We make surprisingly simple modifications to Immix and reference counting. As far as we are aware, this collector is the first conservative reference counter.

The Immix mark-region collector manages memory hierarchically in coarse-grain blocks divided into fine-grain lines [9]. Immix allocates contiguously into empty blocks and lines. Tracing identifies live objects and lines, and mixes marking and copying in the same pass. Whereas generational collectors copy all nursery objects, Immix copies *opportunistically* into lines and blocks that are free when collection begins. When the collector exhausts memory or encounters objects that it cannot move, it simply marks them and their lines live. At the end of collection, it recycles free blocks and lines. To make Immix conservative, we simply start the collection by enumerating the ambiguous roots in stacks and registers, marking their referents live and pinned; the collector never moves them. To ensure that referents are valid, we introduce an *object map*, which identifies objects live at the last collection or allocated since then. Conservative Immix collectors thus limit pinning overheads to the line granularity and maximize copying and locality benefits.

A similarly surprisingly simple change makes deferred reference counting conservative. We start collection by enumerating the ambiguous roots, validating them with the object map, and retaining any object referenced by an ambiguous root, even if its reference count falls to zero.

We implement six conservative collectors and compare to their exact counterparts in a Java VM. We implement prior work — conservative BDW and MCC, and their exact mark-sweep and semi-space counterparts. We design and implement four new conservative collectors: RC$_{cons}$, Immix$_{cons}$, Sticky Immix$_{cons}$, and RC Immix$_{cons}$. Conservative roots degrade all collectors by less than 3% compared to their exact counterparts, except for MCC which degrades over semi-space by 9%. Figure 1 shows that RC Immix$_{cons}$ improves total performance over BDW by 13% on average and up to 41%. RC Immix$_{cons}$ delivers excellent performance, competitive with the fastest exact generational collectors.

In summary, the contributions of this paper are as follows.

1. We examine conservative garbage collection for managed languages.
2. We show that the direct cost of conservative roots is small for Java workloads: excess retention is less than 0.01% and pinned objects are just 0.03% of the heap.
3. We design, implement, and evaluate new and prior conservative collectors and compare to exact collectors.
4. We introduce an optimized object map that filters ambiguous roots to valid objects.

5. We show that Immix lines and opportunistic copying are well matched to conservative garbage collection needs.
6. We extend deferral using the object map and implement the first conservative reference counting collector.
7. We show that RC Immix$_{cons}$ is the first conservative collector to match the performance of exact generational copying collection.

These findings demonstrate that high performance garbage collection is possible for managed languages, whether or not they invest in engineering exact collection.

2. Background and Related Work

This section reviews the mechanisms and requirements for conservative and exact garbage collectors on which we build.

Conservative collectors have thus far been tracing. A tracing garbage collector performs a transitive closure over the object reachability graph, starting with the *roots*, which are references into the heap held by the runtime, including stacks, registers, and static (global) variables [28]. An *exact* garbage collector precisely identifies root references and references between objects while a *conservative* collector must handle *ambiguous references* — values that may or may not be valid pointers. Three broad approaches exist to enumerate references: a) compiler supported exact, b) uncooperative exact, and c) conservative.

2.1 Exact Garbage Collection

Exact garbage collection for managed languages requires cooperation from the compiler and language runtime. The language runtime must identify references from roots and references within the heap (between heap objects). The type system identifies references in heap objects at allocation time. The runtime must dynamically examine the stacks, registers, statics, and any other source of references into the heap to identify root references. Dynamically tracking roots is more challenging if the runtime uses aggressive optimizations, for example, with a just-in-time (JIT) compiler.

Compiler-supported exact The compiler for an exact collector generates and maintains *stack maps* — data structures that, for every point in execution where collection may ensue, report the precise location of every live reference stored by local variables in stacks, or registers. Engineering accurate stack maps is challenging [1, 24]. Precise pointer tracking burdens the compiler with significant bookkeeping in optimizations and intermediate representations [21], and inhibits optimizations, such as code motion.

Nonetheless, many mature high performance managed runtimes use exact root enumeration, such as .NET for C#, and HotSpot, Jikes RVM, and J9 VMs for Java. The interpreter and/or JIT compilers in these systems maintain precise stack maps for every point in execution where a garbage collection may occur. The garbage collector walks each thread's stack frame-by-frame, consulting pre-generated stack maps to enumerate the location of all live references. Because these systems are exact, the collector is free to move objects and redirect program references. All of these systems implement exact copying generational collectors, which are the best performing collectors [9, 11, 31].

Uncooperative exact Exact uncooperative systems are also in use for strongly typed languages that are implemented with C as an intermediate language [7, 23, 27]. In principle, strong typing allows precise pointer identification, but a stock C compiler loses that type precision. Instead, these runtimes dynamically maintains a *shadow stack,* a separate data structure for each frame that identifies precisely the set of live object references. This approach avoids conservatism and the engineering cost of introducing precise pointer tracking and stack map generation within the C compiler, but it does so at the cost of explicitly, dynamically maintaining a shadow stack with the set of live references. This cost is significant because stack operations are frequent and it must perform shadow operations for every stack operation involving references.

2.2 Conservative Garbage Collection

A conservative collector must reason about *ambiguous references* — values that may or may not be valid pointers.

Ambiguous references To ensure correctness, a conservative collector must retain and not move the referent of an ambiguous reference and must retain any transitively reachable objects. The collector must retain the referent in case the ambiguous reference is a valid pointer. The collector must not change the ambiguous reference in case it is a value, not a pointer. In addition, it must carefully manage object metadata. For example, if the collector stores metadata on liveness in an object header, it must guarantee that the referent is a valid object before updating its metadata in order to guarantee that it does not corrupt visible program state.

Ambiguous references thus constrain conservative collectors in the following ways.

- Because ambiguous references may be *valid pointers*, the collector must retain their referents and transitively reachable objects.
- Because ambiguous references may be *values*, the collector may not modify them, pinning the referents.
- In order to avoid corrupting the heap, the collector must guarantee that referents are valid objects before it updates per-object metadata.

The above constraints lead to three consequences. Conservative collectors a) incur excess retention due to their liveness assumptions; b) cannot move (must pin) objects that are targets of ambiguous references; and c) must either filter ambiguous references to assure the validity of the target, or maintain metadata in side structures.

Excess retention Constraint a) leads to a direct space overhead (*excess retention*), because the collector will conservatively mark a dead object as live, as well as all of its transitively reachable descendants. We measure excess retention in a Java VM and show that it is low in Section 5.

Pinning Pinning leads to fragmentation and constrains algorithm design. Because reference types in unsafe languages are ambiguous, *all* references, regardless of whether in the runtime or heap, are ambiguous, and therefore all objects must be pinned. For safe languages, such as Java, C#, JavaScript, PHP, Python, and safe C and C++ variants, the only references that are not well typed are those whose type the compiler does not track, such as local variables in stacks and registers. Therefore, conservative collectors for these languages may move objects that are only the target of well typed references. They will need only to pin objects that are the target of ambiguous roots. Below we describe how pinning influences the design of the two prior classes of conservative collection algorithms in more detail.

Filtering Filtering ambiguous references eliminates those that do not point to viable objects, but increases processing costs for each reference. There are three sources of spurious pointers on a conservatively examined stack. a) Some program value in the stack may by chance correspond to a heap address. b) The compiler may temporarily use pointers, including interior pointers, that are not object references. c) The stack discipline invariably leads to values, including valid references, remaining on the stack well beyond their live range. The collector therefore filters ambiguous references, discarding those that do not point to *valid* objects. Valid objects were either determined live at the last collection or allocated since then. The particular filtering mechanism depends on the collector and we describe them below.

Non-Moving Conservative Collectors The most mature, widely used, and adopted conservative garbage collector is the Boehm-Demers-Weiser (BDW) collector [15]. The BDW collector is a non-moving collector that uses a free-list allocator and a mark-sweep trace to reclaim unused objects. The allocator maintains separate free-lists for each size of object. For unsafe C, the collector is conservative with respect to roots and pointers within the heap. The BDW collector includes a non-moving generational configuration [19].

BDW filters ambiguous references by testing whether they point to a valid, allocated, free-list cell. It first identifies the free-list block into which the reference points and then it establishes the size class for the cells within that block. It tests whether the reference points to the start of a valid cell for that particular block, and finally tests whether that cell is allocated (live and not free). Only references to the start of live cells are treated as valid ambiguous references.

The BDW collector can be configured to exploit type information when available, so that it is conservative only with respect to stacks and registers, and precise with respect to the heap and other roots. These qualities make the BDW collector suitable for other language implementations, particularly initial implementations of managed languages that use C as an intermediate language.

The problem with a non-moving free-list in a managed language setting is that mutator time suffers. Allocating objects by size spreads contemporaneously allocated objects out in memory and induces more cache misses than contiguous bump pointer allocators, such as those used by copying collectors and the Immix collector we use here [9, 11]. Section 6 compares an exact mark-sweep collector and a BDW-style conservative mark-sweep collector, in which only the roots are conservative. Making mark-sweep conservative with respect to stacks and registers only adds 1% overhead. However comparing with copying generational, the design choice of a non-moving free-list imposes a 12% penalty due to degradations in mutator time (Table 4).

For languages with object type precision, a non-moving collector design is overly restrictive since, as we show, most heap objects will not be the target of ambiguous references.

Mostly Copying Collectors Bartlett [8] first described a *mostly copying* conservative collector for memory safe languages, a design that has remained popular [5, 25, 32, 37]. Bartlett's collector is a classic semi-space collector that uses bump pointer allocation. A semi-space collector divides the heap into to-space and from-space. Objects are allocated into to-space. At the start of each collection, the collector flips the spaces.The collector then copies reachable objects out of from-space into the new to-space. At the end of a collection, allocation resumes into the to-space. Bartlett's collector has two twists over the classic semi-space design. First, the to-space and from-spaces are not adjacent halves of a contiguous address space, but instead they are logical spaces comprised of linked lists of discontiguous pages. Second, at the start of each collection, the collector promotes each page referenced by an ambiguous root — the collector unlinks the referent page from the from-space linked list and puts it on the to-space linked list. Thus ambiguously referenced objects are logically promoted into to-space rather than copied. The promoted pages serve as the roots for a final copying collection phase that completes a transitive closure over the heap. Bartlett assumes that all objects on promoted pages are live, exacerbating excess retention.

Attardi et al. [5] improve over Bartlett's MCC by only using the ambiguous referents as roots, rather than using all objects on the ambiguously referenced page as roots. They introduce a *live map*, and during the initial promotion phase they remember the target of each ambiguous pointer. Then during the final copying phase, when scanning promoted pages for roots, they use the live map to select only live objects as sources, significantly reducing excess retention.

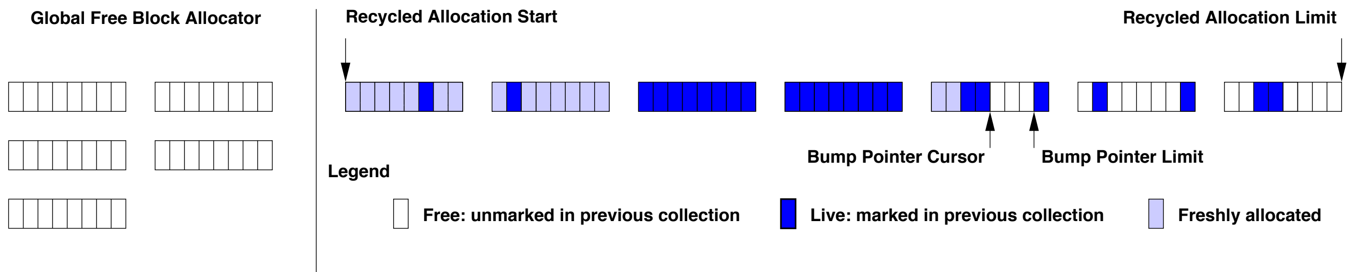


Figure 2. Immix Heap Organization [31]

Many mostly copying collectors, including Apple's WebKit JavaScript VM, use segregated fits free-lists and, like BDW, introspect within the page to determine whether an ambiguous pointer references a valid allocated cell [8, 25, 37]. Hosking's [25] mostly copying collector is concurrent and supports language-level internal references, which requires a broadening of the definition of validity to include ambiguous references that point within valid objects.

Smith and Morrisett [32] take a different approach to filtering ambiguous roots. Since they use a bump pointer rather than a free list, objects are tightly packed, not uniformly spaced. To determine whether an ambiguous reference points to a valid object, they scan the pages from the start. They traverse the contiguous objects on a page introspecting length and skipping to the next object, until the ambiguous reference is reached or passed. If the ambiguous reference matches the start of an object, it is treated as valid, otherwise it is discarded. This mechanism is not efficient. We introduce a low overhead object map to efficiently combine conservatism and bump pointer allocation.

Mostly copying collectors suffer a number of drawbacks. Because pages are not contiguous, objects may not span pages, which leads to wasted space on each page, known as internal fragmentation. Because any page containing a referent of an ambiguous reference is pinned and cannot be used by the allocator, more space is wasted. Section 6 and Figure 6 compare an exact semi-space collector with a Mostly Copying Conservative (MCC) collector with the refinement that we use an *object map* to identify valid objects at the beginning of a collection in a copying space. Conservatism in MCC adds 9% overhead compared to exact semi-space, and 45% compared to copying generational. The performance limitations of conservative mostly copying and non-moving collectors motivate exploring alternative designs.

2.3 Exact Mark-Region and Reference Counting

To create high performance conservative collectors, we make straightforward changes to the Immix mark-region collector, reference counting, and reference counting Immix. This section describes the exact versions of these collectors and Section 3 describes our modifications.

Immix Immix is a mark-region garbage collector that combines bump-pointer allocation with space and time-efficient sweep-to-region collection [9]. The heap is composed of large blocks divided into contiguous lines. Our Immix implementation uses 32 KB blocks and 256 B lines. Objects are allocated into blocks and may span lines. The collector recycles free blocks and lines on partially occupied blocks. The allocator consumes contiguous free lines and free blocks. The garbage collector can leave objects in place, marking the containing line(s), or may copy surviving objects to reduce fragmentation in the same pass. The fine grain lines in Immix are well suited to pinning objects. Figure 2 shows an example during the mutator phase of the heap organization, allocating into contiguous lines in a recycled block. Free blocks are available for allocation to multiple parallel allocators.

Opportunistic Copying A classic copying collector, including MCC, must reserve an equal amount of free memory for copying. Immix may reserve much less or no memory, because it only copies *opportunistically* into free memory available at collection time. If the application wishes to allocate a large object and the allocator cannot satisfy the request, there may be free blocks and partially free blocks. In this case, Immix triggers a defragmentation collection. During a defragmenting collection, Immix chooses source and target blocks. It simply marks live objects and lines on target blocks. It opportunistically copies objects on source blocks into free lines on target blocks. When it exhausts the free memory in target blocks, it simply continues to mark the remaining objects live. Opportunistic copying supports pinning at the granularity of a line using a per-object pin bit.

Sticky Immix We also make Sticky Immix, a generational variant of Immix, conservative. Sticky Immix uses 'sticky mark bits', which follows Demers et al.'s design for generational mark-sweep [9, 19]. Sticky Immix performs generational collection without a copying nursery by marking old objects specially and limiting tracing to new objects during nursery collections. A write barrier records old to young references and adds them to the roots. Tracing only visits young objects and marks them old. Both Sticky Immix and RC Immix reserve a small number of free blocks and

use opportunistic copying to compact surviving young objects. Since most objects die young, this use of opportunistic copying very effectively reduces fragmentation among the nursery objects, creating completely free blocks, and consequently improving application locality.

Reference Counting Reference counting collectors maintain a count of incoming references to each object [18]. Until recently [31], all reference counting algorithms used a free-list allocator. When an object count falls to zero, the collector returns its free memory to the free-list. A straightforward implementation of reference counting requires that the compiler identify every pointer change, decrement the count for the overwritten referent, and increment the count for the new referent. Such an implementation does not require the runtime to maintain precise stacks, registers, or heap references. Objective-C, Perl, Delphi, PHP, and Swift all use naive reference counting, which continuously tracks pointers, rather than periodically examining roots.

Tracking every pointer change is expensive. A standard optimization is to 'defer' mutations to local variables in favor of scanning the stack periodically [20]. Furthermore, straightforward reference counting cannot collect cycles of garbage, so for completeness, either the application program must avoid creating cycles [3, 35, 36], or a separate cycle collector must augment the reference counter. For example, Objective-C and Perl forbid cycles and the Zend PHP VM uses trial deletion to collect cycles [6, 17, 26], which does not require exact root enumeration, but is relatively inefficient [22]. The other alternative is to periodically trace the heap, which requires full enumeration of all roots, either precisely or conservatively. Thus while naive reference counting is popular because it avoids the challenge of root enumeration, it is not high performance. Prior work shows that even simple deferred reference counting adds more than 40% overhead compared to copying generational [30, 31].

We are unaware of any reference counting collectors which use conservatively identified roots.

RC Immix RC Immix combines optimized deferred reference counting with the Immix heap, matching the performance of the fastest generational collectors [30, 31]. RC Immix periodically performs its reference counting work. To work with the Immix heap organization, it counts live objects on each line. When the number of live objects on a line fall to zero, RC Immix reclaims the line. It implements optimizations such as coalescing of multiple mutations to an object and reducing the reference counting bits. It handles cycles by performing periodic backup tracing collections. It implements generational copying behavior by allocating contiguously and then opportunistically copying surviving young objects. It has the immediacy of reference counting, because each collection promptly reclaims both young and mature objects when their count falls to zero.

3. Design

We now describe the design of our object map filtering mechanism for ambiguous roots and our family of conservative Immix and reference counting collectors.

3.1 Object Map Filtering

To precisely identify objects, we filter ambiguous roots with an *object map*, a bitmap which records the precise location of all objects that were live at the last collection or have been allocated since. A few details of maintaining the object map vary from collector to collector, but the fundamentals of the design are common to all.

The bitmap records the location of all potentially live objects. When processing ambiguous references, the collector consults the object map, discarding any reference that does not point to the start of a potentially live object.

Initially the object map is empty. At object allocation time, the allocator sets a bit in the object map that encodes the address of the start of the object. At the start of each collection, the collector first scans the stacks and registers. If a value falls in the range of the heap, the collector consults the object map. If the reference corresponds to an object map entry, it is a valid ambiguous root and the collector adds it to the conservative roots. Otherwise it is discarded.

During collection, the collector must update the object map to account for dead objects. In reference counting, which explicitly identifies dead objects, the collector simply unsets the relevant bit in the object map when it reclaims an object with a zero reference count. Tracing instead directly identifies live objects. After filtering the roots, the tracing collectors zero the entire object map and then the collector reconstructs it by setting a bit for each live object when it traces the object. Because our collectors are parallel, the collector must set or clear the bit atomically to avoid a race to update the containing word among the parallel collector threads. All allocators use thread-local allocation buffers, so there is no race to set the bit at allocation time.

To minimize the object map overhead, we use the x86 `BTS` and `BTR` instructions to set and clear its bits in the atomic modes when appropriate. We empirically established that these instructions outperform (0.6% total time improvement) software bit manipulation instruction sequences, particularly when changing the bit atomically.

Because of object alignment requirements and because the VM uses a specific format for its two word header, the VM can always disambiguate a 'status word' and 'type information block' (TIB) pointer, the two words in every object's header. We use this insight to reduce the object map resolution to one bit per eight bytes. When validating ambiguous pointers, we first determine whether the ambiguous reference points to a valid double word and then examine those words to determine whether the reference points to the start of an object. This optimization halves the space overhead of the object map from 1:32 (3%) to 1:64 (1.5%). It reduces

the mutator L1 data cache misses by 0.7%. By reducing the cache footprint of the object map, we improve mutator locality. The average mutator overhead due to the object map falls from 2.3% to 1.3% as a result of this optimization (Figure 3).

3.2 Conservative Immix and Sticky Immix

Immix's fine-grained heap organization with copying is an excellent match for conservative garbage collection. Most objects are allocated contiguously into 32 KB blocks, and can be copied upon survival. Conservative Immix pins the target objects of ambiguous references at the granularity of 256 B lines. The size of contiguous allocation regions and the associated potential for better locality is thus increased by a factor of eight over MCC, which pins at a page granularity. The granularity of pinning and associated wasted space is also reduced sixteen-fold. Objects referenced from ambiguous roots are pinned on the line(s) they occupy, but Immix may copy all other objects according to its usual heuristics. This feature limits the impact of ambiguous roots to internal line fragmentation.

Immix allocates into both completely empty blocks and partially occupied blocks, but never into used lines. When allocating into an empty block, the corresponding object map entries are first zeroed and then set as each object is allocated. When allocating into a recycled block, the object map areas associated with the free lines in the block are zeroed and the remaining areas are left as-is. Allocation then proceeds and sets the object map bits for each new object.

The Sticky Immix in-place generational collector design [9, 19] makes maintenance of the object map a little more difficult because the tracing phase of the collector is confined to the newly allocated objects that may be scattered throughout the heap. Sticky Immix records each block that it allocates into and then rather than clear the entire object map at the start of collection, it selectively clears the portions that were allocated into. Like other generational collectors, sticky collectors perform periodic full heap collections, during which conservative Sticky Immix clears the entire object map and refreshes it, as described above.

Conservative Immix and Sticky Immix use opportunistic copying. If an object is pinned, the object stays in place. For nursery objects in Sticky Immix and defragmenting collections in both collectors, the collectors identify source and target blocks for copying. If an object is not pinned and there is still free space on a target block, the collectors opportunistically copy unpinned objects from the source blocks to a target block. They otherwise simply mark the object. This process mixes copying and marking in the same collection. In both cases, they set the object map bit.

3.3 Conservative Reference Counting

As mentioned earlier, straightforward (naive) reference counting does not need to identify program roots. However, deferred reference counting depends on root enumeration. Deferral works by ignoring increments and decrements to local variables. It instead periodically establishes the roots, increments all objects that are root-reachable, only then does it reclaim zero reference count objects. It also buffers balancing decrements for each root. It then applies these decrements at the start of the next garbage collection, but after the current root increments [6]. We observe that it is correct to conservatively consider all objects reachable from ambiguous roots to be pinned for the duration of each collection cycle. Objects are only reclaimed if their reference count is zero *and* they not conservatively pinned.

Object map maintenance is relatively simple with reference counting. It sets the object map bits upon allocation, as usual. When an unpinned object's count falls to zero, the collector reclaims the object and clears its object map bit. The reference counter performs periodic cycle collection using a backup tracing algorithm. At each cycle collection, it clears the object map and sets object map bits for each object reached in the cycle collection trace.

3.4 Conservative RC Immix

This work was motivated in part by the insight that RC Immix was likely to be a very good match for conservative collection because it performs as well or better than copying generational, while efficiently supporting pinning at a fine granularity. To realize RC Immix$_{cons}$, we bring together each of the key ideas described above for Immix$_{cons}$ and RC$_{cons}$. Unlike RC, which uses a free-list, RC Immix uses Immix's lines and blocks to perform contiguous allocation and opportunistically copies surviving young objects. RC Immix behaves like a tracing collector with respect to young objects, so we employ the same approach to pinning and object map maintenance for them as we do for Sticky Immix$_{cons}$. Since RC Immix behaves like a reference counting collector with respect to mature objects, we clear object map entries for dead mature objects just as we do for RC$_{cons}$.

4. Methodology

This section presents the software, hardware, and measurement methodologies that we use for evaluation.

Benchmarks. We present results for 18 benchmarks from DaCapo [13], SPECjvm98 [33], and pjbb2005 [12]. The pjbb2005 benchmark is a fixed workload version of SPECjbb2005 [34] 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 include two outliers, mpegaudio and lusearch and in our tables, for completeness, but omit them from the graphs and averages. 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. [38] found a performance bug in the method `QueryParser.getFieldQuery()`, which revision r803664 of Lucene fixes [29]. 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 use this fixed benchmark lusearch-fix.

Jikes RVM & MMTk. We use Jikes RVM release 3.1.3+hg r10761 and MMTk. Jikes RVM [2] is an open source high performance Java virtual machine (VM) written in a slightly extended version of Java. 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 [11].

All of the exact and conservative Immix collectors use 32 KB blocks and 256 B lines. They reserve 2% of the heap for opportunistic copying. Exact and conservative Sticky Immix reserve an additional 2% and target blocks with new allocation for copying nursery survivors. Exact and conservative RC Immix add a dynamic copy reserve, computed based on blocks allocated and the line survival rate at the last garbage collection [31].

All of the garbage collectors we evaluate are parallel [10]. They use thread local allocation for each application thread to minimize synchronization. During collection, the collectors exploit available software and hardware parallelism [16]. To compare collectors, we vary the heap size to understand how well collectors respond to the time – space tradeoff. 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 interpret. 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 frequently executed methods. Using a timer-based approach, it schedules periodic interrupts. At each interrupt, the adaptive system records the currently executing method. A cost model then selects frequently executing methods that 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 upon which 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 [14]. We run each benchmark 20 times (20 invocations) and in Table 4 we report the average and 95% confidence intervals using Student's t-distribution.

Operating System. We use Ubuntu 12.04.3 LTS server distribution and a 64-bit (x86_64) 3.8.0-29 Linux kernel.

Hardware Platform. We report performance results on a 3.4 GHz, 22 nm Core i7-4770 Haswell with 4 cores and 2-way SMT. The two hardware threads on each core share a 32 KB L1 instruction cache, 32 KB L1 data cache, and 256 KB L2 cache. All four cores share a single 8 MB last level cache. A dual-channel memory controller is integrated into the CPU with 8 GB of DDR3-1066 memory.

RC Immix$_{cons}$ is publicly available at `https://jira.codehaus.org/browse/RVM-1085`.

5. Impact of Conservatism

This section performs the first detailed study of the impact of conservatism on collector mechanisms and design in managed languages. It quantifies the effect of conservative root scanning with respect to the number of roots returned and its impact and implications on space consumption (excess retention), filtering, and pinning. Section 6 quantifies the performance impacts.

For this analysis, we modify Jikes RVM to compute statistics that disambiguate exact and ambiguous roots, and their respective transitive closures in the same execution. We examine the state of the stacks, registers, and heap at garbage collection time. We force garbage collections at a fixed periodicity and make the heap sufficiently large that collections are only triggered by our explicit mechanism, never due to space exhaustion. The periodicity of forced collections is measured in bytes, and we tailored this setting for each benchmark so as to induce approximately one hundred collections per execution, which we average across the benchmark execution. We report the full statistics for each benchmark in Table 6 in the appendix. In this section, we report aggregate statistics.

5.1 Ambiguous Pointers

Table 1 shows the impact of conservative scanning on the root set gathered from the stacks and registers. The first row shows the average number of *unique* objects referenced from the stacks and registers when performing an *exact* scan. There are on average 98 unique objects referenced from the stacks and registers at a given garbage collection, rising as high as 263 (pmd) and falling to 35 (compress). The next four rows are all relative to the first row.

The next row indicates the total number of roots returned by an exact scan, as a factor increase over the unique roots.

	avg	min	max
Unique exact roots	*98*	*35*	*263*
All exact roots	2.21×	1.64×	3.85×
All unfiltered conservative roots	8.9×	5.8×	15.1×
All conservative roots	4.7×	2.7×	9.0×
Unique conservative roots	1.6×	1.2×	2.2×

Table 1. Ambiguous Pointers

The average across the benchmarks is 2.21, which indicates that for exact stack and register scans, the total number of roots returned is a bit more than twice that of the unique roots. The level of redundancy among the exact stack and register roots is highest in pmd (3.85×) and lowest in pjbb2005 (1.64×). Redundancy is not surprising since programs often pass the same references among methods, leaving multiple copies on the stack.

The next three rows look at unfiltered, filtered, and unique conservative roots, relative to the unique exact roots. The unfiltered roots are all values in stacks and registers that when viewed as addresses point within the heap. This set is on average 8.9× larger than the set of unique exact roots. The filtered conservative roots are the set of those roots that point to valid objects that were allocated since the last collection or live when last examined. These references are the ambiguous roots. The number of ambiguous roots is about half the number of unfiltered roots, and is 4.7× the size of the set of unique exact roots. The set of unique filtered conservative roots is 1.6× the size the set of unique exact roots, ranging from 1.2× (compress) to 2.2× (sunflow).

In summary, for our Java workloads, conservative scans of stacks and registers return around 60% more unique roots than exact scans.

5.2 Excess Retention

Perhaps the most obvious side effect of conservative collection is *excess retention* — a few false roots may keep many transitively reachable objects artificially alive. We measure excess retention in our instrumented JVM by performing two transitive closures over the heap at each collection, one exact and one conservative. We compare the size of the two closures at each GC and report the average. Table 2 quantifies the effect of excess retention in terms of KB and as a percentage of the live heap.

Excess retention is generally very low, with a handful of benchmarks reporting excess retention of less than 1 KB, a handful at around 20 KB or so, and compress reporting 622 KB. The compress benchmark is small, but it uses several large arrays. Artificially keeping one such array alive has a significant impact. The average excess retention is 44 KB. Normalizing those numbers in terms of the total live heap, excess retention accounts for an insignificant space overhead, 0.02%, and even in the outlier, compress is only 6%.

This analysis shows that excess retention affects very few objects for our Java workloads, even though it is the most obvious and direct manifestation of conservatism. Section 6.4 evaluates the performance effect of artificially increasing the number of objects pinned due to ambiguity.

	avg	min	max
Excess retention	*44 KB*	*0.2 KB*	*622 KB*
Excess retention / live	0.02%	0.001%	6.1%

Table 2. Excess Retention

5.3 Pointer Filtering

The object map and BDW free-list introspection are functionally equivalent. They determine whether an ambiguous pointer refers to an address that contains an object which was either live at the end of the last garbage collection or was allocated since then. If so, the collection retains the ambiguous root. Otherwise it is discarded.

In this comparison, we evaluate the default object map which uses just one bit for each eight bytes because it can disambiguate the two Jikes RVM header words (MS_{OM}). To expose the impact of map density, we also evaluate the object map using one bit for each four bytes, doubling the size of the map ($MS_{OM\times2}$). Using an object map imposes an overhead at allocation time due to updating the map for each new object to indicate its start address.

By contrast, BDW introspection does not require any extra work during allocation. At collection time, checking the validity of an ambiguous reference is simpler with a map than introspecting a free list. On the other hand, the maps must be maintained during collection, accounting for copying of objects (if any) and for the recycling of any dead objects; neither overhead is incurred by BDW filtering.

We use full heap mark-sweep (MS) garbage collection to measure the impact of validating ambiguous references and compare conservative BDW free-list introspection (BDW), the object map described in Section 3.1 (MS_{OM}), and an object map without header word disambiguation, doubling the size of its map ($MS_{OM\times2}$). We normalize to exact MS.

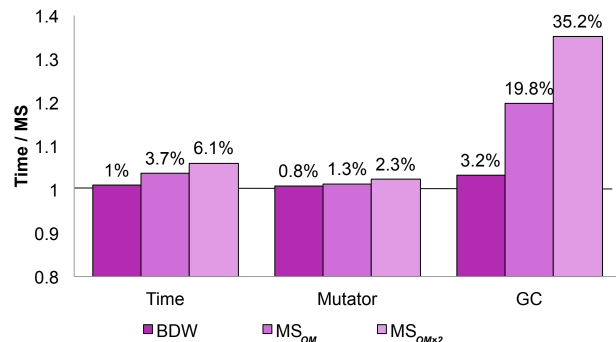


Figure 3. Conservative filtering total, mutator, and collection time overheads in mark-sweep. BDW is cheapest, requiring no additional space or allocation work. The smaller object map in MS_{OM} improves over object map filtering in both mutator and collection time.

		avg	min	max
Actual	*Pinned objects*	*164*	*40*	*435*
	Pinned objects / live	0.03%	0.004%	0.13%
	Pinned bytes	*14 KB*	*5 KB*	*54 KB*
	Pinned bytes / live	0.05%	0.008%	0.28%
MCC	Pinned page / pinned object	0.75	0.56	0.91
	Pinned bytes	*462 KB*	*140 KB*	*1120 KB*
	Pinned bytes / live	2.1%	0.4%	4.6%
	False pinned objects / page	60	27	119
	False pinned bytes	*282 KB*	*102 KB*	*682 KB*
Immix	Pinned line / pinned object	0.89	0.74	0.96
	Pinned bytes	*36 KB*	*10 KB*	*90 KB*
	Pinned bytes / live	0.2%	0.03%	0.4%

Table 3. Pinning Granularity

Figure 3 shows that on average, BDW introspection incurs essentially no mutator time overhead. The main effect is excess retention, which, although small as shown above in Section 5.2, still increases the live heap, incurring a collection-time overhead of 3.2% compared to exact MS, stemming from a increase in the number of collections by 3.6% (not shown). The BDW collection time overhead translates into a 1% total time overhead.

Compared to BDW, object maps incur more overhead due to setting bits at allocation time and a space penalty due to storing the map. All have the same excess retention. A sparse object map ($MS_{OM\times2}$) incurs a 2.3% overhead on the mutator (i.e., the application) compared to exact MS. A sparse object map incurs on average 35.2% collection-time overhead because it performs 13.7% more collections on average. The header word disambiguation improves the object map significantly. The mutator-time overhead for MS_{OM} drops to 1.3% instead of 2.3% and the collection time overhead is 19.8% on average, instead of 35.2% without the optimization.

These statistics reveal that, for a non-moving collector, BDW free-list introspection is the clear winner. However, as we show later, the advantages of copying in other collectors outweigh the penalty of the object map.

5.4 Pinning Granularity

A conservative collector must pin all objects that are the target of ambiguous references, because ambiguous references may be values and therefore cannot be modified. The direct effect of pinning an object will depend on the granularity at which the collector pins objects. BDW incurs no additional space overhead due to pinning, because it never moves any object. The Mostly Copying Collectors (MCC) operate at a page granularity (4 KB), pinning the object and all the other objects on the page as well. The Immix collectors pin at the granularity of a 256 B line and only pin the object, not all objects on the line.

Table 3 reports the impact of pinning at the object, line, and page granularity. The four 'Actual' rows report average number of pinned objects and their footprint in KB. On average, the total number of objects pinned at a given garbage collection is 164 and consume a total of 14 KB. This statistic is consistent with the conservative root set that is on average about 60% larger than the exact roots. The actual pinned objects are only 0.03% of all the live objects and the actual pinned bytes are only 0.05% of the live heap.

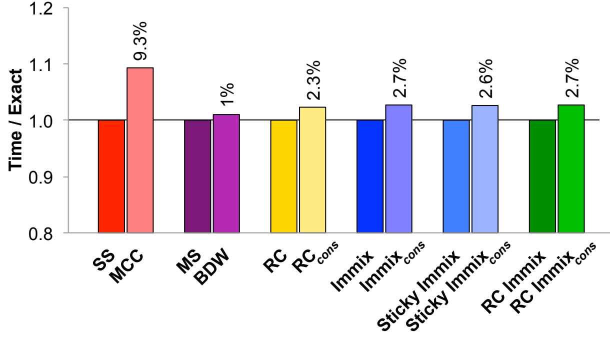
The five 'MCC' rows show the effect of Bartlett-style pinning at a page granularity. The first row shows how many pages are pinned on average by a given object. When more than one pinned object resides on a page, the value is less than one. On average 0.75 pages are pinned by each pinned object. The next row shows how many KB are consumed by the pinned pages. On average, the pinned pages consumed 462 KB which is about 2.1% of the live heap. It next shows the impact of false pinning. Recall that MCC collectors will pin all objects on a pinned page. The fourth 'MCC' row shows that on average around 60 unpinned objects fall on pages with pinned objects, resulting in on average 282 KB of falsely pinned objects at each garbage collection. Although MCC pins a relatively small fraction of the heap (2.1%), it is nearly two orders of magnitude larger than the actual fraction (0.05%) of pinned objects.

The 'Immix' rows in the table show the effect of pinning with Immix's line granularity. This first row shows on average how many lines are pinned by a given object. When more than one object pins a line, the value is less than one. On average 0.89 lines are pinned by each pinned object. The chances of another object pinning a given line is lower than for a page, so the average number of lines pinned grows to 0.89 from 0.75 for pages. The next row shows how many KB are consumed by the pinned lines. On average, pinned lines consume 36 KB, which is about 0.2% of the live heap. Compared to pages, which consume 462 KB, the line granularity of Immix decreases the space footprint by an order of magnitude. Whereas pinning pages effects around 2% of the live heap, pinning lines effects 0.2% of the heap.

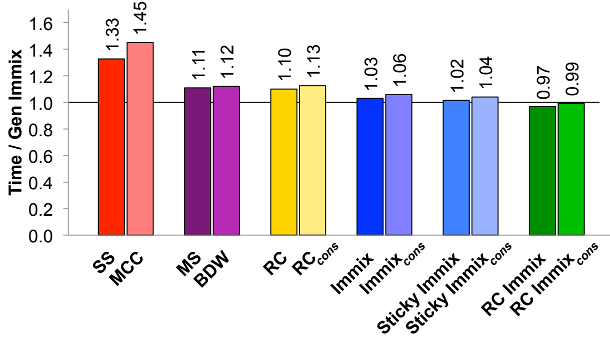
This section establishes that for our Java workloads root processing time and excess retention are not significant problems for conservative collectors, and pinning due to Immix-style lines has roughly an order of magnitude less direct impact than pinning due to MCC pages.

6. Performance Evaluation

This section evaluates the design and implementation of six conservative collectors: MCC, BDW, RC_{cons}, $Immix_{cons}$, Sticky $Immix_{cons}$, and RC $Immix_{cons}$. We compare them to their exact counterparts: semi-space (SS), mark-sweep (MS), RC, Immix, Sticky Immix, and RC Immix. The conservative mark-sweep collector (BDW) is a mature mark-sweep implementation with BDW-style reference filtering. The Mostly Copying Collector (MCC) is a Bartlett-style mostly copying collector that uses our object map to identify valid root referents. RC is a deferred reference counting



(a) Conservative relative to their exact variants. Except for MCC, conservative roots impose very little overhead.



(b) Overall performance relative to exact Gen Immix. RC Immix$_{cons}$ matches exact Gen Immix.

Figure 4. Geometric means of total performance for exact and conservative collectors at 2×minimum heap size. Lower is better.

collector that uses a free-list heap organization and collects cycles with a backup mark-sweep collector.

6.1 Conservative versus Exact Variants

We first evaluate performance penalties incurred by conservative garbage collection by comparing six different exact collectors to their conservative counterpart. This experiment holds the algorithms constant to explore the direct impact of ambiguous roots and pinning, as opposed to their indirect impact on algorithmic choice. Figure 4(a) shows that, except for MCC, the conservative collectors are within 1 to 2.7% of their exact counterparts. MCC suffers because pinning at a page granularity reduces mutator locality and induces fragmentation, resulting in more garbage collections (measured but not shown here). BDW has the lowest overhead because introspecting on the free-list is cheap and only performed at collection time, whereas maintaining the object map incurs small allocation and collection time costs. Section 5 demonstrated that excess retention, the number of pinned objects, and the cost of maintaining the object map and filtering objects are all low for Java benchmarks. That analysis explains why five of the conservative collectors see negligible overhead relative to their exact variants.

Figure 4(b) summarizes the results for all twelve collectors relative to Gen Immix. Gen Immix is a mature high performance copying generational collector that has been the Jikes RVM production collector since 2009. All of the collectors that use a free-list (MS, BDW, RC, and RC$_{cons}$) suffer significant performance penalties compared to Gen Immix. For example, BDW is 12% slower and RC$_{cons}$ is 13% slower than Gen Immix. The heap organization is the dominating effect as shown in prior work [9, 31], rather than exact or conservative root processing.

All of the exact and conservative Immix collectors outperform the free-list collectors. Prior work shows that degradations in mutator locality explain this difference [9, 11, 31]. A free-list degrades cache miss rates because the free-list allocator spreads contemporaneously allocated objects out in memory on different cache lines. In contrast, the bump pointer allocator places contemporaneously allocated objects contiguously in space, often sharing cache lines, improving their locality.

Exact Sticky Immix is only 2% slower and Sticky Immix$_{cons}$ is only 4% slower than Gen Immix. The best performing conservative collector is RC Immix$_{cons}$. Even though conservatism slows it down, it is still 1% faster than Gen Immix.

6.2 Total, Mutator, and Collection Time

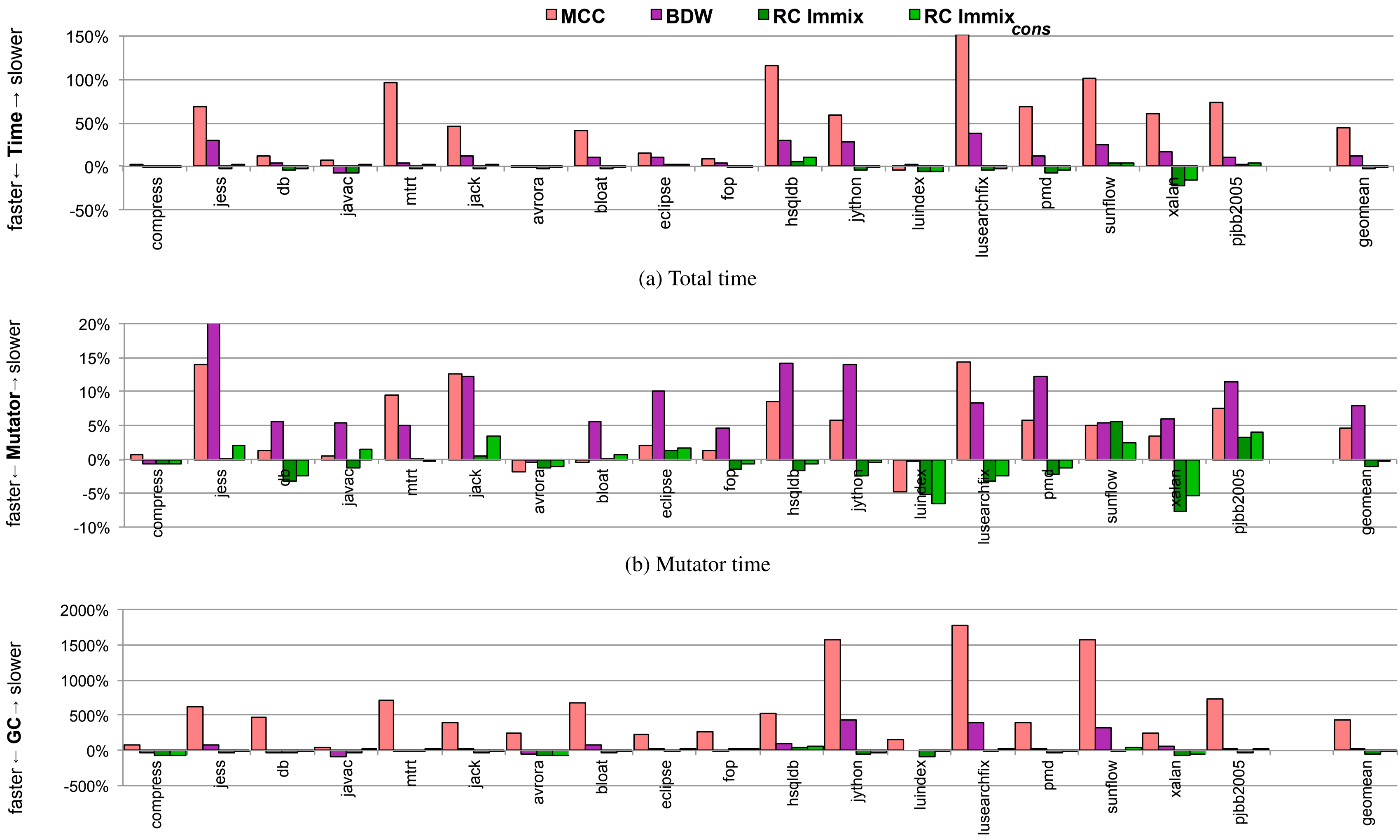
This section presents a more detailed per-benchmark performance analysis of total, mutator, and garbage collection times. For simplicity of exposition, we restrict this analysis to the best performing exact collector (Gen Immix), the best performing conservative collector (RC Immix$_{cons}$), its exact counterpart (RC Immix), and the prior conservative collectors (MCC, BDW) with a heap 2× the minimum in which all benchmarks execute. We present the numeric results in Table 4 and graph them in Figure 5.

The geometric mean in Figure 5(a) and the bottom of the four 'time' columns of Table 4 show that at this heap size, Gen Immix, RC Immix and RC Immix$_{cons}$ perform similarly on total execution time, while BDW performs 12% slower, and MCC performs 45% slower on average across our Java benchmarks. RC Immix$_{cons}$ lags RC Immix by just 2%, and is 1% better on average than Gen Immix, the production collector. RC Immix$_{cons}$ tracks RC Immix total performance closely across the benchmarks, following RC Immix's excellent performance on luindex, pmd, and xalan.

The five benchmarks where RC Immix$_{cons}$ degrades most against RC Immix are javac, jack, hsqldb, lusearch, and xalan. The javac, jack, and xalan benchmarks have higher mutator overhead (2.5-3%) compared to RC Immix. On javac, lusearch, and xalan, RC Immix$_{cons}$ has higher garbage collection overhead compared to RC Immix. The javac, lusearch, and xalan benchmarks have higher number of collections (18-25%) compared to RC Immix. The javac benchmark is a very memory-sensitive benchmark and the object map increases the heap pressure, increasing the number of collections. The pinning of objects disturbs the locality of the mutator, and

Benchmark	GenImmix			RC Immix			RC Immix$_{cons}$			BDW			MCC		
	milliseconds			Normalized to GenImmix											
	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	1760 ±0.3	1741 ±0.2	19 ±10.0	0.99 ±0.2	0.99 ±0.2	0.25 ±2.0	0.98 ±0.2	0.99 ±0.2	0.27 ±2.1	0.99 ±0.2	0.99 ±0.1	0.67 ±5.6	1.01 ±0.2	1.01 ±0.1	1.73 ±13.4
jess	355 ±0.3	323 ±0.2	32 ±2.6	0.98 ±0.8	1.00 ±0.8	0.76 ±2.0	1.01 ±0.4	1.02 ±0.3	0.88 ±2.6	1.31 ±0.4	1.26 ±0.3	1.79 ±3.9	1.69 ±0.9	1.14 ±0.3	7.18 ±15.9
db	1238 ±0.3	1209 ±0.3	29 ±2.2	0.96 ±0.5	0.97 ±0.5	0.74 ±1.4	0.98 ±0.4	0.98 ±0.4	0.93 ±1.7	1.05 ±0.5	1.05 ±0.5	0.68 ±4.1	1.12 ±0.6	1.01 ±0.4	5.79 ±16.2
javac	773 ±0.2	661 ±0.2	112 ±1.1	0.93 ±4.7	0.99 ±0.8	0.62 ±28.4	1.02 ±5.0	1.02 ±0.8	1.06 ±29.9	0.93 ±0.3	1.05 ±0.3	0.20 ±0.4	1.07 ±0.6	1.00 ±0.3	1.46 ±3.8
mpegaudio	1103 ±0.0	1103 ±0.0	0 ±0.0	1.00 ±0.3	1.00 ±0.3	0.00 ±0.0	1.00 ±0.0	1.00 ±0.0	0.00 ±0.0	1.00 ±0.0	1.00 ±0.0	0.00 ±0.0	0.98 ±0.2	0.98 ±0.2	0.00 ±0.0
mtrt	245 ±1.5	215 ±1.6	30 ±2.9	0.98 ±1.2	1.00 ±1.2	0.84 ±4.6	1.01 ±2.7	1.00 ±2.8	1.05 ±8.1	1.04 ±1.2	1.05 ±1.2	0.98 ±3.8	1.97 ±3.1	1.09 ±1.4	8.17 ±23.7
jack	496 ±0.3	453 ±0.2	43 ±2.7	0.98 ±0.5	1.00 ±0.4	0.67 ±2.4	1.02 ±0.8	1.03 ±0.5	0.86 ±4.6	1.12 ±0.3	1.12 ±0.2	1.08 ±3.0	1.46 ±0.9	1.13 ±0.4	4.91 ±11.4
mean	**811** ±0.4	**767** ±0.4	**44** ±3.1												
geomean				**0.97**	**0.99**	**0.60**	**1.00**	**1.01**	**0.77**	**1.07**	**1.09**	**0.75**	**1.34**	**1.06**	**4.02**
avrora	2266 ±0.3	2250 ±0.3	16 ±3.3	0.98 ±0.2	0.99 ±0.2	0.24 ±9.9	0.98 ±0.2	0.99 ±0.3	0.27 ±9.3	0.99 ±0.3	1.00 ±0.3	0.52 ±2.8	1.00 ±0.3	0.98 ±0.3	3.40 ±13.3
bloat	2179 ±0.4	2047 ±0.5	132 ±1.3	0.98 ±1.0	1.00 ±1.1	0.63 ±1.4	1.00 ±0.8	1.01 ±0.8	0.81 ±2.7	1.10 ±0.4	1.06 ±0.4	1.86 ±2.7	1.41 ±0.6	0.99 ±0.5	7.79 ±8.9
eclipse	11272 ±0.9	10654 ±1.0	618 ±1.1	1.00 ±1.2	1.01 ±1.2	0.87 ±2.1	1.02 ±0.9	1.02 ±1.0	1.06 ±2.4	1.11 ±0.9	1.10 ±1.0	1.18 ±1.7	1.15 ±0.9	1.02 ±0.9	3.31 ±3.1
fop	579 ±0.5	562 ±0.5	17 ±2.3	0.99 ±0.4	0.99 ±0.4	1.02 ±3.8	1.00 ±0.4	0.99 ±0.4	1.11 ±4.0	1.04 ±0.5	1.05 ±0.5	0.95 ±2.9	1.09 ±0.5	1.01 ±0.4	3.71 ±11.6
hsqldb	706 ±0.5	561 ±0.1	145 ±2.5	1.06 ±0.5	0.98 ±0.1	1.36 ±2.8	1.11 ±0.4	0.99 ±0.1	1.58 ±2.9	1.31 ±0.6	1.14 ±0.3	1.94 ±3.8	2.16 ±2.6	1.09 ±3.1	6.33 ±11.4
jython	2416 ±0.4	2335 ±0.4	81 ±1.7	0.96 ±0.3	0.98 ±0.3	0.52 ±1.1	0.98 ±0.5	1.00 ±0.5	0.65 ±3.4	1.28 ±0.4	1.14 ±0.4	5.43 ±9.3	1.58 ±0.7	1.06 ±0.6	16.69 ±23.0
luindex	637 ±7.8	632 ±7.8	5 ±6.8	0.94 ±6.1	0.95 ±6.2	0.04 ±8.4	0.94 ±5.4	0.93 ±5.5	0.98 ±5.5	1.00 ±7.8	1.00 ±7.8	1.70 ±9.8	0.97 ±5.6	0.95 ±5.6	2.62 ±18.3
lusearch	1306 ±0.4	782 ±0.6	524 ±0.4	0.62 ±0.4	0.79 ±0.5	0.36 ±0.3	0.68 ±0.6	0.81 ±0.8	0.49 ±0.7	1.37 ±1.0	0.94 ±0.7	2.03 ±1.7	2.51 ±1.6	0.95 ±0.7	4.85 ±3.5
lusearchfix	539 ±1.3	497 ±1.3	42 ±1.2	0.95 ±1.3	0.97 ±1.4	0.78 ±1.0	0.98 ±1.4	0.98 ±1.4	1.04 ±1.5	1.39 ±1.7	1.08 ±1.5	4.98 ±7.4	2.51 ±2.8	1.14 ±1.6	18.80 ±20.8
pmd	621 ±0.9	521 ±0.8	100 ±3.5	0.92 ±0.9	0.98 ±0.9	0.64 ±3.3	0.96 ±1.1	0.99 ±0.9	0.81 ±4.6	1.11 ±1.6	1.12 ±0.9	1.07 ±8.1	1.69 ±1.8	1.06 ±1.0	4.98 ±14.7
sunflow	1725 ±1.1	1619 ±1.2	106 ±0.9	1.05 ±1.2	1.06 ±1.3	0.88 ±3.2	1.05 ±0.9	1.03 ±0.9	1.35 ±4.4	1.25 ±1.1	1.05 ±1.0	4.29 ±5.8	2.01 ±1.7	1.05 ±0.9	16.75 ±12.4
xalan	754 ±0.6	579 ±0.7	175 ±1.0	0.79 ±0.6	0.92 ±0.7	0.34 ±0.5	0.85 ±0.6	0.95 ±0.8	0.51 ±0.6	1.17 ±1.2	1.06 ±1.1	1.55 ±2.2	1.61 ±1.0	1.03 ±0.8	3.52 ±3.9
mean	**2154** ±1.2	**2023** ±1.3	**131** ±2.2												
geomean				**0.96**	**0.98**	**0.51**	**0.98**	**0.99**	**0.84**	**1.15**	**1.07**	**1.81**	**1.49**	**1.03**	**6.73**
pjbb2005	2870 ±0.4	2606 ±0.3	264 ±2.1	1.01 ±0.9	1.03 ±0.4	0.76 ±7.7	1.04 ±1.5	1.04 ±0.3	1.03 ±16.8	1.11 ±0.4	1.11 ±0.3	1.09 ±2.4	1.74 ±2.0	1.07 ±0.3	8.25 ±25.1
min	245	215	5	0.79	0.92	0.04	0.85	0.93	0.27	0.93	0.99	0.20	0.97	0.95	1.46
max	11272	10654	618	1.06	1.06	1.36	1.11	1.04	1.58	1.39	1.26	5.43	2.51	1.14	18.80
mean	**1746** ±0.9	**1637** ±0.9	**109** ±2.5												
geomean				**0.97**	**0.99**	**0.55**	**0.99**	**1.00**	**0.83**	**1.12**	**1.08**	**1.31**	**1.45**	**1.05**	**5.68**

Table 4. Total, mutator, and collection performance at 2×minimum heap size with confidence intervals. Figure 5 graphs these results. We report milliseconds for Gen Immix and normalize the others to Gen Immix. (We exclude mpegaudio and lusearch from averages, see Methodology.) RC Immix$_{cons}$ is 2% slower than RC Immix and *still* slightly faster than production exact Gen Immix.



(a) Total time

(b) Mutator time

(c) Garbage collection time

Figure 5. Total, mutator, and collector performance for MCC, BDW, RC Immix, RC Immix$_{cons}$ normalized to Gen Immix for 18 benchmarks at 2× minimum heap size. RC Immix$_{cons}$ matches two exact high performance collectors.

for javac, xalan and lusearch it also introduces line fragmentation that increases the number of collections. In several cases, these benchmarks have higher than average numbers of conservative roots. For example, 1.7× for javac, 2.3× for lusearch, and 1.9× for xalan, where the average is 1.6× (see Table 6 in the Appendix). However, these effects are modest. Although RC Immix$_{cons}$ degrades javac, jack, hsqldb, lusearch, and xalan the most compared to exact RC Immix, RC Immix$_{cons}$ is still faster than Gen Immix on average.

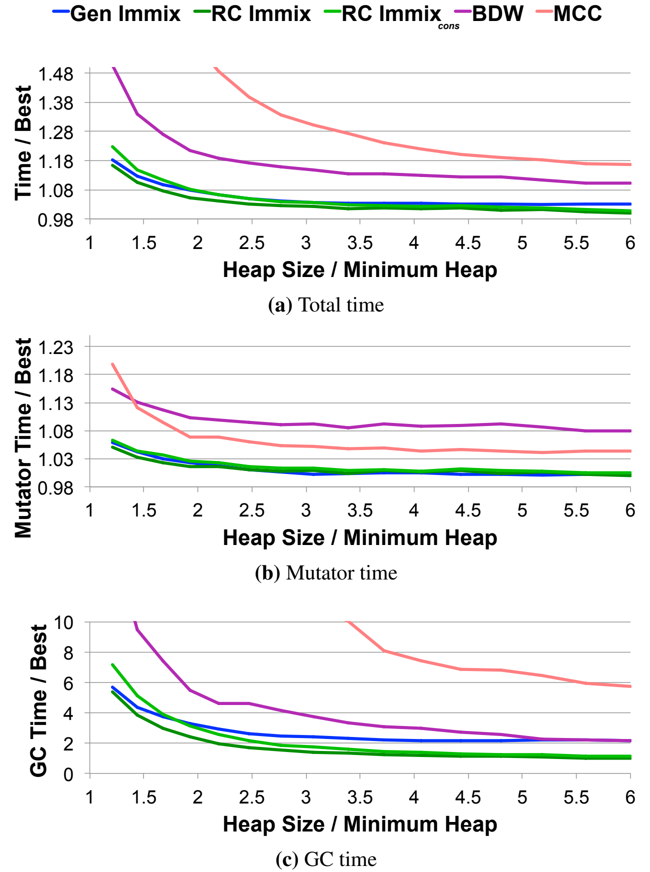
Figure 5(b) and the four 'time$_{mu}$' columns of Table 4 show that the mutator time is responsible for the total time results for the most part; Gen Immix, RC Immix and RC Immix$_{cons}$ perform similarly on mutator time, while BDW performs about 8% slower, and MCC performs about 5% slower on average across our suite of Java benchmarks. RC Immix$_{cons}$ is only 1% slower than RC Immix on mutator time, with no programs degrading mutator time by more than 3%. Gen Immix, RC Immix and RC Immix$_{cons}$ all use write barriers which impose a direct mutator overhead [39]. Nonetheless, despite not requiring a write barrier, BDW consistently suffers the worst mutator overhead, 8% on average.

Our BDW collector does not use an object map, and has no other mutator-time overheads directly associated with conservatism, so based on previous experiments [9, 11, 31], we attribute the slowdown to the loss of locality (explained in more detail in Section 6.1). Despite RC Immix$_{cons}$ having the mutator-time burden of maintaining an object map and a write barrier, its locality advantages are enough to deliver better mutator performance than BDW.

Figure 5(c) and the four 'time$_{gc}$' columns of Table 4 show the relative cost of garbage collection among the four collectors. Both RC Immix and RC Immix$_{cons}$ perform very well with respect to garbage collection time, outperforming Gen Immix by 45% and 17% respectively. While RC Immix improves collector time on all but two programs, RC Immix$_{cons}$ slows down 7 and improves 11 compared to Gen Immix. BDW performs worst on all but 6 benchmarks. BDW performs exceptionally well only on javac, which has an interesting lifetime behavior that builds up a large structure and then releases it all, four times over. This pattern can defeat generational collection because the survival rate for each generational collection will tend to be relatively high until the application releases the data structures.

MCC performs much worse than BDW and its huge garbage collection cost is the main reason for the overall 45% slowdown. MCC degrades 9% over standard semi-space collector, but neither are space efficient because they reserve half the heap for copying.

The three collectors that exploit the weak generational hypothesis do very well on all benchmarks. RC Immix and RC Immix$_{cons}$ do better than Gen Immix because they use reference counting for mature objects, which means that those objects are promptly reclaimed, whereas Gen Immix has to wait for sporadic full heap collections to reclaim space from dead mature objects.



(a) Total time

(b) Mutator time

(c) GC time

Figure 6. The performance of MCC, BDW, Gen Immix, RC Immix, and RC Immix$_{cons}$ normalized to the best system as a function of heap size. RC Immix$_{cons}$ collection time degrades a bit more in the smallest heap sizes compared to Gen Immix, but remains competitive in all heap sizes.

Summarizing, RC Immix$_{cons}$ performs extremely well. It suffers only about 1% overhead in mutator time and a similar overhead in collection time compared to its exact counterpart RC Immix. At this heap size and with Java workloads, RC Immix$_{cons}$ outperforms the mature and well tuned Gen Immix collector. The 13% advantage of RC Immix$_{cons}$ over BDW comes from: 1) much better mutator performance due to the bump pointer operating over coarse grained allocation regions, 2) further improvements to the mutator performance due to locality benefits that come from defragmentation with optimistic copying, and 3) much better garbage collection performance due to RC Immix$_{cons}$'s ability to exploit the weak generational hypothesis notwithstanding pinning with ambiguous roots.

6.3 Sensitivity to Heap Size

Garbage collection is fundamentally a time-space tradeoff, which this section examines by varying the heap size. Figure 6 shows the average total time, mutator time, and garbage

collection time for each system as a function of heap size. In each graph, performance is normalized to the best performance data point on that graph, so the best result has a value of 1.0. Figure 6(a) shows the classic time-space tradeoff curves expected of garbage collected systems, with BDW and MCC consistently slower compared to the other collectors. The graphs reveal that RC Immix and RC Immix$_{cons}$ are very similar, with a slow divergence in total time as the heap becomes smaller because RC Immix$_{cons}$ has a slightly larger heap and collects more often. Once heap sizes are tight, Gen Immix starts to outperform RC Immix$_{cons}$. Figure 6(b) shows that the relationship among the five collectors' mutator performance is almost unchanged in moderate heap sizes. For smaller heap sizes, they all degrade. BDW has the worst mutator performance except at the smallest heap size where BDW outperforms MCC because MCC disturbs locality by frequently copying nursery objects that have not had sufficient time to die. Figure 6(c) shows the relationship among the five collectors' garbage collection performance. RC Immix and RC Immix$_{cons}$ have better garbage collection performance than Gen Immix and MCC has the worst garbage collection performance. BDW garbage collection performance approaches Gen Immix as the heap becomes large and no collector is invoked frequently.

In summary, conservative Immix variants perform very close to their exact counterparts, and RC Immix$_{cons}$ performs as well or better than the best exact generational collector across a wide range of heap sizes.

6.4 Discussion and Wider Applicability

Although our empirical results are for Java, we believe that other languages will benefit from these algorithms.

Conservatism and Pinning The Immix conservative collector designs apply to any setting with ambiguous references, including fully conservative systems. However, the major performance advantage comes from opportunistic copying of unpinned objects; opportunities which are nonexistent when *all* references are ambiguous.

To explore the potential benefit of transitioning an existing managed language runtime to RC Immix$_{cons}$ first requires quantifying the relative fraction of ambiguous references in representative applications. Ambiguous references will be influenced by language elements and values in the stack and heap references. The environment also influences ambiguous references. For example, JavaScript may have larger numbers of conservatively pinned objects because the browser and document model may refer to JavaScript objects and are typically implemented in C.

Because all of our benchmarks pin so few objects, we explore how much pinning Immix can tolerate while maintaining its performance advantages. We conduct a simple experiment that artificially increases the number of pinned objects by *factors* of 2 to 32 compared to RC Immix$_{cons}$ with 0.2% average pinned in Java. We find that in a modest 2× heap, performance was degraded compared to RC Immix$_{cons}$ by 0.7% to 11% respectively, as shown in Table 5.

	Increased Pinning				
Heap Size	**2×** (0.4%)	**4×** (0.8%)	**8×** (1.6%)	**16×** (3.2%)	**32×** (6.4%)
2×	0.7%	1.8%	3.4%	6.8%	11%
3×	0.8%	1.1%	2.2%	2.3%	5.3%

Table 5. Performance effect of increasing pinning of objects by factors of 2× to 32× compared to RC Immix$_{cons}$ with 0.2% average pinned. The percentage of objects pinned is in parentheses. A 32-fold increase in pinning results in 11% slowdown in a 2× heap and 5.3% slowdown in a 3× heap.

Of course, other languages may pin more or less than Java. The fewer pinned objects, the more likely an Immix heap organization and opportunistic copying can improve locality and performance. The next step for attaining Immix performance advantages would be to modify the heap organization to use lines and blocks and implement a full-heap tracing Immix collector (Immix$_{cons}$). Our measurements show that even this simple system has the potential to deliver 5% or more total performance improvement.

Performance Potential One issue that may dampen the effects of heap organization and garbage collector efficiency is code quality. If the language implementation is immature and uses an interpreter or generates poor quality code, the collector's effect on overall performance will likely dampen. To test this hypothesis, we intentionally crippled our runtime, first by disabling optimization of application code and then also by deoptimizing the runtime code itself, including the garbage collector. The first scenario mimics a mature VM with low code quality (*mature*). The second approximates an immature VM with low code quality (*immature*). We measured both startup and steady state performance.

We find that RC Immix$_{cons}$ and Immix$_{cons}$ offered measurable, though dampened, advantages in all scenarios. This result suggests that the Immix heap structure will benefit both immature and high performance runtimes. Comparing with BDW implementations in the same scenarios, the benefits were most modest during startup (1% for 'mature' and 5% for 'immature'), which is unsurprising because the performance of other parts of the runtime, including the classloader and baseline JIT will dominate during startup. We were interested to find that in steady state, the immature VM scenario benefitted by 8%, more than the mature VM scenario at 4%. Presumably the low code quality of the mature VM scenario dominates, whereas in the immature VM, all elements are slow, so the locality and algorithmic benefits from Immix offer performance advantages. In all of these scenarios, RC Immix$_{cons}$ and Immix$_{cons}$ performed about the same, which suggests that the advantages of reference counting mature objects do not become apparent unless the VM and the code quality are both well optimized.

In summary, even while a VM is maturing, if very few objects are pinned, conservative Immix and RC Immix should improve performance. Their benefits are likely to grow as the VM itself matures and generated code quality improves.

7. Conclusion

VM developers often choose not to implement exact garbage collection because of the substantial software engineering effort it demands. Instead they have taken one of three tacks: 1) naive reference counting, 2) conservative non-moving mark-sweep with a free-list, or 3) conservative MCC with page pinning. For example, Objective-C, Perl, and Delphi use naive reference counting, Chakra uses non-moving mark-sweep, and WebKit uses MCC. A variety of prior work suggests and we confirm that these garbage collection algorithms sacrifice a lot of performance.

The contributions of this paper are the design and implementation of a high performance conservative collector for managed languages. This collector combines an object map to identify valid objects, Immix mark-region collection to limit the impact of pinning to a line granularity, and deferred reference counting to increase the immediacy of reclaiming old objects. We observe that we can pin the referents of ambiguous roots at a fine grain with an Immix line, which minimizes pinning overheads and maximizes locality benefits. We use opportunistic copying to mitigate the cost of pinning because it combines marking of pinned objects and copying of unpinned objects as space allows. We simply use an object map to determine that referents are objects, and then conservatively retain and pin their targets. We can capture the high performance and prompt reclamation of deferred reference counting even with ambiguous roots. No previous work combines ambiguous roots with reference counting. The resulting RC Immix$_{cons}$ collector attains efficient generational behavior, efficient pinning, and the fast reclamation of old objects. Combining these collector mechanisms in this novel way leads to a very surprising result: high-performance conservative garbage collection.

Acknowledgements We thank Filip Pizlo and Tony Hosking for encouraging us to seriously investigate the performance potential of Bartlett-style conservative collectors, and thank Daniel Frampton for his feedback on the manuscript.

References

[1] O. Agesen, D. Detlefs, and J. E. Moss. Garbage collection and local variable type-precision and liveness in java virtual machines. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 269–279, New York, NY, USA, 1998. ACM. doi: 10.1145/277650.277738.

[2] 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.

[3] Apple Inc. Transitioning to ARC release notes, Aug. 2013. URL https://developer.apple.com/library/mac/releasenotes/ObjectiveC/RN-TransitioningToARC.

[4] Apple Inc. *The Swift Programming Language*. Swift Programming Series. Apple Inc., June 2014.

[5] G. Attardi and T. Flagella. A customisable memory management framework. In *Proceedings of the 6th Conference on USENIX Sixth C++ Technical Conference - Volume 6*, CTEC'94, pages 8–8, Berkeley, CA, USA, 1994. USENIX Association.

[6] 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.

[7] J. Baker, A. Cunei, T. Kalibera, F. Pizlo, and J. Vitek. Accurate garbage collection in uncooperative environments revisited. *Concurrency and Computation: Practice and Experience*, 21 (12):1572–1606, 2009. doi: 10.1002/cpe.1391.

[8] J. F. Bartlett. Compacting garbage collection with ambiguous roots. *SIGPLAN Lisp Pointers*, 1(6):3–12, Apr. 1988. ISSN 1045-3563. doi: 10.1145/1317224.1317225.

[9] 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.

[10] 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. doi: 10.1109/ICSE.2004.1317436.

[11] 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.

[12] 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.

[13] 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.

[14] 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. doi: 10.1145/1378704.1378723.

[15] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18 (9):807–820, Sept. 1988. ISSN 0038-0644. doi: 10.1002/spe.4380180902.

[16] 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.

[17] T. W. Christopher. Reference count garbage collection. *Software Practice and Experience*, 14(6):503–507, June 1984. doi: 10.1002/spe.4380140602.

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

[19] A. Demers, 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.

[20] 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.

[21] A. Diwan, E. Moss, and R. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 273–282, New York, NY, USA, 1992. ACM. doi: 10.1145/143095.143140.

[22] 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.

[23] F. Henderson. Accurate garbage collection in an uncooperative environment. In *Proceedings of the 3rd International Symposium on Memory Management, ISMM 2002, Berlin, Germany, June 20 - 21, 2002*, pages 150–156. ACM, 2002. ISBN 1-58113-539-4. doi: 10.1145/512429.512449.

[24] M. Hirzel, A. Diwan, and J. Henkel. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *ACM Trans. Program. Lang. Syst.*, 24(6):593–624, Nov. 2002. ISSN 0164-0925. doi: 10.1145/586088.586089.

[25] A. L. Hosking. Portable, mostly-concurrent, mostly-copying garbage collection for multi-processors. In *Proceedings of the 5th International Symposium on Memory Management, ISMM 2006, Ottawa, Canada, June 10 - 11, 2006*, pages 40–51. ACM, 2006. ISBN 1-59593-221-6. doi: 10.1145/1133956.1133963.

[26] R. D. Lins. Cyclic reference counting with lazy mark-scan. *Inf. Process. Lett.*, 44(4):215–220, 1992. doi: 10.1016/0020-0190(92)90088-D.

[27] LLVM. Accurate garbage collection with LLVM, Feb. 2014. URL http://llvm.org/docs/GarbageCollection.html.

[28] 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.

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

[30] 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*, pages 73–84. ACM, 2012. doi: 10.1145/2258996.2259008.

[31] R. Shahriyar, S. M. Blackburn, X. Yang, and K. S. McKinley. Taking off the gloves with reference counting Immix. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications, OOPSLA'13, Indianapolis, Indiana, USA, Oct, 2013*, pages 93–110. ACM, 2013. doi: 10.1145/2509136.2509527.

[32] F. Smith and G. Morrisett. Comparing mostly-copying and mark-sweep conservative collection. In *Proceedings of the 1st International Symposium on Memory Management, ISMM 1998, Vancouver, BC, Canada, October 17 - 19, 1998*, pages 68–78. ACM, 1998. ISBN 1-58113-114-3. doi: 10.1145/301589.286868.

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

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

[35] L. D. Stein. Devel::Cycle - Find memory cycles in objects, 2003. URL http://search.cpan.org/~lds/Devel-Cycle-1.11/lib/Devel/Cycle.pm.

[36] J. R. Thomas, M. Cantu, and A. Bauer. Reference counting and object harvesting in Delphi. *Dr. Dobb's*, May 2013. URL http://www.drdobbs.com/mobile/reference-counting-and-object-harvesting/240155664.

[37] WebKit. The WebKit open source project, 2014. URL http://trac.webkit.org/browser/trunk/Source/JavaScriptCore/heap.

[38] 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.

[39] X. Yang, S. M. Blackburn, D. Frampton, and A. L. Hosking. Barriers reconsidered, friendlier still! In *Proceedings of the 11th International Symposium on Memory Management, ISMM 2012, Beijing, China, June 15 - 16, 2012*, pages 37–48. ACM, 2012. doi: 10.1145/2258996.2259004.

Appendix

This appendix presents individual benchmark statistics that support the aggregate analysis in Section 5. Table 6 includes the basic statistics on the heap, exact roots, and conservative roots for each benchmark. It further quantifies their effects on filtering, excess retention, and pinning. We examine the number of pinned objects and how much memory fragmentation this causes MCC pages and Immix line pinning to consume with an object map.

The 'Live Heap' column shows the size of live object graph in MB. As mentioned in Section 5, we compute all of the analysis in the table by repeatedly performing full heap garbage collections and measuring and comparing statistics within each collection using exact and conservative roots. We targeted about 100 GCs per benchmark and the 'Force GCs' column shows that the actual number of GCs ranges from 72 to 144.

The set of 'Roots' columns show the raw number of 'Exact' unique roots and all roots. The three columns of 'Conservative' root statistics are normalized to the exact unique roots. While the 'all' conservative column shows a factor of 8.9 more conservative roots are processed, filtering reduces them (filt.) and only a factor of 1.6 are unique (uniq).

The 'Excess Retention' columns show in KB and as a percentage how many additional objects are transitively reachable from the conservative roots and thus kept live that an exact collector would have reclaimed. Since one root could transitively reach the whole entire heap, even one conservative roots could have a large effect. However, we do not observe this behavior. In these Java Benchmarks, only one benchmark (compress) has excess retention greater than 0.3%. It uses a few large arrays and retaining even one live array has a large impact.

The 'Pinned Space' quantifies the exact number of objects pinned ('Objects'), which is the same as BDW will pin, and the effect of Immix line pinning and MCC page pinning. MCC pins two orders of magnitude more objects than BDW or line pinning. The last two columns in the table quantifies how much of that increase is due to the false pinning of other objects on the page — they account for about half of the excess retention (282 KB of 462 KB). Immix line pinning is extremely effective at limiting the impact of ambiguous roots to just 0.2% of heap objects. Section 5 includes more discussion on these statistics and their implications.

Benchmark	Live Heap MB	No. Force GCs	Roots (/exact unique) Exact uniq	Exact all	Conservative all	Conservative filt.	Conservative uniq	Excess Retention KB	Excess Retention %	Pinned Space (/pinned obj, KB, % live) Objects obj.	Objects KB	Immix lines line	Immix lines KB	Immix lines %	MCC pages page	MCC pages KB	MCC pages %	False Pinning MCC /page	False Pinning MCC KB
compress	10.0	81	*35*	2.16	7.1	3.2	1.2	622.3	6.1	*40*	6	0.96	10	0.1	0.88	140	1.4	89.1	102
jess	8.7	75	*76*	1.96	8.3	4.7	1.8	0.6	0.0	*134*	10	0.89	30	0.3	0.72	388	4.4	51.3	226
db	14.4	144	*38*	1.99	7.1	3.0	1.2	0.3	0.0	*46*	6	0.96	11	0.1	0.88	161	1.1	119.1	138
javac	14.9	105	*74*	2.10	7.8	4.6	1.7	0.6	0.0	*127*	9	0.87	28	0.2	0.71	362	2.4	77.0	275
mtrt	13.0	72	*73*	1.81	7.4	4.0	1.6	0.6	0.0	*113*	8	0.74	21	0.2	0.57	258	1.9	53.4	181
jack	8.1	103	*46*	2.01	7.5	3.6	1.5	0.5	0.0	*69*	7	0.88	15	0.2	0.77	212	2.6	51.0	116
avrora	17.8	138	*108*	1.95	9.3	4.9	1.4	2.3	0.0	*150*	13	0.91	34	0.2	0.69	413	2.3	71.9	309
bloat	21.7	78	*61*	2.04	6.7	3.4	1.4	13.7	0.1	*85*	7	0.90	19	0.1	0.79	267	1.2	50.7	139
chart	22.9	108	*58*	1.78	5.8	2.7	1.3	60.4	0.3	*72*	5	0.86	16	0.1	0.76	219	0.9	55.0	128
eclipse	53.0	89	*96*	2.17	8.9	4.3	1.5	3.7	0.0	*141*	11	0.96	34	0.1	0.91	512	0.9	67.8	324
fop	22.2	97	*71*	2.33	6.5	3.5	1.3	0.2	0.0	*92*	6	0.96	22	0.1	0.85	314	1.4	89.7	261
hsqldb	64.9	117	*70*	2.16	10.7	5.1	1.5	18.6	0.0	*105*	8	0.92	24	0.0	0.79	333	0.5	60.1	225
jython	53.7	128	*81*	2.84	13.1	8.0	1.8	2.1	0.0	*145*	7	0.94	34	0.1	0.84	488	0.9	74.9	316
luindex	16.3	133	*53*	2.14	9.0	4.7	1.7	12.8	0.1	*85*	5	0.94	20	0.1	0.82	277	1.7	66.2	198
lusearch	15.2	168	*124*	1.89	10.8	5.8	2.3	30.7	0.2	*274*	0.8	20	55	0.4	0.66	721	4.6	29.5	249
lusearchfix	15.3	110	*126*	1.85	10.1	5.4	2.1	2.9	0.0	*259*	21	0.84	54	0.3	0.68	710	4.5	36.2	294
pmd	35.0	74	*263*	3.85	11.6	6.8	1.5	24.1	0.1	*397*	20	0.84	83	0.2	0.64	1022	2.9	53.7	682
sunflow	19.2	101	*193*	2.65	13.6	8.1	2.2	48.0	0.2	*407*	54	0.80	82	0.4	0.56	904	4.6	27.0	393
xalan	23.9	106	*233*	3.73	15.1	9.0	1.9	13.8	0.1	*435*	54	0.83	90	0.4	0.64	1120	4.6	37.9	572
pjbb2005	187.1	92	*106*	1.64	10.5	5.2	2.0	7.7	0.0	*212*	15	0.93	49	0.0	0.79	673	0.4	59.0	476
min	8.1	72	*35*	1.64	5.8	2.7	1.2	0.2	0.0	*40*	5	0.74	10	0.0	0.56	140	0.4	27.0	102
max	187.1	144	*263*	3.85	15.1	9.0	2.2	622.3	6.1	*435*	54	0.96	90	0.4	0.91	1120	4.6	119.1	682
mean	**34.0**	**103.9**	***98***	**2.21**	**8.9**	**4.7**	**1.6**	**44.0**	**0.0**	***164***	**14**	**0.89**	**36**	**0.1**	**0.75**	**462**	**1.7**	**59.3**	**282**

Table 6. Individual benchmark statistics on live heap size, exact roots, conservative roots, excess retention, and pinning. The table presents arithmetic mean for quantities and geometric mean for percentages. The text of the appendix explains each column and Section 5 analyses the aggregate meaning in more detail. Overall, this table shows that conservative roots have only a small impact on root scanning work, excess retention, and pinning at an Immix-line granularity. In particular, although all conservative roots expand the potential root set by a factor of 8.9, they are still few, and after filtering, they only expand the collector work by a factor of 1.6. Excess retention is low at 0.02%. Pinning at at page granularity effects 1.7% of objects on average, whereas pinning at a line granularity effects only 0.1% objects.