High Performance Reference Counting and Conservative Garbage Collection

Rifat Shahriyar

A thesis submitted for the degree of
Doctor of Philosophy
The Australian National University

April 2015

Except where otherwise indicated, this thesis is my own original work.

Rifat Shahriyar
15 April 2015

to my mother, Ferdous Kabir

Acknowledgments

This work would not have been possible without the generous support and assistance of many individuals and organizations. I only hope that I have remembered to give all of them the credit they deserve.

First and foremost I offer my sincerest gratitude to my supervisor, Professor Steve Blackburn. This thesis would not have been possible without his immense support, knowledge, guidance and patience. Thanks Steve for helping me happily sail through this academic journey. I feel very lucky to have such an awesome supervisor like Steve.

I would also like to thank my other supervisor, Professor Kathryn McKinley for helping me throughout my doctoral study. Her informative discussions, feedback and cooperative nature has always motivated and guided me. Her curiosity for proper and thorough understanding always inspired me and makes my thesis much richer.

I would also like to thank Dr. Daniel Frampton for helping me during the initial phase of my doctoral study. He made things understand very easily and his informative discussions were very helpful.

I would also like to thank the Australian National University for helping me financially throughout the duration of my studies and stay in Australia.

I would also like to express my gratitude to all my fellow students for rendering their friendliness and support, which made my stay in Australia very pleasant. Thank you Vivek Kumar, Ting Cao, Xi Yang, James Bornholt, Tiejun Gao, Yi Lin, Kunshan Wang, and Luke Angove for the helpful discussions. I feel very lucky to be able to work with these excellent and friendly fellows.

I would also like to thank my friends in Australia especially Wayes Tushar, Saikat Islam, Shaila Pervin, Md Mohsin Ali, Shakila Khan Rumi, Swakkhar Shatabda, Kh. Mahmudul Alam, Masaba Adneen, Mirza Adnan Hossain, Ashika Basher, Abdullah Al Mamun, Samia Israt Ronee, little Aaeedah, Fazlul Hasan Siddiqui, Nevis Wadia, little Afeef, Rakib Ahmed, Tofazzal Hossain, Masud Rahman, and last but not the least cute little Ehan. They care about me and are always there to provide assistance when I need it. My life in Australia is much more colorful with their company.

My thesis would be incomplete without acknowledging the love and support of my family members, my mother Ferdous Kabir, my grandmother Fakrujjahan, my sister Shampa Shahriyar and my brother in law S M Abdullah. Finally, I would like to acknowledge my late father, Mohammad Shahriyar, who is my ideal.

Abstract

Garbage collection is an integral part of modern programming languages. It automatically reclaims memory occupied by objects that are no longer in use. Garbage collection began in 1960 with two algorithmic branches — *tracing* and *reference counting*. Tracing identifies live objects by performing a transitive closure over the object graph starting with the stacks, registers, and global variables as roots. Objects not reached by the trace are implicitly dead, so the collector reclaims them. In contrast, reference counting explicitly identifies dead objects by counting the number of incoming references to each object. When an object's count goes to zero, it is unreachable and the collector may reclaim it.

Garbage collectors require knowledge of every reference to each object, whether the reference is from another object or from within the runtime. The runtime provides this knowledge either by continuously keeping track of *every* change to each reference or by periodically enumerating all references. The collector implementation faces two broad choices — *exact* and *conservative*. In exact garbage collection, the compiler and runtime system precisely identify all references held within the runtime including those held within stacks, registers, and objects. To exactly identify references, the runtime must introspect these references during execution, which requires support from the compiler and significant engineering effort. On the contrary, conservative garbage collection does not require introspection of these references, but instead treats each value ambiguously as a potential reference.

Highly engineered, high performance systems conventionally use *tracing* and *exact* garbage collection. However, other well-established but less performant systems use either *reference counting* or *conservative* garbage collection. Reference counting has some advantages over tracing such as: a) it is easier implement, b) it reclaims memory immediately, and c) it has a local scope of operation. Conservative garbage collection is easier to implement compared to exact garbage collection because it does not require compiler cooperation. Because of these advantages, both reference counting and conservative garbage collection are widely used in practice. Because both suffer significant performance overheads, they are generally not used in performance critical settings. This dissertation carefully examines reference counting and conservative garbage collection to understand their behavior and improve their performance.

My thesis is that reference counting and conservative garbage collection can perform as well or better than the best performing garbage collectors.

The key contributions of my thesis are: 1) An in-depth analysis of the key design choices for reference counting. 2) Novel optimizations guided by that analysis that significantly improve reference counting performance and make it competitive with a well tuned tracing garbage collector. 3) A new collector, RC Immix, that replaces the traditional free-list heap organization of reference counting with a line and block

heap structure, which improves locality, and adds copying to mitigate fragmentation. The result is a collector that outperforms a highly tuned production generational collector. 4) A conservative garbage collector based on RC Immix that matches the performance of a highly tuned production generational collector.

Reference counting and conservative garbage collection have lived under the shadow of tracing and exact garbage collection for a long time. My thesis focuses on bringing these somewhat neglected branches of garbage collection back to life in a high performance setting and leads to two very surprising results: 1) a new garbage collector based on reference counting that outperforms a highly tuned production generational tracing collector, and 2) a variant that delivers high performance conservative garbage collection.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis addresses the challenges and opportunities of achieving high performance for two areas of memory management that have historically suffered from poor performance, but are still widely used — reference counting and conservative garbage collection.

1.1 Problem Statement

Today garbage collection is ubiquitous; almost every managed programming language has support for garbage collection. It automatically reclaims memory occupied by objects that are no longer in use. It frees the programmer from manually dealing with memory deallocation. The benefits of garbage collection are increased reliability, a decoupling of memory management from other software engineering concerns, and less developer time spent chasing memory management errors. The memory manager is key to overall performance of managed applications because it includes the direct cost of garbage collection as well as the indirect cost of locality due to its layout of objects in memory.

Garbage collection algorithms have been built upon two branches since 1960 — *tracing* and *reference counting*. Tracing indirectly identifies garbage, and directly identifies live objects by performing a transitive closure over the object graph starting with the stacks, registers, and global variables as roots [McCarthy, 1960]. Reference counting directly identifies garbage by maintaining a count of incoming references to each object [Collins, 1960]. It has immediacy of reclamation and simple implementation and is thus used in well-established systems for PHP, Perl, and Objective-C.

Garbage collector implementations have two choices — *exact* and *conservative*. In exact garbage collection, the runtime precisely identifies all references, including those in stacks, registers, and objects, which requires support from the compiler and a significant engineering effort. A conservative garbage collector does *not* exactly identify these references, instead it inspects all values held within the runtime and dynamically tests whether they point to a valid object [Bartlett, 1988; Boehm and Weiser, 1988; Demers et al., 1990; Smith and Morrisett, 1998; WebKit, 2014]. Although conservatism forces some restrictions upon the runtime, the compiler and runtime are free of the onerous implementation issues of exactly tracking references in all code.

Conservative garbage collection is used to provide automatic memory management support for unmanaged languages such as C/C++, and managed languages such as JavaScript.

Highly engineered systems, such as HotSpot, J9, and .NET, conventionally use tracing and exact garbage collection for high performance. Many other well-established systems use either reference counting or conservative garbage collection implementations, such as PHP and Chakra for JavaScript. These systems are widely used, but suffer significant performance overheads compared to exact generational tracing garbage collection. This thesis addresses this problem.

In the following, we briefly describe reference counting and conservative garbage collection. They are described in more detail in Section 2.5 and 2.7.

Reference Counting The two algorithmic roots of the garbage collection family tree were born within months of each other in 1960 [McCarthy, 1960; Collins, 1960]. Reference counting works by keeping a count of incoming references to each object and collecting objects when their count falls to zero [Collins, 1960]. Therefore all that is required of its implementation is to notice each pointer change, increment the target object's count whenever a pointer to it is created and decrement the target object's count when a pointer to it is overwritten. The collector reclaims objects when their count reaches zero and then decrements all of their descendants' counts. This algorithm is simple, immediate, and requires no global computation. The simplicity of this naive implementation is particularly attractive and thus widely used in well-established systems such as PHP, Perl and Objective-C. By contrast, tracing collectors start with a set of roots, which requires the runtime to enumerate all references into the heap held within the runtime such as, global variables, stacks, and registers. The collector performs a transitive closure from these roots, marking each reachable object as live. The collector then reclaims unmarked objects.

However, naive reference counting is slow. Intercepting every pointer mutation, including those to the stacks and registers is costly. Several approaches reduce this cost. Deferral overlooks changes to the stacks and registers by periodically enumerating the old and new values [Deutsch and Bobrow, 1976; Bacon et al., 2001]. Coalescing elides repeated changes to the same heap references [Levanoni and Petrank, 2001, 2006]. Ulterior reference counting ignores operations for young objects [Blackburn and McKinley, 2003]. Even with these optimizations, reference counting is slow. We compare a state-of-the-art reference counting collector and a highly tuned mark-sweep implementation and find that reference counting is over 30% slower than its tracing counterpart. The best performing collectors are generational tracing [Blackburn et al., 2004a; Blackburn and McKinley, 2008] and are widely used for Java and C#. We find that reference counting is over 40% slower than a generational tracing collector. Consequently, reference counting is not used in any high performance systems today. In spite of being one of the major algorithmic branches of garbage collection, the performance of reference counting is still poor. By contrast, researchers have extensively studied tracing collectors. We claim that with proper understanding of its behavior and different key design points, reference counting has the potential

to produce high performance garbage collection alongside tracing. By bringing one of the two major algorithmic branches back into contention, language implementers will be presented with a much richer choice of implementation alternatives.

Conservative Garbage Collection Managed language semantics guarantee that the runtime can exactly identify references held within heap objects because type of each heap object is known by the runtime. However, exactly identifying references held within dynamic runtime state such as stacks and registers requires compiler cooperation. Modern managed languages such as Java and C# use exact garbage collection by enforcing a strict compiler disciple and runtime interface. Many implementations of other popular languages, including PHP, Perl, JavaScript, and Objective-C, avoid the engineering headache of exact garbage collection by using conservative garbage collection or naive reference counting.

A conservative garbage collector is constrained by its reliance on ambiguous references [Bartlett, 1988; Boehm and Weiser, 1988; Demers et al., 1990; Smith and Morrisett, 1998; WebKit, 2014]. The collector must (1) retain the referent since the ambiguous reference may actually be a pointer, and (2) pin (not move) the referent (object held by the reference) since the ambiguous reference may actually be a value and therefore cannot be modified. These constraints prevented previous conservative collectors for managed languages from achieving performance competitive to generational tracing because either (1) they forgo moving objects altogether, which is essential to the best performing collectors, or (2) they pin any page targeted by an ambiguous root, incurring significant overheads. Conservative root identification treats every value in the stacks, registers, and statics as a potential ambiguous root. The collector first checks whether the value falls within the range of valid heap addresses and if so, applies filtering to determine whether it points to a valid object. Conservative garbage collection is widely used but current implementations incur significant performance overheads compared to the high performance exact tracing collectors that move objects.

1.2 Scope and Contributions

The aim of this thesis is to address the performance barriers affecting reference counting and conservative garbage collection. Reference counting and conservative garbage collection are widely used, but prior implementations suffer from poor performance. We achieve high performance reference counting with novel optimizations guided by detailed analysis of its key design points and changing its free-list heap organization. We build conservative variants of existing exact garbage collectors with negligible overhead. We demonstrate that conservatism is compatible with high performance even though the prior conservative implementations do not achieve it. We achieve high performance conservative garbage collection by building a conservative garbage collector on top of our fast reference counting with changed heap structure, and with a low overhead object map to validate ambiguous references. With our contribu-

tions, for the first time, the performance of both reference counting and conservative garbage collection are competitive with the best copying generational tracing collectors.

This thesis uses Java as a representative of modern high-level languages and the Jikes RVM virtual machine as the managed runtime implementation [Alpern et al., 2000, 2005]. Java is, as of 2014, one of the most popular programming languages in use [TIOBE, 2014a,b] and Jikes RVM won 2012 SIGPLAN Software Award because of its high quality and modular design that made it easy for researchers to develop, share, and compare advances in programming language implementation [Jikes, 2012]. To evaluate different garbage collectors, we implemented them in Jikes RVM. The methodology and insights developed here should be applicable beyond this specific context. The analyses and optimizations we introduce throughout the thesis are general and can be implemented in other JVMs as well as in non-Java managed languages.

Optimizing Reference Counting We first show that a prior state-of-the-art reference counting implementation using a free list lags the performance of a highly tuned tracing collector by more than 30% on average. We perform an in-depth analysis of the key design choices of reference counting with a free list: a) how to store the count, b) how to correctly maintain the count, and c) how to collect cycles of garbage. We confirm that limited bit reference counting is a good idea because the vast majority of objects have a maximum count of seven or less [Jones et al., 2011]. We identify that young objects are responsible for more than 70% of increments and decrements and that on average only 10% of them survive. We introduce two novel optimizations that eliminate reference counting operations for non-surviving young objects, significantly reduce the workload for the cycle detector, and use limited bit reference counting to significantly improve performance. The ultimate outcome is the first reference counting implementation with performance competitive with a well tuned tracing collector. However because it uses a free list, its performance still lags the best performing generational collectors.

Reference Counting Immix Free-list allocation divides memory into cells of various fixed sizes [Wilson et al., 1995]. It allocates an object into a free cell in the smallest size class that accommodates the object instead of following the allocation order. A free-list allocator suffers from fragmentation and poor cache locality [Blackburn et al., 2004a]. Moreover, unlike tracing, reference counting is a local operation, which suggests that it is not possible to move objects to mitigate fragmentation. For state-of-the-art reference counting, we identify the free-list heap organization as the major performance bottleneck due to the poor locality it induces of the application and the instruction load of piecemeal zeroing. We identify two opportunities for copying with reference counting. We introduce a new collector, RC Immix, that replaces the free-list heap organization of reference counting with the line and block heap structure introduced by the Immix collector [Blackburn and McKinley, 2008]. Immix is a mark-region collector that uses a simple bump pointer to allocate objects

into regions of contiguous memory where objects may span lines in a block, but not blocks. RC Immix also integrates Immix's opportunistic copying with reference counting to mitigate fragmentation. This is the first collector to perform copying with pure reference counting. The ultimate outcome is that RC Immix outperforms the highly tuned production generational Immix collector in Jikes RVM [Blackburn and McKinley, 2008].

Fast Conservative Garbage Collection We next examine conservative garbage collection for safe languages in a modern context and surprisingly find that two sources of overhead associated with conservatism — excess retention and the number of pinned objects — are normally very low (less than 1%). We show that these modest overheads are dominated by the consequences of the prior work's reliance on non-copying free list and page grained pinning, and that for significant use cases, such as in managed languages, these designs are needlessly pessimistic. We identify Immix's line and block structure and opportunistic copying as a good match for conservative collection. Most of the existing conservative collectors use a non-moving free list, or pin at page granularity [Bartlett, 1988; Smith and Morrisett, 1998; WebKit, 2014] whereas Immix can pin at line granularity, reducing the amount of pinned heap space by an order of magnitude. We use a low overhead object map to determine the validity of ambiguous references. We introduce the design and implementation of the conservative variants of existing garbage collectors that leverage these mechanisms. We implemented conservative reference counting and conservative variants of Immix collectors with very low performance penalty compared to their exact counterparts. We also introduce the design and implementation of a high performance conservative collector, RC Immix$_{cons}$, that combines RC Immix with conservative root scanning. RC Immix$_{cons}$ matches the performance of a highly optimized generational copying collector. As far as we know, this is the first conservative reference counting collector and the first conservative collector to match the performance of a high performance generational copying collector.

1.3 Meaning

Today reference counting and conservative garbage collection are widely used, but generally in non-performance critical settings because their implementations suffer significant performance overheads. We improve the performance of both reference counting and conservative garbage collection significantly to the point where they are competitive with the best copying generational tracing collectors. With these advancements, language implementers now have a much richer choice of implementation alternatives both algorithmically (reference counting or tracing) and implementation-wise (exact or conservative) that can all achieve high performance. These insights and advances are likely to particularly impact the development of new and emerging languages, where the implementation burden of tracing and exactness is often the critical factor in the first implementation of a language.

1.4 Thesis Outline

The body of this thesis is structured around the three key contributions outlined above. Chapter 2 provides an overview of garbage collection and surveys relevant garbage collection literature. It provides more detailed background on previous reference counting optimizations and conservative garbage collectors. Chapter 3 discusses our experimental methodology.

Chapters 4, 5, and 6 comprise the main body of the thesis, covering the three key contributions. Chapter 4 identifies the key design choices for reference counting, provides a detailed quantitative analysis with respect to those design points, introduces novel optimizations guided by the analysis that eliminates reference counting operations for short lived young objects, and makes its performance competitive with full heap mark-sweep tracing collector. Chapter 5 identifies the free-list heap organization as the remaining major performance bottleneck for reference counting and introduces a new collector, RC Immix, that replaces the free list with the line and block organization of Immix, performs copying to mitigate fragmentation and as a result outperforms a highly tuned production generational collector. Chapter 6 first computes the direct cost of conservativeness, finding that conservative roots do not induce very much excess retention, 1% or less for our Java benchmarks. It then identifies the line and block heap organization and opportunistic copying mechanism in the Immix collector as a good match for conservative collection, and introduces a low overhead object map to validate ambiguous references. It introduces conservative reference counting and conservative Immix collector using these mechanisms with very low performance penalty, and develops a high performance conservative garbage collector, RC Immix$_{cons}$, that matches performance of a generational copying collector.

Finally Chapter 7 concludes the thesis, describing how the contributions have identified, quantified, and addressed the challenges of achieving high performance reference counting and high performance conservative garbage collection. It further identifies key future directions for research.

Chapter 2

Background

This chapter provides background information on garbage collection basics, reference counting, Immix garbage collection, and conservative garbage collection to place the research contributions in context.

This chapter starts with a brief introduction to the field of garbage collection in Section 2.1. Section 2.2 outlines the garbage collection terminology, Section 2.3 outlines the key components of garbage collection, and Section 2.4 describes the canonical garbage collectors. Section 2.5 provides a detailed background on reference counting with different optimizations over the last 50 years or so. Section 2.6 provides an overview of the Immix collector. Section 2.7 provides background on conservative garbage collection.

2.1 Overview of Garbage Collection

Garbage collection is an integral part of modern programming languages. It frees the programmer from manually dealing with memory deallocation for every object they create. Garbage collection was introduced in LISP [McCarthy, 1960] and it has gained popularity through Java and .NET. It is also included in languages such as Haskell, JavaScript, PHP, Perl, Python, and Smalltalk. For a more complete discussion of the fundamentals of garbage collection see [Jones et al., 2011; Wilson, 1992].

Programs require data to execute and this data is typically stored in memory. Memory can be allocated: a) statically where memory requirements for the data are fixed ahead-of-time, b) on the stack where the lifetime of the data is tightly bound with the currently executing method, and c) dynamically, where memory requirements are determined during execution – potentially changing between individual executions of the same program. Dynamically allocated memory can be managed either explicitly or automatically by the program. Popular programming languages, such as C/C++ require the programmer to explicitly manage memory through primitives such as the C function *malloc* and *free*, which is tedious and error-prone. Managed languages, such as Java/.NET use a garbage collector to automatically free memory.

The purpose of garbage collection is to reclaim memory that is no longer in use by the program. Determining precisely when an object will no longer be accessed is

difficult in general, so garbage collectors usually rely on reachability to conservatively approximate liveness. An object is reachable if it is transitively reachable from the running program state. Objects that are not reachable are garbage. A garbage collector differentiates between objects in the program that are no longer reachable and thus the program is guaranteed never to access (garbage/dead) and objects that are reachable (non-garbage/live). The collector then frees the memory that garbage objects are occupying.

2.2 Terminology

The area of memory used for dynamic object allocation is known as the **heap**. The process of reclaiming unused memory is known as **garbage collection**, a term coined by McCarthy [1960]. Following Dijkstra et al. [1976], from the point of view of the garbage collector, the term **mutator** refers the application or program that mutates the heap. Collectors that must stop the mutator to perform collection work are known as **stop the world** collectors, as compared to **concurrent** or **on-the-fly** collectors which reclaim objects while the application continues to execute. Collectors that employ more than one thread to do the collection work are **parallel** collectors. A parallel collector can either be stop the world or concurrent. The term **mutator time** is used to denote the time when the mutator is running and the term **GC time** is used to denote the time when the garbage collector is running. A garbage collector that checks the liveness of all objects in the heap at each collection is known as a **full heap** collector, as compared to a **generational** or **incremental** collector which may collect only part of the heap. Some garbage collectors require interaction with the running mutator. These interactions are generally implemented with **barriers**. A barrier is inserted by the compiler on every read or write to track reference read or mutation. The most common form of barrier is a **write barrier**, which is invoked whenever the mutator writes to a reference in the heap. Some collectors require knowledge of the **runtime roots**, all references into the heap held by runtime including stacks, registers, statics, and JNI.

2.3 Garbage Collection Algorithms

Automatic memory management consists of three key components: a) object allocation, b) garbage identification, and c) garbage reclamation. Different garbage collection algorithms employ different approaches to handle each of the components.

2.3.1 Object Allocation

The allocator plays a key role in mutator performance since it determines the placement and thus locality of objects. There are two techniques used for object allocation — contiguous allocation and free-list allocation.

Contiguous Allocation Contiguous memory allocation appends new objects by incrementing a pointer by the size of the new object [Cheney, 1970]. The allocator places objects contiguously in memory in allocation order. Such allocators are simple and fast at allocation time and provide excellent locality to the mutator because objects allocated together in time are generally used together [Blackburn et al., 2004a]. Allocating them contiguously thus provides spatial cache line and page reuse. Contiguous allocation generally performs synchronized allocation of larger chunks [Garthwaite and White, 1998; Alpern et al., 1999; Berger et al., 2000], which are then assigned to a single allocation thread that performs fast and unsynchronized contiguous allocation within the chunk.

Free-list Allocation Free-list allocation divides memory into cells of various fixed sizes [Wilson et al., 1995], known as a free list. Each free list is unique to a size and is composed from blocks of contiguous memory. Free-list allocation allocates an object into a free cell in the smallest size class that accommodates the object. So it places objects in memory based on their size and free memory availability rather than allocation order. When an object becomes free, the allocator returns the cell containing the object to the free list for reuse. Free-list allocation suffers two notable shortcomings. First, it is vulnerable to fragmentation of two kinds. It suffers from internal fragmentation when objects are not perfectly matched to the size of their containing cell, and it suffers external fragmentation when free cells of particular sizes exist, but the allocator requires cells of another size. Second, it suffers from poor locality because it often positions contemporaneously allocated objects in spatially disjoint memory [Blackburn et al., 2004a].

2.3.2 Garbage Identification

There are two techniques for identifying garbage — reference counting and tracing. All garbage collection algorithms in the literature use one of these techniques for garbage identification.

Reference Counting Reference counting directly identifies garbage. It keeps track for each object a count of the number of incoming references to it held by other objects, known as a *reference count*. When a reference count falls to zero, the associated object is garbage. It is incomplete because it cannot detect cycle of garbage.

Tracing Tracing indirectly identifies garbage by directly identifying all live objects. It performs a transitive closure over the object graph, starting with the *roots* – references in the stacks, registers, statics and JNI. It identifies each visited object as live. All objects that were not visited during the trace are identified as garbage.

2.3.3 Garbage Reclamation

Once the collector identifies the garbage objects there are several techniques that reclaim the space.

Direct to free list With reference counting, the collector may directly return the space used by garbage objects to a free list when an objects reference count falls to zero.

Sweep A sweep traverses over all the allocated objects in the heap, freeing the space associated with dead objects. For tracing garbage collection approaches that indirectly identify garbage, some form of sweep is required to free the space of the garbage objects. A sweep may operate over individual free list cells and larger regions of memory.

Compaction Compaction rearranges live objects within a region to create larger regions of free memory for future allocation. A classic example is sliding compaction where all live objects are compressed into a contiguous region of used memory, leaving a contiguous region of memory free for future allocation [Styger, 1967].

Evacuation Live objects can be evacuated from a region of memory and copied into another, to create an entire evacuated free region for future allocation. Evacuation requires two different regions of memory. Evacuation can be particularly effective when there are very few survivors since the cost is proportional to moving the survivors, and naturally aligns itself with tracing as the identification approach.

2.4 Canonical Garbage Collectors

The canonical garbage collectors use the above approaches for object allocation, garbage identification, and object reclamation.

2.4.1 Mark-Sweep

The first garbage collection algorithm was created as part of the LISP system [McCarthy, 1960], and is today known as mark-sweep. Mark-sweep is a tracing collector that runs in two simple phases. The *mark* phase performs a transitive closure over the object graph, marking each object as it is visited. The *sweep* phase performs a scan over all of the objects. If an object is not marked, it is unreachable and can be collected. On the other hand, if an object is marked, it is reachable, so the collector can clear the mark and retain it. It is possible for part of this sweeping phase to be performed by the mutator, a technique called *lazy sweeping* [Hughes, 1982]. Lazy sweeping reduces garbage collection pause time in stop the world collectors and can improve overall performance, due to improved cache behavior as sweep operations and subsequent allocations are performed on the same page in quick succession.

Mark-sweep is very efficient at collection time, the fastest and simplest full heap garbage collection mechanism available. This advantage is offset by slow downs it imposes on the mutator. Its free-list allocator forces the mutator to allocate objects in the discovered holes surrounding live objects. This policy often results in lower allocation performance, but the dominant effect is poor locality of reference due to objects being spread over the heap. Mark-sweep also suffers from fragmentation, where the total available memory may be sufficient to support an allocation request, but an empty, contiguous region of sufficient size may not be available.

2.4.2 Reference Counting

Reference counting was the second garbage collection algorithm published, also for the LISP system [Collins, 1960]. Reference counting operates under a fundamentally different strategy from that of tracing garbage collectors. Reference counting tracks for each object a count of the number of incoming references to it held by other objects, termed as the *reference count*. Whenever a reference is created or copied, the collector increments the reference count of the object it references, and whenever a reference is destroyed or overwritten, the collector decrements the reference count of the object it references. Write barriers implement counting. If an object's reference count reaches zero, the object has become inaccessible, and the collector reclaims the object and decrements the reference count of all objects referenced by that reclaimed object. Removing a single reference can potentially lead to many objects being freed. A common variation of the algorithm allows reference counting to be made incremental; instead of freeing an object as soon as its reference count becomes zero, it is added to a list of unreferenced objects, and periodically one or more items from this list are freed [Weizenbaum, 1969]. This naive algorithm is simple, inherently incremental, and requires no global computation. The simplicity of this naive implementation is particularly attractive and thus widely used, including in well-established systems such as PHP, Perl, and Python.

Reference counting has two clear limitations. It is unable to collect cycles of garbage because a cycle of references will self-sustain non-zero reference counts. Moreover, naive implementation of reference counting performs increment and decrement operations on every reference operation, including those to variables in stacks and registers. This overhead is further increased on multi-threaded systems, because the collector must perform reference count updates atomically to maintain correct counts. Prior optimizations overcome some of these limitations (see Section 2.5), and this thesis overcomes the remaining performance limitations.

2.4.3 Semi-Space

Semi-space collection is a tracing collector that uses evacuation to reclaim memory. The available heap memory is divided into two equal sized regions. During the execution of the program, the program allocates and uses objects in one region while the other region is empty. When garbage collection is triggered, the region

containing objects is labeled as *from-space* and the empty region is labeled as *to-space*. The garbage collector performs a transitive closure over the object graph as in a mark-sweep collector. But instead of simply marking the object, each live object is copied when it is first encountered from the from-space to the to-space. It leaves a forwarding pointer to the new location in the old location. The collector then updates all references to the old location so that they point to the copied objects in the to-space. At the end of the collection, all reachable objects now reside in the to-space and the collector reclaims the whole from-space. At the start of each collection, roles of the spaces are switched. Initial implementations of semi-space collectors used a recursive algorithm [Minsky, 1963; Fenichel and Yochelson, 1969], which has unbound depth, but Cheney [1970] later implemented a simple iterative algorithm.

Semi-space collection makes less efficient use of memory because it must reserve half of the total memory to ensure there is space to copy all objects in the worst case. It can be expensive in collection time when all live objects need to be copied, but if very few objects survive it is efficient. It has good allocation performance and induces good mutator locality because contemporaneously allocated objects are allocated contiguously. During copying, the semi-space collector automatically compacts the heap, so it does not suffer from fragmentation.

2.4.4 Mark-Compact

Mark-compact collection aims to combine the benefits of both semi-space and mark-sweep collection. It addresses the fragmentation often seen in the mark-sweep collection by compacting objects into contiguous regions of memory. But it does so in place rather than relying on the large reserved space required by the semi-space collection. While this in-place transition saves space, it involves significant additional collection effort because it must traverse objects many times. The simplest form is sliding compaction, originally implemented in LISP-2 as a four phase algorithm [Styger, 1967]. In the first phase, the collector performs a transitive closure over the object graph, marking objects as they are visited. In the second phase, the collector calculates the future location of each marked object and remembers that location for each object. In the third phase, the collector updates all references to reflect the addresses calculated in the second phase. In the fourth and final phase, the collector copies objects to their new locations. Copying is done in strict address order to ensure that no live data is overwritten.

The additional phases of simple compaction algorithms make them significantly more expensive than simple mark-sweep or semi-space collection. While optimized versions of mark-compact collectors exist, they are rarely used as the primary collector in high performance systems. However, compaction is commonly combined with mark-sweep collection to provide a means to escape fragmentation issues, and is sometimes used alongside semi-space collection to allow execution to continue when memory is tight [Sansom, 1991]. Compaction does, however, have the advantage of excellent mutator locality because it preserves allocation order, and has very low space overheads.

2.4.5 Mark-Region

Mark-region collection combines contiguous allocation of semi-space with the collection strategy of mark-sweep. The motivation is to achieve the mutator performance of semi-space and the collection performance of mark-sweep. In terms of allocation, mark-region is similar to semi-space, with objects allocated into contiguous regions of memory. In terms of collection, mark-region is similar to mark-sweep, but instead of sweeping individual objects it sweeps regions; regions with no reachable objects are made available for allocation. A mark-region collection consists of two phases. The *mark* phase performs a transitive closure over the object graph, it marks objects as well their regions as they are visited. The *sweep* phase scans all the regions, and regions that were not marked in the mark phase are made available for future allocation. Immix provides the first detailed analysis and description of a mark-region collector [Blackburn and McKinley, 2008], although a mark-region approach was previously used in Oracle's JRockit and IBM's production virtual machines.

Mark-region collection combines excellent allocation performance and good collection performance, with good mutator locality. But it is susceptible to issues with fragmentation, because a single live object may keep an entire region alive and unavailable for reuse. To combat this problem, mark-region collectors often employ techniques to relocate objects in memory to reduce fragmentation. The JRockit collector performs compaction of the heap at each collection, the IBM collector performs whole heap compaction when necessary. Immix combats this problem by defining hierarchy of two regions (blocks divided into lines) and then adds lightweight defragmentation as required when memory is in demand. Immix is described further in Section 2.6.

2.4.6 Generational

Generational garbage collection is based on the weak generational hypothesis that 'most object dies young' [Ungar, 1984; Lieberman and Hewitt, 1983] and was at that time the most important advancement in garbage collection since the first collectors were developed in 1960. Generational collectors divide the heap into regions for objects of different ages, and perform more frequent collections on more recently allocated objects and less frequent collections on the oldest objects. The youngest generation is generally known as the *nursery* and a *minor* collection collects only the nursery. The space containing the oldest objects is known as the *mature* space. A full heap *major* collection collects both young and old generation. During a minor collection, generational collectors must remember all references from the mature space into the nursery and assume they are live. A *generational write barrier* takes note of references from older generations to younger generations, keeping them in a *remembered set* for use during minor collections. The remembered set in combination with the standard root set provide the starting point for a transitive closure across all live objects within the nursery. A partial copying collection can be performed during this closure, with live objects evacuated from the nursery into the mature space. When the weak generational hypothesis holds, this collection is very efficient

because only a small fraction of nursery objects survive and must be copied into the mature space. Various mature space strategies are possible.

In generational collection, objects are allocated in the nursery contiguously, providing good allocation performance and better mutator locality. The collection performance is also very good compared to full heap collectors. The majority of high performance collectors (e.g., HotSpot, J9, and .NET) are generational collectors.

In the next section, we describe prior work on reference counting with different optimizations on which we build.

2.5 Reference Counting Garbage Collection

The first account of reference counting was published by George Collins in 1960, just months after John McCarthy described tracing garbage collection [McCarthy, 1960]. Reference counting directly identifies dead objects by keeping a count of the number of references to each object, freeing the object when its count reaches zero.

2.5.1 Naive Reference Counting

Collins' simple, *immediate* reference counters count *all* references, in the heap, stacks, registers and local variables. The compiler inserts increments and decrements on referents where ever references are created, copied, destroyed, or overwritten. Because such references are very frequently mutated, immediate reference counting has a high overhead. However, immediate reference counting needs very minimal runtime support, so is a popular implementation choice due to its low implementation burden. The algorithm requires just barriers on every pointer mutation and the capacity to identify all pointers within an object when the object dies. The former is easy to implement, for example compilers for statically and dynamically typed languages directly and easily identify pointer references, as do *smart pointers* in C++; while the latter can be implemented through a destructor. Objective-C, Perl, Delphi, PHP, and Swift [Stein, 2003; Thomas et al., 2013; Apple Inc., 2013, 2014] use naive reference counting.

Collins' reference counting algorithm suffers from significant drawbacks including: a) an inability to collect cycles of garbage, b) overheads due to tracking frequent stack pointer mutations, c) overheads due to storing the reference count, and d) overheads due to maintaining counts for short lived objects. We now briefly outline five important optimizations developed over the past fifty years to improve over Collins' original algorithm.

2.5.2 Deferral

To mitigate the high cost of maintaining counts for rapidly mutated references, Deutsch and Bobrow [1976] introduced deferred reference counting. Deferred reference counting ignores mutations to frequently modified variables, such as those stored in reg-

isters and on the stacks. Deferral requires a two phase approach, dividing execution into distinct mutation and collection phases. This tradeoff reduces reference counting work significantly, but delays reclamation. Since deferred references are not accounted for during the mutator phase, the collector counts other references and places zero count objects in a zero count table (ZCT), deferring their reclamation. Periodically in a GC reference counting phase, the collector enumerates all deferred references from the stacks and registers into a root set and then reclaims any object in the ZCT that is not a referent of the root set.

Bacon et al. [2001] eliminate the zero count table by buffering decrements between collections. Initially the buffered decrement set is empty. At collection time, the collector temporarily increments a reference count to each object in the root set and then processes all of the buffered decrements. Deferred reference counting performs all increments and decrements at collection time. At the end of the collection, it adds a buffered decrement for every root. Although much faster than naive immediate reference counting, deferred reference counting typically uses stack maps [2.7.1] to enumerate all live pointers from the stacks. Stack maps are an engineering impediment, which discourages many reference counting implementations from including deferral [Jibaja et al., 2011].

2.5.3 Coalescing

Levanoni and Petrank [2001, 2006] observed that all but the first and last in any chain of mutations to a reference within a given window can be *coalesced*. Only the initial and final states of the reference are necessary to calculate correct reference counts. Intervening mutations generate increments and decrements that cancel each other out. This observation is exploited by remembering (logging) only the initial value of a reference field when the program mutates it between periodic reference counting collections. At each collection, the collector need only apply a decrement to the initial value of any overwritten reference (the value that was logged), and an increment to the latest value of the reference (the current value of the reference).

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

2.5.4 Ulterior

Blackburn and McKinley [2003] introduced *ulterior reference counting*, a hybrid collector that combines copying generational collection for the young objects and reference counting for the old objects. They observe that most mutations are to the nursery objects. Ulterior reference counting extends deferral to include all nursery objects. It restricts copying to nursery object and reference counting to old objects, the object demographics for which they perform well. It safely ignores mutations to select heap objects. Ulterior reference counting is not difficult to implement, but the implementation is a hybrid, and thus manifests the implementation complexities of both a standard copying nursery and a reference counted heap. Azatchi and Petrank [2003] independently proposed the use of reference counting for the old generation and tracing for the young generation. During a nursery collection, their collector marks all live nursery objects, and sweeps the rest. Their reclamation is thus proportional to the entire nursery size, rather than the survivors as in a copying nursery. They also explored the use of the reference counting buffers of update coalescing as a source of inter-generational pointers. This collector is also hybrid, so manifests the implementation complexities of two orthodox collectors.

2.5.5 Age-Oriented

Paz et al. [2005] introduced *age oriented* collection, which aimed to exploit the generational hypothesis that most objects die young. Their age-oriented collector uses a reference counting collection for the old generation and a tracing collection for the young generation that establishes reference counts during tracing. This collector provides a significant benefit as it avoids performing expensive reference counting operations for the many young objects that die. It does not perform copying. Like ulterior reference counting, this collector is a hybrid, so manifests the implementation complexities of two orthodox collectors.

2.5.6 Collecting Cyclic Objects

Reference counting suffers from the problem that cycles of objects will sustain non-zero reference counts, and therefore cannot be collected. There exist two general approaches to deal with cyclic garbage: *backup tracing* [Weizenbaum, 1969; Frampton, 2010] and *trial deletion* [Christopher, 1984; Martinez et al., 1990; Lins, 1992; Bacon and Rajan, 2001; Paz et al., 2007]. Paz et al. [2007] compared backup tracing with trial deletion and found that backup tracing performed slightly better and predicted in future trial deletion will outperform backup tracing. Later Frampton [2010] conducted a detailed study of cycle collection and showed that backup tracing, starting from the roots, performs significantly better than trial deletion and has more predictable performance characteristics.

2.5.6.1 Backup Tracing

Backup tracing performs a mark-sweep style trace of the entire heap to eliminate cyclic garbage. The only key difference to a classical mark-sweep is that during the sweep phase, decrements must be performed from objects found to be garbage for their descendants into the live part of the heap. To support backup tracing, each object needs to be able to store a mark state during tracing. Backup tracing can also be used to restore *stuck* reference counts due to use of limited bits to store the count. Backup tracing must enumerate live root references from the stacks and registers, which requires stack maps. For this reason, naive reference counting implementations usually do not perform backup tracing.

2.5.6.2 Trial Deletion

Trial deletion collects cycles by identifying groups of self-sustaining objects using a partial trace of the heap in three phases. In the first phase, the sub-graph rooted from a selected candidate object is traversed, with reference counts for all outgoing pointers (temporarily) decremented. Once this process is complete, reference counts reflect only external references into the sub-graph. If any object's reference count is zero then that object is only reachable from within the sub-graph. In the second phase, the sub-graph is traversed again, and outgoing references are incremented from each object whose reference count did not drop to zero. Finally, the third phase traverses the sub-graph again, sweeping all objects that still have a reference count of zero. The original implementation was due to Christopher [1984] and has been optimized over time [Martinez et al., 1990; Lins, 1992; Bacon and Rajan, 2001; Paz et al., 2007].

Lins [1992] performed cycle detection lazily, periodically targeting the set of candidate objects whose counts experienced decrements to non-zero values. Bacon and Rajan [2001] performed each phase over all candidates in a group after observing that performing the three phases for each candidate sequentially like Lins could exhibit quadratic complexity. Bacon and Rajan also used simple static analysis to exclude the processing of objects they could identify as inherently acyclic. Paz et al. [2007] combines the work of Bacon and Rajan [2001] with update coalescing that significantly reduces the set of candidate objects. They also develop new filtering techniques to filter out a large fraction of the candidate objects. They also integrate their cycle collector with the age oriented collector [2.5.5] that greatly reduces the load on the cycle collector.

2.5.7 Reference Counting and the Free List

Free lists support immediate and fast reclamation of individual objects, which makes them particularly suitable for reference counting. Other systems, such as evacuation and compaction, must identify and move live objects before they may reclaim any memory. Also, free lists are a good fit to backup tracing used by many reference

counters. Free lists are easy to *sweep* because they encode free and occupied memory in separate metadata. The sweep identifies and retains live objects and returns memory occupied by dead objects to the free list.

Blackburn et al. [2004a] show that contiguous allocation in a copying collector delivers significantly better locality than free-list allocation in a mark-sweep collector. When contiguous allocation is coupled with copying collection, the collector must update all references to each moved object [Cheney, 1970], a requirement that is at odds with reference counting's local scope of operation. Because reference counting does not perform a closure over the live objects, in general, a reference counting collector does not know of and therefore cannot update all pointers to an object it might otherwise move. Thus far, this prevented reference counting from copying and using a contiguous allocator.

This section provided a detailed overview on how reference counting works, its shortcomings, and different optimizations to overcome some of them. This section also explained why reference counting uses a free list for allocation. These are the necessary background for Chapter 4. In the next section, we describe Immix garbage collection.

2.6 Immix Garbage Collection

As introduced in Section 2.4.5, mark-region memory managers use a simple bump pointer to allocate objects into regions of contiguous memory [Blackburn and McKinley, 2008]. A tracing collection marks each object and marks its containing region. Once all live objects have been traced, it reclaims unmarked regions. This design addresses the locality problem in free-list allocators. A mark-region memory manager can choose whether to move surviving objects or not. By contrast, evacuating and compacting collectors must copy, leading them to have expensive space (semi-space) or time (mark-compact) collection overheads compared to mark-sweep collectors. Mark-region collectors are vulnerable to fragmentation because a single live object may keep an entire region alive and unavailable for reuse, and thus must copy some objects to attain space efficiency and thus performance.

2.6.1 Heap Organization: Lines and Blocks

Immix is a mark-region collector that uses a region hierarchy with two sizes: lines, which target cache line locality, and blocks, which target page level locality [Blackburn and McKinley, 2008]. Each block is composed of lines, as shown in Figure 2.1. The bump pointer allocator places new objects contiguously into empty lines and skips over occupied lines. Objects may span lines, but not blocks. Several small objects typically occupy a line. Immix uses a bit in the header to indicate whether an object straddles lines, for efficient line marking. Immix recycles free lines in partially free blocks, allocating into them first. Figure 2.1 shows how Immix allocates objects contiguously in empty lines and blocks.

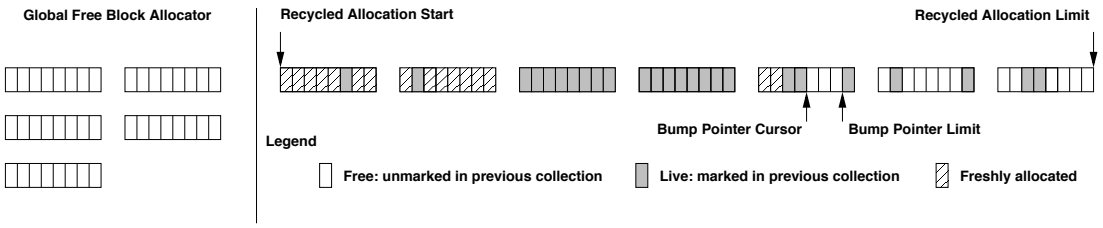


Figure 2.1: Immix Heap Organization [Blackburn and McKinley, 2008]

2.6.2 Opportunistic Copying

Immix tackles fragmentation using *opportunistic* defragmentation, which mixes marking with copying. At the beginning of a collection, Immix determines whether to defragment, e.g., based on fragmentation levels. Blocks with free lines at the start of a collection indicate fragmentation because although available, the memory was not usable by the mutator. Furthermore, the live/free status for these blocks is up-to-date from the prior collection. During a defragmenting collection, Immix uses two bits in the object header to differentiate between marked and forwarded objects. At the beginning of a defragmenting collection, Immix identifies source and target blocks based on the statistics from the previous collection. During the mark trace, when Immix first encounters an object that resides on a source block and there is still available memory for it on a target block, Immix copies the object to a target block, leaving a forwarding pointer. Otherwise Immix simply marks the object as usual. When Immix encounters forwarded objects while tracing, it updates the reference accordingly. This process is opportunistic, since it performs copying until it exhausts free memory to defragment the heap. The result is a collector that combines the locality of a copying collector and the collection efficiency of a mark-sweep collector with resilience to fragmentation. The best performing production collector in Jikes RVM is generational Immix (Gen Immix), which consists of a copying nursery space and an Immix old space [Blackburn and McKinley, 2008].

This section provided a detailed overview of the Immix garbage collection. It described the line and block heap organization of Immix with contiguous allocation and the opportunistic copying mechanism to mitigate fragmentation. These are the necessary background for Chapter 5. In the next section, we describe conservative garbage collection.

2.7 Conservative Garbage Collection

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. An *exact* garbage collector precisely identifies root ref-

erences 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.

2.7.1 Requirements and Complexities of 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 [Agesen et al., 1998; Hirzel et al., 2002]. Precise pointer tracking burdens the compiler with significant bookkeeping in optimizations and intermediate representations [Diwan et al., 1992]. It also inhibits some 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 [Blackburn et al., 2004a; Blackburn and McKinley, 2008].

Uncooperative exact Exact uncooperative systems are also in use for strongly typed languages that are implemented with C as an intermediate language [Henderson, 2002; Baker et al., 2009; LLVM, 2014]. 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.

2.7.2 Ambiguous References

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

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.

2.7.3 Non-Moving Conservative Collectors

The most mature, widely used, and adopted conservative garbage collector is the Boehm-Demers-Weiser (BDW) collector [Boehm and Weiser, 1988]. 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 [Demers et al., 1990].

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 object 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 [Blackburn et al., 2004a; Blackburn and McKinley, 2008].

For languages with object type precision, a non-moving collector design is overly restrictive since, as we will show in Chapter 6, most heap objects will not be the target of ambiguous references.

2.7.4 Mostly Copying Collectors

Bartlett [1988] first described a *mostly copying* conservative collector for memory safe languages, a design that has remained popular [Hosking, 2006; Smith and Morrisett, 1998; Attardi and Flagella, 1994; WebKit, 2014]. 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 and Flagella [1994] 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.

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 [Bartlett, 1988; Hosking, 2006; WebKit, 2014]. Hosking's [Hosking, 2006] 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 [1998] 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.

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.

This section provided detailed overview of the mechanisms and requirements for conservative and exact garbage collection. It described the implications of conservative garage collection due to ambiguous references — excess retention, pinning, and filtering. It also described existing non-moving and mostly copying conservative collectors. These are the necessary background for Chapter 6.

2.8 Summary

This chapter introduced key background material. We discussed reference counting with different existing optimizations, which provides necessary background for Chapter 4. We discussed the Immix garbage collection, which along with reference counting provides necessary background for Chapter 5. We further discussed conservative garbage collection, which provides necessary background for Chapter 6. Before we move to the primary contributions of the thesis, we next give an overview of our experimental methodology.

Chapter 3

Experimental Methodology

This section presents software, hardware, and measurement methodologies that we use throughout the evaluations presented in this thesis.

3.1 Benchmarks

We draw 20 benchmarks from DaCapo [Blackburn et al., 2006], SPECjvm98 [SPEC, 1999], and pjbb2005 [Blackburn et al., 2005]. The pjbb2005 benchmark is a fixed workload version of SPECjbb2005 [SPEC, 2006] with 8 warehouses that executes 10,000 transactions per warehouse. We do not use SPECjvm2008 because that suite does not hold workload constant, so is unsuitable for GC evaluations unless modified. Since a few DaCapo 9.12 benchmarks do not execute on our virtual machine, we use benchmarks from both 2006-10-MR2 and 9.12 Bach releases of DaCapo to enlarge our suite.

We omit two outliers, mpegaudio and lusearch, from our figures and averages, but include them grayed-out in tables, for completeness. The mpegaudio benchmark is a very small benchmark that performs almost zero allocation. The lusearch benchmark allocates at three times the rate of any other. The lusearch benchmark derives from the 2.4.1 stable release of Apache Lucene. Yang et al. [2011] found a performance bug in the method `QueryParser.getFieldQuery()`, which revision r803664 of Lucene fixes [Seeley, 2009]. The heavily executed `getFieldQuery()` method unconditionally allocated a large data structure. The fixed version only allocates a large data structure if it is unable to reuse an existing one. This fix cuts total allocation by a factor of eight, speeds the benchmark up considerably, and reduces the allocation rate by over a factor of three. We patched the DaCapo lusearch benchmark with just this fix and we call the fixed benchmark lusearch-fix.

3.2 Software Platform

3.2.1 Jikes RVM and MMTk

We use Jikes RVM and MMTk for all of our experiments. Jikes RVM [Alpern et al., 2000, 2005] is an open source high performance Java virtual machine (VM) written

in a slightly extended version of Java. We use Jikes RVM release 3.1.3+hg r10761 to build each of our systems. 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 [Blackburn et al., 2004a].

All of the garbage collectors we evaluate are parallel [Blackburn et al., 2004b]. They use thread local allocation for each application thread to minimize synchronization. During collection, the collectors exploit available software and hardware parallelism [Cao et al., 2012]. 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 have a bytecode interpreter. Instead, a fast template-driven baseline compiler produces machine code when the VM first encounters each Java method. The adaptive compilation system then judiciously optimizes the most frequently executed methods. Using a timer-based approach, it schedules periodic interrupts. At each interrupt, the adaptive system records the currently executing method. Using a cost model, it then selects frequently executing methods it predicts will benefit from optimization. The optimizing compiler compiles these methods at increasing levels of optimizations.

3.2.2 Performance Analysis

To reduce perturbation due to dynamic optimization and to maximize the performance of the underlying system that we improve, we use a *warmup replay* methodology. Before executing any experiments, we gathered compiler optimization profiles from the 10th iteration of each benchmark. When we perform an experiment, we execute one complete iteration of each benchmark without any compiler optimizations, which loads all the classes and resolves methods. We next apply the benchmark-specific optimization profile and perform no subsequent compilation. We then measure and report the subsequent iteration. This methodology greatly reduces non-determinism due to the adaptive optimizing compiler and improves underlying performance by about 5% compared to the prior replay methodology [Blackburn et al., 2008]. We run each benchmark 20 times (20 invocations) and report the average. We also report 95% confidence intervals for the average using Student's t-distribution.

3.2.3 Program and Collector Analyses and Statistics

To perform detailed program and collector analysis of different program and collector behaviors, such as maximum reference count and excess retention, we instrument relevant garbage collection configuration to gather information on different metrics while running the benchmarks. This instrumentation does not affect the garbage collection workload (the exact same objects are collected with or without the instrumentation). The instrumentation slows the collector down considerably, but these analyses are for understanding behaviors, not used in production executions, this slowdown is

irrelevant. We do not use the instrumentation for any of our performance studies.

3.3 Operating System

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

3.4 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.

Chapter 4

Optimizing Reference Counting

Among garbage collection algorithms, reference counting has some interesting advantages such as the possibility of simpler implementation, local scope of operation, and immediacy of reclamation. Traditional reference counting is however very slow compared to tracing and consequently has not been used in high performance systems. This chapter identifies key design choices for reference counting, presents a detailed analysis of reference counting with respect to those design choices, and then shows how to optimize reference counting. As far as we are aware, this is the first such quantitative study of reference counting. The analysis confirms that limited bit reference counting is a good idea because the majority of objects have a low reference count. The analysis also identifies that the majority of the reference count increments and decrements are due to objects that are very short lived. This chapter introduces optimizations that overlook reference counting operations for short lived young objects and significantly improve the performance of reference counting. The result is the first reference counting implementation that is competitive with a full heap mark-sweep tracing collector.

This chapter is structured as follows. Section 4.2 describes the key design choices for reference counting — storing and accurately maintaining the reference count and collecting cyclic garbages. Section 4.3 describes the quantitative analysis of those key design points for different benchmarks. Section 4.4 discusses and evaluates different choices for limited bit reference counting. Section 4.5 describes the two novel optimizations that eliminate reference counting operations for short lived young objects in detail and their effect on performance. Section 4.6 evaluates our improved reference counting with a well tuned full heap mark-sweep tracing collector and with some existing high performance collectors.

This chapter describes work published as "Down for the Count? Getting Reference Counting Back in the Ring" [Shahriyar, Blackburn, and Frampton, 2012].

4.1 Introduction

In an interesting twist of fate, the two algorithmic roots of the garbage collection family tree were born within months of each other in 1960, both in the Communications of the ACM [McCarthy, 1960; Collins, 1960]. On the one hand, reference

counting [Collins, 1960] *directly* identifies garbage by noticing when an object has no references to it, while on the other hand, tracing [McCarthy, 1960] identifies live objects and thus only *indirectly* identifies garbage (those objects that are not live). Reference counting offers a number of distinct advantages over tracing, namely that it: a) can reclaim objects as soon as they are no longer referenced, b) is inherently incremental, and c) uses object-local information rather than global computation. Nonetheless, for a variety of reasons, reference counting is rarely used in high performance settings and has been somewhat neglected within the garbage collection literature. Other than the systems developed in this thesis, we are not aware of any high performance system that relies on reference counting. However, reference counting is popular among new languages with relatively simple implementations. The latter is due to the ease with which naive reference counting can be implemented, while the former is due to reference counting's limitations. We compare the state-of-the-art reference counting implementation on which we build and a well tuned mark-sweep implementation and find that reference counting is over 30% slower than its tracing counterpart. The goal of this chapter is to revisit reference counting, understand its shortcomings, and address some of its limitations.

We identify the key design points of reference counting and evaluate the intrinsic behaviors such as maximum reference counts of Java benchmarks with respect to those design points. Our analysis of benchmark intrinsics motivates three optimizations: 1) using just a few bits to maintain the reference count, 2) eliding reference count operations for newly allocated objects, and 3) allocating new objects as dead, avoiding a significant overhead in deallocating them. We then conduct an in-depth performance analysis of reference counting and mark-sweep, including combinations of each of these optimizations. We find that together these optimizations eliminate the reference counting overhead, leading to performance consistent with high performance mark-sweep.

In summary, this chapter makes the following contributions.

1. We identify and evaluate key design choices for reference counting implementations.
2. We conduct an in-depth quantitative study of intrinsic benchmark behaviors with respect to those design choices.
3. Guided by our analysis, we introduce optimizations that greatly improve reference counting performance.
4. We conduct a detailed performance study of reference counting and mark-sweep, showing that our optimizations eliminate the overhead of reference counting.

Our detailed study of intrinsic behaviors should help other garbage collector implementers design more efficient reference counting algorithms. Our optimizations remove the performance barrier to using reference counting rather than mark-sweep, thereby making the locality and immediacy of reference counting compelling and providing the platform on which the subsequent chapters in this thesis build.

4.2 Design Space

We now explore the design space for reference counting implementations. In particular, we explore strategies for: 1) storing the reference count, 2) maintaining an accurate count, and 3) dealing with cyclic objects. We describe each of these and survey major design alternatives.

4.2.1 Storing the Count

Each object has a reference count associated with it. This section considers the choices for storing the count. This design choice is a tradeoff between the space required to store the count, and the complexity of accurately managing counts with limited bits.

Use a dedicated word per object By using a dedicated word we can guarantee that the reference count will never overflow. In a 32-bit address space, in the worst case, if every word of memory pointed to a single object, the count would only be 2^{30}. However, an additional header word has a significant cost, not only in terms of space, but also time, as allocation rate is also affected. For example, the addition of an extra 32-bit word to the object header incurs an overhead of 2.5% in total time and 6.2% in GC time, on average across our benchmark suite when using Jikes RVM's production garbage collector.

Use a field in each object's header. Object headers store information to support runtime operations such as virtual dispatching, dynamic type checking, synchronization, and object hashing. Although header bits are valuable, it may be possible to use a small number of bits to store the reference count. The use of a small number of bits means that the reference counter must handle *overflow*, where a count reaches a value too large for small number of bits. Two basic strategies to deal with overflow exist: 1) an auxiliary data structure, such as a hash table, stores accurate counts, 2) *sticky* counts, once they overflow future increments and decrements are ignored. In the latter case, one may depend on a backup tracing cycle collector to either restore counts or directly collect the object [Jones et al., 2011].

4.2.2 Maintaining the Count

There are several ways to maintain accurate reference count as mentioned in Section 2.5.

1. *Naive* immediate reference counting that counts *all* references including the heap, stacks, and registers [2.5.1]

2. *Deferral* ignores mutations to frequently modified variables, such as those stored in registers and on the stacks [2.5.2]

3. *Coalescing* ignores many mutations to heap references by observing that all but the first and last in any chain of mutations to a reference within a given window can be coalesced [2.5.3]

4. *Ulterior* reference counting combines copying generational collection for the young objects and reference counting for the old objects [2.5.4]

5. *Age-oriented* collector uses a reference counting collection for the old generation and a tracing collection for the young generation that establishes reference counts during tracing [2.5.5]

4.2.3 Collecting Cyclic Objects

Reference counting alone cannot collect cyclic garbage. There exist two general approaches to deal with cyclic garbage as mentioned in Section 2.5.6.

1. *Trial deletion* collects cycles by identifying groups of self-sustaining objects using a partial trace of the heap

2. *Backup tracing* performs a mark-sweep style trace of the entire heap to eliminate cyclic garbage

Cycle collection is not the focus of this chapter, however some form of cycle collection is essential for completeness of reference counting. We use backup tracing, which performs substantially better than trial deletion and has more predictable performance characteristics [2.5.6]. Backup tracing also provides a solution to the problem of reference counts that become stuck due to limited bits. We invoke the backup tracing cycle collector when the memory available for allocation falls below a specific threshold (in our case it is 4 MB, but other values or fractions are possible).

4.3 Analysis of Reference Counting Intrinsics

Recall that despite the implementation advantages of simple immediate reference counting, reference counting is rarely used in high performance settings because historically it has been comprehensively outperformed by tracing collectors. To help understand the sources of overhead and identify opportunities for improvement, we now study the behavior of standard benchmarks with respect to operations that are *intrinsic* to reference counting. In particular, we focus on metrics that are neither user-controllable nor implementation-specific.

4.3.1 Analysis Methodology

We instrument Jikes RVM to identify, record, and report statistics for every object allocated. We control the effect of cycle collection by performing measurements with cycle collection policies at both extremes (*always* collect cycles versus *never* collect cycles) and report when this affects the analysis. To perform our analysis, we instrument

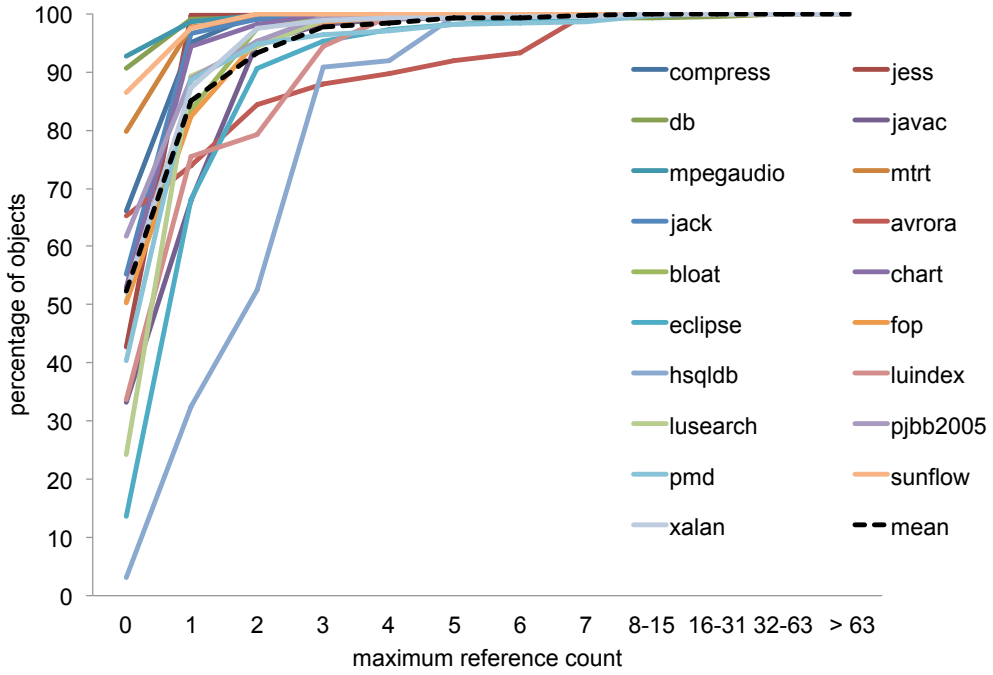


Figure 4.1: Most objects have very low maximum reference counts. This graph plots the cumulative frequency distribution of maximum reference counts among objects in each benchmark.

the standard configuration of reference counting in MMTk to gather information on different metrics while running the benchmarks. See Section 3.2.3 for more methodology details.

Note that the analysis of intrinsic properties we present in this Section *does not* depend on Jikes RVM or MMTk. The measurements we make here could have been made on any other JVM to which we had access to the source.

4.3.2 Distribution of Maximum Reference Counts

We start by measuring the distribution of *maximum reference counts*. For each object our instrumented JVM keeps track of its maximum reference count, and when the object dies we add the object's maximum reference count to a histogram. In Table 4.1 we show the cumulative maximum reference count distributions for each benchmark. For example, the table shows that for the benchmark `eclipse`, 68.2% of objects have a maximum reference count of just one, and 95.4% of all objects have a maximum reference count of three. On average, across all benchmarks, 99% of objects have a maximum reference count of six or less. The data in Table 4.1 is displayed pictorially in Figure 4.1.

max count	mean	compress	jess	db	javac	mpegaudio	mtrt	jack	avrora	bloat	chart	eclipse	fop	hsqldb	luindex	lusearch	pjbb2005	pmd	sunflow	xalan
0	**52.4**	66.1	42.7	90.6	33.2	92.7	79.8	55.2	65.3	50.5	52.9	13.6	50.3	3.1	33.7	24.2	61.8	40.4	86.5	52.2
1	**85.1**	95.0	99.7	99.1	67.8	98.7	97.7	96.5	73.8	83.6	94.3	68.2	82.5	32.4	75.4	89.3	88.9	88.6	97.5	87.3
2	**93.2**	99.9	99.8	99.2	95.0	99.8	99.1	99.1	84.3	97.5	98.2	90.6	95.3	52.6	79.3	94.4	95.3	94.9	100	97.5
3	**97.7**	100	99.9	99.3	98.2	100	99.7	99.8	87.9	99.2	100	95.4	98.7	90.9	94.4	98.9	99.5	96.3	100	98.9
4	**98.5**	100	99.9	99.3	99.1	100	99.8	100	89.7	99.5	100	97.3	99.4	92.0	99.5	100	99.7	97.0	100	99.2
5	**99.2**	100	99.9	99.3	99.4	100	99.8	100	91.9	99.9	100	98.3	99.6	99.6	99.7	100	99.8	98.2	100	99.3
6	**99.3**	100	99.9	99.3	99.4	100	99.8	100	93.3	99.9	100	98.9	99.7	99.8	99.8	100	99.9	98.5	100	99.4
7	**99.7**	100	99.9	99.3	99.5	100	99.9	100	99.9	99.9	100	99.2	99.8	99.9	99.9	100	99.9	98.7	100	99.5
8-15	**99.9**	100	100	99.4	99.7	100	99.9	100	99.9	100	100	99.8	99.9	100	100	100	100	99.8	100	99.9
16-31	**99.9**	100	100	99.5	99.9	100	100	100	99.9	100	100	99.9	100	100	100	100	100	99.9	100	99.9
32-63	**100**	100	100	100	99.9	100	100	100	99.9	100	100	100	100	100	100	100	100	100	100	100
> 63	**100**	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100

Table 4.1: Most objects have very low maximum reference counts. Here we show the cumulative frequency distribution of maximum reference counts among objects in each benchmark. For many benchmarks, 99% of objects have maximum counts of 2 or less.

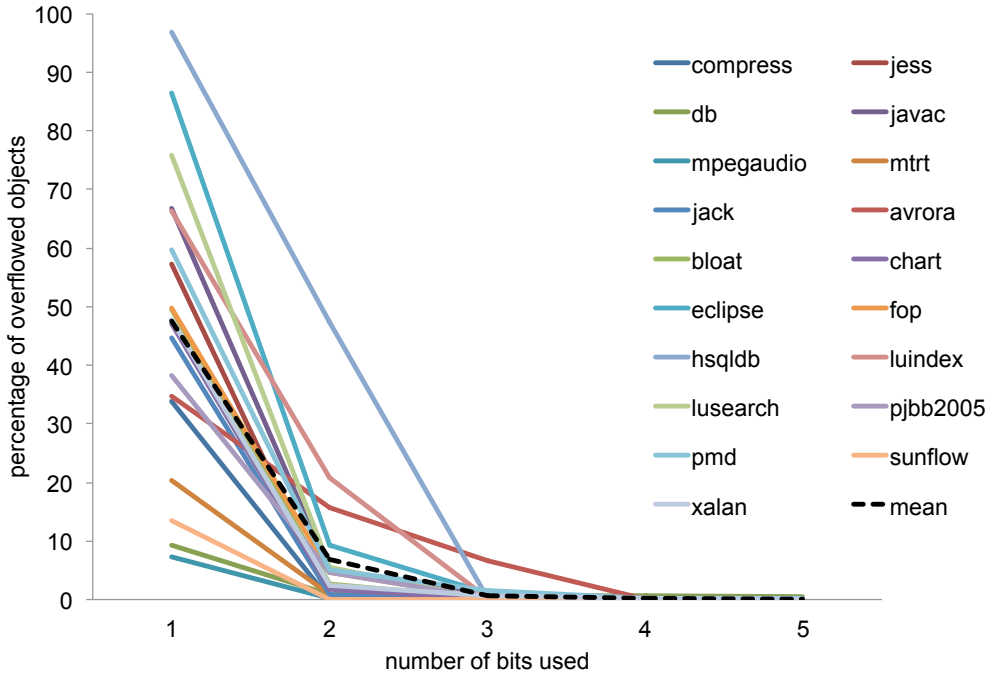


Figure 4.2: The number of objects which suffer overflowed reference counts drops off rapidly as the number of available bits grows from two to five.

4.3.3 Limited Reference Count Bits and Overflow

When the number of bits available for storing the reference count is restricted, the count may overflow. In Table 4.2 we show for different sized reference count fields, measurements of: a) the percentage of *objects* that would ever overflow, and b) the percentage of *reference counting operations* that act on overflowed objects. The first measure indicates how many objects at some time had their reference counts overflow. An overflowed reference count will either be stuck until a backup trace occurs, or will require an auxiliary data structure if counts are to be unaffected. The second measure shows how many operations occurred on objects that were already stuck, and is therefore indicative of how much overhead an auxiliary data structure may experience.

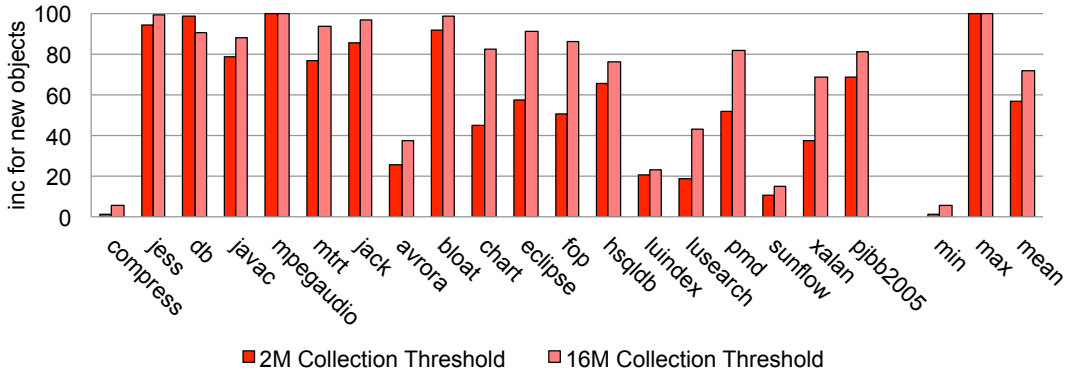
Results for reference count fields sized from one to five bits are shown in Table 4.2. For example, the table shows that when three bits are used, only 0.65% of objects experience overflow, and for compress and mpegaudio, none overflow. Although the percentage of overflowed objects is less than 1%, it is interesting to note that these overflowed objects attract nearly 23% of all increment and decrement operations, on average. Overflowed objects thus appear to be highly popular objects. The data in Table 4.2 is displayed pictorially in Figure 4.2.

bits used	mean	compress	jess	db	javac	mpegaudio	mtrt	jack	avrora	bloat	chart	eclipse	fop	hsqldb	luindex	lusearch	pjbb2005	pmd	sunflow	xalan
percentage of overflowed objects																				
1	**47.65**	33.93	57.31	9.37	66.77	7.29	20.23	44.76	34.74	49.54	47.08	86.36	49.68	96.89	66.31	75.83	38.23	59.62	13.54	47.83
2	**6.75**	0.08	0.16	0.80	4.96	0.16	0.95	0.95	15.74	2.54	1.83	9.38	4.73	47.38	20.75	5.57	4.67	5.15	0.01	2.47
3	**0.65**	0	0.08	0.68	0.59	0	0.16	0.01	6.69	0.10	0.02	1.15	0.31	0.21	0.16	0.01	0.14	1.53	0.01	0.59
4	**0.11**	0	0.06	0.68	0.28	0	0.08	0	0.06	0.05	0.01	0.24	0.10	0.01	0.01	0.01	0.02	0.26	0.01	0.17
5	**0.06**	0	0.03	0.49	0.12	0	0.03	0	0.06	0.03	0.01	0.07	0.05	0.01	0.01	0.01	0.01	0.14	0.01	0.06
percentage of increments on overflowed objects																				
1	**100**	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
2	**41.2**	8.2	77.5	96.9	19.9	2.5	35.2	3.6	28.6	71.7	17.2	25.5	20.0	39.7	63.8	16.3	68.4	61.8	84.7	41.9
3	**22.7**	0	76.8	83.6	12.2	0	29.9	0	17.1	51.3	14.1	10.9	8.9	5.0	0.9	8.0	35.3	14.5	35.0	27.3
4	**17.8**	0	75.6	55.1	9.7	0	25.8	0	14.8	36.2	13.1	7.0	7.0	4.9	0	7.4	18.9	10.1	34.3	18.2
5	**13.4**	0	73.6	16.7	7.5	0	22.2	0	11.1	20.5	11.2	5.0	5.5	4.8	0	6.5	14.2	8.6	33.0	14.5
percentage of decrements on overflowed objects																				
1	**100**	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
2	**43.0**	10.4	77.5	96.9	21.3	2.4	40.0	3.5	28.4	71.9	17.8	32.4	20.1	48.1	64.4	17.4	69.8	65.6	84.7	44.0
3	**23.3**	0	76.7	83.6	13.0	0	34.8	0	17.0	51.4	14.6	13.7	7.3	6.1	0.9	8.5	36.1	15.3	35.0	28.7
4	**18.3**	0	75.5	55.1	10.3	0	30.3	0	14.6	36.3	13.5	9.0	5.3	5.9	0	7.9	19.3	10.8	34.3	19.2
5	**13.8**	0	73.6	16.7	8.1	0	26.3	0	11.0	20.5	11.6	6.6	4.0	5.8	0	6.9	14.4	9.1	33.0	15.3

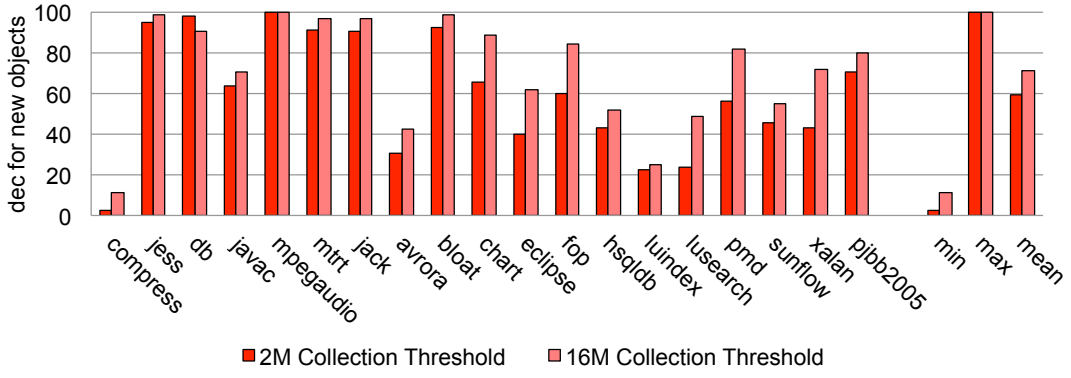
Table 4.2: Reference count overflow is infrequent when a modest number of bits are used. The top third of this table shows the number of objects which ever suffer overflow when 1, 2, 3, 4, or 5 bits are used for reference counts. The middle third shows how many increments are applied to overflowed objects. The bottom third shows how many decrements are applied to overflowed objects.

types	collection	mean	compress	jess	db	javac	mpegaudio	mtrt	jack	avrora	bloat	chart	eclipse	fop	hsqldb	luindex	lusearch	pjbb2005	pmd	sunflow	xalan
percentage of increments																					
new	2M	**57**	1.1	94.1	98.5	78.5	99.8	76.5	85.2	25.4	91.5	44.9	57.6	50.4	65.4	20.6	18.5	68.7	51.5	10.7	37.6
	16M	**71**	5.3	99.1	90.3	87.8	99.8	93.5	96.8	37.6	98.8	82.4	91.0	85.9	76.2	23.1	43.4	81.4	82.0	15.1	68.7
scalar	2M	**16**	0.8	0.1	0	9.0	0	2.7	2.6	69.3	0.1	5.9	0.5	6.5	14.2	64.8	20.7	12.2	18.2	57.3	11.3
	16M	**18**	0.8	0	0	8.0	0	3.0	1.1	61.5	0	6.2	0.5	4.4	14.0	69.7	40	14.1	12.6	79.3	18.3
array	2M	**1**	0	0	0.6	2.6	0	0	0.4	0	0.1	0	1.0	0	6.7	5.1	0.2	0.4	0.3	0	4.1
	16M	**2**	0.1	0	9.6	2.4	0	0	0.4	0	0	0.2	0.4	0	6.8	5.1	0.3	1.6	0.2	0	2.8
root	2M	**27**	98.1	5.8	0.9	9.9	0.2	20.7	11.7	5.3	8.3	49.1	41.0	43.0	13.7	9.5	60.6	18.7	29.9	32.0	47.0
	16M	**9**	93.8	0.8	0.1	1.8	0.2	3.5	1.7	0.9	1.1	11.2	8.2	9.7	3.0	2.2	16.3	3.0	5.2	5.6	10.1
percentage of decrements																					
new	2M	**60**	2.4	95.0	97.7	63.5	99.8	90.8	90.4	30.7	92.1	65.3	39.7	60.1	42.9	22.2	24.0	70.5	56.4	45.5	43.2
	16M	**71**	11.3	98.9	90.3	70.5	99.8	96.7	97.0	42.2	98.6	88.7	61.5	84.4	51.7	24.8	48.8	80	81.6	54.9	71.5
scalar	2M	**15**	0.8	0.4	0.1	16.1	0	1.0	2.1	64.4	0.3	4.6	3.6	4.0	25.8	63.5	19.7	13.2	15.1	34.4	10.7
	16M	**15**	0.9	0.4	0	14.9	0	0.9	0.8	57.0	0.1	4.0	4.1	1.8	25.3	67.9	33.7	14.5	9.8	41.4	14.9
array	2M	**1**	0	0	0.5	2.5	0	0	0.3	0	0.2	0	0.9	0	6.2	5.1	0.8	0.3	0.3	0	3.7
	16M	**2**	0	0	9.4	2.4	0	0	0.4	0	0.1	0	0.3	0	6.2	5.2	1.3	1.1	0.2	0	2.2
root	2M	**21**	96.7	4.5	1.7	7.7	0.2	6.9	7.2	4.8	7.2	28.3	37.7	29.7	12.5	9.1	53.8	14.0	25.1	20.1	40
	16M	**8**	87.3	0.6	0.3	1.3	0.2	1.1	1.7	0.8	1.0	4.9	7.2	5.7	2.7	2.1	12.8	2.1	4.5	3.7	7.8
cycle	2M	**3**	0.1	0.1	0	10.3	0	1.3	0	0	0.1	1.8	18.1	6.2	12.7	0.1	1.8	2.0	3.1	0	2.3
	16M	**4**	0.6	0.1	0	10.9	0	1.4	0	0	0.2	2.4	26.9	8.1	14.1	0.1	3.3	2.3	4.0	0	3.6

Table 4.3: New objects account for a large percentage of increment and decrement operations. This table shows the sources of increment (top) and decrement (bottom) operations when collections are forced at 2 MB and 16 MB intervals. In all cases new objects dominate.



(a) Increments



(b) Decrements

Figure 4.3: New objects are responsible for the majority of reference counting operations. We show here the percentage of (a) increments and (b) decrements that are due to objects allocated within the most recent 2 MB and 16 MB of allocation.

4.3.4 Sources of Reference Counting Operations

Table 4.3 shows for each benchmark the origin of the increment and decrement operations. In each case we account for the operations as being due to: a) newly allocated objects (*new*), b) mutations to non-new *scalar* and *array* objects, and c) temporary operations due to *root* reachability when using deferred reference counting. For decrements, we also include a fifth category that represents decrements that occur during *cycle* collection. We performed this measurement with collections artificially triggered at a range of intervals from 2 MB to 16 MB, and report only 2 MB and 16 MB to show the significant differences. This choice of interval is guided by typical nursery sizes for a generational garbage collector [2.4.6]. The definition of *new* is anything allocated within the last interval, so as the interval becomes larger, a larger fraction of the live objects are *new*. This measurement related to *new* objects depends on the object header size and alignment restrictions, which may slightly vary in different JVMs.

On average, 71% of increments and 71% of decrements are performed upon newly

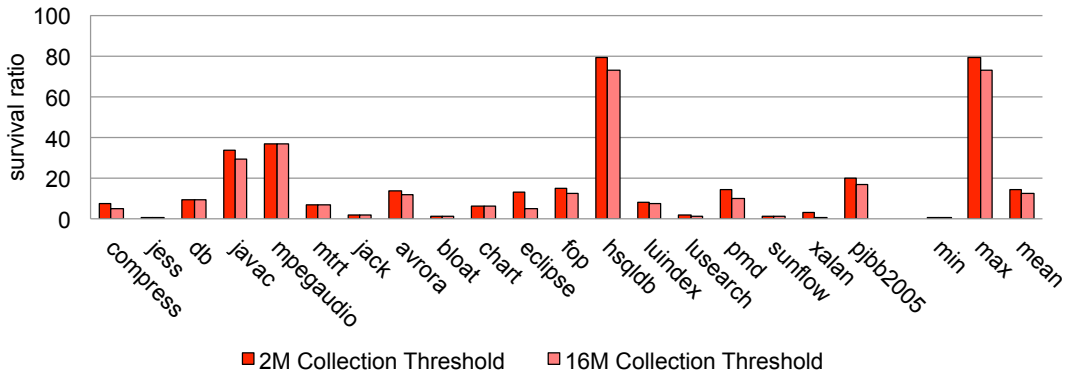


Figure 4.4: Most benchmarks have very low object survival ratios. This graph shows the percentage of objects that survive beyond 2 MB and 16 MB of allocation.

allocated objects (over 90% for some benchmarks). For most benchmarks increments and decrements to non-new objects are low (around 9-10%), consistent with previous findings [Blackburn and McKinley, 2003]. Around 10% of operations are due to root reachability. Cycle collection performs 4% of decrements.

Figures 4.3(a) and 4.3(b) illustrate data from Table 4.3 graphically, showing the percentage of increments and decrements due to new objects, where *new* is defined in terms of both 2 MB and 16 MB allocation windows.

Conventionally, when using deferred reference counting, new objects are born *live*, with a temporary increment of one. A corresponding decrement is enqueued and applied at the next collection. Thus a highly allocating benchmark will incur a large number of increments and decrements simply due to the allocation of objects. Furthermore, newly allocated objects are relatively more frequently mutated, so contribute further to the total count of reference counting operations.

Table 4.4 shows the percentage of increments as a function of maximum reference count. For example, the table shows that on average 31% of increments are performed for objects having maximum reference count of one and 18% increments are performed for objects having maximum reference count of two. Interestingly, on average 17% of increments are due to objects with very high maximum reference counts ($>$63).

Figure 4.4 shows that most benchmarks have a survival ratio of under 10%, indicating that over 90% of objects are unreachable by the time of the first garbage collection. This information and the data which shows that new objects attract a disproportionate fraction of increments and decrements confirms previous suggestions that new objects are likely to be a particularly fruitful focus for optimization of reference counting [Blackburn and McKinley, 2003; Azatchi and Petrank, 2003; Paz et al., 2005].

max count	mean	compress	jess	db	javac	mpegaudio	mtrt	jack	avrora	bloat	chart	eclipse	fop	hsqldb	luindex	lusearch	pjbb2005	pmd	sunflow	xalan
0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	**31**	72.1	17.3	0.8	23.4	67.5	50.2	77.5	2.7	6.2	63.4	32.8	38.2	8.8	5.9	54.1	13.6	18.3	10.5	28.2
2	**18**	27.3	5.0	0.1	47.7	27.2	7.6	12.5	53.7	5.6	13.2	29.2	31.2	20.9	3.1	19.2	7.4	7.9	4.8	24.6
3	**9**	0.6	0.1	0.1	9.5	5.3	7.1	7.9	5.4	1.4	8.6	9.5	12.3	37.7	16.5	16.5	10.5	3.2	0	10.6
4	**5**	0	0	0	3.2	0	1.2	1.5	4.0	0.4	0.3	5.6	5.5	1.4	52.9	4.5	7.5	7.7	0	1.2
5	**4**	0	0	0	1.0	0	0.3	0.3	5.6	0.6	0.1	3.6	1.6	25.3	4.2	0	6.8	26.7	0	0.5
6	**1**	0	0	0	0.4	0	0.2	0.2	3.7	0.1	0	2.6	0.8	0.5	3.6	0	8.2	4.8	0	0.6
7	**2**	0	0	0	0.5	0	0.3	0.1	16.8	0.1	0	1.8	0.8	0.2	4.0	0	1.7	7.3	0	0.4
8-15	**3**	0	0.1	0.1	1.9	0	3.0	0.1	0	0.4	0	5.6	2.0	0.3	9.8	0	3.9	11.6	0	13.1
16-31	**2**	0	0.2	4.6	2.2	0	4.5	0	0	6.0	0.4	3.0	1.7	0	0	0	21.3	1.3	0	1.8
32-63	**7**	0	0.2	89.9	1.6	0	2.2	0	0.1	27.8	1.5	1.8	1.5	0	0	0.2	3.1	2.2	0	4.5
> 63	**17**	0	77.0	4.4	8.4	0	23.5	0	8.1	51.4	12.4	4.5	4.6	4.9	0.1	5.5	16.1	8.9	84.7	14.5

Table 4.4: 49% of increment and decrement operations occur on objects with maximum reference counts of just one or two. This table shows how increment operations are distributed as a function of the maximum reference count of the object the increment is applied to.

4.3.5 Efficacy of Coalescing

Coalescing [2.5.3] is most effective when individual reference fields are mutated many times, allowing the reference counter to avoid performing a significant number of reference count operations. To determine whether this expectation matches actual behavior, we compare the total number of reference mutation operations to the number of reference mutations observable by coalescing (i.e., where the final value of a reference field does not match the initial value). We control the window over which coalescing occurs by triggering collection after set volumes of application allocation (from 2 MB to 8 MB).

Table 4.5 shows, for example, that with a window of 8 MB, coalescing observes 50.5% and 92.2% of reference mutations for compress and jess respectively. For a few benchmarks, such as avrora, luindex, and sunflow, coalescing is extremely effective, eliding 90% or more of all reference mutations. However, for many benchmarks, coalescing is not particularly effective, eliding less than half of all mutations. In addition to measuring this for all objects, we separately measure operations over *new* objects — those allocated since the start of the current time window. This data shows that coalescing is significantly more effective with old objects. This is consistent with the idea that frequently mutated objects tend to be long lived, and is not inconsistent with the prior observation [Blackburn and McKinley, 2003] that most mutations occur to young objects (since over the life of a program, young objects typically outnumber old objects by around 10:1). Table 4.6 provides a different perspective by showing the breakdown of total reference mutations per unit time (millisecond).

4.3.6 Cyclic Garbage

Table 4.7 shows key statistics for each benchmark related to cyclic garbage. For each benchmark we show: 1) the percentage of objects that can be reclaimed by pure reference counting, and 2) the percentage of objects that are part of a cyclic graph when unreachable, so can only be reclaimed via cycle collection, and 3) the percentage of objects that are statically known to be acyclic (i.e., an object of that type can never transitively refer to itself). Note that 2) may not be directly participating in a cycle but may be referenced by a cycle. These results show that the importance of cycle collection varies significantly between benchmarks, with some benchmarks relying heavily on cycle collection (javac, mpegaudio, eclipse, hsqldb and pmd) while the cycle collector is responsible for reclaiming almost no memory (less than 1% for jess, db, jack, avrora, bloat and sunflow).

4.4 Limited Bit Reference Counting

Because the vast majority of objects have low maximum reference counts, the use of just a few bits for the reference counting is appealing. The idea has been proposed before [Jones et al., 2011], but to our knowledge has not been systematically analyzed. Key insights that can be drawn from our analysis are that most objects have maxi-

trigger	mean	compress	jess	db	javac	mpegaudio	mtrt	jack	avrora	bloat	chart	eclipse	fop	hsqldb	luindex	lusearch	pjbb2005	pmd	sunflow	xalan
percentage of pointer field changes seen (overall)																				
2M	**36.4**	48.0	92.2	26.9	54.7	0.1	47.9	53.3	10.0	12.1	71.9	54.8	60.6	43.8	3.9	27.7	23.9	32.3	8.9	19.2
4M	**36.2**	49.6	92.2	26.4	54.5	0.1	47.9	53.3	10.0	12.1	71.9	55.9	60.6	43.8	3.9	22.2	22.1	32.3	8.9	19.2
8M	**36.2**	50.5	92.2	26.1	54.1	0.1	47.9	53.3	10.0	11.8	71.9	55.9	60.6	43.8	3.8	22.1	23.4	32.3	8.9	19.1
percentage of pointer field changes seen (for new objects)																				
2M	**48.5**	53.3	92.4	28.6	61.1	0.1	48.8	56.4	31.3	12.0	79.1	58.3	67.4	67.7	25.5	74.8	45.9	41.3	48.3	28.6
4M	**46.5**	53.3	92.4	28.5	59.7	0.1	48.8	56.0	30.8	12.1	78.1	57.4	66.5	66.9	13.5	58.4	45.6	40.2	48.0	27.4
8M	**45.8**	53.2	92.4	28.2	59.1	0.1	48.8	55.2	30.8	11.7	76.4	57.0	66.4	65.5	9.4	56.2	45.2	40.0	48.3	26.7
percentage of pointer field changes seen (for old objects)																				
2M	**10.3**	12.8	45.3	13.7	29.0	20.0	4.0	1.2	0.1	21.0	18.1	9.6	10.5	2.7	1.0	0.2	2.7	1.8	0.8	0.3
4M	**9.9**	14.1	38.2	8.0	30.4	20.0	3.5	1.0	0.0	22.1	18.6	13.5	11.4	1.3	1.2	0.1	1.8	1.6	0.7	0.2
8M	**9.7**	14.5	32.7	4.9	29.2	21.1	2.6	1.1	0.0	28.6	19.0	12.9	11.5	0.7	0.9	0.1	1.4	1.5	0.7	0.1

Table 4.5: Coalescing elides around 64% of pointer field changes on average, and around 90% for old objects. This table shows the percentage of mutations that *are* seen by coalescing given three different collection windows. The top third shows the overall average. The middle third shows results for new objects. The bottom third shows old objects.

types	mean	compress	jess	db	javac	mpegaudio	mtrt	jack	avrora	bloat	chart	eclipse	fop	hsqldb	luindex	lusearch	pjbb2005	pmd	sunflow	xalan
Scalar	**8027**	2	3165	896	7689	3185	2319	9046	3305	36862	3234	2432	1872	11162	10051	10059	13080	11558	13894	8695
Array	**1788**	0	3026	7806	1010	1	3069	2099	15	551	106	3242	72	3080	937	280	2723	1320	62	4566
Bulk	**114**	0	2023	0	6	0	0	6	0	4	3	91	2	7	0	0	0	11	0	7
Total	**9928**	2	8214	8702	8705	3186	5389	11151	3320	37417	3343	5765	1945	14249	10989	10339	15804	12890	13955	13268

Table 4.6: References are mutated around 10 million times per second on average. This graph shows the rate of mutations per millisecond for each benchmark, broken down by scalars, arrays and bulk copy operations.

types	mean	compress	jess	db	javac	mpegaudio	mtrt	jack	avrora	bloat	chart	eclipse	fop	hsqldb	luindex	lusearch	pjbb2005	pmd	sunflow	xalan
pure rc objects	**84**	92	99.7	100	77	64	93	99.9	99.8	99	94	47	81	27	91	84	86	80	99.99	90
cyclic objects	**16**	8	0.3	0	23	36	7	0.1	0.2	1	6	53	19	73	9	16	14	20	0.01	10
acyclic objects	**38**	55	18	4	34	44	3	38	16	49	49	54	46	35	49	28	35	21	97	44

Table 4.7: The importance of cycle collection. This table shows that on average 84% of objects can be collected by reference counting without a cycle collector, and that about half of these, on average 38% of all objects are inherently acyclic. These results vary considerably among the benchmarks.

mum reference counts of seven or less, and that objects with high maximum reference counts account for a disproportionate fraction of reference counting operations. The former motivates using around three bits for storing the count, while the latter suggests that any strategy for dealing with overflow must not be too expensive since it is likely to be heavily invoked. We now describe three strategies for dealing with reference count overflow.

4.4.1 Hash Table on Overflow

When an object's reference count overflows, the reference count can be stored in a hash table. Increments and decrement are performed in the hash table until the reference count drops below the overflow threshold, at which point the hash table entry is released. Each entry in the hash table requires two words, one word for the object (key) and one word for the count (value). We measure the size of hash table across the benchmarks and find that 1 MB table is sufficient for all benchmarks.

4.4.2 Stuck on Overflow and Ignored

When an object's count overflows, it may be left stuck at the overflow value and all future increments and decrements will be ignored. Reference counting is thus unable to collect these objects, so they must be recovered by a backup tracing cycle collector. Note that a trial deletion cycle collector cannot collect such objects.

4.4.3 Stuck on Overflow and Restored by Backup Trace

A refinement to the previous case has the backup trace *restore* reference counts within the heap during tracing, by incrementing the target object's count for each reference traversed. Although this approach imposes an additional role upon the backup trace, it has the benefit of freeing the backup trace from performing recursive decrement operations for collected objects.

4.4.4 Evaluation

Figure 4.5 shows our experimental evaluation of these strategies. In Jikes RVM we have up to one byte (8 bits) available in the object header for use by the garbage collector. We use two bits to support the dirty state for coalescing, one bit for the mark-state for backup tracing, and the remaining five bits to store the reference count. All results are normalized to MMTk's default reference counting configuration, *Standard RC*, a coalescing deferred collector using an additional header word that is fully precise and that uses backup tracing cycle collector.

As we mentioned before, an additional header word has a cost, not only in terms of space, but also time, as allocation rate is also affected. This result is visible in mutator time (Figure 4.5(b)) where all the three strategies are more than 2% faster than *Standard RC*.

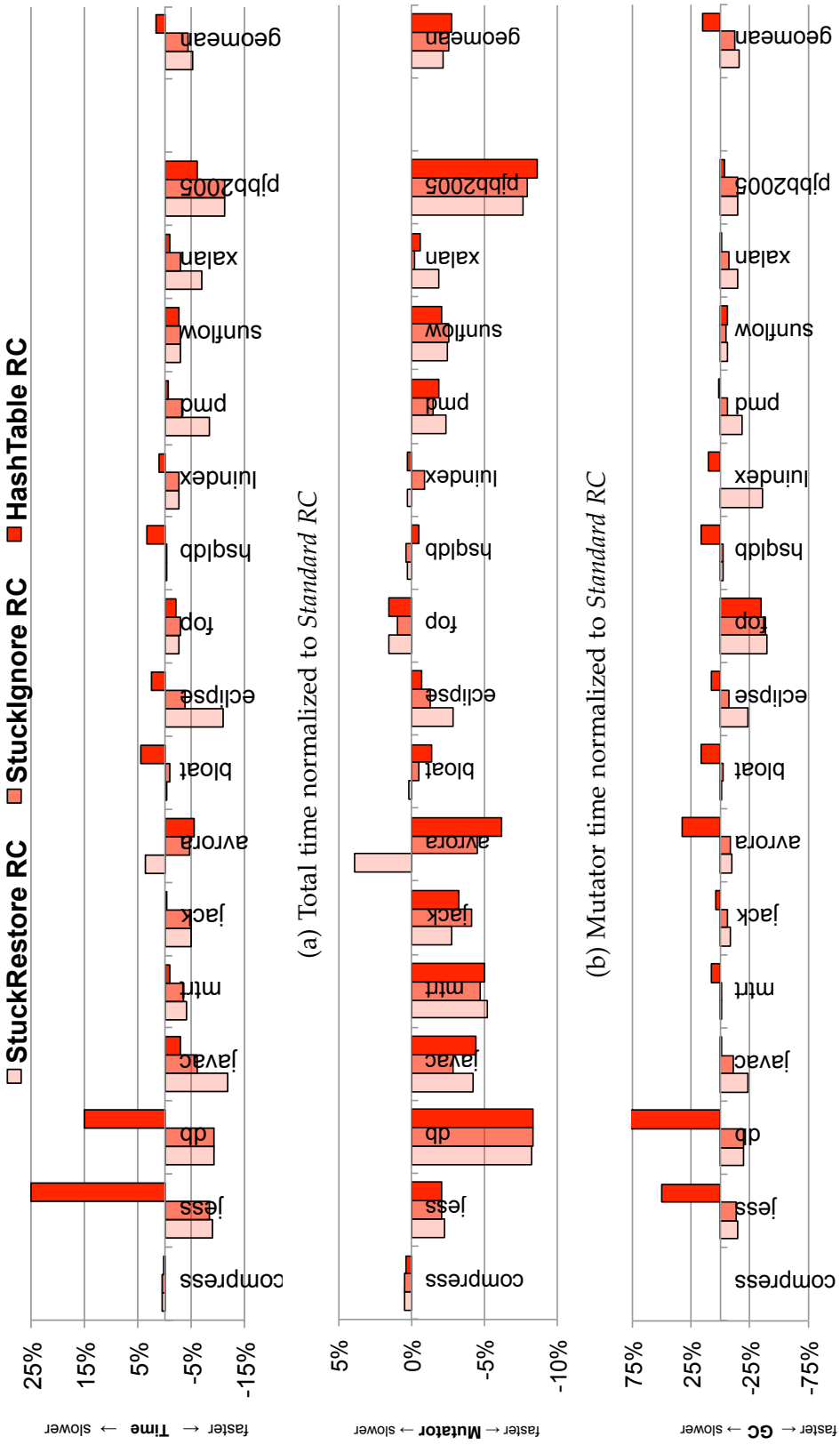


Figure 4.5: These graphs show three strategies for dealing with overflow. Results vary greatly among benchmarks.

For the majority of the benchmarks *HashTable RC* performs poorly, with *Standard RC* 2% better in total time (Figure 4.5(a)) and 17% better in collection time (Figure 4.5(c)) than *HashTable RC* on average. The performance of jess and db is much worse in *HashTable RC* compared to other benchmarks. This was predicted by our analysis, which showed that these benchmarks had high rates of reference counting operations on overflowed objects. While *HashTable RC* benefits from not requiring an additional header word, this benefit is outweighed by the cost of performing increment and decrement operations in the hash table. In *HashTable RC*, the processing of increments and decrements are 34% and 19% slower than in *Standard RC*, respectively.

Given the poor performance of the hash table approach, we turn our attention to the systems that use backup tracing to collect objects with sticky reference counts, *StuckIgnore RC* and *StuckRestore RC*. Both *StuckIgnore RC* and *StuckRestore RC* outperform *Standard RC* (by 4% and 5% respectively). This is primarily due to no longer requiring an additional header word, although there is also some advantage from ignoring reference counting operations. Comparing the two sticky reference count systems, *StuckRestore RC* performs slightly better in both total time and collection time. Backup tracing in *StuckRestore RC* performs more work than *StuckIgnore RC* because it restores the count for the objects. But as mentioned earlier, during backup tracing if any object's reference count is zero then only the object is reclaimed and count of the descendants are not decremented, giving *StuckRestore RC* a potential advantage.

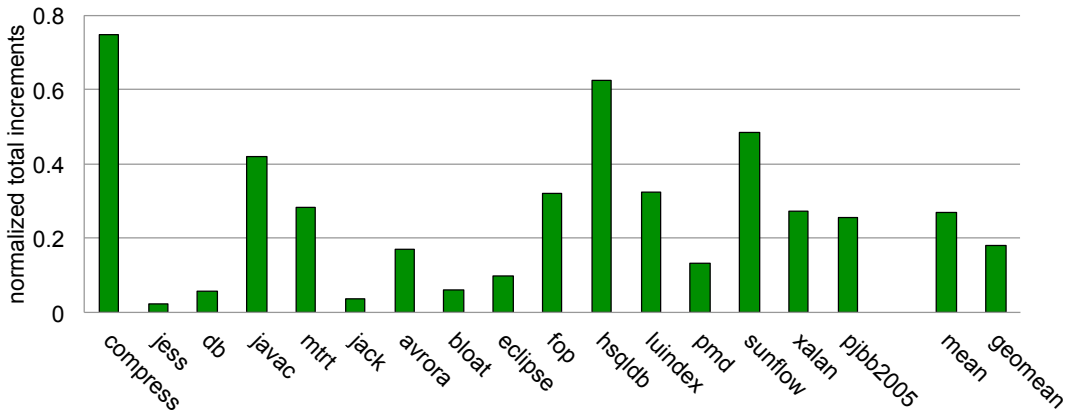
We also measured (but do not show here) the three overflow strategies with an additional header word, to factor out the source of difference with *Standard RC*. In this scenario, the extra word is not used to store the reference count but simply acts as a placeholder to evaluate the impact of the space overhead. In that case, *StuckIgnore RC* performs same as *Standard RC* and *StuckRestore RC* only marginally outperformed *Standard RC* (by 1% in total time), indicating that most of their advantage comes from the use of a small reference counting field.

4.5 Improving Reference Counting

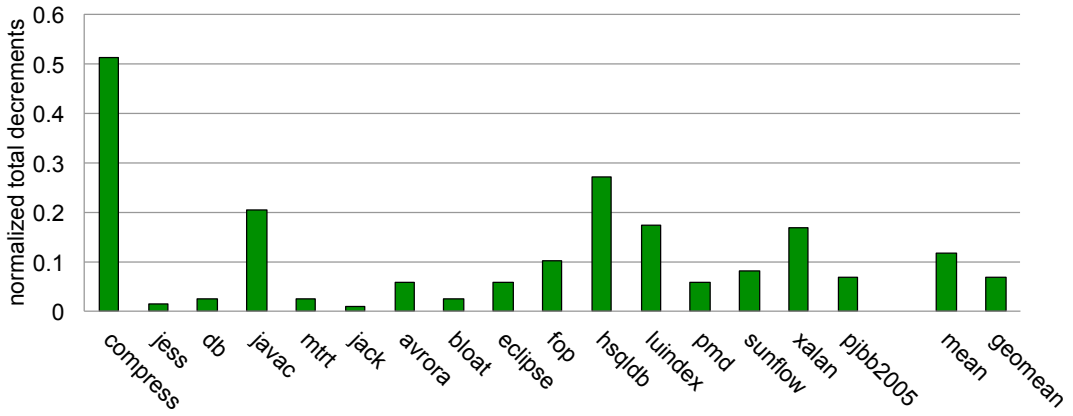
Our analysis shows that reference counting overheads are dominated by the behavior of new objects, and yet the vast majority of those objects do not survive a single collection. We use these observations of new objects to optimize reference counting.

4.5.1 Exploiting the Weak Generational Hypothesis

We leverage two insights that allow us to ignore new objects until their first collection, at which point they can be processed lazily, as they are discovered. First, coalescing reference counting uses a dirty bit in each object's header to ignore mutations to objects between their initial mutation and the bit being reset at collection time. A collector that ignores new objects could straightforwardly use this mechanism. Second, in a deferred reference counter, any new object reachable from either the roots or old objects will be included in the set of increments. Furthermore, the set of increments



(a) Increments



(b) Decrements

Figure 4.6: Lazy mod-buf insertion and born dead optimizations for new objects greatly reduce the number of reference counting operations necessary compared to *Standard RC*. The effectiveness varies substantially among the benchmarks. On average over 80% of increments and 90% of decrements are eliminated.

will only include references to new objects that are live. We further observe that if new objects are allocated as logically dead, and only made live upon discovery, then a significant fraction of expensive freeing operations can be avoided, since the vast majority of objects do not survive the first collection.

We start by considering the treatment of new objects in a collector that uses deferred reference counting and coalescing, and we use this as our point of comparison. In such a collector, new objects are allocated dirty with a reference count of one. The object is added to the deque of decrements (*dec-buf*), and a decrement cancelling the reference count of one is applied once the *dec-buf* is processed at the next collection [2.5.2]. The object is also added to the deque of modified objects (*mod-buf*) used by the coalescing mechanism. At the next collection, the collector processes the *mod-buf* and applies an increment for each object that the processed object now points to. Because all references within a new object are initially *null*, the coalescing mechanism

does not need to explicitly generate decrements corresponding to outgoing pointers from the initial state of the object [2.5.3].

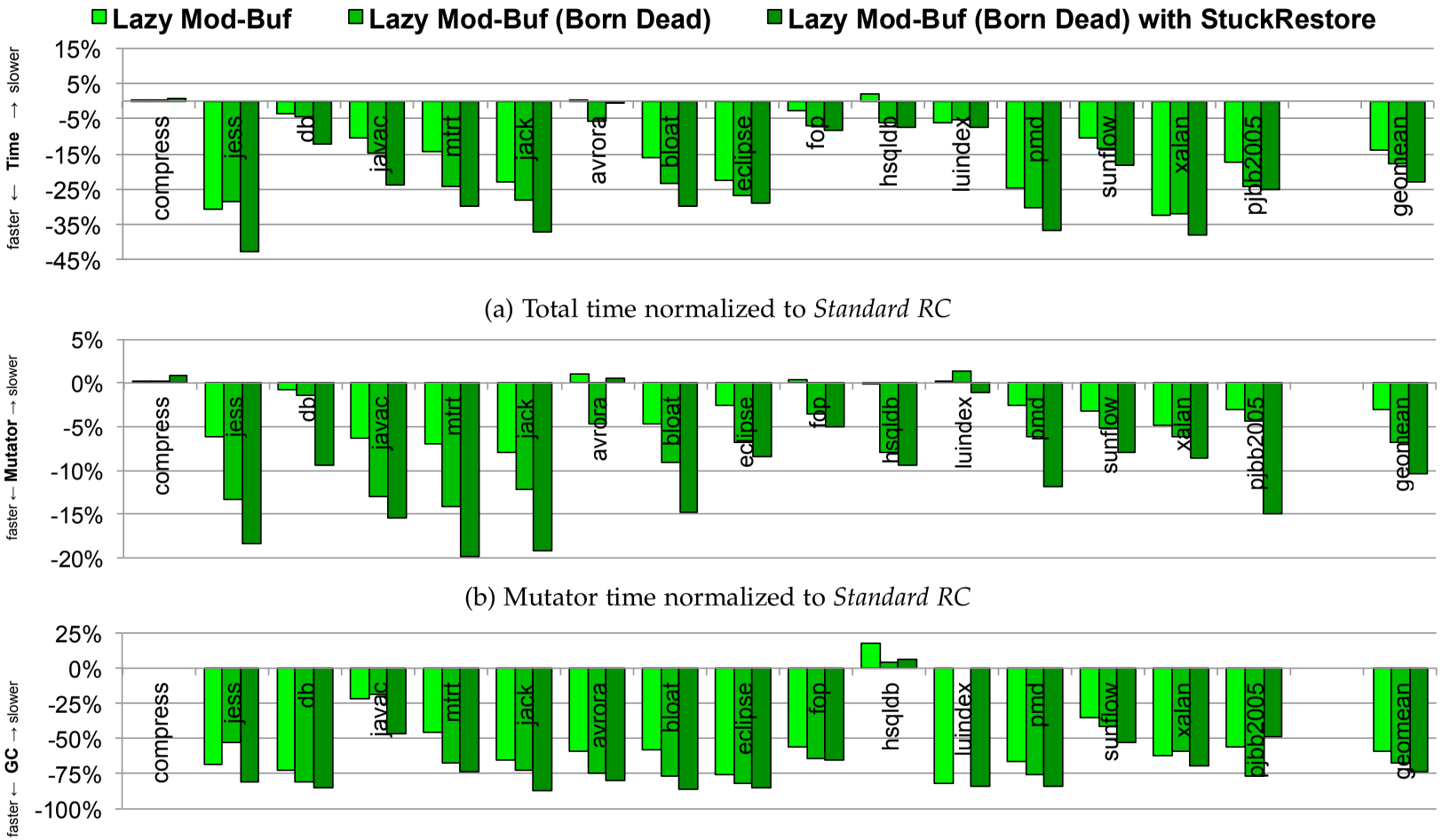
4.5.1.1 Lazy Mod-Buf Insertion Optimization

Our first optimization is to not add new objects to the *mod-buf*. Instead, we add a *new* bit to the object header, and add objects lazily to the *mod-buf* at collection time, only if they are encountered during the processing of increments. During collection, whenever the subject of an increment is marked as new, the object's new bit is cleared, and the object is pushed onto the *mod-buf*. Because in a coalescing deferred reference counter, all references from roots and old objects will increment all objects they reach, our approach will retain all new objects directly reachable from old objects and the roots. Because each object processed on the *mod-buf* will increment each of its children, our scheme is transitive. Thus new objects are effectively traced. However, rather than combing reference counting and tracing to create a hybrid collector [Blackburn and McKinley, 2003; Azatchi and Petrank, 2003; Paz et al., 2005], our scheme achieves a similar result via a very simple optimization to existing reference counting collector. It reduces the reference counting bits from five to four due to the *new* bit, but only 0.11% objects suffer overflow instead of 0.06% for five bits [Table 4.2]. This optimization required only very modest changes to MMTk's existing reference counting collector. Figure 4.6(a) shows the massive reductions in the total number of increments (over 80%).

4.5.1.2 Born Dead Optimization

As a simple extension of the above optimization, instead of allocating objects live, with a reference count of one and a compensating decrement enqueued to the *dec-buf*, our second optimization allocates new objects as dead with a reference count of zero and does not enqueue a decrement. This inverts the presumption: the reference counter does not need to identify those new objects that are *dead*, but it must rather identify those that are *live*. This inversion means that work is done in the infrequent case of a new object being reachable, rather that the common case of it being dead. New objects are only made live when they receive their first increment while processing the *mod-buf* during collection time. Our optimization removes the need for creating compensating decrements and avoids explicitly freeing short lived objects. Figure 4.6(b) shows the massive reduction in the total number of decrements (over 90%).

We evaluate the performance of lazy mod-buf insertion and born dead optimization for new objects. Figures 4.7(a), 4.7(b), and 4.7(c) (the leftmost and midmost columns) show the effect of the optimizations on total time, mutator time, and garbage collection time respectively relative to orthodox deferred reference counting with coalescing (*Standard RC*). The first optimization (*Lazy Mod-Buf*) improves over *Standard RC* by 14% in total time, 3% in mutator time, and 59% in collection time, on average, over the set of benchmarks. The two optimizations combined (*Lazy*



(a) Total time normalized to *Standard RC*

(b) Mutator time normalized to *Standard RC*

(c) Garbage collection time normalized to *Standard RC*

Figure 4.7: Lazy mod-buf insertion and born dead optimizations for new objects reduce total time by around 18% compared to *Standard RC*. The combined effect of our optimizations is a 23% improvement in total time compared to *Standard RC*.

types	mean	compress	jess	db	javac	mpegaudio	mtrt	jack	avrora	bloat	eclipse	fop	hsqldb	luindex	lusearch	pjbb2005	pmd	sunflow	xalan
Standard RC	**16**	8	0.3	0	23	36	7	0.1	0.2	1	53	19	73	9	16	14	20	0.01	10
With our optimization	**10**	1	0.3	0	16	36	7	0.1	0.01	1	4	14	71	8	1	13	11	0.01	1

Table 4.8: Exploiting the weak generational hypothesis results in a 37% reduction in dead cyclic objects, from 16% to 10% on average, when cycle collection is triggered every 4 MB of allocation. These results vary considerably among the benchmarks, and is as high as 92% for eclipse and 90% for xalan.

Mod-Buf (Born Dead)) are 18% faster in total time, 7% faster in mutator time, and 67% faster in collection time than *Standard RC* on average.

4.5.1.3 Reducing Cyclic Objects

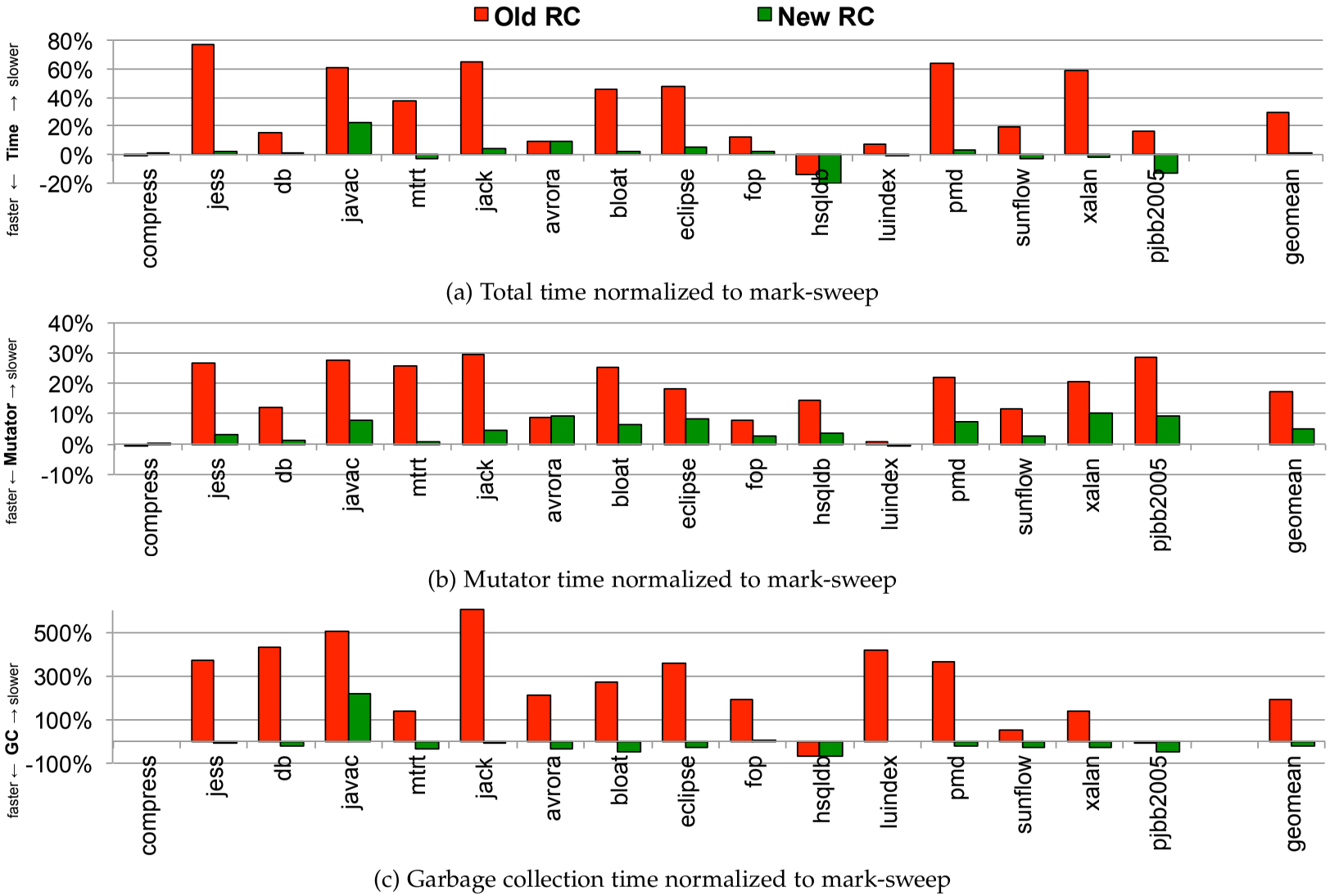
The lazy mod-buf insertion and born dead optimizations for new objects also reduce the impact of cyclic garbage. If a cycle is dead by the time of the first GC, it is implicitly collected, and thus does not place a load on the reference counting collector or its cycle detection mechanism. Table 4.8 shows that when cycle collections are artificially triggered every 4 MB of allocation, the number of cycles found by the cycle detector is reduced by 37%, from 16% to 10% by applying our optimizations. The effect of this optimization is reduced heap pressure due to dead cyclic objects that would otherwise remain in the heap until the next cycle collection.

4.5.2 Bringing It All Together

Figure 4.7 (the rightmost column) also presents an evaluation of the impact of the three most effective optimizations operating together: a) limited bits for the reference count and restore counts during backup trace, b) lazy *mod-buf* insertion, and c) born dead. The combined effect of these optimizations is 23% faster in total time (Figure 4.7(a)), 10% faster in mutator time (Figure 4.7(b)), and 74% faster in collection time (Figure 4.7(c)) compared to our base case (*Standard RC*), on average over the benchmarks. This substantial improvement over an already optimized reference counting implementation should change perceptions about reference counting and its applicability to high performance contexts.

4.6 Results

The conventional wisdom has been that reference counting is totally uncompetitive compared to a modern mark-sweep collector [Blackburn and McKinley, 2003]. Figure 4.8 shows the evaluation of *Standard RC* and *Lazy Mod-Buf (Born Dead) with StuckRestore* against a well tuned mark-sweep collector. Consistent with conventional wisdom, *Standard RC* performs substantially worse than mark-sweep, slowing down by 30%. However, our optimized reference counter, *Lazy Mod-Buf (Born Dead) with*

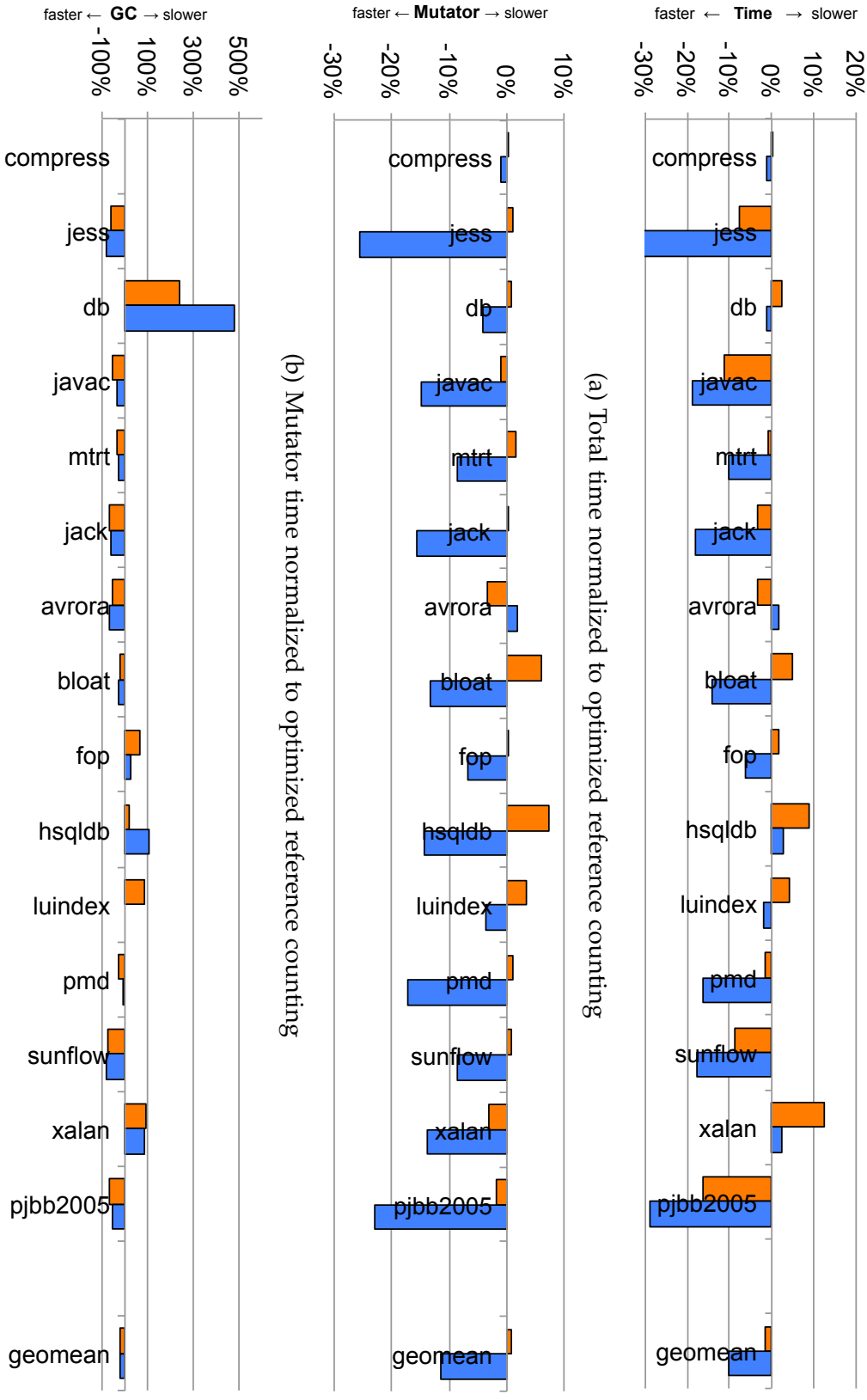


(a) Total time normalized to mark-sweep

(b) Mutator time normalized to mark-sweep

(c) Garbage collection time normalized to mark-sweep

Figure 4.8: Our optimized reference counting closely matches mark-sweep, while standard reference counting performs 30% worse.



(a) Total time normalized to optimized reference counting

(b) Mutator time normalized to optimized reference counting

(c) Garbage collection time normalized to optimized reference counting

Figure 4.9: Sticky Immix outperforms our optimized reference counting collector by 10%. The combination of the optimized reference counter and the Immix heap layout appears to be promising.

StuckRestore, is able to entirely eliminate the overhead and performs marginally faster than mark-sweep on average, and is at worst 23% worse than mark-sweep (javac whose performance largely depends on the triggering of cycle collection) and at best 20% better than mark-sweep (hsqldb).

We also compare our improved reference counting with sticky mark bits collectors [Demers et al., 1990; Blackburn and McKinley, 2008]. These collectors are similar to ours in that they combine generational ideas in a non-moving or less aggressive moving context. However, they use tracing and they use a write barrier to avoid tracing the whole heap at every collection. Like our approach, they identify new objects using bits in the object header to treat them separately. Immix is a mark-region based tracing garbage collector that allocates contiguously in lines and blocks, and perform opportunistic copying to mitigate fragmentation. It achieves space efficiency, fast reclamation, and mutator performance. Much of its performance advantage over mark-sweep is due to its line and block heap organization. Sticky MS and Sticky Immix are the non-moving generational variant of mark-sweep and Immix. Figure 4.9 shows that our improved reference counting collector is only 1% slower than Sticky MS and 10% slower than Sticky Immix. Sticky Immix should therefore be a good indicator of the performance of our improved reference counting projected onto the Immix heap organization. This is an exciting prospect because Sticky Immix is only 2% slower than Jikes RVM's production collector. An exploration of this collector is the focus of the next chapter.

4.7 Summary

This chapter identifies that a reference counting implementation following the prior state-of-the-art lags the performance of a highly tuned mark-sweep tracing collector by 30% on average. The chapter introduced a detailed quantitative analysis of reference counting's key design points. The analysis reveals that: a) the vast majority of reference counts are low, less than five, b) many reference count increments and decrements are to newly allocated young objects but few of them survive, and c) many cycles die with young objects. This chapter introduces two novel optimizations that overlook reference counting operations for short lived newly allocated objects in a limited bit reference counting system. The resulting reference counting collector, which is a deferred coalescing collector that uses limited bits to store the count and optimizations that exploit the weak generational hypothesis, performs the same as a highly tuned full heap mark-sweep tracing collector.

The improved reference counting collector is still 10% slower than production generational tracing collectors; a substantial barrier to adoption in performance-critical settings. The next chapter will focus on how to attack that performance gap and make reference counting competitive with high performance generational tracing collectors.

Chapter 5

Reference Counting Immix

The previous chapter introduced the first reference counting collector with performance matching a full heap mark-sweep tracing collector, and within 10% of a high performance generational collector, Gen Immix. Though a major advance, this 10% gap remains a substantial barrier to adoption in performance-conscious application domains. This chapter identifies reference counting's free-list heap organization as the major source of the remaining performance gap. We introduce a new collector, RC Immix, that replaces the free-list heap structure of reference counting with the line and block heap structure of the Immix collector. RC Immix extends the reference counting algorithm to count live objects on each line. RC Immix also integrates Immix's opportunistic copying with reference counting to mitigate fragmentation. In RC Immix, reference counting offers efficient collection performance and the line and block heap organization delivers excellent mutator locality and efficient allocation. With these advances, RC Immix closes the 10% performance gap, matching and outperforming a highly tuned production generational collector, Gen Immix.

This chapter is structured as follows. Section 5.2 presents motivating analysis, showing that a free-list allocator suffers significant performance overheads compared to a contiguous allocator due to poor cache locality and additional instructions required for zeroing. Section 5.3 describes the design of the RC Immix collector, which combines our improved reference counting and the Immix collector. Section 5.4 evaluates RC Immix, comparing it to a production generational collector, Gen Immix, and some other existing high performance collectors.

This chapter describes work published as "Taking Off the Gloves with Reference Counting Immix" [Shahriyar, Blackburn, Yang, and McKinley, 2013].

5.1 Introduction

The reference counting collector described in the previous chapter, which we will call RC, removed the 30% performance gap compared to a full heap mark-sweep tracing collector, but RC is still 10% slower than a high performance generational tracing collector, which is a substantial barrier to adoption in performance-critical settings.

This chapter identifies that the major source of the 10% gap is that RC's free-list heap layout has poor cache locality and imposes instruction overhead. Poor locality

occurs because free-list allocators typically disperse contemporaneously allocated objects in memory, which degrades locality compared to allocating them together in space [Blackburn et al., 2004a; Blackburn and McKinley, 2008]. Instruction overheads are greater in free lists, particularly when programming languages require objects be pre-initialized to zero. While a contiguous allocator can do bulk zeroing very efficiently, a free-list allocator must zero object-by-object, which is inefficient [Yang et al., 2011].

To solve these problems, we introduce Reference Counting Immix (RC Immix). RC Immix uses the allocation strategy and the line and block heap organization introduced by Immix mark-region garbage collection [2.6]. Recall that Immix places contemporaneously created objects contiguously in free lines within blocks. Immix allocates into partially free blocks by efficiently skipping over occupied lines. Objects may span lines, but not blocks. Immix reclaims memory at a line and block granularity. The granularity of reclamation is the key mismatch between reference counting and Immix. RC Immix resolves this problem. Reference counting reclaims objects, whereas Immix reclaims lines and blocks. The design contributions of RC Immix are as follows. 1) RC Immix extends the reference counter to count live objects on a line. When the live object count of a line is zero, RC Immix reclaims the free line. 2) RC Immix extends *opportunistic copying*, which mixes copying with leaving objects in place. RC Immix adds *proactive* copying, which combines reference counting and copying to compact newly allocated live objects. RC Immix on occasion *reactively* copies old objects during cycle detection to eliminate fragmentation.

Combining copying and reference counting is novel and surprising. Unlike tracing, reference counting is inherently local, and therefore in general, the set of incoming references to a live object is not known. However, we observe two important opportunities. First, in a reference counter that coalesces increments and decrements [2.5.3], since each new object starts with no references to it, the first collection must enumerate *all* references to that new object, presenting an opportunity to move that object proactively. We find that when new objects have a low survival rate, the remaining live objects are likely to cause fragmentation. We therefore copy new objects, which is very effective in small heaps. Second, since completeness requires a tracing cycle collection phase, RC Immix seizes upon this opportunity to incorporate reactive defragmentation of older objects. In both cases, we use opportunistic copying, which mixes copying and leaving objects in place, and thus can stop copying when it exhausts available memory.

Two engineering contributions of RC Immix are improved handling of roots and sharing the limited header bits to serve triple duty for reference counting, backup cycle collection with tracing, and opportunistic copying. The combination of these innovations results in a collector that attains great locality for the mutator and very low overhead for reference counting. Measurements on our Java benchmarks show that for all but the smallest of heap sizes RC Immix outperforms the best high performance collector in the literature. In some cases RC Immix performs substantially better.

In summary, this chapter makes the following contributions.

1. We identify heap organization as the remaining performance bottleneck for reference counting.

2. We merge reference counting with the heap structure of Immix by marrying per-line live object counts with object reference counts for reclamation.

3. We identify two opportunities for copying objects — one for young objects and one that leverages the required cycle collector — further improving locality and mitigating fragmentation both proactively and reactively.

4. We develop and evaluate RC Immix, which improves performance by 12% on average compared to RC and sometimes much more, matching or outperforming the fastest production and eliminating the performance barrier to using reference counting.

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

5.2 Motivation

This section motivates our approach. We start with a critical analysis of the performance of our previous chapter's improved reference counter, which we now refer to simply as RC. This analysis shows that inefficiencies derive from: 1) remaining reference counting overheads and 2) poor locality and instruction overhead due to the free-list heap structure.

5.2.1 Motivating Performance Analysis

All previous reference counting implementations in the literature use a free-list allocator because when the collector determines that an object's count is zero, it may then immediately place the freed memory on a free list (See Section 2.5.7). We start our analysis by understanding the performance impact of this choice, using hardware performance counters. We then analyze RC further to establish its problems and opportunities for performance improvements.

5.2.1.1 Free-List and Contiguous Allocation

Recall from Section 2.3.1 and 2.6:

1. A *free-list* allocator organizes memory into k size-segregated free lists and allocates new objects into a free cell in the smallest size class that accommodates the object.

2. A *contiguous* allocator allocates new objects by incrementing a 'bump' pointer by the size of the new object.

Mutator	Immix	Mark-Sweep	Semi-Space
Time	1.000	1.100	1.007
Instructions Retired	1.000	1.071	1.000
L1 Data Cache Misses	1.000	1.266	0.966

Table 5.1: The *mutator* characteristics of mark-sweep relative to Immix using the geometric mean of the benchmarks. GC time is excluded. Free-list allocation increases the number of instructions retired and L1 data cache misses. Semi-space serves as an additional point of comparison.

3. A *mark-region* allocator uses a simple bump pointer to allocate new objects into regions of contiguous memory.

First, to explore the performance impact of free-list allocation, we evaluate *mutator* time, which is the total time minus the collector time, in Table 5.1. We measure mark-region collector Immix, mark-sweep using a free list, and semi-space [2.4.3], across a suite of benchmarks. We compare mutator time of Immix to mark-sweep to cleanly isolate the performance impact of the free-list allocator versus the Immix allocator. Mark-sweep uses the same free-list implementation as RC, and neither Immix nor mark-sweep use barriers in the mutator. We also compare to semi-space. Semi-space is the canonical example of a contiguous allocator and thus an interesting limit point, but it is incompatible with reference counting. The semi-space data confirms that Immix is very close to the ideal for a contiguous allocator.

The contiguous bump allocator has two advantages over the free list, both of which are borne out in Table 5.1. The combined effect is 10% performance advantage. The first advantage of a contiguous allocator is that it improves the cache locality of contemporaneously allocated objects by placing them on the same or nearby cache lines, and interacts well with modern memory systems. Our measurements in Table 5.1 confirm prior results [Blackburn et al., 2004a], showing that a free list adds 27% more L1 data cache misses to the mutator, compared to the Immix contiguous allocator. This degradation of locality has two related sources. 1) Contemporaneously allocated objects are much less likely to share a cache line when using a free list. 2) A contiguous allocator touches memory sequentially, priming the prefetcher to fetch lines before the allocator writes new objects to them. On the other hand, a free-list allocator disperses new objects, defeating hardware prefetching prediction mechanisms. Measurements by Yang et al. [2011] show these prefetching effects.

The second advantage of contiguous allocation is that it uses fewer instructions per allocation, principally because it zeros free memory in bulk using substantially more efficient code [Yang et al., 2011]. The allocation itself is also simpler because it only needs to check whether there is sufficient memory to accommodate the new object and increase the bump pointer, while the free-list allocator has to look up and update the metadata to decide where to allocate. We inspect generated code and confirm the result of Blackburn et al. [2004a] — that in the context of a Java

Mutator	Sticky Immix	RC	Immix
Time	1.000	1.139	0.984
Instructions Retired	1.000	1.092	0.972
L1 Data Cache Misses	1.000	1.329	1.018

Table 5.2: The *mutator* characteristics of RC and Sticky Immix, which except for heap layout have similar features. GC time is excluded. RC's free-list allocator increases instructions retired and L1 cache misses. Immix serves as a point of comparison.

optimizing compiler, where the size of most objects is statically known, the free-list allocation sequence is only slightly more complex than for the bump pointer. The overhead in additional instructions shown in Table 5.1 is therefore solely attributable to the substantially less efficient cell-by-cell zeroing required by a free-list allocator. We measure a 7% increase in the number of retired instructions due to the free list compared to Immix's contiguous allocator.

5.2.1.2 Analyzing RC Overheads

We use a similar analysis to examine mutator overheads in RC by comparing to Sticky Immix [Blackburn and McKinley, 2008; Demers et al., 1990], a generational variant of Immix. We choose Sticky Immix for its similarities to RC. Both collectors a) are mostly non-moving, b) have generational behavior, and c) use similar write barriers. This comparison holds as much as possible constant but varies the heap layout between free list and contiguous.

Table 5.2 compares mutator time, retired instructions, and L1 data cache misses of RC and Sticky Immix. The mutator time of RC is on average 13.9% slower than Sticky Immix, which is reflected by the two performance counters we report. 1) RC has on average 9.2% more mutator retired instructions than Sticky Immix. 2) RC has on average 33% more mutator L1 data cache misses than Sticky Immix. These results are consistent with the hypothesis that RC's use of a free list is the principal source of overhead compared to Sticky Immix, and motivates our design that combines reference counting with the Immix heap structure.

5.3 Design of RC Immix

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

In addition, RC Immix seizes two opportunities for defragmentation using *proactive* and *reactive* opportunistic copying. When identifying new objects for the first time,

it opportunistically copies them, proactively defragmenting. When it, on occasion, performs cyclic garbage collection, RC Immix performs reactive defragmentation.

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

5.3.1 RC and the Immix Heap

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

5.3.1.1 RC Immix Line and Block Reclamation

RC Immix detects free lines by tracking the number of live objects on a line. RC Immix replaces Immix's line mark with a per-line live object count, which counts the number of live objects on the line. It does not count incoming references to the line. Each object is born dead in RC, with a zero reference count to elide all reference counting work for short lived objects. In RC Immix, each line is also born dead with a zero live object count to similarly elide all line counting work when a newly allocated line only contains short lived objects. RC only increments an object's reference count when it encounters it during the first GC after the object is born, either directly from a root or due to an increment from a live mutated object. We propagate this laziness to per-line live object counts in RC Immix.

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

5.3.1.2 Limited Bit Count

In Jikes RVM, one byte (eight bits) is available in the object header for use by the garbage collector. RC uses all eight bits. It uses two bits to log mutated objects for the purposes of coalescing increments and decrements, one bit for the mark state

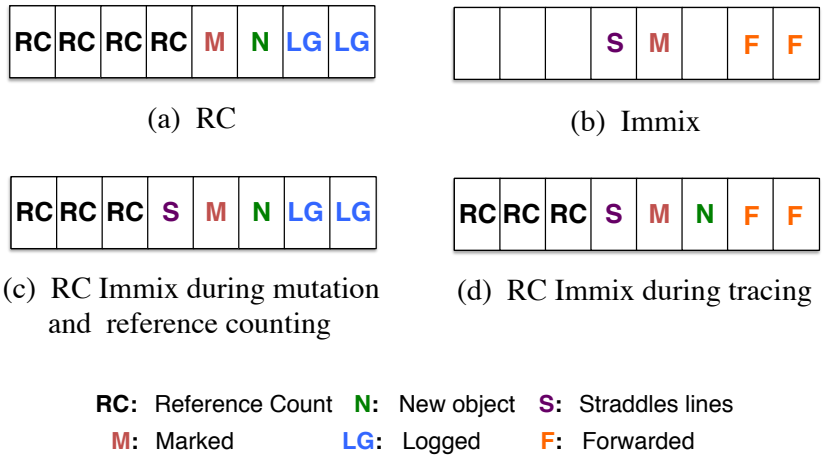


Figure 5.1: How RC, Immix, and the different phases of RC Immix use the eight header bits.

for backup cycle tracing, one bit for identifying new objects, and the remaining four bits to store the reference count. Figure 5.1(a) illustrates how RC fully uses all its eight header bits. Table 4.2 shows that four bits for the reference count is sufficient to correctly count references to more than 99.8% of objects.

To integrate RC and Immix, we need some header bits in objects for Immix-specific functionality as well. The base Immix implementation requires four header bits, fewer header bits than RC, but three bits store different information than RC. Both Immix and RC share the requirement for one mark bit during a tracing collection. Immix however requires one bit to identify objects that span multiple lines and two bits when it forwards objects during copying. (Copying collectors, including Immix and RC Immix, first copy the object and then store a forwarding pointer in the original object's header.) Figure 5.1(b) shows the Immix header bits.

Immix and RC Immix both require a bit to identify objects that may span lines to correctly account for live and dead lines. Immix and RC Immix both use an optimization called conservative marking which means this bit is only set for objects that are larger than one line, which empirically is relatively uncommon [2.6]. Immix stores its line marks in per-block metadata and RC Immix does the same. Immix and RC Immix both need to forward objects during copying. Forwarding uses two bits during a collection to record the forwarding state (not forwarded, being forwarded, or forwarded).

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

We therefore put the two bits to use as follows. 1) During mutation, RC Immix

follows RC, using the logged bits to mark modified objects that it has remembered for coalescing. 2) During a reference counting collection, RC Immix follows RC. For old objects, RC Immix performs increments and decrements as specified by coalescing and then clears the two bits. 3) For new objects and during cycle collection, RC Immix follows Immix. It sets the now cleared bits to indicate that it has forwarded an object and at the end of the collection, reclaims the memory. RC Immix thus overloads the two bits for coalescing and forwarding. Figure 5.1(c) shows how RC Immix uses the header bits during mutation and reference counting. Figure 5.1(d) shows how RC Immix repurposes the logged bits for forwarding during a collection. All the other bits remain the same in both phases.

Consequently, we reduce the number of reference counting bits to three. Three bits will lead to overflow in just 0.65% of objects on average, as shown in Table 4.2. When a reference count is about to overflow, it remains stuck until a cycle collection, at which time it is reset or left stuck depending on the correct count.

Several optimizations and languages such as C# require pinning. Pinned objects are usually identified by a bit in the header. The simplest way to add pinning is to steal another bit from the reference count, reducing it to two bits. A slightly more complex design adds pinning to the logged and forwarded bits, since each of logged and forwarding only require three states. When we evaluated stealing a reference count bit for pinning, it worked well (see Section 5.4.3), so we did not explore the more complex implementation. Our default RC Immix configuration does *not* use pinning.

5.3.2 Cycle Collection and Defragmentation

5.3.2.1 Cycle Collection

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

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

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

5.3.2.2 Defragmentation with Opportunistic Copying

Reference counting is a local operation, meaning that the collector is only aware of the number of references to an object, not their origin. Therefore it is generally not possible to move objects during reference counting. However, RC Immix seizes upon two important opportunities to copy objects and thus mitigate fragmentation. First, we observe that when an object is subject to its first reference counting collection, *all* references to that object will be traversed, giving us a unique opportunity to move the object during a reference counting collection. Because each object is unreferenced at birth, at its first GC, the set of all increments to a new object must be the set of *all* references to that object. Second, we exploit the fact that cycle collection involves a global trace, and thus presents another opportunity to copy objects. In both cases, we use *opportunistic* copying. Opportunistic copying mixes copying with in-place reference counting and marking such that it can stop copying when it exhausts the available memory.

5.3.2.3 Proactive Defragmentation

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

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

Our heuristic seeks to mimic the behavior of a generational collector, while making the copy reserve as small as possible. Ideally, an oracle would tell us the survival rate of the next collection (e.g., 10%) and the collector would size the copy reserve accordingly. We seek to emulate this policy by using past survival rate to predict the future. Computing fine-gain byte or object survival in production requires looking up every object's size, which is too expensive. Instead, we use line survival rate as an estimate of byte survival rate. We compute line survival rates of partially full blocks

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

Table 5.3: Benchmark characteristics. Bytes allocated into the RC Immix heap and minimum heap, in MB. The average survival rate as a percentage of bytes, objects, lines, and blocks measured in an instrumentation run at 1.5× the minimum heap size. Block survival rate is too coarse to predict byte survival rates. Line survival rate is fairly accurate and adds no measurable overhead.

	Heap Size		
Heuristic	**1.2×**	**1.5×**	**2×**
MAX	0.990	0.984	0.976
EXP	0.994	0.990	0.982

Table 5.4: Two proactive copying heuristics and their performance at 1.2, 1.5 and 2 times the minimum heap size, averaged over all benchmarks. Time is normalized relative to Gen Immix. Lower is better.

when we scan the line marks in a block to recycle its lines. This computation adds no measurable overhead.

Table 5.3 shows the average byte, object, line, and block percentage survival rates. Block survival rates significantly over-predict actual byte survival rates. Line survival rates over-predict as well, but much less. The difference between line and block survival rate is an indication of fragmentation. The larger the difference between the two, the more live objects are spread out over the blocks and the less likely a fresh allocation of a multi-line object will fit in the holes (contiguous free lines).

We experimented with a number of heuristics and choose two effective ones. We call our default copy reserve heuristic MAX. MAX simply takes the maximum survival rate of the last N collections (4 in our experiments). Also good, but more complex, is a heuristic we call EXP. EXP computes a moving window of survival rates in buckets of N bytes of allocation (32 MB in our experiments) and then weights each bucket by an exponential decay function (1 for the current bucket, 1/2 for the next oldest, 1/4, and so on). Table 5.4 shows that the simple MAX heuristic performs well.

5.3.2.4 Reactive Defragmentation

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

A key empirical question for cycle detection and defragmentation is how often to perform them. If we perform them too often, the system loses its incrementality and pays both reference counting and tracing overheads. If we perform them too infrequently, it takes a long time to reclaim objects kept alive by dead cycles and

Threshold		Heap Size		
Cycle	Defrag	1.2×	1.5×	2×
1%	**1%**	**0.990**	**0.983**	**0.975**
5%	5%	0.996	0.983	0.976
10%	10%	1.023	0.993	0.980

Table 5.5: Sensitivity to frequency of cycle detection and reactive defragmentation at 1.2, 1.5 and 2 times the minimum heap size, averaged over all benchmarks. Time is normalized relative to Gen Immix. Lower is better.

the heap may suffer a lot of fragmentation. Both waste memory. This threshold is necessarily a heuristic. We explore thresholds as a function of heap size.

We use the following principle for our heuristic. If at the end of a collection, the amount of free memory available for allocation falls below a given threshold, then we mark the next collection for cycle collection. We can always include defragmentation with cycle detection, or we can perform it less frequently. Triggering cycle collection and defragmentation more often enables applications to execute in smaller minimum heap sizes, but will degrade performance. Depending on the scenario, this choice might be desirable. We focus on performance and use a free memory threshold which is a fraction of the total heap size. We experiment with a variety of thresholds to pick the best values for both and show the results for three heap sizes in Table 5.5. Based on the results in Table 5.5, we use 1% for both.

5.3.3 Reference versus Object Level Coalescing

When Levanoni and Petrank [2001] first described coalescing of reference counts, they remembered the address and value of each reference when it was first mutated. However, in practice it is easier to remember the address and contents of each object when the first of its reference fields is mutated. In the first case, the collector compares the GC-time value of the reference with the remembered value and decrements the count for the object referred to by the remembered reference and increments the count for the object referred to by the latest value of the reference. With object level coalescing, each reference within the object is remembered and compared. The implementation challenge is due to the need to only remember each reference once, and therefore efficiently record somewhere that a given reference had been remembered. Using a bit in the object's header makes it easy to do coalescing at object granularity. Both RC and RC Immix use object level coalescing. As part of this work, we implemented reference level coalescing. We did this by stealing a high order bit within each reference to record whether that reference had been remembered. We then map two versions of each page to a single physical page (each one corresponding to the two possible states of the high order bit). We must also modify the JVM's object equality tests to ensure that the stolen bit is ignored in any equality test. We were disappointed to find that despite the low overhead bit stealing approach we devised, we saw no

performance advantage in using reference level coalescing. Indeed, we observed a small slowdown. We investigated and noticed that reference level coalescing places a small but uniform overhead on each pointer mutation, but the potential benefit for the optimization is dominated by the young object optimizations implemented in our improved reference counting (RC) and RC Immix. As a result, we use object level coalescing in RC Immix.

5.3.4 Optimized Root Scanning

The existing implementation of the RC algorithm treats Jikes RVM's boot image as part of the root set, enumerating each reference in the boot image at each collection. We identified this as a significant bottleneck in small heaps and instead treat the boot image as a non-collected part of the heap, rather than part of the root set. This very simple change, which logs mutations to boot image objects instead of scanning them at each collection, delivers a significant performance boost to RC in modest heaps and is critical to RC Immix's performance in small heaps (Figure 5.3(a)).

5.4 Results

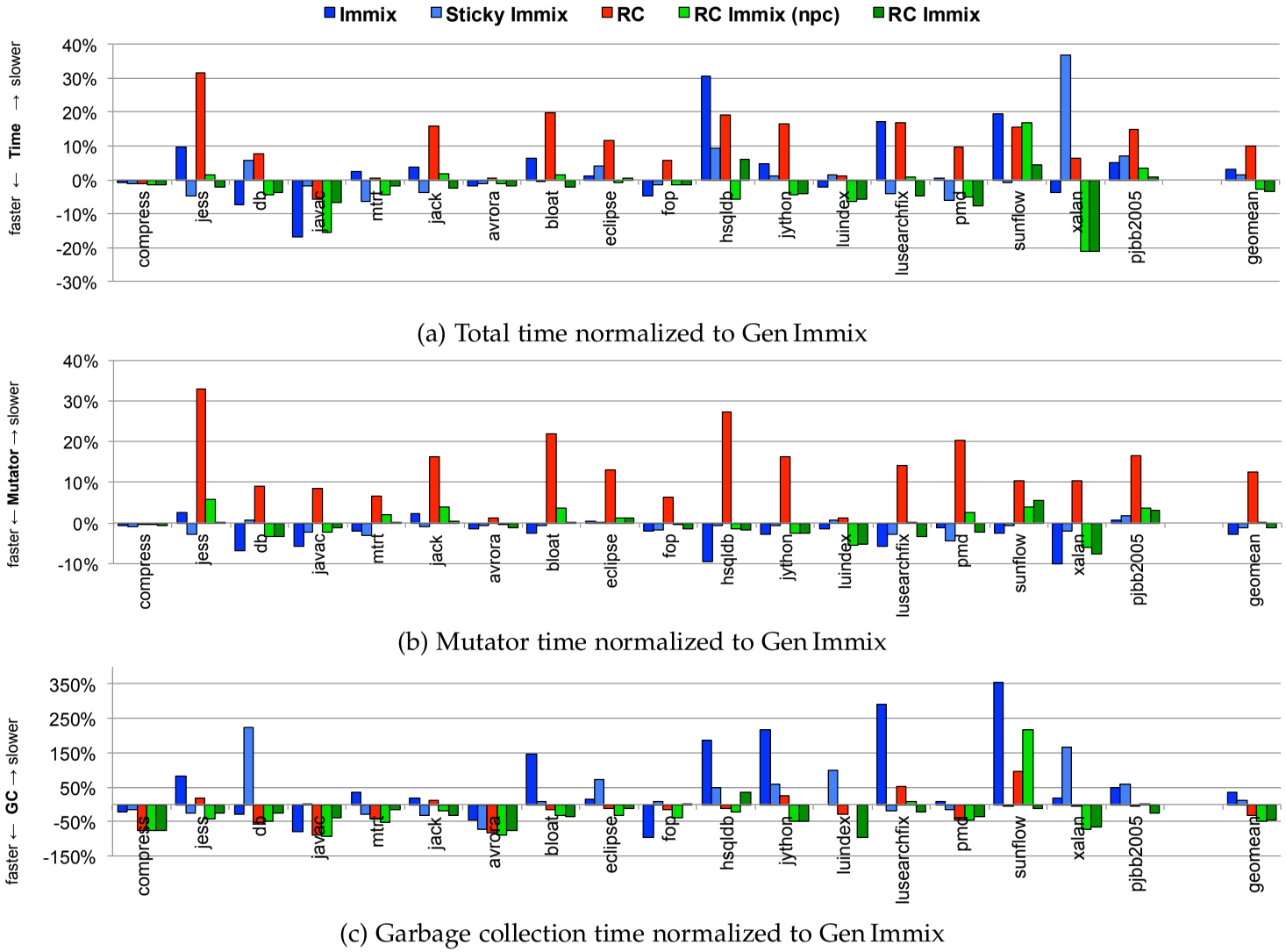
We first compare RC Immix with other collectors at a moderate 2× heap size, then consider sensitivity to available memory, and perform additional in-depth analysis.

5.4.1 RC Immix Performance Overview

Table 5.6 and Figure 5.2 compare total time, mutator time, and garbage collection time of RC Immix and RC Immix without proactive copying (npc) against a number of collectors. The figure illustrates the data and the table includes raw performance as well as relative measurements of the same data. This analysis uses a moderate heap size of 2× the minimum in which all collectors can execute each benchmark. We explore the space-time tradeoff in more detail in Section 5.4.2. In Figure 5.2(c), results are missing for some configurations on some benchmarks. In each of these cases, either the numerator or denominator or both performed no GC (see Table 5.6).

The table and figure compare six collectors.

1. Full heap Immix [2.6].
2. Gen Immix, which uses a copying nursery and an Immix mature space [Blackburn and McKinley, 2008].
3. Sticky Immix, which uses Immix with an in-place generational adaptation [Blackburn and McKinley, 2008; Demers et al., 1990].
4. RC from Chapter 4.
5. RC Immix (npc) which excludes proactive copying and performs well in moderate to large heaps due to very low collection times.
6. RC Immix as described in the previous section.



(a) Total time normalized to Gen Immix

(b) Mutator time normalized to Gen Immix

(c) Garbage collection time normalized to Gen Immix

Figure 5.2: RC Immix performs 3% better than Gen Immix, the highest performance generational collector, at 2× minimum heap size.

Benchmark	Gen Immix			Immix			Sticky Immix			RC			RC Immix (npc)			RC Immix		
	milliseconds			————			————			*Normalized to Gen Immix*			————			————		
	time	$time_{mu}$	$time_{gc}$	time	$time_{mu}$	$time_{gc}$	time	$time_{mu}$	$time_{gc}$	time	$time_{mu}$	$time_{gc}$	time	$time_{mu}$	$time_{gc}$	time	$time_{mu}$	$time_{gc}$
compress	1760 ±0.3	1741 ±0.2	19 ±10.0	0.99 ±0.2	0.99 ±0.2	0.79 ±6.1	0.99 ±0.2	0.99 ±0.2	0.86 ±6.9	0.99 ±0.2	1.00 ±0.2	0.24 ±2.2	0.99 ±0.2	1.00 ±0.1	0.24 ±2.2	0.99 ±0.2	0.99 ±0.2	0.25 ±2.0
jess	355 ±0.3	323 ±0.2	32 ±2.6	1.10 ±0.4	1.03 ±0.2	1.82 ±4.6	0.95 ±0.3	0.97 ±0.3	0.75 ±2.0	1.32 ±0.4	1.33 ±0.3	1.19 ±2.8	1.02 ±0.4	1.06 ±0.3	0.59 ±1.9	0.98 ±0.8	1.00 ±0.8	0.76 ±2.0
db	1238 ±0.3	1209 ±0.3	29 ±2.2	0.93 ±0.6	0.93 ±0.6	0.71 ±2.5	1.06 ±0.3	1.01 ±0.3	3.22 ±5.7	1.08 ±0.4	1.09 ±0.4	0.49 ±2.8	0.96 ±0.4	0.97 ±0.5	0.51 ±1.1	0.96 ±0.5	0.97 ±0.5	0.74 ±1.4
javac	773 ±0.2	661 ±0.2	112 ±1.1	0.83 ±0.2	0.94 ±0.2	0.19 ±0.5	0.98 ±0.4	0.98 ±0.4	1.03 ±1.6	0.94 ±0.2	1.09 ±0.2	0.11 ±0.2	0.84 ±0.3	0.98 ±0.3	0.07 ±0.2	0.93 ±4.7	0.99 ±0.8	0.62 ±28.4
mpegaudio	1103 ±0.0	1103 ±0.0	0 ±0.0	1.00 ±0.0	1.00 ±0.0	0.00 ±0.0	1.00 ±0.2	1.00 ±0.2	0.00 ±0.0	1.00 ±0.0	1.00 ±0.0	0.00 ±0.0	1.00 ±0.2	1.00 ±0.2	0.00 ±0.0	1.00 ±0.3	1.00 ±0.3	0.00 ±0.0
mtrt	245 ±1.5	215 ±1.6	30 ±2.9	1.02 ±1.3	0.98 ±1.2	1.34 ±4.5	0.94 ±1.3	0.97 ±1.2	0.71 ±5.2	1.01 ±1.2	1.07 ±1.2	0.57 ±3.9	0.95 ±2.2	1.02 ±2.5	0.49 ±3.1	0.98 ±1.2	1.00 ±1.2	0.84 ±4.6
jack	496 ±0.3	453 ±0.2	43 ±2.7	1.04 ±0.3	1.02 ±0.2	1.19 ±3.1	0.96 ±0.3	0.99 ±0.2	0.67 ±2.1	1.16 ±0.3	1.16 ±0.2	1.13 ±2.6	1.02 ±0.7	1.04 ±0.7	0.81 ±2.1	0.98 ±0.5	1.00 ±0.4	0.67 ±2.4
mean	**811** ±0.4	**767** ±0.4	**44** ±3.6															
geomean				**0.98**	**0.98**	**0.83**	**0.98**	**0.98**	**1.00**	**1.08**	**1.12**	**0.46**	**0.96**	**1.01**	**0.35**	**0.97**	**0.99**	**0.60**
avrora	2266 ±0.3	2250 ±0.3	16 ±3.3	0.98 ±0.3	0.98 ±0.3	0.54 ±2.1	0.99 ±0.2	0.99 ±0.2	0.27 ±2.6	1.01 ±0.3	1.01 ±0.3	0.18 ±1.1	0.99 ±0.3	1.00 ±0.3	0.10 ±0.3	0.98 ±0.2	0.99 ±0.2	0.24 ±9.9
bloat	2179 ±0.4	2047 ±0.5	132 ±1.3	1.07 ±1.0	0.98 ±1.0	2.47 ±4.5	1.00 ±0.5	0.99 ±0.6	1.08 ±1.6	1.20 ±0.6	1.22 ±0.6	0.86 ±1.3	1.02 ±0.7	1.04 ±0.7	0.67 ±2.2	0.98 ±1.0	1.00 ±1.1	0.63 ±1.4
eclipse	11272 ±0.9	10654 ±1.0	618 ±1.1	1.01 ±0.8	1.00 ±0.9	1.15 ±2.9	1.04 ±0.9	1.00 ±1.0	1.72 ±1.9	1.12 ±0.9	1.13 ±1.0	0.88 ±1.1	0.99 ±0.8	1.01 ±0.8	0.66 ±1.4	1.00 ±1.2	1.01 ±1.2	0.87 ±2.1
fop	579 ±0.5	562 ±0.5	17 ±2.3	0.95 ±0.5	0.98 ±0.4	0.05 ±10.0	0.99 ±0.4	0.98 ±0.4	1.07 ±2.2	1.06 ±0.5	1.06 ±0.5	0.86 ±1.7	0.99 ±0.6	1.00 ±0.4	0.62 ±9.7	0.99 ±0.4	0.99 ±0.4	1.02 ±3.8
hsqldb	706 ±0.5	561 ±0.1	145 ±2.5	1.30 ±2.3	0.91 ±2.9	2.85 ±5.1	1.09 ±1.1	0.99 ±0.1	1.48 ±5.8	1.19 ±0.5	1.27 ±0.2	0.88 ±1.6	0.94 ±0.4	0.99 ±0.1	0.78 ±1.4	1.06 ±0.5	0.98 ±0.1	1.36 ±2.8
jython	2416 ±0.4	2335 ±0.4	81 ±1.7	1.05 ±0.5	0.97 ±0.4	3.18 ±9.0	1.01 ±0.4	0.99 ±0.4	1.58 ±2.4	1.17 ±0.7	1.16 ±0.8	1.27 ±2.0	0.96 ±0.4	0.97 ±0.4	0.50 ±3.3	0.96 ±0.3	0.98 ±0.3	0.52 ±1.1
luindex	637 ±7.8	632 ±7.8	5 ±6.8	0.98 ±6.8	0.99 ±6.8	0.00 ±0.0	1.02 ±9.3	1.01 ±9.3	1.99 ±17.7	1.01 ±8.0	1.01 ±8.1	0.71 ±4.4	0.94 ±6.2	0.95 ±6.2	0.00 ±0.0	0.94 ±6.1	0.95 ±6.2	0.04 ±8.4
lusearch	1306 ±0.4	782 ±0.6	524 ±0.4	1.21 ±0.7	0.79 ±0.7	1.83 ±1.1	1.34 ±1.4	0.85 ±0.5	2.06 ±3.3	0.89 ±0.7	1.07 ±1.0	0.63 ±0.6	0.63 ±0.6	0.80 ±0.8	0.36 ±0.4	0.62 ±0.4	0.79 ±0.5	0.36 ±0.3
lusearchfix	539 ±1.3	497 ±1.3	42 ±1.2	1.17 ±1.2	0.94 ±1.2	3.89 ±4.8	0.96 ±1.4	0.97 ±1.5	0.80 ±1.3	1.17 ±1.4	1.14 ±1.5	1.52 ±1.9	1.01 ±1.3	1.00 ±1.4	1.08 ±2.0	0.95 ±1.3	0.97 ±1.4	0.78 ±1.0
pmd	621 ±0.9	521 ±0.8	100 ±3.5	1.01 ±0.8	0.99 ±0.8	1.10 ±3.3	0.94 ±0.7	0.96 ±0.7	0.85 ±2.6	1.10 ±0.9	1.20 ±0.8	0.56 ±1.8	0.95 ±0.8	1.02 ±0.8	0.55 ±2.7	0.92 ±0.9	0.98 ±0.9	0.64 ±3.3
sunflow	1725 ±1.1	1619 ±1.2	106 ±0.9	1.19 ±1.1	0.97 ±1.0	4.56 ±8.3	0.99 ±1.4	0.99 ±1.5	0.96 ±1.3	1.16 ±1.0	1.10 ±1.0	1.94 ±2.9	1.17 ±1.1	1.04 ±0.9	3.18 ±8.4	1.05 ±1.2	1.06 ±1.3	0.88 ±3.2
xalan	754 ±0.6	579 ±0.7	175 ±1.0	0.96 ±0.5	0.90 ±0.6	1.18 ±1.3	1.37 ±1.4	0.98 ±0.7	2.65 ±5.4	1.07 ±1.0	1.10 ±0.9	0.93 ±3.1	0.79 ±0.7	0.94 ±0.9	0.29 ±0.4	0.79 ±0.6	0.92 ±0.7	0.34 ±0.5
mean	**2154** ±1.3	**2023** ±1.3	**131** ±2.2															
geomean				**1.06**	**0.96**	**1.90**	**1.03**	**0.99**	**1.14**	**1.11**	**1.13**	**0.84**	**0.97**	**1.00**	**0.60**	**0.96**	**0.98**	**0.51**
pjbb2005	2870 ±0.4	2606 ±0.3	264 ±2.1	1.05 ±0.7	1.01 ±0.4	1.48 ±5.4	1.07 ±0.4	1.02 ±0.3	1.60 ±2.9	1.15 ±0.9	1.17 ±0.4	0.97 ±8.4	1.04 ±0.5	1.04 ±0.4	1.02 ±3.6	1.01 ±0.9	1.03 ±0.4	0.76 ±7.7
min	245	215	5	0.83	0.90	0.00	0.94	0.96	0.27	0.94	1.00	0.11	0.79	0.94	0.00	0.79	0.92	0.04
max	11272	10654	618	1.30	1.03	4.56	1.37	1.02	3.22	1.32	1.33	1.94	1.17	1.06	3.18	1.06	1.06	1.36
mean	**1746** ±0.9	**1637** ±0.9	**109** ±2.5															
geomean				**1.03**	**0.97**	**1.37**	**1.02**	**0.99**	**1.11**	**1.10**	**1.13**	**0.69**	**0.97**	**1.00**	**0.51**	**0.97**	**0.99**	**0.55**

Table 5.6: Total, mutator, and collection performance at 2× minimum heap size with confidence intervals. Figure 5.2 graphs these results. We report milliseconds for Gen Immix and normalize the others to Gen Immix. (We exclude mpegaudio and lusearch from averages.) RC Immix performs 3% better than production Gen Immix.

Mutator	Gen Immix	RC	RC Immix
Time	1.000	1.126	0.989
Instructions Retired	1.000	1.094	1.012
L1 Data Cache Misses	1.000	1.313	1.043

Table 5.7: Mutator performance counters show RC Immix solves the instruction overhead and poor locality problems in RC. Applications executing RC Immix compared with Gen Immix in a moderate heap size of 2× the minimum execute the same number of retired instructions and see only a slightly higher L1 data cache miss rate. Comparing RC to RC Immix, RC Immix reduces cache miss rates by around 20%.

We normalize to Gen Immix since it is the best performing in the literature across all heap sizes and consequently is the default production collector in Jikes RVM. All of the collectors, except RC, defragment when there is an opportunity, i.e., there are partially filled blocks without fresh allocation and fragmentation is high, as described in Section 5.3.2.

These results show that RC Immix outperforms the best performing garbage collector at this moderate heap size and completely eliminates the reference counting performance gap. The time_{gc} result shows that, not surprisingly, Immix, the only full heap collector that does not exploit any generational behaviors, has the worst collector performance, degrading by on average 37%. Since garbage collection time is a relatively smaller influence on total time in a moderate heap, all but RC perform similarly on total time. At this heap size RC Immix performs the same as RC Immix (npc), but its worse-case degradation is just 6% while its best case improvement is 21%. By comparison, RC Immix (npc) has a worst case degradation of 17% and best case improvement of 21%. Table 5.6 and Figure 5.2(c) show that RC Immix (npc) has the best garbage collection time, outperforming Gen Immix by 49%. As we show later, RC Immix has an advantage over RC Immix (npc) when memory is tight and fragmentation is a bigger issue.

The time_{mu} columns of Table 5.6 and Figure 5.2(b) show that RC Immix matches or beats the Immix collectors with respect to mutator performance and improves significantly over RC in a moderate heap. The reasons that RC Immix improves over RC in total time stem directly from improvements in mutator performance. RC mutator time is 13% worse than any other system, as we reported in Table 5.2 and discussed in Section 5.2. RC Immix completely eliminates this gap in mutator performance.

Table 5.7 summarizes the reasons for RC Immix's improvement over RC by showing the number of mutator retired instructions and mutator L1 data cache misses for RC and RC Immix normalized to Gen Immix. RC Immix solves the instruction overhead and poor locality problems in RC because by using a bump pointer, it wins twice.

First, it gains the advantage of efficient zeroing of free memory in lines and blocks, rather than zeroing at the granularity of each object when it dies or is recycled in the

free list (see Section 5.2 and measurements by Yang et al. [2011]). Second, it gains the advantage of contiguous allocation in memory of objects allocated together in time. This heap layout induces good cache behavior because objects allocated and used together occupy the same cache line, and because the bump pointer marches sequentially through memory, the hardware prefetcher correctly predicts the next line to fetch, so it is in cache when the program (via the memory allocator) accesses it. Prefetching measurements by Yang et al. [2011] quantify this effect. Table 5.7 shows that compared to RC, RC Immix reduces cache misses by around 20% (1.043/1.313). Gen Immix has slightly lower cache miss rates than RC Immix, which makes sense because it always allocates new objects contiguously (sequentially) whereas RC Immix sometimes allocates into partially full blocks and must skip over occupied lines.

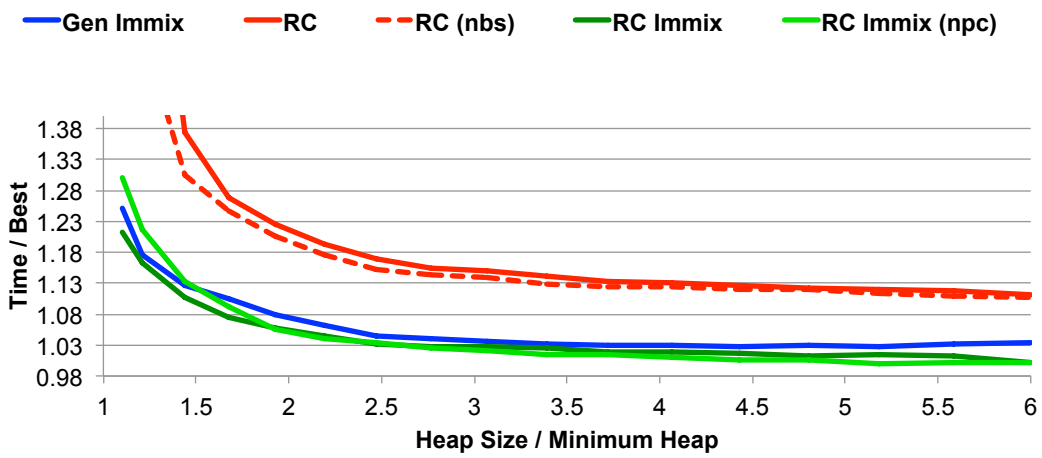
5.4.2 Variable Heap Size Analysis

Garbage collection is fundamentally a time-space tradeoff, which this section examines by varying the heap size. Figure 5.3 evaluates RC Immix performance as a function of available memory. Each of the three graphs varies heap size between 1× and 6× the minimum in which all collectors can execute the benchmark. 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. The graphs plot total time, mutator time, and garbage collection time as a geometric mean of the benchmarks, showing Gen Immix, RC, RC (nbs) with no boot image scanning (Section 5.3.4), RC Immix and RC Immix (npc) with no proactive copying. Figure 5.3(a) shows total time, and reveals that RC Immix dominates RC at all heap sizes, and consistently outperforms Gen Immix at heap sizes above 1.2× the minimum. Figures 5.3(b) and 5.3(c) reveal the source of the behavior of RC Immix seen in Figure 5.3(a). Figure 5.3(b) reveals that the mutator performance of RC Immix is consistently good. This graph makes it clear that the underlying heap structure has a profound impact on mutator performance. Figure 5.3(c) shows that in garbage collection time, RC Immix outperforms RC in tighter heaps, matches Gen Immix at heap size 1.2× the minimum and outperforms Gen Immix at heap sizes above 1.3× the minimum.

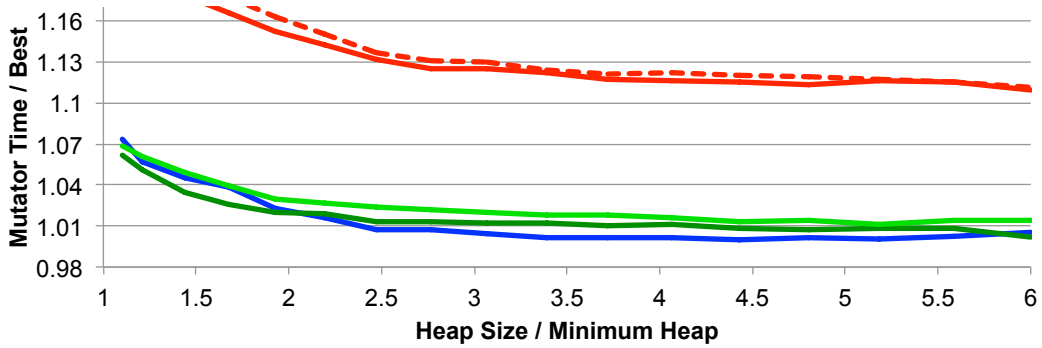
5.4.3 Pinning

We conducted a simple experiment to explore the tradeoff associated with dedicating a header bit for pinning (see Section 5.3.1). While a pinning bit could be folded into the logged and forwarding bits, in this case we simply trade pinning functionality for reduced reference counting bits. In Jikes RVM, pinning can be utilized by the Java standard libraries to make IO more efficient, so although no Java application can exploit pinning directly, there is a potential performance benefit to providing pinning support.

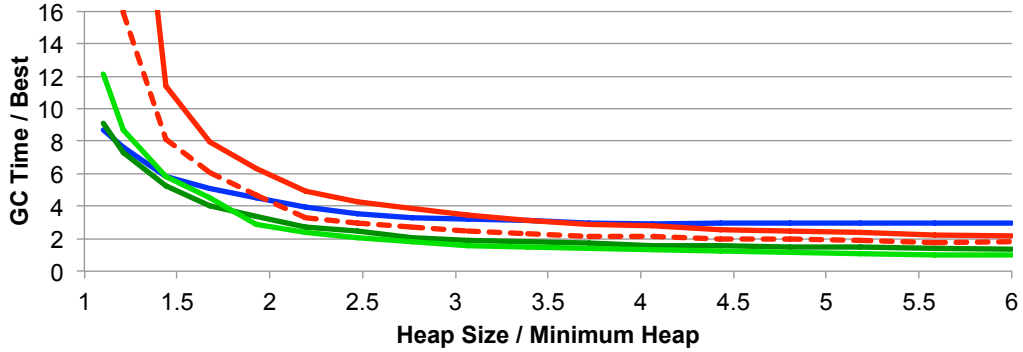
Table 5.8 shows the result of a simple experiment with three configurations at three heap sizes. The performance numbers are normalized to Gen Immix and represent the geometric mean of all benchmarks. In the first row, we have three reference



(a) Total time



(b) Mutator time



(c) Garbage collection time

Figure 5.3: The performance of Gen Immix, RC, RC with no boot image scanning, RC Immix, and RC Immix with no proactive copying as a function of heap size.

Bits Used		Heap Size		
count	pin	1.2×	1.5×	2×
3	0	0.994	0.986	0.976
2	0	0.990	0.982	0.979
2	1	0.988	0.981	0.974

Table 5.8: Performance sensitivity of RC Immix with pinning bit at 1.2, 1.5 and 2 times the minimum heap size, averaged over all benchmarks. Time is normalized relative to Gen Immix. Lower is better.

counting bits and no pinning support, which is the default configuration used in RC Immix. Then we reduce the number of reference counting bits to two, *without* adding pinning. Finally we use two reference counting bits and add support for pinning. The results show that to the first approximation, the tradeoff is not significant, with the performance variations all being within 0.6% of each other. Although the variations are small, the numbers are intriguing. We see that at the 2× heap, the introduction of pinning improved total performance by around 0.5% when holding the reference counting bits constant.

5.4.4 Benchmark Analysis

Table 5.6 reveals that sunflow is significantly slower on RC Immix (npc) than Gen Immix, whereas xalan and lusearch are significantly faster when using RC Immix. We now analyze these outlier results.

Sunflow Table 5.6 shows that sunflow is 17% slower in total time on RC Immix (npc) than Gen Immix, and that this slowdown is *entirely* due to a garbage collection slowdown of 3.18×. The source of this problem appears to be high fragmentation among surviving young objects in sunflow. It was this observation that encouraged us to explore proactive defragmentation, and this benchmark shows that the strategy is hugely effective, as RC Immix is only 5% slower than Gen Immix. sunflow has a high allocation rate [Yang et al., 2011], and our observation that Gen Immix does a large number of nursery collections, but no mature space collections at 2× minimum heap size confirms this behavior. RC Immix (npc) does a large number of collections, many of which are defragmenting cycle collections, and yet sunflow has few cycles [4.3.6]. Furthermore, Table 5.3 shows that although the line survival rate for sunflow is 5%, the block survival rate is a remarkable 99%. This indicates that surviving objects are scattered in the heap generating fragmentation, thus Immix blocks are being kept alive unnecessarily. We also established empirically that sunflow's performance degraded substantially if the standard defragmentation heuristic was made less aggressive.

Xalan Both RC Immix (npc) and RC Immix perform very well on xalan, principally because they have lower GC time than Gen Immix. RC Immix (npc) has 71% lower GC time than Gen Immix and RC Immix has 66% lower GC time than Gen Immix. xalan has a large amount of medium lifetime objects, which can only be recovered by a full heap collection with Gen Immix, but are recovered in a timely way in both RC Immix (npc) and RC Immix.

Lusearch RC Immix performs much better on lusearch than Gen Immix. In fact Gen Immix has substantially worse mutator time than any other system. This result is due to the bug in lusearch that causes the allocation of a very large number of medium sized objects, leading Gen Immix to perform over 800 nursery collections, destroying mutator locality. The allocation pathology of lusearch is established [3.1] and is the reason why we use lusearch-fix in our results, exclude lusearch from all of our aggregate (mean and geomean) results, and leave it greyed out in Table 5.6. If we were to include lusearch in our aggregate results then both RC Immix (npc) and RC Immix would be 5% faster in geomean than Gen Immix.

5.5 Summary

Prior to this work, all reference counting implementations including RC, the collector we introduced in Chapter 4, use a free-list allocator. A free-list allocator suffers from fragmentation and poor cache locality compared to a contiguous allocator [Blackburn et al., 2004a]. The chapter identifies the free-list heap organization as the main reason for the 10% performance gap between RC and a production generational tracing collector, Gen Immix. This chapter introduces a new collector, RC Immix, that replaces the free list with the line and block heap structure of Immix, and is the first to combine copying with reference counting, which mitigates fragmentation. RC Immix outperforms a highly tuned production generational collector (Gen Immix) — the first reference counting collector to achieve that milestone.

RC Immix, like other high performance collectors and each of the collectors discussed up to this point in the thesis, is an exact garbage collector. It depends on exact identification of references in the stacks and registers, which requires compiler cooperation and significant engineering effort. Many implementations of other popular languages avoid the engineering headache of exact garbage collection by using conservative garbage collection but do so at the cost of significant performance overheads. The next chapter will focus on achieving high performance garbage collection in conservative settings.

Chapter 6

Fast Conservative Garbage Collection

This chapter examines conservative garbage collection for managed languages. Conservative garbage collectors are robust to ambiguity due to either language imprecision or lack of engineering support for exactness. However, conservatism comes with a significant performance overhead. This chapter identifies the non-moving free-list heap structure as the main source of this overhead. We present the design and implementation of conservative Immix and conservative RC Immix collectors, which overcome the limitations of a free-list heap structure. The conservative RC Immix collector matches the performance of a well tuned exact generational copying collector, Gen Immix, showing that conservative garbage collection is compatible with high performance for managed languages.

This chapter is structured as follows. Section 6.2 describes the design of our object map filtering mechanism for ambiguous roots and our family of conservative Immix and reference counting collectors including conservative RC Immix. Section 6.3 performs the detailed study of the impact of conservatism on collector mechanisms and design including excess retention, ambiguous root filtering, and pinning. Section 6.4 evaluates our family of conservative collectors with their exact counterparts and evaluates conservative RC Immix collector with high performance exact RC Immix collector, with the best copying generational exact tracing collector Gen Immix, and with two widely used conservative collector Boehm-Demers-Weiser (BDW) and Mostly Copying (MCC).

This chapter describes work published as "Fast Conservative Garbage Collection" [Shahriyar, Blackburn, and McKinley, 2014].

6.1 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 [2.7.2].

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 chapter explores conservative collectors for managed languages 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 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 the choice of 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 [2.7.3]. 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 collector [2.7.4]. 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 [Blackburn et al., 2004a; Blackburn and McKinley, 2008]. To quantify this performance cost, we implement and compare them to a copying generational collection (Gen Immix), the production collector in Jikes RVM. Figure 6.1 summarizes our results, plotting geometric mean of total (mutator + collector) time on our 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 [2.7.1].

An alternative tactic for limiting the compiler burden is naive reference counting, which is used by Objective-C, Perl, Delphi, PHP, and Swift [2.5.1]. These collectors never examine the stack because the compiler or interpreter simply inserts increments and decrements when the program changes an object reference. In Chapter 4 and 5 we quantify the penalty of deferred reference counting, which eliminates increments and

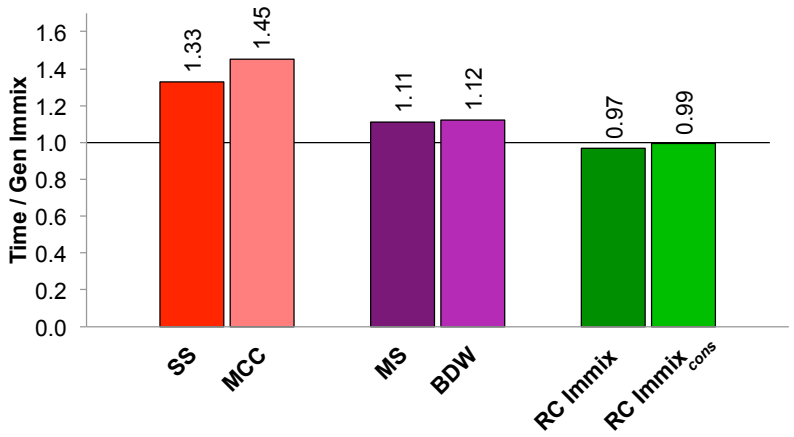


Figure 6.1: Performance of exact semi-space (SS), conservative MCC, exact mark-sweep (MS), conservative BDW, exact RC Immix, and conservative RC Immix$_{cons}$ normalized to exact Gen Immix at a moderate heap size. Lower is better. Prior conservative collectors sacrifice performance. RC Immix$_{cons}$ performs similarly to Gen Immix and RC Immix, the best exact collectors.

decrements on local variables and is thus faster than naive reference counting, 40% compared to a copying generational collector [Deutsch and Bobrow, 1976; Apple Inc., 2014]. All of these language implementations either forbid cycles, leak cycles, or perform costly trial deletion [2.5.6]. Naive reference counting imposes an even larger performance sacrifice.

This chapter 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.

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 6.1 shows that RC Immix$_{cons}$ improves total performance over BDW by 13% on average and results later in the chapter show improvements up to 41%. RC Immix$_{cons}$ delivers excellent performance, competitive with the fastest exact generational collectors.

In summary, this chapter makes the following contributions.

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.
8. We describe how other managed languages may achieve these results.

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

6.2 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.

6.2.1 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 Jikes RVM uses a specific format for its two word header, Jikes RVM 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 6.2).

6.2.2 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 [Blackburn and McKinley, 2008; Demers et al., 1990] 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 (Immix$_{cons}$) and Sticky Immix (Sticky Immix$_{cons}$) 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.

6.2.3 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 [2.5.2]. 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 conservative reference counting (RC$_{cons}$). 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.

6.2.4 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 conservative RC Immix (RC Immix$_{cons}$), we bring together each of the key ideas described above for Immix$_{cons}$ and RC$_{cons}$. 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$_{cons}$.

6.3 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 6.4 quantifies the performance impacts. We report the full statistics for each benchmark in Table 6.1. Then we report aggregate statistics in more detail.

Full Statistics Table 6.1 presents individual benchmark statistics that support the subsequent aggregate analysis. It 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 table presents arithmetic mean for quantities and geometric mean for percentages.

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.

The 'Live Heap' column shows the size of live object graph in MB. 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).

			Roots							Pinned Space								False Pinning	
	Live Heap	No. Force	(/exact unique) Exact		Conservative			Excess Retention		(/pinned obj, KB, % live) Objects		Immix lines			MCC pages			MCC	
Benchmark	MB	GCs	uniq	all	all	filt.	uniq	KB	%	obj.	KB	line	KB	%	page	KB	%	/page	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 6.1: Individual benchmark statistics on live heap size, exact roots, conservative roots, excess retention, and pinning.

	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 6.2: Ambiguous Pointers

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.

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.

The next four subsections discuss these results in mote detail and pull out statistics from Table 6.1 in a summary form.

6.3.1 Ambiguous Pointers

Table 6.2 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. 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

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

Table 6.3: Excess Retention

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, but the total numbers of roots is still low.

6.3.2 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 6.3 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.

6.3.3 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 collector 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 6.2.1 (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 6.2 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 6.3.2, 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.

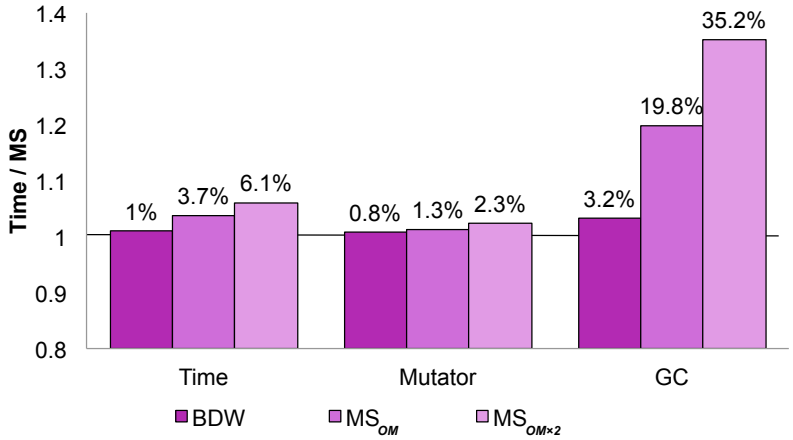


Figure 6.2: Conservative filtering total, mutator, and collection time overheads in mark-sweep. BDW is cheapest, requiring no additional space or allocation work. The smaller object map in MS_{OM} improves over object map filtering in both mutator and collection time.

		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 6.4: Pinning Granularity

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.

6.3.4 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 6.4 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. Section 6.4.4 evaluates the performance effect of artificially increasing the number of objects pinned due to ambiguity.

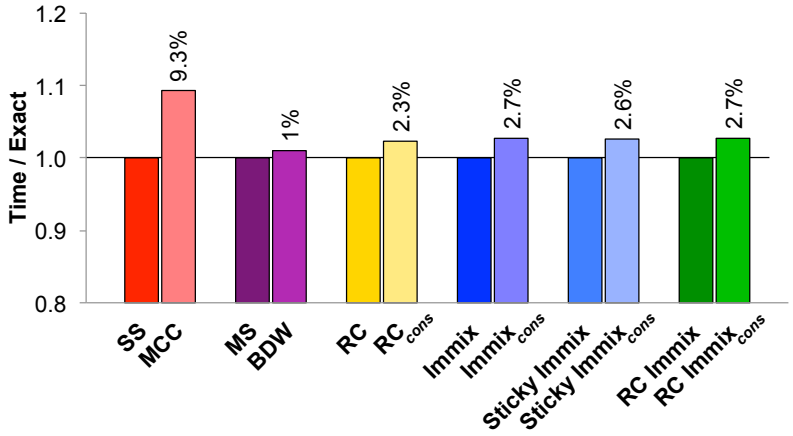
This section establishes that for our Java workloads root processing time and excess retention are not significant problems for conservative collectors, and pinning due to Immix-style lines has roughly an order of magnitude less direct impact than pinning due to MCC pages.

6.4 Performance Evaluation

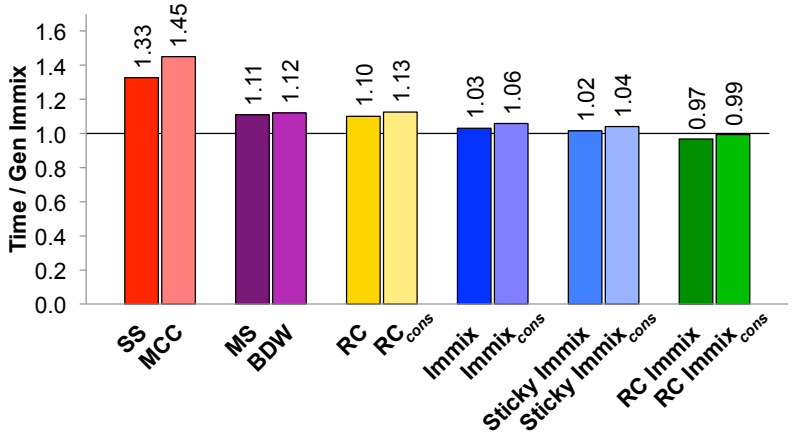
This section evaluates the design and implementation of six conservative collectors: MCC, BDW, RC_{cons}, Immix_{cons}, Sticky Immix_{cons}, and RC Immix_{cons}. We compare them to their exact counterparts: semi-space (SS), mark-sweep (MS), RC, Immix, Sticky Immix, and RC Immix. The conservative mark-sweep 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.

6.4.1 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 6.3(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



(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.

Figure 6.3: Geometric means of total performance for exact and conservative collectors at 2× minimum heap size. Lower is better.

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 6.3 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 6.3(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 the previous chapter [5.2], rather than exact or conservative root processing.

All of the exact and conservative Immix collectors outperform the free-list collectors. Prior work and the previous chapter show that degradations in mutator locality explain this difference [Blackburn et al., 2004a; Blackburn and McKinley, 2008]. 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.

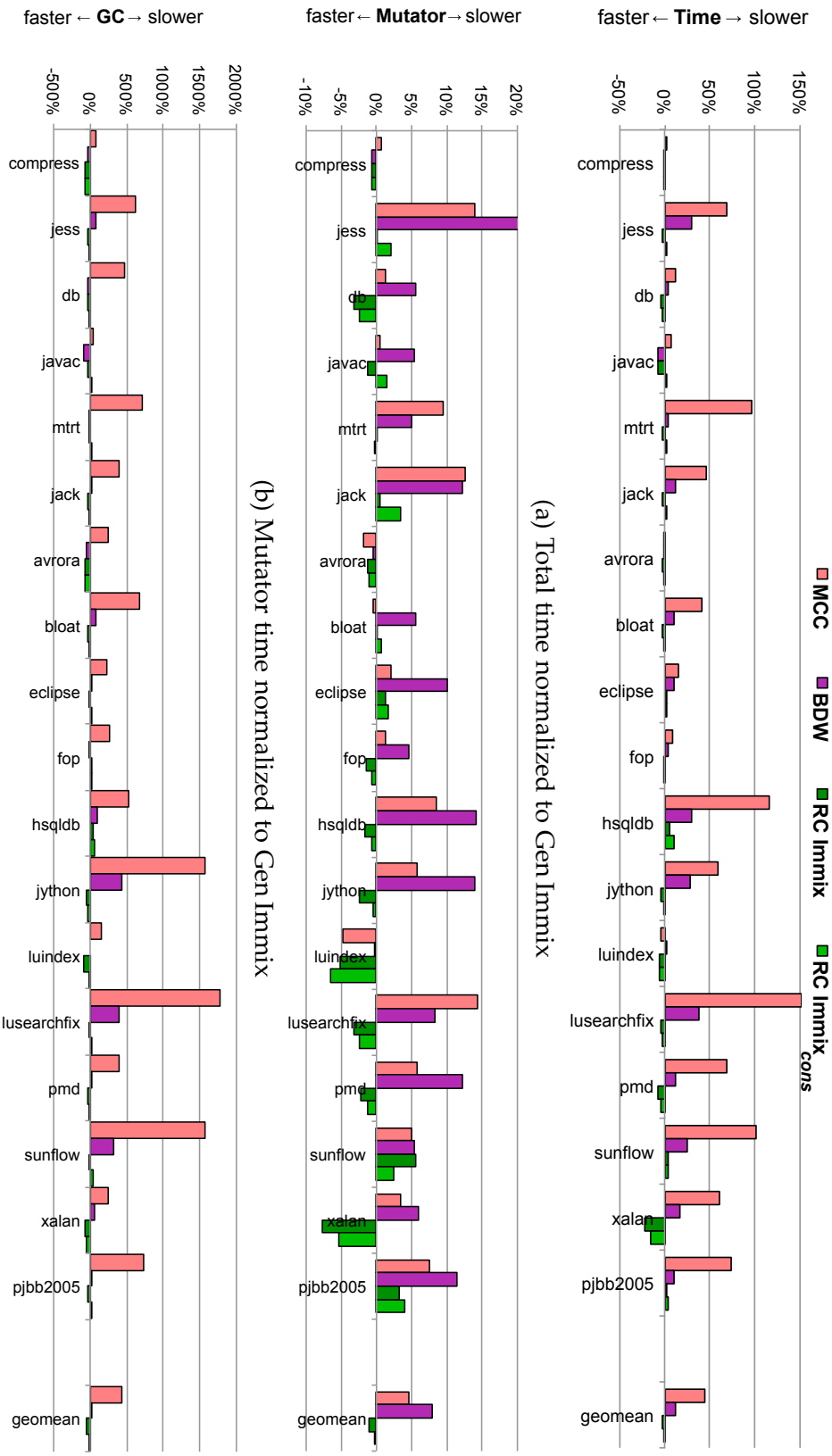
Exact Sticky Immix is only 2% slower and Sticky Immix$_{cons}$ is only 4% slower than Gen Immix. The best performing conservative collector is RC Immix$_{cons}$. Even though conservatism slows it down, it is still 1% faster than Gen Immix.

6.4.2 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 6.5 and graph them in Figure 6.4.

The geometric mean in Figure 6.4(a) and the bottom of the four 'time' columns of Table 6.5 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



(a) Total time normalized to Gen Immix

(b) Mutator time normalized to Gen Immix

(c) Garbage collection time normalized to Gen Immix

Figure 6.4: RC Immix$_{cons}$ matches two exact high performance collectors, RC Immix and Gen Immix, at 2× minimum heap size.

Benchmark	**Gen Immix**			**RC Immix**			**RC Immix$_{cons}$**			**BDW**			**MCC**		
	milliseconds			*Normalized to Gen Immix*											
	time	time$_{mu}$	time$_{gc}$	time	time$_{mu}$	time$_{gc}$	time	time$_{mu}$	time$_{gc}$	time	time$_{mu}$	time$_{gc}$	time	time$_{mu}$	time$_{gc}$
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 6.5: Total, mutator, and collection performance at 2× minimum heap size with confidence intervals. Figure 6.4 graphs these results. We report milliseconds for Gen Immix and normalize the others to Gen Immix. (We exclude mpegaudio and lusearch from averages.) RC Immix$_{cons}$ is 2% slower than RC Immix and *still* slightly faster than production exact Gen Immix.

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 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 6.1). 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 6.4(b) and the four 'time$_{mu}$' columns of Table 6.5 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 [Yang et al., 2012]. Nonetheless, despite not requiring a write barrier, BDW consistently suffers the worst mutator overhead, 8% on average.

BDW collector does not use an object map, and has no other mutator time overheads directly associated with conservatism, so based on the results in Chapter 5 and previous experiments [Blackburn et al., 2004a; Blackburn and McKinley, 2008], we attribute the slowdown to the loss of locality. 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 6.4(c) and the four 'time$_{gc}$' columns of Table 6.5 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 seven and improves eleven compared to Gen Immix. BDW performs worst on all but six 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% in total time 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.

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 well tuned production collector, Gen Immix. 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.

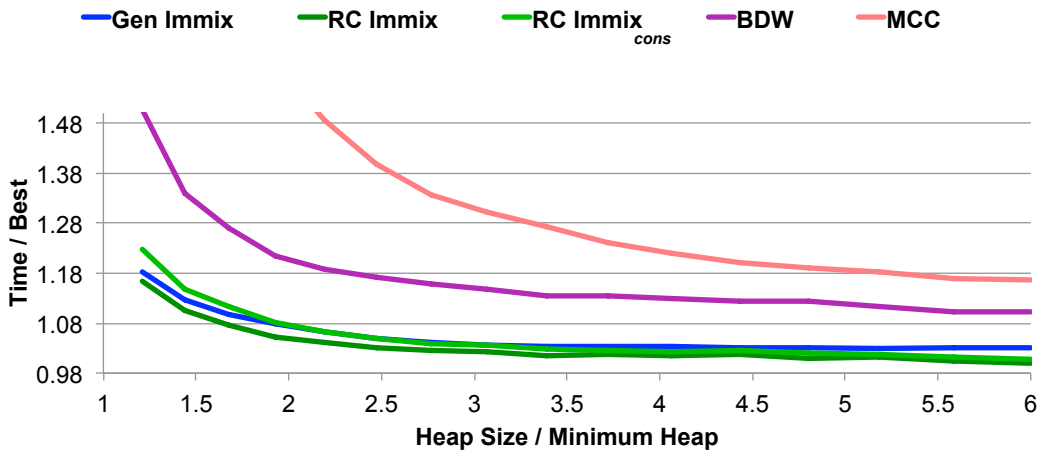
6.4.3 Sensitivity to Heap Size

Garbage collection is fundamentally a time-space tradeoff, which this section examines by varying the heap size. Figure 6.5 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 6.5(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 6.5(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 6.5(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.

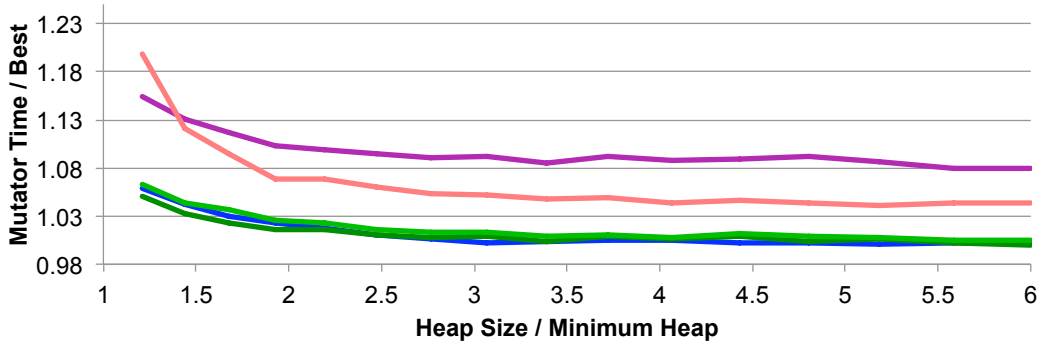
In summary, conservative Immix variants perform very close to their exact counterparts, and RC Immix$_{cons}$ performs as well or better than the best exact generational collector across a wide range of heap sizes.

6.4.4 Discussion and Wider Applicability

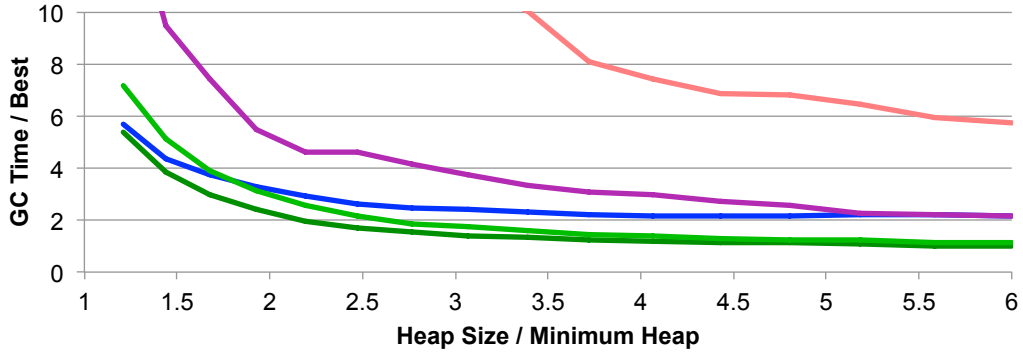
Although our empirical results are for Java, we believe that other languages will benefit from these algorithms.



(a) Total time



(b) Mutator time



(c) Garbage collection time

Figure 6.5: The performance of MCC, BDW, Gen Immix, RC Immix, and RC Immix$_{cons}$ as a function of heap size.

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

Table 6.6: 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.

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 stacks 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 6.6.

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 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.

6.5 Summary

This chapter examines conservative garbage collection for managed languages. Conservative garbage collection avoids the engineering headache of exact garbage collection but suffers significant performance overheads. We find that both excess retention and the number of objects that can't be moved because of conservatism are very low for Java. We identify the non-moving free-list heap structure as the main source of the performance overhead for existing conservative garbage collectors. This chapter identifies Immix's line and block heap structure as a good match for conservative collection because it provides better mutator locality and finer line granularity pinning. We introduce a low overhead object map to validate ambiguous references. With these mechanisms, this chapter introduces the design and implementation of conservative variants of existing garbage collector including conservative Immix and reference counting. In particular, the conservative RC Immix collector, RC Immix$_{cons}$ matches the performance of a highly optimized copying generational collector, Gen Immix. This is the first conservative collector to achieve this milestone.

Chapter 7

Conclusion

Garbage collection design and implementation are both characterized by stark choices. Garbage collection *designs* must choose between tracing and reference counting. Garbage collector *implementations* must choose between exact and conservative collection. Performance concerns have lead to tracing and exact collection dominating, a choice evident today in highly engineered systems such as HotSpot, J9, and .NET. However, many other well-established systems use either reference counting or conservative garbage collection, including implementations for widely used languages such as PHP and JavaScript. Today reference counting and conservative garbage collection are widely used, but generally in non-performance critical settings because their implementations suffer significant overheads.

This thesis addresses the performance barriers affecting reference counting and conservative garbage collection. We achieve high performance reference counting with novel optimizations guided by detailed analysis of its key design points, and by changing its free-list heap organization to the block and line hierarchy of Immix for better mutator locality. We achieve high performance conservative garbage collection by building a conservative garbage collector on top of our fast reference counting with its Immix heap structure, and with a low overhead mechanism to validate ambiguous references. With our contributions, for the first time, the performance of both reference counting and conservative garbage collection are competitive with the best copying generational tracing collectors.

We conduct a comprehensive analysis of reference counting, show that its performance lags mark-sweep by over 30%, and measure a number of behaviors intrinsic to reference counting, which give insight into its behavior and opportunities for improvement. We identify two significant optimizations that eliminate reference counting operations for short lived young objects and together they entirely eliminate the performance gap with mark-sweep. On their own, these advances close but do not eliminate the performance gap between reference counting and the best generational tracing collector.

We identify heap organization as the principal source of this remaining performance gap for reference counting. Until this thesis, reference counting has always used a free list because it offered a constant time operation to immediately reclaim each dead object. Unfortunately, optimizing for reclamation time neglects the more essential performance requirement of cache locality on modern systems. We show that

indeed reference counting in a free list heap suffers poor locality compared to contiguous and the Immix hierarchical memory organizations. Our insight is that although contiguous heap organization and freeing at object granularity are incompatible, the Immix heap organization and reference counting are compatible. We describe the design and implementation of a new reference counting collector, RC Immix. The key contributions of our work are an algorithm for performing per-line live object counts and the integration of proactive and reactive opportunistic copying. We show how to copy new objects proactively to mitigate fragmentation and improve locality. We further show how to combine reactive defragmentation with backup cycle detection. In RC Immix, reference counting offers efficient collection, while the line and block heap organization offers efficient allocation and better mutator locality, and as a result RC Immix outperforms the best generational tracing collector, Gen Immix.

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. We describe 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 ambiguous root referents at a fine grain with an Immix line, which minimizes pinning overheads and maximizes locality benefits. Immix's opportunistic copying mitigates the cost of pinning because it combines marking of pinned objects and copying of unpinned objects as space allows. 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, RC Immix$_{cons}$ matches the performance of the best generational tracing collector, Gen Immix.

7.1 Future Work

The following sections focus on potential future directions for this line of work.

7.1.1 Root Elision in Reference Counting

A key advantage of reference counting over generational collection is that it continuously collects mature objects. The benefits are borne out by the improvements we see in xalan for both RC Immix and RC Immix$_{cons}$, which has many medium lived objects. These objects are promptly reclaimed by both reference counting and RC Immix, but are not reclaimed by a generational collector until a full heap collection occurs. However, this timely collection of mature objects does not come for free. Unlike a

nursery collection in a generational collector, a deferred reference counting collector must enumerate all roots, including all pointers from the stacks and all pointers from globals (statics). We realize that it might be possible to greatly reduce the workload of enumerating roots by selectively enumerating only those roots that have changed since the last GC. In the case of globals/statics this could be achieved either by a write barrier or by keeping a shadow set of globals. We note that the latter may be feasible because the amount of space consumed by global pointers is typically very low. In the case of the stack, we could utilize a return barrier [Yuasa et al., 2002] to only scan the parts of the stack that have changed since the last GC.

7.1.2 Concurrent Reference Counting and RC Immix

All of the garbage collectors we presented are stop the world parallel collectors. That means all of the mutator threads must stop during garbage collection and multiple collector threads perform the GC work. Sometimes with strict requirements for responsiveness, it is necessary to allow mutator to progress during a collection cycle. Concurrent garbage collection [Steele, 1975; Dijkstra et al., 1978; Bacon and Rajan, 2001; Pizlo et al., 2007, 2008; McCloskey et al., 2008] is used to implement this system where both collector and mutator threads are running and progressing simultaneously. Bacon and Rajan [2001] introduced concurrent cycle collection and Levanoni and Petrank [2001, 2006] introduced on-the-fly reference counting collection that uses update coalescing to reduce the concurrency overheads. It should be possible to make both our improved reference counting and the RC Immix collector concurrent to support high performance systems that need to reduce pause times.

7.1.3 Reference Counting in Soft Real Time Systems

Unlike tracing, in reference counting, the majority of objects are reclaimed as soon as they can no longer be referenced, and in an incremental fashion, without long pauses for collection cycles and with clearly defined lifetimes of each object. In soft real time systems short pauses are very important to maintain responsiveness [Baker, 1978; Bacon et al., 2003b,a]. So reference counting may be more suitable for soft real time systems rather than tracing, but poor performance lessens its appeal. Though high performance tracing garbage collectors exist for soft real time systems, our improvements may ignite the possible use of reference counting in soft real time systems.

7.1.4 Applicability to Non-Java Languages

Our analysis and design of reference counting and conservative garbage collection are all implemented in a Java virtual machine and with Java benchmarks. But we believe that they are applicable to other languages as well. For example, PHP and Objective-C uses reference counting, and Chakra VM and WebKit uses conservative garbage collection. Our analysis can be done for other languages and based on the outcome of the analysis, our design can be implemented there.

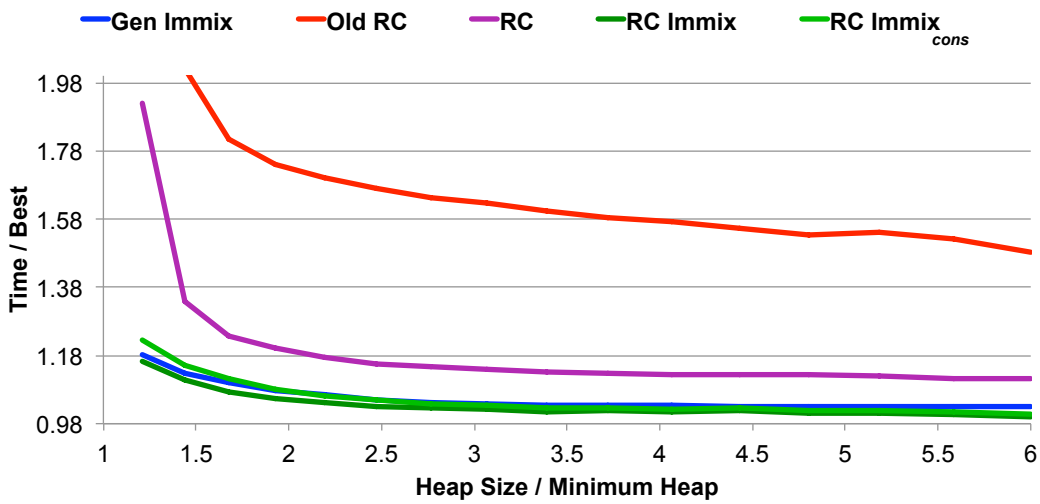


Figure 7.1: Key results of the thesis — the performance of Gen Immix, old RC, RC, RC Immix, and RC Immix$_{cons}$ as a function of heap size.

7.2 Final Words

The main contribution of this thesis is the design and implementation of new algorithms and mechanisms for reference counting and conservative garbage collection that significantly improve performance to the point where they are competitive with today's best copying generational tracing collectors. Figure 7.1 summarizes this result. It shows the performance improvement we achieved compared to Gen Immix, starting from Old RC to RC (Chapter 4), and then RC Immix (Chapter 5), and RC Immix$_{cons}$ (Chapter 6). With these advancements, language implementers now have a much richer choice of implementation alternatives both algorithmically (reference counting or tracing) and implementation-wise (exact or conservative), without compromising performance. These insights and advances are likely to particularly impact the development of new and emerging languages, where the implementation burden of tracing and exactness is often the critical factor in the first implementation.

Bibliography

Agesen, O.; Detlefs, D.; and Moss, J. E., 1998. Garbage collection and local variable type-precision and liveness in java virtual machines. In *ACM Conference on Programming Language Design and Implementation, PLDI'98, Montreal, Quebec, Canada, June 17-19, 1998*, 269–279. ACM. doi:10.1145/277650.277738. (cited on page 20)

Alpern, B.; Attanasio, C. R.; Barton, J. J.; Burke, M. G.; Cheng, P.; Choi, J.-D.; Cocchi, A.; Fink, S. J.; Grove, D.; Hind, M.; Hummel, S. F.; Lieber, D.; Litvinov, V.; Mergen, M.; Ngo, T.; Russell, J. R.; Sarkar, V.; Serrano, M. J.; Shepherd, J.; Smith, S.; Sreedhar, V. C.; Srinivasan, H.; and Whaley, J., 2000. The Jalapeño virtual machine. *IBM Systems Journal*, 39, 1 (February 2000), 211–238. doi:10.1147/sj.391.0211. (cited on pages 4 and 25)

Alpern, B.; Attanasio, C. R.; Cocchi, A.; Lieber, D.; Smith, S.; Ngo, T.; Barton, J. J.; Hummel, S. F.; Sheperd, J. C.; and Mergen, M., 1999. Implementing Jalapeño in Java. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications, OOPSLA'99, Denver, Colorado, USA, November 1-5, 1999*, 314–324. ACM. doi:10.1145/320384.320418. (cited on page 9)

Alpern, B.; Augart, S.; Blackburn, S. M.; Butrico, M.; Cocchi, A.; Cheng, P.; Dolby, J.; Fink, S. J.; Grove, D.; Hind, M.; McKinley, K. S.; Mergen, M.; Moss, J. E. B.; Ngo, T.; Sarkar, V.; and Trapp, M., 2005. The Jikes RVM Project: Building an open source research community. *IBM Systems Journal*, 44, 2 (2005), 399–418. doi:10.1147/sj.442.0399. (cited on pages 4 and 25)

Apple Inc., 2013. Transitioning to ARC release notes. https://developer.apple.com/library/mac/releasenotes/ObjectiveC/RN-TransitioningToARC. (cited on page 14)

Apple Inc., 2014. *The Swift Programming Language*. Swift Programming Series. Apple Inc. (cited on pages 14 and 77)

Attardi, G. and Flagella, T., 1994. A customisable memory management framework. In *Proceedings of the 6th conference on USENIX Sixth C++ Technical Conference - Volume 6, CTEC'94, Cambridge, MA, USA, 1994*, 8–8. USENIX Association. (cited on pages 22 and 23)

Azatchi, H. and Petrank, E., 2003. Integrating generations with advanced reference counting garbage collectors. In *Compiler Construction, 12th International Conference, CC 2003, Warsaw, Poland, April, 2003*, vol. 2622 of *Lecture Notes in Computer Science*, 185–199. Springer Berlin Heidelberg. doi:10.1007/3-540-36579-6_14. (cited on pages 16, 39, and 48)

Bacon, D. F.; Attanasio, C. R.; Lee, H. B.; Rajan, V. T.; and Smith, S., 2001. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *ACM Conference on Programming Language Design and Implementation, PLDI'01, Snowbird, UT, USA, June 20-22, 2001*, 92–103. ACM. doi:10.1145/378795.378819. (cited on pages 2 and 15)

Bacon, D. F.; Cheng, P.; and Rajan, V. T., 2003a. Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for java. In *Proceedings of the 2003 Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES'03, San Diego, California, USA, June 11-13, 2003*, 81–92. ACM. doi:10.1145/780732.780744. (cited on page 99)

Bacon, D. F.; Cheng, P.; and Rajan, V. T., 2003b. A real-time garbage collector with low overhead and consistent utilization. In *ACM Symposium on the Principles of Programming Languages, POPL'03, New Orleans, Louisisana, USA, January 15-17, 2003*, 285–298. ACM. doi:10.1145/604131.604155. (cited on page 99)

Bacon, D. F. and Rajan, V. T., 2001. Concurrent cycle collection in reference counted systems. In *European Conference on Object-Oriented Programming, Budapest, Hungary, June 18 - 22, 2001*, 207–235. LNCS. doi:10.1007/3-540-45337-7_12. (cited on pages 16, 17, and 99)

Baker, H. G., 1978. List processing in real time on a serial computer. *Communications of the ACM*, 21, 4 (1978), 280–294. doi:10.1145/359460.359470. (cited on page 99)

Baker, J.; Cunei, A.; Kalibera, T.; Pizlo, F.; and Vitek, J., 2009. Accurate garbage collection in uncooperative environments revisited. *Concurrency and Computation: Practice and Experience*, 21, 12 (2009), 1572–1606. doi:10.1002/cpe.1391. (cited on page 20)

Bartlett, J. F., 1988. Compacting garbage collection with ambiguous roots. *SIGPLAN Lisp Pointers*, 1, 6 (Apr. 1988), 3–12. doi:10.1145/1317224.1317225. (cited on pages 1, 3, 5, 22, and 23)

Berger, E. D.; McKinley, K. S.; Blumofe, R. D.; and Wilson, P. R., 2000. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-IX, Cambridge, MA, USA, November 12-15, 2000*, 117–128. ACM. doi:10.1145/378993.379232. (cited on page 9)

Blackburn, S. M.; Cheng, P.; and McKinley, K. S., 2004a. 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 10–14, 2004*, 25–36. ACM. doi:10.1145/1005686.1005693. (cited on pages 2, 4, 9, 18, 20, 22, 26, 56, 58, 74, 76, 89, and 92)

Blackburn, S. M.; Cheng, P.; and McKinley, K. S., 2004b. Oil and water? High performance garbage collection in Java with MMTk. In *The 26th International Conference on Software Engineering, ICSE'04, Edinburgh, Scotland, May 23-28, 2004*, 137–146. ACM/IEEE. doi:10.1109/ICSE.2004.1317436. (cited on page 26)

Blackburn, S. M.; Garner, R.; Hoffmann, C.; Khan, A. M.; McKinley, K. S.; Bentzur, R.; Diwan, A.; Feinberg, D.; Frampton, D.; Guyer, S. Z.; Hirzel, M.; Hosking, A.; Jump, M.; Lee, H.; Moss, J. E. B.; Phansalkar, A.; Stefanović, D.; VanDrunen, T.; von Dincklage, D.; and Wiedermann, B., 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications, OOPSLA'06, Portland, OR, USA, October 22-26, 2006*, 169–190. ACM. doi:10.1145/1167473.1167488. (cited on page 25)

Blackburn, S. M.; Hirzel, M.; Garner, R.; and Stefanović, D., 2005. pjbb2005: The pseudojbb benchmark. http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005. (cited on page 25)

Blackburn, S. M. and McKinley, K. S., 2003. Ulterior reference counting: Fast garbage collection without a long wait. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications, OOPSLA'03, Anaheim, CA, USA, October 26-30, 2003*, 344–358. ACM. doi:10.1145/949305.949336. (cited on pages 2, 16, 39, 41, 48, and 50)

Blackburn, S. M. and McKinley, K. S., 2008. 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 7-13, 2008*, 22–32. ACM. doi:10.1145/1379022.1375586. (cited on pages xv, 2, 4, 5, 13, 18, 19, 20, 22, 53, 56, 59, 67, 76, 80, 89, and 92)

Blackburn, S. M.; McKinley, K. S.; Garner, R.; Hoffman, C.; Khan, A. M.; Bentzur, R.; Diwan, A.; Feinberg, D.; Frampton, D.; Guyer, S. Z.; Hirzel, M.; Hosking, A.; Jump, M.; Lee, H.; Moss, J. E. B.; Phansalkar, A.; Stefanović, D.; VanDrunen, T.; von Dincklage, D.; and Wiedermann, B., 2008. Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century. *Communications of the ACM*, 51, 8 (Aug. 2008), 83–89. doi:10.1145/1378704.1378723. (cited on page 26)

Boehm, H.-J. and Weiser, M., 1988. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18, 9 (Sep. 1988), 807–820. doi:10.1002/spe.4380180902. (cited on pages 1, 3, and 22)

Cao, T.; ; Blackburn, S. M.; Gao, T.; and McKinley, K. S., 2012. 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, USA, June 9-13, 2012*, 225–236. ACM/IEEE. doi:10.1145/2366231.2337185. (cited on page 26)

Cheney, C. J., 1970. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13, 11 (Nov. 1970), 677–678. doi:10.1145/362790.362798. (cited on pages 9, 12, and 18)

Christopher, T. W., 1984. Reference count garbage collection. *Software Practice and Experience*, 14, 6 (Jun. 1984), 503–507. doi:10.1002/spe.4380140602. (cited on pages 16 and 17)

Collins, G. E., 1960. A method for overlapping and erasure of lists. *Communications of the ACM*, 3, 12 (December 1960), 655–657. doi:10.1145/367487.367501. (cited on pages 1, 2, 11, 14, 29, and 30)

Demers, A.; Weiser, M.; Hayes, B.; Boehm, H.; Bobrow, D.; and Shenker, S., 1990. Combining generational and conservative garbage collection: framework and implementations. In *ACM Symposium on the Principles of Programming Languages, POPL'90, San Francisco, California, USA, January 1990*, 261–269. ACM. doi:10.1145/96709.96735. (cited on pages 1, 3, 22, 53, 59, 67, and 80)

Deutsch, L. P. and Bobrow, D. G., 1976. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19, 9 (September 1976), 522–526. doi:10.1145/360336.360345. (cited on pages 2, 14, and 77)

Dijkstra, E. W.; Lamport, L.; Martin, A. J.; Scholten, C. S.; and Steffens, E. F. M., 1976. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46*. Springer-Verlag. (cited on page 8)

Dijkstra, E. W.; Lamport, L.; Martin, A. J.; Scholten, C. S.; and Steffens, E. F. M., 1978. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21, 11 (Nov. 1978), 965–975. doi:10.1145/359642.359655. (cited on page 99)

Diwan, A.; Moss, E.; and Hudson, R., 1992. Compiler support for garbage collection in a statically typed language. In *ACM Conference on Programming Language Design and Implementation, PLDI'92, San Francisco, California, USA, June 17-19, 1992*, 273–282. ACM. doi:10.1145/143095.143140. (cited on page 20)

Fenichel, R. R. and Yochelson, J. C., 1969. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12 (Nov. 1969), 611–612. doi:10.1145/363269.363280. (cited on page 12)

Frampton, D., 2010. *Garbage Collection and the Case for High-level Low-level Programming*. Ph.D. thesis, Australian National University. http://cs.anu.edu.au/~Daniel.Frampton/DanielFrampton_Thesis_Jun2010.pdf. (cited on page 16)

Garthwaite, A. and White, D., 1998. The gc interface in the evm. Technical report, Sun Microsystems, Inc. (cited on page 9)

Henderson, F., 2002. 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*, 150–156. ACM. doi:10.1145/512429.512449. (cited on page 20)

Hirzel, M.; Diwan, A.; and Henkel, J., 2002. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *ACM Transactions on Programming Languages and Systems*, 24, 6 (Nov. 2002), 593–624. doi:10.1145/586088.586089. (cited on page 20)

Hosking, A. L., 2006. 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*, 40–51. ACM. doi:10.1145/1133956.1133963. (cited on pages 22 and 23)

Hughes, R. J. M., 1982. A semi-incremental garbage collection algorithm. *Software - Practice and Experience*, 12 (1982), 1081–1082. doi:10.1002/spe.4380121108. (cited on page 10)

Jibaja, I.; Blackburn, S. M.; Haghighat, M. R.; and McKinley, K. S., 2011. Deferred gratification: Engineering for high performance garbage collection from the get go. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, MSPC 2011, San Jose, CA, USA, June 5, 2011*, 58–65. ACM. doi:10.1145/1988915.1988930. (cited on page 15)

Jikes, 2012. Jikes rvm. http://www.sigplan.org/Awards/Software/2012. (cited on page 4)

Jones, R. E.; Hosking, A.; and Moss, J. E. B., 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman and Hall/CRC Applied Algorithms and Data Structures Series. http://gchandbook.org/. (cited on pages 4, 7, 31, and 41)

Levanoni, Y. and Petrank, E., 2001. An on-the-fly reference counting garbage collector for Java. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications, OOPSLA'01, Tampa, FL, USA, October 14-18, 2001*, 367–380. ACM. doi:10.1145/504282.504309. (cited on pages 2, 15, 66, and 99)

Levanoni, Y. and Petrank, E., 2006. An on-the-fly reference-counting garbage collector for Java. *ACM Transactions on Programming Languages and Systems*, 28, 1 (January 2006), 1–69. doi:10.1145/1111596.1111597. (cited on pages 2, 15, and 99)

Lieberman, H. and Hewitt, C., 1983. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26, 6 (Jun. 1983), 419–429. doi:10.1145/358141.358147. (cited on page 13)

Lins, R. D., 1992. Cyclic reference counting with lazy mark-scan. *Information Processing Letters*, 44, 4 (1992), 215–220. doi:10.1016/0020-0190(92)90088-D. (cited on pages 16 and 17)

LLVM, 2014. Accurate garbage collection with LLVM. http://llvm.org/docs/GarbageCollection.html. (cited on page 20)

Martinez, A. D.; Wachenchauzer, R.; and Lins, R. D., 1990. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34, 1 (February 1990), 31–35. doi:10.1016/0020-0190(90)90226-N. (cited on pages 16 and 17)

McCarthy, J., 1960. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3, 4 (April 1960), 184–195. doi:10.1145/367177.367199. (cited on pages 1, 2, 7, 8, 10, 14, 29, and 30)

McCloskey, B.; Bacon, D. F.; Cheng, P.; and Grove, D., 2008. Staccato: A parallel and concurrent real-time compacting garbage collector for multiprocessors. Technical Report RC24504, IBM Research. (cited on page 99)

Minsky, M., 1963. A LISP garbage collector algorithm using serial secondary storage. Technical report, Massachusetts Institute of Technology. (cited on page 12)

Paz, H.; Bacon, D. F.; Kolodner, E. K.; Petrank, E.; and Rajan, V. T., 2007. An efficient on-the-fly cycle collection. *ACM Transactions on Programming Languages and Systems*, 29, 4 (August 2007). doi:10.1145/1255450.1255453. (cited on pages 16 and 17)

Paz, H.; Petrank, E.; and Blackburn, S. M., 2005. Age-oriented concurrent garbage collection. In *Compiler Construction, 14th International Conference, CC 2005, Edinburgh, UK, April 4-8, 2005*, vol. 3443 of *Lecture Notes in Computer Science*, 121–136. Springer Berlin Heidelberg. doi:10.1007/978-3-540-31985-6_9. (cited on pages 16, 39, and 48)

Pizlo, F.; Frampton, D.; Petrank, E.; and Steensgaard, B., 2007. Stopless: a real-time garbage collector for multiprocessors. In *Proceedings of the 6th International Symposium on Memory Management, ISMM 2007, Montreal, Quebec, Canada, October 21-22, 2007*, 159–172. ACM. doi:10.1145/1296907.1296927. (cited on page 99)

Pizlo, F.; Petrank, E.; and Steensgaard, B., 2008. A study of concurrent real-time garbage collectors. In *ACM Conference on Programming Language Design and Implementation, PLDI'08, Tucson, AZ, USA, June 7-13, 2008*, 33–44. ACM. doi:10.1145/1375581.1375587. (cited on page 99)

Sansom, P., 1991. Dual-mode garbage collection. In *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, 283–310. Department of Electronics and Computer Science, University of Southampton. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.24.1020. (cited on page 12)

Seeley, Y., 2009. JIRA issue LUCENE-1800: QueryParser should use reusable token streams. https://issues.apache.org/jira/browse/LUCENE-1800. (cited on page 25)

Shahriyar, R.; Blackburn, S. M.; and Frampton, D., 2012. 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*, 73–84. ACM. doi:10.1145/2258996.2259008. (cited on page 29)

Shahriyar, R.; Blackburn, S. M.; and McKinley, K. S., 2014. Fast conservative garbage collection. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications, OOPSLA'14, Portland, Oregon, USA, October 20-24, 2014*. ACM. doi:10.1145/2660193.2660198. (cited on page 75)

Shahriyar, R.; Blackburn, S. M.; Yang, X.; and McKinley, K. S., 2013. Taking off the gloves with reference counting Immix. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications, OOPSLA'13, Indianapolis, Indiana, USA, October 26-31, 2013*, 93–110. ACM. doi:10.1145/2509136.2509527. (cited on page 55)

Smith, F. and Morrisett, G., 1998. 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*, 68–78. ACM. doi:10.1145/301589.286868. (cited on pages 1, 3, 5, 22, and 23)

SPEC, 1999. *SPECjvm98, Release 1.03*. Standard Performance Evaluation Corporation. http://www.spec.org/jvm98. (cited on page 25)

SPEC, 2006. *SPECjbb2005 (Java Server Benchmark), Release 1.07*. Standard Performance Evaluation Corporation. http://www.spec.org/jbb2005. (cited on page 25)

Steele, G. L., 1975. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18, 9 (Sep. 1975), 495–508. doi:10.1145/361002.361005. (cited on page 99)

Stein, L. D., 2003. Devel::Cycle - Find memory cycles in objects. http://search.cpan.org/~lds/Devel-Cycle-1.11/lib/Devel/Cycle.pm. (cited on page 14)

Styger, P., 1967. LISP 2 garbage collector specifications. Technical Report Technical Report TM-3417/500/00 1, System Development Cooperation. (cited on pages 10 and 12)

Thomas, J. R.; Cantu, M.; and Bauer, A., 2013. Reference counting and object harvesting in Delphi. *Dr. Dobb's*, (May 2013). http://www.drdobbs.com/mobile/reference-counting-and-object-harvesting/240155664. (cited on page 14)

TIOBE, 2014a. Tiobe index for java. http://www.tiobe.com/index.php/content/paperinfo/tpci/Java.html. (cited on page 4)

TIOBE, 2014b. Tiobe index for may 2014. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. (cited on page 4)

Ungar, D., 1984. Generation Scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SDE 1, 1984*, 157–167. ACM. doi:10.1145/800020.808261. (cited on page 13)

WebKit, 2014. The WebKit open source project. http://trac.webkit.org/browser/trunk/Source/JavaScriptCore/heap. (cited on pages 1, 3, 5, 22, and 23)

Weizenbaum, J., 1969. Recovery of reentrant list structures in Lisp. *Communications of the ACM*, 12, 7 (July 1969), 370–372. doi:10.1145/363156.363159. (cited on pages 11 and 16)

Wilson, P. R., 1992. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management, IWMM'92, St. Malo, France, September 17-19, 1992*, vol. 637 of *Lecture Notes in Computer Science*, 1–42. Springer Berlin Heidelberg. http://dl.acm.org/citation.cfm?id=645648.664824. (cited on page 7)

Wilson, P. R.; Johnstone, M. S.; Neely, M.; and Boles, D., 1995. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management, IWMM'95, Kinross, Scotland, UK, Sep 27-29, 1995*, vol. 986 of *Lecture Notes in Computer Science*, 1–116. Springer Berlin Heidelberg. doi:10.1007/3-540-60368-9_19. (cited on pages 4 and 9)

Yang, X.; Blackburn, S. M.; Frampton, D.; and Hosking, A. L., 2012. Barriers reconsidered, friendlier still! In *Proceedings of the 11th International Symposium on Memory Management, ISMM 2012, Beijing, China, June 15 - 16, 2012*, 37–48. ACM. doi:10.1145/2258996.2259004. (cited on page 92)

Yang, X.; Blackburn, S. M.; Frampton, D.; Sartor, J. B.; and McKinley, K. S., 2011. Why nothing matters: The impact of zeroing. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications, OOPSLA'11, Portland, Oregon, USA, October 22 - 27, 2011*, 307–324. ACM. doi:10.1145/2048066.2048092. (cited on pages 25, 56, 58, 71, and 73)

Yuasa, T.; Nakagawa, Y.; Komiya, T.; and Yasugi, M., 2002. Return barrier. In *Proceedings of the International Lisp Conference*. (cited on page 99)