Down for the Count? Getting Reference Counting Back in the Ring *

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

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

Daniel Frampton
Australian National University
Daniel.Frampton@anu.edu.au

Abstract

Reference counting and tracing are the two fundamental approaches that have underpinned garbage collection since 1960. However, despite some compelling advantages, reference counting is almost completely ignored in implementations of high performance systems today. In this paper we take a detailed look at reference counting to understand its behavior and to improve its performance. We identify key design choices for reference counting and analyze how the behavior of a wide range of benchmarks might affect design decisions. As far as we are aware, this is the first such quantitative study of reference counting. We use insights gleaned from this analysis to introduce a number of optimizations that significantly improve the performance of reference counting.

We find that an existing modern implementation of reference counting has an average 30% overhead compared to tracing, and that in combination, our optimizations are able to completely eliminate that overhead. This brings the performance of reference counting on par with that of a well tuned mark-sweep collector. We keep our in-depth analysis of reference counting as general as possible so that it may be useful to other garbage collector implementers. Our finding that reference counting can be made directly competitive with well tuned mark-sweep should shake the community's prejudices about reference counting and perhaps open new opportunities for exploiting reference counting's strengths, such as localization and immediacy of reclamation.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection)

General Terms Design, Performance, Algorithms, Measurement

Keywords Reference Counting, Memory Management, Garbage Collection, Java

1. Introduction

In an interesting twist of fate, the two fundamental branches of the garbage collection family tree were born within months of each other in 1960, both in the Communications of the ACM [10, 20]. On the one hand, reference counting [10] *directly* identifies garbage by noticing when an object has no references to it, while on the other hand, tracing [20] 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 remains somewhat neglected within the garbage collection literature. The goal of this work is to revisit reference counting, understand its shortcomings, and address some of its limitations. 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 hope to give new life to this much neglected branch of the garbage collection literature.

Reference counting works by keeping a count of incoming references to each object and collecting objects when their count falls to zero. Therefore in principle all that is required is a write barrier that notices each pointer change, decrementing the target object's count when a pointer to it is overwritten and incrementing the target object's count whenever a pointer to it is created. This 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. By contrast, tracing collectors must start with a set of *roots*, which requires the runtime to enumerate all pointers into the heap from global variables, the stacks, and registers. Root enumeration thus requires deep integration with the runtime and can be challenging to engineer [14].

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. We do not address this limitation, which can be overcome with a backup demand-driven tracing mechanism. However, reference counting is also slow. Naive reference counting is extremely costly because of the overhead of intercepting every pointer mutation, including those to the registers and stacks. High performance reference counting overlooks changes to the stacks and registers [12] and may even elide many changes to heap references [16]. However, even high performance reference counting is slow. We compare high performance reference counting and mark-sweep implementations and find that reference counting is over 30% slower than its tracing counterpart.

We reconsider reference counting. We start by identifying key design parameters and evaluating the intrinsic behaviors of Java workloads with respect to those design points. For example, we study the distribution of maximum reference counts across Java benchmarks. 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

* This work supported by the Australian Research Council DP0666059.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISMM'12, June 15–16, 2012, Beijing, China.
Copyright © 2012 ACM 978-1-4503-1350-6/12/06…$10.00

objects, and 3) allocating new objects as dead, avoiding a significant overhead in deallocating them. We then conduct an in-depth performance analysis of mark-sweep and reference counting, 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.

This paper 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 reference counting, 3) guided by our analysis, we introduce optimizations that greatly improve reference counting performance, and 4) we conduct a detailed performance study of reference counting and mark-sweep, showing that our optimizations eliminate the overhead of reference counting.

We hope that the insights and optimizations brought to light in this paper may give new life to reference counting. Our detailed study of intrinsic behaviors will 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 incrementality, locality, and immediacy of reference counting compelling.

2. Background and 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 data structures. We describe each of these and survey major design alternatives.

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 trade-off between the space required to store the count, and the complexity of accurately managing counts when limited bits are available.

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) have an auxiliary data structure such as a hash table to store accurate counts, 2) have *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 count or directly collect the object [15].

2.2 Maintaining the Count

Simple, *immediate* reference counters count *all* references, both on the heap and in local variables. Whenever references are created, copied, destroyed, or overwritten, increment and decrement operations are performed on the referents. 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 when performance is not the highest priority. 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 through the use of *smart pointers* in C++, while the latter can be implemented through a destructor. In contrast, tracing collectors must be able to identify all pointers held in the runtime state, such as those in stacks, registers, and global variables. To identify all pointers from the stack into the heap, the runtime must implement *GC maps*, which are generally difficult to implement and maintain correctly.

Deferred Deutsch and Bobrow [12] introduced deferred reference counting. In contrast to the immediate reference counting described above, deferred reference counting ignores mutations to frequently modified variables such as those stored in registers and on the stack. Periodically, these references are enumerated into a root set, and any objects that are neither in the root set nor referenced by other objects in the heap may be collected. They achieve this directly by maintaining a zero count table that holds all objects known to have a reference count of zero. This zero count table is enumerated, and any object that does not have a corresponding entry in the root set is identified as garbage. Bacon et al. [3] avoid the need to maintain a zero count table by buffering decrements between collections. At collection time, elements in the root set are given a temporary increment while processing all of the buffered decrements. Deferred reference counting performs all increments and decrements during collection time. Although much faster than immediate reference counting, these schemes require *GC maps*, removing the implementation advantage over tracing.

Coalescing Heap references are mutated very frequently: even with stack mutations deferred, we measured millions of reference mutations per second. Levanoni and Petrank [16, 17] observed that all but the first and last in any chain of mutations to a given reference within a given window could 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 can be exploited by remembering only the initial value of a reference field between periodic reference counting collections. At each of these collections, only the objects referred to by the initial (stored) and current values of the reference field need to be updated. Levanoni and Petrank implemented coalescing using *object remembering*. The first time an object has a reference modified since the last collection: a) the mutated object is marked dirty and all outgoing reference values are remembered; b) all future reference mutations for that (now dirty) object are ignored; c) during the next collection the remembered object is scanned, increments are performed on all outgoing pointers, decrements are performed on all remembered outgoing references, and the dirty flag is cleared. New objects are remembered and allocated dirty, ensuring that outgoing references are incremented at the next collection. No old values are recorded for new objects because all outgoing references start as *null*.

Generational Blackburn and McKinley [4] introduced *ulterior reference counting*, a hybrid collector that combines copying generational collection for the young objects and reference counting for the old objects. It restricts copying and reference counting to the object demographics for which they perform well and safely ignores mutations to select heap objects. It can achieve high performance with reduced pause time. Ulterior reference counting is not difficult to implement, but the implementation is a hybrid, and thus

manifests the complexities of both a standard copying nursery and a reference counted heap.

Age-Oriented Paz et al. [21] 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 provides a significant benefit as it avoids performing expensive reference counting operations for the many young objects that die. Like ulterior reference counting, this collector is a hybrid, so manifests the complexities of two orthodox collectors.

2.3 Collecting Cyclic Objects

As discussed above, reference counting alone cannot collect all garbage. Objects can form a cycle, where a group of objects point to each other, maintaining non-zero reference counts. There exist two general approaches to deal with cyclic garbage: *backup tracing* [22] and *trial deletion* [2, 9, 18, 19]. Frampton [13] conducted a detailed study of cycle collection.

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 as described in Section 2.1.

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 [9] and has been optimized over time [2, 18, 19].

Cycle collection is not the focus of this paper, however some form of cycle collection is essential for completeness. We use backup tracing, which performs substantially better than trial deletion and has more predictable performance characteristics [13]. Backup tracing also provides a solution to the problem of reference counts that become stuck due to limited bits.

3. Analysis of Reference Counting Intrinsics

Recall that despite the implementation advantages of simple immediate reference counting, reference counting is rarely used because it is 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.

3.1 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 vs. *never* collect cycles) and report when this affects the analysis.

Jikes RVM We use Jikes RVM and MMTk for all experiments. Jikes RVM [1] is a high performance research JVM with a well-tuned garbage collection infrastructure MMTk [7]. Jikes RVM is open source written almost entirely in a slightly extended Java. 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. To ensure performance and repeatability, all of our experiments were run using Jikes RVM's replay compilation feature. We use the most recent version, which executes one iteration of each benchmark using only the unoptimized baseline compiler, before using user-provided profile information to optimize hot methods all at once, prior to the second iteration. The second iteration of the benchmark then executes this optimized version of the code. This approach offers the performance of steady state in an adaptively optimized system, whilst avoiding the non-determinism of adaptive compilation.

MMTk 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 [6]. To perform our analysis, we instrument the standard configuration of reference counting to gather information on different metrics while running the benchmarks. This instrumentation does not affect the garbage collection workload (the exact same set of objects is collected with or without the instrumentation). The instrumentation slows the collector down considerably, but since this part of our analysis is not concerned with collector performance, this slowdown is irrelevant. We do not use the instrumentation for our subsequent performance study. All of the collectors we evaluate are parallel, including the standard reference counting we use as our baseline. The optimizations we present here are correct with respect to parallel collection and the results we present here exploit parallel collection.

We use mark-sweep as our representative tracing collector and principal point of comparison because it utilizes the same heap organization and allocator as the reference counters. We compare our best reference counting system with the high performance Immix tracing collector [5], but this is not our main point of comparison because the principal advantage of the Immix collector is its unique heap organization which is orthogonal to our optimizations.

Benchmarks We use 19 benchmarks from the DaCapo and SPEC benchmark suites in all the measurements and performance studies taken in this paper. SPEC provides both Java client and server side benchmarks. The DaCapo suite [8] is a suite of non-trivial real-world open source Java applications. We use the superset of all benchmarks from DaCapo 2006 and DaCapo 9.12 that can run successfully with Jikes RVM, using the more recent version of any given benchmark when the opportunity exists. We identified a nominal minimum heap size for each benchmark by finding the minimum heap size in which the benchmark could successfully complete using any of the three systems we evaluate (standard reference counting, our optimized reference counting, and mark-sweep). Unless otherwise stated we conduct all of our performance experiments while holding the heap size constant at $2\times$ the minimum heap size, which is a modest size.

Experimental Platform The results we present here were measured on a modern Core i5 670 dual-core processor with two-way SMT, a clock rate of 3.4 GHz, 4 MB of last level cache, and 4 GB of RAM. We conducted our evaluation on a range of modern and older x86 processors and found that our analysis and optimizations are robust.

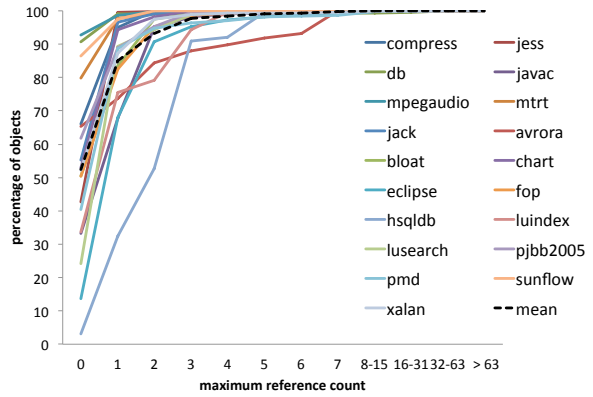


Figure 1. Most objects have very low maximum reference counts. This graph plots the cumulative frequency distribution of maximum reference counts among objects in each benchmark.

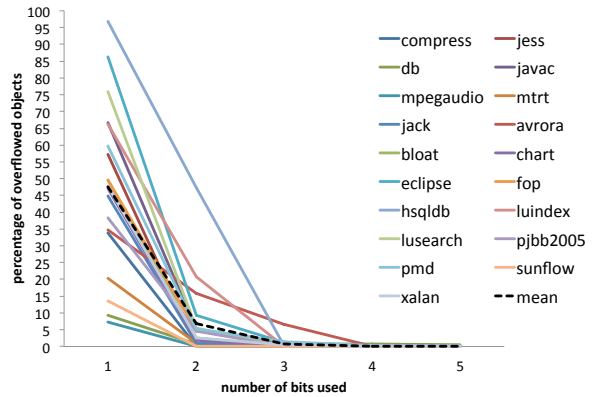


Figure 2. The number of objects which suffer overflowed reference counts drops off rapidly as the number of available bits grows from two to five.

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.

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 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 1 is displayed pictorially in Figure 1.

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 2 we show for different sized reference count fields, measurements of: a) the fraction of *objects* that would ever overflow, and b) the fraction 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 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 2 is displayed pictorially in Figure 2.

3.4 Sources of Reference Counting Operations

Table 3 shows for each benchmark the origin of the increment and decrement operations. In each case we account for the operations

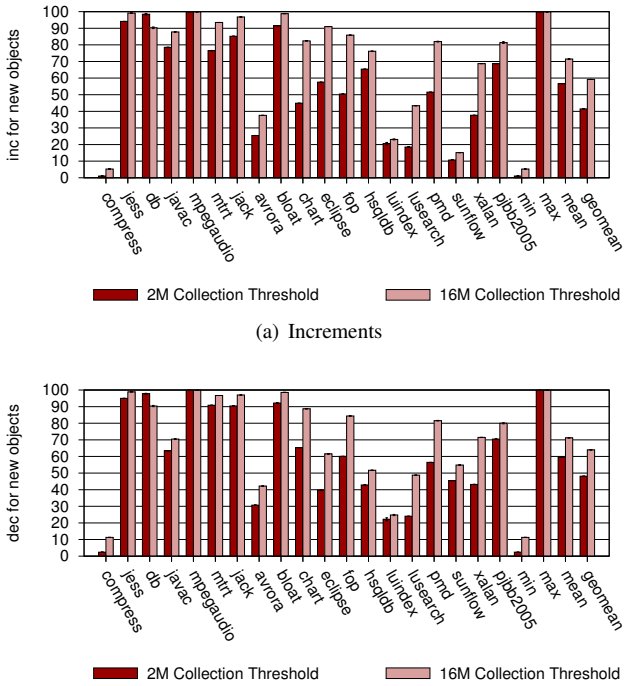


Figure 3. New objects are responsible for the majority of reference counting operations. We show here the fraction of (a) increments and (b) decrements that are due to objects allocated within the most recent 2 MB and 16 MB of objects allocated.

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 2MB to 16MB, and report only 2MB and 16MB to show the significant differences. The definition of 'new' is anything allocated within the last interval, so as the interval becomes larger, a larger fraction the live objects are 'new'.

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 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.

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 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 trigger	mean	compress	jess	db	javac	mpegaudio	mtrt	jack	avrora	bloat	chart	eclipse	fop	hsqldb	luindex	lusearch	pjbb2005	pmd	sunflow	xalan
breakdown 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
breakdown 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 3. New objects account for a large fraction 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.

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. 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.

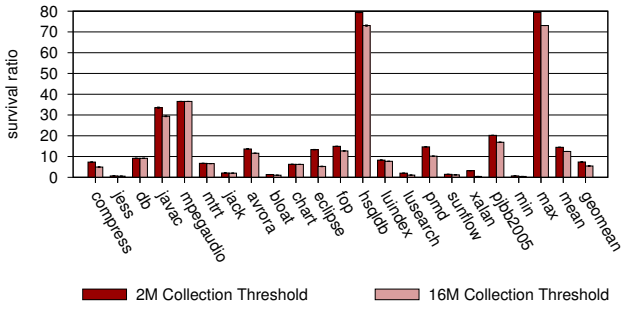


Figure 4. Most benchmarks have very low object survival ratios. This graph shows the percentage of objects that survive beyond 2 MB and 16 MB of allocation.

On average 71% of increments and 71% of decrements are performed upon newly 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 [4]. Around 10% of operations are due to root reachability. 4% of decrements are performed during cycle collection.

Figures 3(a) and 3(b) illustrate data from Table 3 graphically, showing the fraction 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 shows the fraction 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 shows that most benchmarks have 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 [4, 21].

3.5 Efficacy of Coalescing

Coalescing 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 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 [4] 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 6 provides a different perspective by showing the breakdown of total reference mutations per unit time (millisecond).

3.6 Cyclic Garbage

Table 7 shows key statistics for each benchmark related to cyclic garbage. For each benchmark we show: 1) the fraction of objects that can be reclaimed by pure reference counting, and 2) the fraction of objects that are part of a cyclic graph when unreachable, so can only be reclaimed via cycle collection, and 3) the fraction 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).

collection 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 5. Coalescing elides around 64% of pointer field changes on average, and around 90% for old objects. This table shows the fraction 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 6. References are mutated around 10 million times per second on average, on our 3.4 GHz Core i5. 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**	91	99.7	100	77	64	93	99.9	99.8	99	94	47	82	27	91	84	87	80	99.99	90
cyclic objects	**16**	9	0.3	0	23	36	7	0.1	0.2	1	6	53	18	73	9	16	13	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 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 on average 38% of all objects are inherently acyclic. These results vary considerably among the benchmarks.

4. Improving Reference Counting

We now explore two areas for optimization that arise from our analysis of the intrinsic data presented in the previous section. We describe the insights, evaluate the designs, and evaluate the ideas in combination.

4.1 Storing the Reference Count

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 [15], but to our knowledge has not been systematically analyzed. Key insights that can be drawn from our intrinsic analysis are that most objects have maximum 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.

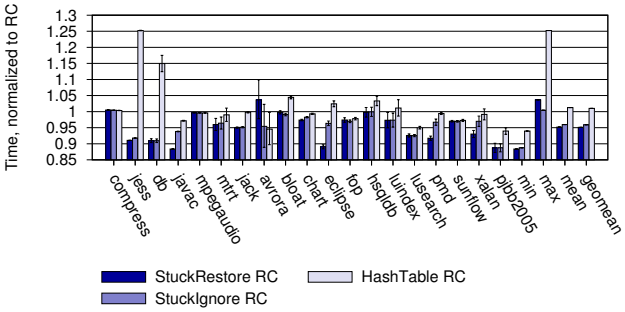
Hash table on overflow (*HashTable RC*) When an object's reference count overflows, the reference count can be stored in a hash table. Increments and decrement are performed in the hash table until the reference count drops below the overflow threshold, at which point the hash table entry is released. 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.

Stuck and Ignored on Overflow (*StuckIgnore RC*) 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 the backup tracing cycle collector (note that a trial deletion cycle collector cannot collect such objects).

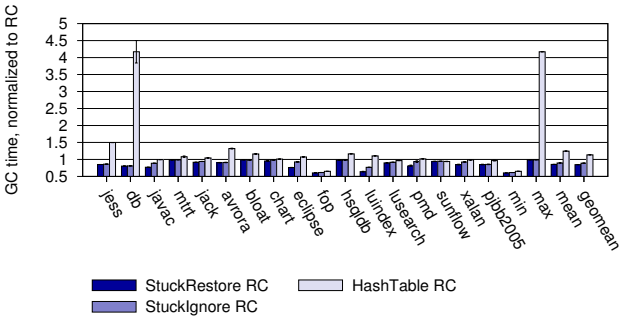
Stuck and Restored on Overflow (*StuckRestore RC*) A refinement to the previous case has the backup trace *restore* reference counts within the heap during tracing, by incrementing the target object's count for each reference traversed. Although this approach imposes an additional role upon the backup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

In Figure 5 we evaluate 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 and that uses backup tracing cycle collector.

For the majority of the benchmarks *HashTable RC* performs poorly, with *Standard RC* 1% better in total time (Figure 5(a)) and 13% better in collection time (Figure 5(b)) 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

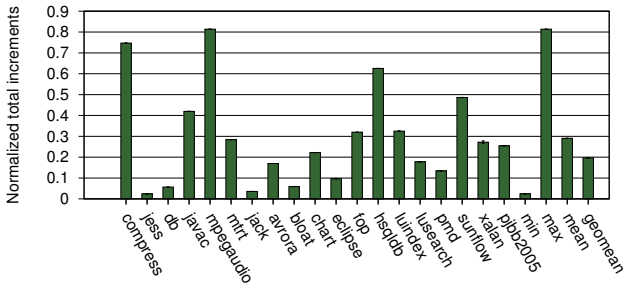


(a) Total Time

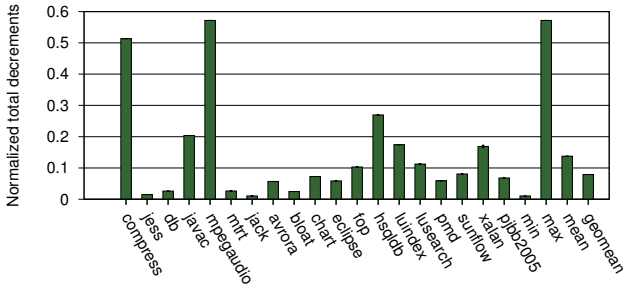


(b) GC Time

Figure 5. Using a hash table to account for reference count overflow is not a good solution. These graphs show three strategies for dealing with overflow. Results vary greatly among benchmarks.



(a) Increments



(b) Decrements

Figure 6. Lazy treatment of new objects greatly reduces the number of reference counting operations necessary compared to *Standard RC*. The effectiveness varies greatly among the benchmarks.

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 30% and 17% 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.

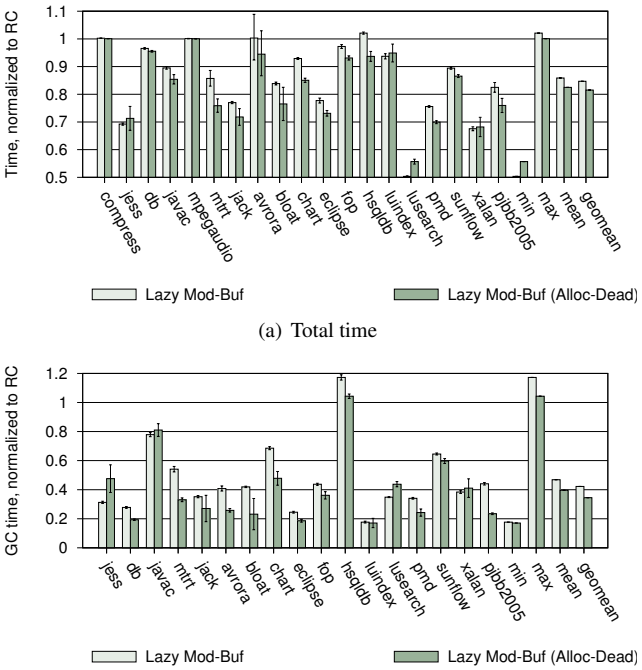
4.2 Lazy Treatment of New Objects

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. Two previous systems proposed hybrid collectors that successfully exploited this property. Blackburn and McKinley [4] combined copying generational collection and reference counting, using the copying collector to absorb the impact of the young objects. Paz et al. developed a similar scheme that combined mark-sweep collection with reference counting [21]. Like the previous work, we propose to avoid reference counting operations on new objects. However, our goal is to do so within the framework of reference counting, without creating a hybrid by introducing another collector.

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 will only include references to new objects that are live.

We further observe that if new objects are allocated 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.

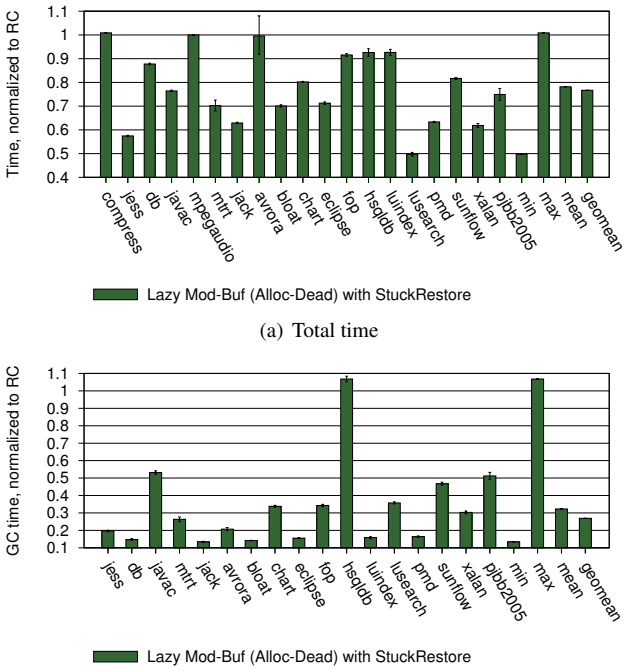
We start by considering the treatment of new objects in a collector that uses deferred reference counting and coalescing, and we use this as our point of comparison. In such a collector, new objects are allocated dirty with a reference count of one. The object is added



(a) Total time

(b) GC time

Figure 7. Lazy treatment of new objects reduces total time by around 20% compared to *Standard RC*. Most of this benefit comes from not eagerly adding new objects to the *mod-buf*.



(a) Total time

(b) GC time

Figure 8. The combined effect of our optimizations is a 24% improvement in total time compared to *Standard RC*. Both jess and lusearch see nearly two-fold improvements.

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 [12]. 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 points to. Because all references are initially *null*, the coalescing mechanism does not need to explicitly generate decrements corresponding to outgoing pointers from the initial state of the object [16].

Lazy Mod-Buf Insertion 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. 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 [4, 21], our scheme achieves a similar result via a very simple optimization to existing reference counting collector. This optimization required only very modest changes to MMTk's existing reference counting collector.[1] Figure 6(a) shows the massive reduction in the total number of increments.

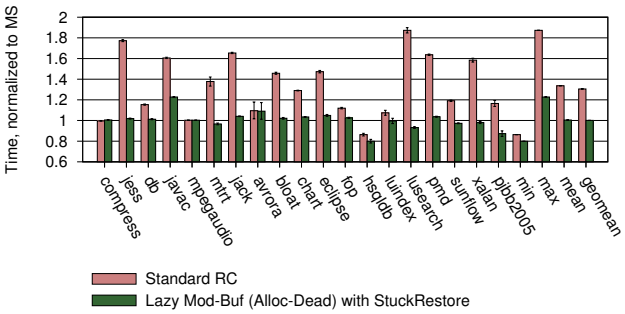
Allocate As Dead As a simple extension of the above optimization, instead of allocating objects live, with a reference count of one and a compensating decrement enqueued to the *dec-buf*, our second optimization allocates new objects as dead and does not enqueue a decrement. This inverts the presumption: the reference counter does not need to identify those new objects that are *dead*, but it must rather identify those that are reachable. 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 6(b) shows that decrements are reduced by over 80%.

We evaluate performance of both optimizations for lazy treatment of new objects. Figures 7(a) and 7(b) show the effect of the optimizations on total 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 16% in total time and 58% in collection time, on average, over the set of benchmarks. The two optimizations combined (*Lazy Mod-Buf (Alloc-Dead)*) are 19% faster in total time and 66% faster in collection time than *Standard RC* on average.

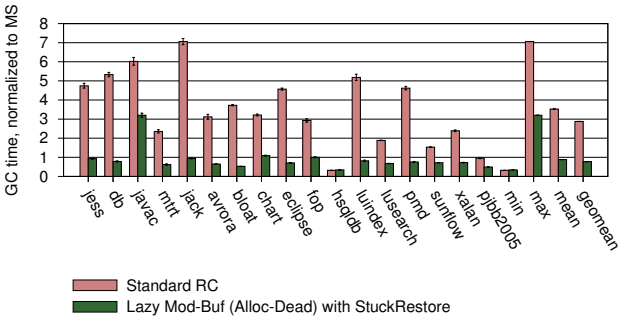
4.3 Bringing It All Together

Figure 8 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) allocate as dead. The combined effect of these optimizations is 24% faster in total time (Figure 8(a)) and 74% faster in collection time (Figure 8(b)) 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.

[1] We have contributed our code to Jikes RVM.

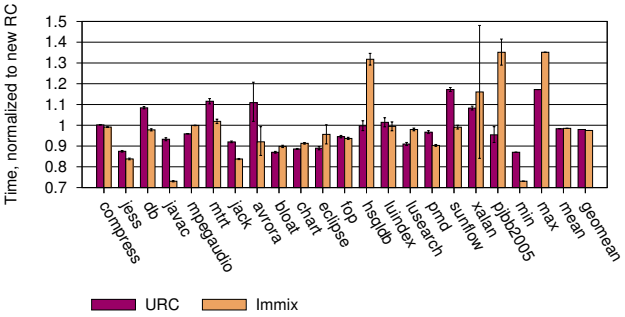


(a) Total time

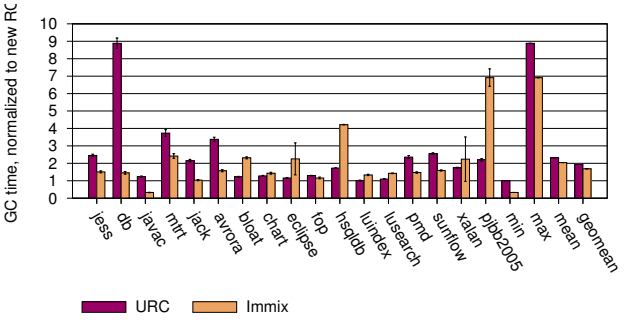


(b) GC time

Figure 9. Our optimized reference counting very closely matches mark-sweep, while standard reference counting performs 30% worse.



(a) Total time



(b) GC time

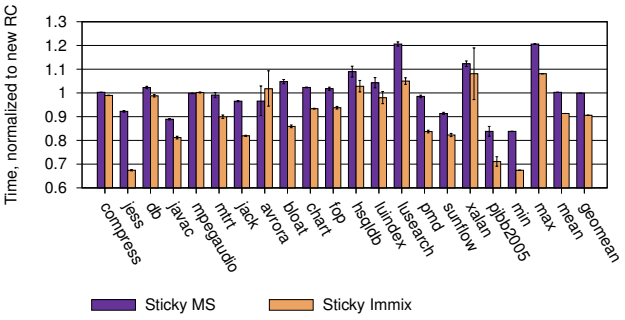
Figure 10. Our optimized reference counting collector also performs very well compared to ulterior reference counting and Immix. Our collector lags URC by 2% and Immix by 3% on average.

5. Back In The Ring

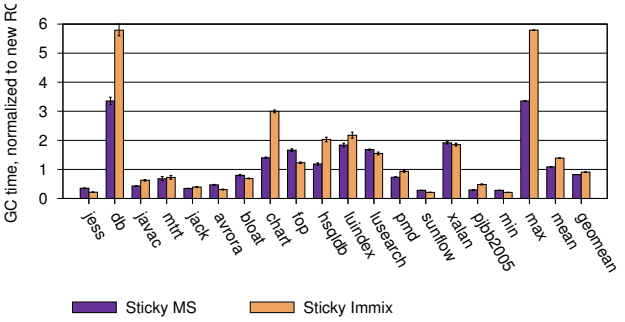
The conventional wisdom is that reference counting is totally uncompetitive compared to a modern mark-sweep collector [4]. Figure 9 shows the evaluation of *Standard RC* and *Lazy Mod-Buf (Alloc-Dead)* against a well tuned mark-sweep collector. Consistent with conventional wisdom, *Standard RC* performs substantially worse than mark-sweep, slowing down by 30%. However, our optimized reference counter, *Lazy Mod-Buf (Alloc-Dead)*, is able to entirely eliminate the overhead and perform marginally faster than mark-sweep on average, and is at worst 22% worse than mark-sweep (javac whose performance largely depends on the triggering of cycle collection) and at best 21% better than mark-sweep (hsqldb).

We compared our improved reference counting with ulterior reference counting [4] and Immix [5]. Ulterior reference counting combines copying generational collection for the young objects and reference counting for the old objects. Immix is a mark-region based tracing garbage collector with opportunistic defragmentation, which mixes copying and marking in a single pass. It achieves space efficiency, fast reclamation, and mutator performance. Much of its performance advantage over mark-sweep is due to its heap organization. Figure 10 shows that our improved reference counting is 2% slower than ulterior reference counting and 3% slower than Immix.

We also compare our improved reference counting with sticky mark bits collectors [5, 11]. These collectors are similar to ours in that they combine generational ideas in a non-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. Figure 11 shows that our improved reference counting collector performs the same as Sticky MS and 10% slower than



(a) Total time



(b) GC time

Figure 11. 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.

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 3% slower than Jikes RVM's production collector.

6. Conclusion

Of the two fundamental algorithms on which the garbage collection literature is built, reference counting has lived in the shadow of tracing. It has a niche among language developers for whom either performance or completeness is not essential, and is unused by mature high performance systems, despite a number of intrinsic advantages such as promptness of recovery and dependence on local rather than global state. The basis for its poor standing is that high performance reference counting significantly lags high performance tracing algorithms in performance.

We have conducted a comprehensive analysis of reference counting, confirmed that its performance lags mark-sweep by over 30%, and measured a number of reference counting intrinsics which give insight into its behavior and opportunities for improvement. We have identified two significant optimizations which together entirely eliminate the performance gap with mark-sweep. Unlike prior work, our optimizations are not hybrids, but modest changes to orthodox reference counting that significantly improve its performance.

Our hope is that our optimizations and our analysis of reference counting behavior will give new life to reference counting garbage collection.

References

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

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

[3] D. F. Bacon, C. R. Attanasio, H. B. Lee, V. T. Rajan, and S. Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *ACM Conference on Programming Language Design and Implementation*, pages 92–103, Snowbird, UT, USA, 2001. doi: 10.1145/378795.378819.

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

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

[6] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*, pages 137–146, Edinburgh, Scotland, UK, 2004. doi: 10.1109/ICSE.2004.1317436.

[7] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *Proceedings of the ACM Conference on Measurement & Modeling Computer Systems*, pages 25–36, New York, NY, USA, 2004. doi: 10.1145/1005686.1005693.

[8] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*, pages 169–190, Portland, OR, USA, 2006. doi: 10.1145/1167473.1167488.

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

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

[11] A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Proceedings of the Seventeenth Annual ACM Symposium on the Principles of Programming Languages*, pages 261–269, San Francisco, CA, USA, 1990. doi: 10.1145/96709.96735.

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

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

[14] I. Jibaja, S. M. Blackburn, M. R. Haghighat, and K. S. McKinley. Deferred gratification: Engineering for high performance garbage collection from the get go. In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, pages 58–65, San Jose, CA, USA, 2011. doi: 10.1145/1988915.1988930.

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

[16] Y. Levanoni and E. Petrank. An on-the-fly reference counting garbage collector for Java. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*, pages 367–380, Tampa, FL, USA, 2001. doi: 10.1145/504282.504309.

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

[18] R. D. Lins. Cyclic reference counting with lazy mark-scan. *Information Processing Letters*, 44(4):215–220, December 1992. doi: 10.1016/0020-0190(92)90088-D.

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

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

[21] H. Paz, E. Petrank, and S. M. Blackburn. Age-oriented concurrent garbage collection. In *International Conference on Compiler Construction*, Edinburgh, Scotland, UK, 2005. doi: 10.1007/978-3-540-31985-6_9.

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