THE AUSTRALIAN NATIONAL UNIVERSITY

TR-CS-02-06

Fast Garbage Collection without a Long Wait

Stephen M Blackburn and
Kathryn S McKinley

November 2002

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

Fast Garbage Collection without a Long Wait

Stephen M Blackburn[*]
Department of Computer Science
Australian National University
Canberra, ACT, 0200, Australia
Steve.Blackburn@cs.anu.edu.au

Kathryn S McKinley
Department of Computer Sciences
University of Texas at Austin
Austin, TX, 78712, USA
mckinley@cs.utexas.edu

ABSTRACT

General purpose garbage collectors have yet to combine short pause times with fast throughput. Collectors such as reference counting attain short pause times with significant performance penalties. Pure and hybrid generational collectors can achieve high throughput and have modest average pause times, but occasionally collect the whole heap and consequently incur long pauses. This paper introduces *RC-hybrid*, which combines copying nursery collection and reference counting the older generation to achieve both goals. Key to our algorithm is a generalization of deferred reference counting which allows RC-hybrid to safely ignore mutations to nursery objects. RC-hybrid thus restricts copying and reference counting to the objects for which they perform well. Copying collectors' bump pointer allocation is extremely fast, and collection time is proportional to the live objects whose survival rates are low and bounded in a fixed size nursery. Reference counting time is proportional to object mutations and the number of dead objects, both of which are typically low for older objects. We further bound time spent reference counting with collection triggers and buffering. We compare RC-hybrid with pure reference counting, a state-of-the-art copying and mark-sweep hybrid, and a number of other collectors. We show that RC-hybrid combines fast throughput, and low average and maximum pause times.

Categories and Subject Descriptors

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

General Terms

Design, Performance, Algorithms

Keywords

reference counting, copying, generational hybrid, Java

[*]This work is supported by NSF ITR grant CCR-0085792, and DARPA grant F33615-01-C-1892. Any opinions, findings, conclusions, or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

1. Introduction

A long standing and unachieved goal for general purpose garbage collectors is to combine short pause times with excellent throughput [16]. This goal is especially important for large server and interactive applications. (We focus in this paper on general purpose collectors as opposed to real-time collectors with hard deadlines [12, 19].) General purpose collectors [6, 12, 15, 20] that achieve consistently short pause times, such as reference counting, perform much worse than collectors such as a copying/mark-sweep generational hybrid [5] and Beltway [8] that are tuned for high performance. Figure 1 illustrates this point by plotting the average and maximum pause time behavior, and throughput of pure reference counting (RC), variable-nursery copying generational (VGEN), mark-sweep (MS), and a generational copying/mark-sweep hybrid (MS-hybrid) on two benchmarks. These results demonstrate that the low pause times of reference counting can significantly degrade total performance, and better performing collectors have poor maximum pause times which are due to occasional full heap collects. This dichotomy is also true of other collectors [6, 8, 12, 16, 20].

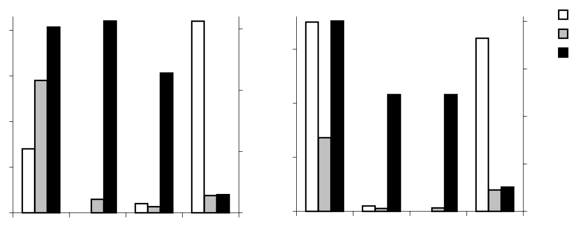


Figure 1: Throughput and Pause Time Behavior of Four General Purpose Collectors.

We present a new hybrid that achieves both goals by exploiting the different mutation and lifetime behaviors of young and old objects, and matching the best performing collectors and allocators to each context. Most young objects die quickly (the weak generational hypothesis [24]) and most pointer mutations are among young objects [3, 23]. These conditions favor a collection algorithm proportional to the live objects that does not keep track of pointers among young objects — fast bump-pointer allocation into a nursery collected by tracing and copying the reachable objects. The space overhead associated with copying is limited to just the nursery, and the collection time is a function of the survival rate. Objects that survive nursery collections mutate infrequently, and are likely to live a long time. These conditions favor a space efficient allocator and a collection algorithm proportional to the dead objects and pointer mutations — a free-list allocator and a reference counting collector.

We generalize deferred reference counting [14, 7, 18] to allow mutations of class variables and selected heap pointers to be ignored by the reference counter. This generalization provides the key to building an efficient hybrid reference counting collector. Previous work noticed that the reference counter should not keep track of the frequent updates to the stack and registers. Having excluded these pointers from the reference counts, collecting objects with a reference count of zero is *deferred* until the collector scans and updates the reference counts for stack and register object referents [14]. We extend deferral to heap pointers and statics (class variables). We do not reference count nursery objects, nor do we count pointers from the nursery to the reference counted space. We thus restrict the load of the reference counting collector to objects with low mutation and death rates. We use a special write barrier which in addition to remembering pointers into the nursery, stores increments and decrements of old-to-old pointer mutations in buffers (Bacon et al. also use buffers [6, 7]). Consequently, if a young object points to an old one and does not get promoted, we never reference count it. These objects are responsible for between 89% and 99% of object mutations in typical object-oriented programs [9, 23]. During the course of a nursery collection, all live nursery objects must be scanned. RC-hybrid exploits this phase to adjust the reference counts of objects referred to by surviving nursery objects. After a nursery collection, all objects are in the old space and RC-hybrid processes all the buffered increments and decrements, and frees garbage. We use the synchronous version of Bacon et al.'s cycle collection mechanism to free cyclic garbage and gain completeness [6, 7].

We achieve good throughput because the nursery performs well on young objects, and it effectively limits the reference counter to those objects that have relatively long lifetimes and mutate infrequently. This basic organization also tends to have low pause times because the time to collect a fixed-size nursery is bounded, and the old space is collected incrementally rather than through occasional full heap collections. However, the cycle detection algorithm we use is not fully incremental, and if all of the old objects are in one large cycle, we will incur a long pause. One of our benchmarks exhibits this behavior. For other situations, we introduce a number of triggers and exploit buffering to limit the time RC-hybrid spends freeing reference counted objects on any one collection, which if needed, buffers some frees until the next collection.

Our contributions are a new copying/reference-counting hybrid, a generalization of deferred reference counting which makes this hybrid possible, and a write barrier that provides correctness and efficiency. After we review related work, Section 3 describes this algorithm in detail. We use the Jikes RVM and JMTk, a new memory management toolkit for Java. JMTk contains implementations of all of the collectors we study. Hybrid collectors share implementations with their non-hybrid counterparts, and all collectors share common mechanisms (e.g., free lists, remembered sets). Our experiments thus truly compare policies rather than implementation subtleties. We evaluate and compare RC-hybrid with a number of other collectors in Section 5, and show that it is able to couple performance comparable to a state-of-the-art copying/mark-sweep generational collector on throughput (on average 5% worse), with much lower (up to a factor of 10) maximum pause times on the SPEC JVM benchmarks and a fixed workload variant of SPEC JBB. RC-hybrid actually has shorter pause times than a pure reference counting algorithm in many cases since it filters the reference counting load. We also present write barrier and other collector statistics that show how we achieve these results.

2. Related Work

This section explains deferred reference counting and its performance in detail, and then overviews other incremental approaches and high throughput collectors.

Reference counting tracks the number of pointers to each object by continuously monitoring pointer mutations [13, 16]. Each time a mutation overwrites a reference to o_{old} with a reference to o_{new}, it increments o_{new}'s reference count, and decrements o_{old}'s. If an object's reference count becomes zero, the collector reclaims it. Its work is proportional to the number of object mutations and dead objects. Its main advantage is that the work of garbage detection is spread out over every mutation, and it is thus very incremental. However, if a lot of objects become unreachable at once, the incrementality of reference counting suffers unless the collector bounds the number of objects it collects at once by *buffering* some of the processing [16] for the next collection, or performs collection concurrently. The main disadvantage of reference counting the entire heap is that the total cost of tracking all the pointers is expensive and seriously degrades mutator performance.

Deferred reference counting avoids monitoring certain heavily mutated pointer sources by occasionally checking their contents [14]. It thus does less work and finds garbage later. For example, it can safely ignore register and stack operations if object deletion is deferred until it examines the contents of the registers and stacks. Deutch and Bobrow achieve this via a *zero count table* (ZCT), which records objects with reference counts of zero [14]. Occasionally, they scan the stacks and registers, and any object in the ZCT which is *not* pointed to by the registers and stacks, and still has a reference count of zero is safely freed.

Bacon et al. [6, 7] build on this idea in a concurrent collector by *buffering* increments and decrements, and updating reference counts only during periodic collections. Instead of using a ZCT, their collector periodically examines the stacks and registers and applies temporary increments for each stack pointer into the heap. For each temporary increment, a matching decrement is applied in the next collection. To avoid race conditions, Bacon et al. used *epochs* to ensure that increments are always applied before decrements.

Previous work only applies deferral to registers and stacks. We generalize deferral to include other pointer sources such as heap objects and statics (class variables). We use buffering and only apply reference counts after examining and including deferred pointer sources. Specifically, we ignore mutations to nursery objects and are able to account for pointers from this space by exploiting the scan of live nursery objects that occurs as part of each nursery collection.

Other incremental collectors include MOS and concurrent collectors. The mature object space (MOS) collector traces and copies objects, incrementally packing connected objects together [15]. It achieves completeness without full heap collections and can be configured to be highly incremental, yielding low pause times. However, this comes at a performance cost, with objects potentially undergoing numerous collections before being identified as garbage. Concurrent tracing collectors [10] use a special write barrier to accommodate interference by the mutator in the tracing phase. The significant overheads of concurrent tracing has been addressed by dedicating a separate CPU to the task of collection [6, 7, 20]. We consider solutions that do not require additional CPUs to combine short pause times with good performance.

Generational algorithms exploit the low rate of object survival for the new *nursery* objects using tracing [5, 8, 16, 24]. *Tracing* identifies dead objects indirectly—by tracing the live objects and excluding those that it did not trace [16]. The cost of tracing algorithms is thus proportional to the number of live objects.

The two broad approaches to tracing are copying and mark-sweep. A copying collector copies all live objects into another space. A mark-sweep collector marks live objects, identifies all unmarked objects, and frees them. Copying collectors use monotonic (*bump-pointer*) allocation, and mark-sweep collectors use *free-list* allocation. Bump-pointer allocation is fast, but copying collectors pay a space penalty because they must hold an equal space in reserve for copying. Free-list allocation is slower, but needs no copy reserve.

Because of the high mortality of nursery objects [24], generational collectors copy *nursery* survivors to an older generation [5, 8, 16]. They repeatedly collect the nursery, and only collect the older generation when the heap is full. The older generation may be either a copied space (classic generational), or a mark-sweep collected space (MS-hybrid). Beltway collectors [8] generalize over classic copying generational collectors by adding incremental collection in independent *belts* (analogous to generations). Beltway configurations outperform generational collectors [8]. (We plan to have performance comparisons to Beltway in the final paper, but the current implementation of JMTk is just weeks old and does not yet include Beltway configurations.)

In this paper, we compare performance with MS-hybrid which combines the best of copying and mark sweep, using bump-pointer allocation but storing long-lived objects in a space efficient mark-sweep space. Previous work shows that it performs very well in practice [5], and several commercial JVMs use it. Generational collection on average tracks the time to collect the nursery, but it does not remove the need for full heap collection. In the worst case, all these collectors must pause while the collector traces the entires, full heap.

3. The RC-Hybrid Collector

The RC-hybrid is a generational collector. In the nursery, it uses a bump pointer allocator and a copying collector with collection costs proportional to the number of live objects. It performs well on nursery objects, since most are short-lived, and since the nursery is relatively small, the copy reserve overhead is limited. It uses a free-list allocator with a reference counting collector on the old generation with collection costs proportional to the number of dead objects and pointer updates. Since old objects have low object mortality and few pointer updates, it performs well. Since both components are incremental, it can recover garbage promptly and attain short pause times.

This section first presents generalized deferred reference counting. We then describe our collector organization, mechanics, and our write barrier that remembers pointers into the nursery and reference counts older objects. To control worst case pause times, we propose and implement a number of techniques.

3.1 Generalized Deferred Reference Counting

Reference counting shifts much of the load of collection onto the mutator by continuously monitoring pointer updates. Deferred reference counting limits the mutator load to changes to heap objects, avoiding the heavily mutated registers and stacks. We generalize this idea, and exclude pointers in class variables and selected heap objects from the mutator reference counting load.

The correctness of deferred reference counting depends on scanning stacks and registers before deleting any objects. More generally, deferred reference counting must enumerate and account for *all live pointer fields ignored by the mutator* before any reference counted object can be deleted, whether those pointer fields are in the registers, stacks, class variables or heap objects.

Jikes RVM stores class variables in a single array and the tracing collectors routinely enumerate them as part of the root scan. It is therefore easy to extend deferral to ignore mutations to class variables.

The efficient enumeration of ignored pointer fields in heap objects is less obvious. There are two clear possibilities: perform a traversal to identify those objects containing pointers which were ignored, or use a remembered set to record all such pointer fields which contain references to the reference counted heap. Both of these approaches could be quite expensive.

In RC-hybrid, we ignore pointer fields in nursery objects. By piggy backing the nursery collection, we enumerate the pointers within the surviving (live) nursery objects for free. Thus RC-hybrid removes the heavily mutated nursery objects from the mutator reference counting load, while cheaply maintaining the invariant of generalized deferred reference counting.

3.2 Organization and Mechanics

We organize RC-hybrid as follows. We have two generations: the young *nursery* space and the old reference counted space. We have a special write barrier to track pointers into the young space and reference count the old that we describe in detail below.

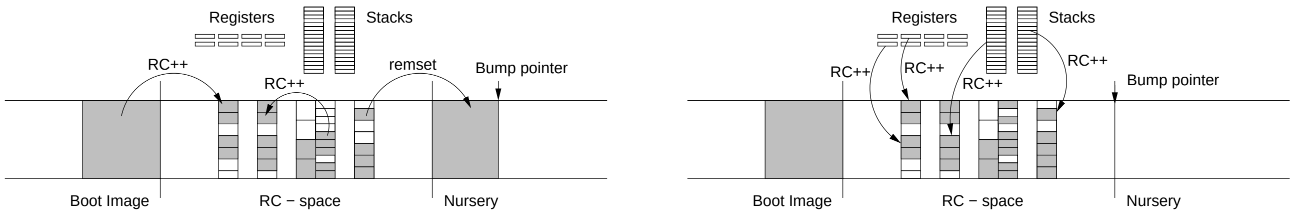
The mutator allocates into a nursery using a bump pointer. This type of allocator is faster than any of the alternatives. We defer reference counting the registers, stacks, class variables and the nursery. The old space uses a free-list allocator. It divides blocks of memory into size classes [17, 25] and keeps a free list for each block. (We list the sizes in Section 4.4.) The old space allocator puts objects in the smallest size class possible. It allocates a block to a size class when no block contains a free object of the right size. It changes the size class for a block only if the block is completely free.

When the nursery is full, the collector traces the live objects which are reachable either from the root set or from the old space, and copies (allocates) the survivors into the old, reference counted space. Each time an old space object is encountered in the root set, we follow Bacon et al.'s approach and generate a temporary increment, which is offset by a decrement during the next collection. When the nursery collection is complete, all objects are in the old space, and all the reference count updates are in the increment and decrement buffers. RC-hybrid then applies all of the increments, followed by the decrements. It frees objects with reference count zero, and decrements their children. During the decrement phase, we identify cyclic garbage candidates and collect them using Bacon et al.'s algorithm [6, 7], which we summarize in Section 3.4.

3.3 Write Barrier

The *write-barrier* records pointers into the nursery from the old space and records increments and decrements for reference counting. Figure 3 shows the Java code for our barrier. Its fast path ignores all pointers from the nursery. It uses a remembered set to record the source of old space to nursery pointers. It reference counts old objects by storing the new target in the increment buffer and the old target in a decrement buffer. Boot image objects are not reference counted so do not accrue decrements or increments. By not recording nursery to nursery pointer mutations (similar to the classic generational barrier), it eliminates 90% to 99% of pointer stores depending on the nursery size, as others have shown [9, 23] and as we demonstrate even for small nursery sizes in Section 5. For this same reason, we inline the fast path which does not record pointers and make a method call for the slow path. This selective inlining makes the code itself run slightly faster, and reduces the load on the JIT compiler [9].

We arrange memory such that the nursery is in high memory, the old space in middle memory, and the boot image in low memory,



(a) During mutator activity

(b) During GC

Figure 2: The Organization of the RC-hybrid Collector

as shown in Figure 2. If the pointer source is in the nursery (line 4 in Figure 3), we do not remember it. Otherwise, if the source points into the nursery from either the old space or the boot image, we remember it (line 11). If the pointer that this store overwrites (`old`, line 14) points into the old space, we insert a decrement. If the target is points into the old space we insert an increment. (See Section 4.5 for a discussion of the boot image.)

```
 1 private void writeBarrier(VM_Address src,
 2                           VM_Address tgt)
 3   throws VM_PragmaInline {
 4   if (src.LT(NURSERY_START))
 5     writeBarrierSlow(src, tgt);
 6 }
 7 private void writeBarrierSlow(VM_Address src,
 8                               VM_Address tgt)
 9   throws VM_PragmaNoInline {
10   if (tgt.GE(NURSERY_START)) {
11     remset.insert(src);
12   } else {
13     VM_Address old = VM_Magic.getMemoryAddress(src);
14     if (old.GE(RC_START) && old.LT(NURSERY_START))
15       decBuffer.push(old);
16     if (tgt.GE(RC_START))
17       incBuffer.push(tgt);
18   }
19 }
```

Figure 3: The RC-hybrid Write Barrier

3.4 Cycles

RC-hybrid also collects cycles. We follow the synchronous algorithm from Bacon and Rajan [7]. On every collection, the algorithm creates a candidate set of potential cycle roots from the decrements which do not go to zero. It colors these objects purple and puts them on a list. At the end of a collection, if the purple object is still live, it computes a transitive closure coloring the root and all reachable objects gray and decrementing their reference counts. It then finds all of the gray roots with reference count zero (cyclic garbage headers) and recursively frees them and their children with zero reference counts. For non-garbage objects, it restores the reference counts.

3.5 Controlling Pause Times

Nursery collection and reference counting times combine to determine pause times. We can simply control the nursery component by keeping the nursery size small. Since copying collection is proportional to the amount of live objects, we must not make it too small or more objects will survive, placing unnecessary load on the old generation. We show both these behaviors in Section 5.5. In the worst and pathological case, the bound on the pause must accommodate all nursery objects surviving.

The worst case for the reference counting space is that it is completely full and all of the objects die at once. For cycle detection, we can limit the number of potential root cycle (purple) objects we process at once. However, if the entire old space is one dead cycle, we will not be able to guarantee a short pause unless we abandon cycle collection, perform concurrent cycle detection [6, 7], or design an incremental cycle detection algorithm. One of our benchmarks exhibits this behavior.

To control pause times in other situations, we experiment with a trigger and buffering of deletions. The trigger computes the volume of meta data which sums the size of the remembered sets, the increment buffer, the decrement buffer, and the purple set. When this sum crosses a limit, we prematurely trigger a collection. We experiment with fixed size limits in Section 5.5. Another choice is to make the limit proportional to the heap size.

The final mechanism buffers the decrements and object deletions until the next collection. During the reference counting phase, we must first process all of the increments. As we then process the decrements, find dead objects, and delete them, we can bound the additional time we spend freeing by simply not processing all the decrements, leaving them in the decrement buffer for the next collection. We call this mechanism a pause time guideline. It works for cyclic and non-cyclic garbage. We implement it by examining a timer after we delete some threshold number of objects in the reference counting phase.

4. Methodology

This section first describes the Jikes RVM and the characteristics of the PowerMac G4 on which we do all experiments. We then briefly overview the range of collectors we study and how they work. As we pointed out in the previous section, all of these collectors share a common infrastructure, and reuse the same components unless otherwise noted. We then describe some features of the benchmarks we use in our experiments.

4.1 The Jikes RVM

We use the Jikes RVM (formerly known as Jalapeño) for our experiments with a new memory management tool kit JMTk (see Section 4.4). Jikes RVM is a high performance VM written in Java with an aggressive optimizing compiler [2, 1]. Jikes RVM offers three compiler choices: *baseline*, a quick non-optimizing compiler for all methods; *optimizing*, an aggressive optimizing compiler for all methods; and *adaptive*, it initially uses baseline and adaptively recompiles hot methods with the optimizing compiler [4]. The adaptive compiler uses sampling to select optimization candidates, and thus tends to make slightly different choices for each execution which are influenced by changes in the collector and write barrier. This non-determinism can make the adaptive compiler a difficult platform for any detailed study, but we use it for this study because it places the most realistic load on the system. This introduces variations in the load on the garbage collector because the write barrier for each collector is part of the runtime system as well as the program and induces both different mutator behavior and collector load [9]. Jikes RVM can be configured with various levels of ahead-of-time compilation. A minimal configuration only precompiles those classes essential to bootstrapping the VM (which does not include the optimizing compiler). We use the configuration which precompiles as much as possible, including key libraries and the optimizing compiler. We also turn off assertion checking for our experiments.[1]

4.2 Experimental Platform

We perform all of our experiments on a 1 GHz PowerMac G4, with 32KB on-chip L1 data and instruction caches, a 256KB unified L2 cache, 2MB L3 off-chip cache, and 512MB of memory, running PPC Linux 2.4.19. We run each benchmark five times and use the fastest of these.

4.3 Collectors

In addition to RC-hybrid, we use four collectors in our study (VGEN, MS, MS-hybrid and RC). We include two others (SS and FGEN) in our discussion below for clarity. We first summarize each collector and then discuss them in more detail.

SS: The semi-space collector uses one policy on the whole heap: bump-pointer allocation with a collector that traces and copies live objects.

FGEN: The fixed-size nursery generational collector uses the SS policy in a fixed-size nursery and an old space.

VGEN: The variable-size nursery generational collector [3] is the same as FGEN, except that the nursery size varies based on the size of the old generation, so it triggers collections at different points.

MS: The mark-sweep collector uses one policy on the whole heap: a free-list allocator and a collector that traces and marks live objects, and then reclaims unmarked objects.

MS-hybrid: The generational mark-sweep hybrid uses a SS policy in a fixed-size nursery, and a MS policy in the old space. It copies nursery survivors into the old space.

RC: The deferred reference-counting collector uses one policy on the whole heap: a free-list allocator and a collector that periodically processes mutator increments and decrements and deletes objects with reference count of zero.

RC-hybrid: The generational reference counting hybrid uses SS on a fixed-size nursery, and a RC policy on the old space. It copies nursery survivors into the old space. See Section 3.

We use several categories to describe these collectors further. The *generational* collectors divide the heap into a nursery and old generation and collect each independently. They are FGEN, VGEN, MS-hybrid, and RC-hybrid. The *whole heap* collectors, SS, MS, and RC, scavenge the entire heap on every collection. Pure *copying* collectors, SS, FGEN, and VGEN, only ever copy objects, and the pure *non-copying* collectors are RC and MS. For each collector, we now describe the collection triggering mechanisms, the write barrier, space overhead, and time overhead. We use the same mechanisms to scan objects and enumerate the stack and static pointers into the heap for all the collectors, although the actions taken on the enumeration vary. We fix the heap size in our implementation to insure fair comparisons, and for the purposes of this discussion.

SS: A semi-space collector [11] divides the heap in half, *to-space* and *from-space*, reserving half for copying into (since in the worst case all objects could survive) and half for allocation. It allocates using a bump pointer into the to-space until it is full. It then swaps to-space and from-space, scans all of the reachable objects in from-space, copies them to new to-space, and begins allocating into to-space again. It does not have a write barrier. Collection time is proportional to the number of survivors. Its performance suffers because it repeatedly copies objects that survive for a long time, and its pause time suffers because it collects the entire heap every time.

Each of the four generational collectors (FGEN, VGEN, MS-hybrid and RC-Hybrid) use the same allocation and collection mechanisms (and code!) for the nursery with the same resultant space and time overheads.

FGEN: The fixed-sized nursery two generational collector uses the SS policy in the nursery and old space. Filling the nursery of size *N* triggers a collection that copies the survivors to the old generation. Filling the entire usable heap triggers a whole heap collection in which FGEN copies survivors into the old generation. To collect the nursery independently of the higher generation, FGEN tracks pointers from the older generation into the nursery. We use a standard generational write barrier [9] that tests if the source is beyond the fixed nursery limit. Because object lifetimes typically follow the weak-generational hypothesis, it performs well. Its average pause time is also good because it is proportional to the nursery survivors, but its worst case pause time is proportional to collecting the entire heap.

VGEN: The variable-nursery generational copying collector [3] makes efficient use of memory by allowing the nursery to grow to consume all usable memory not consumed by the older generation. It collects the nursery only when both generations consume all usable memory. When the older generation consumes all usable memory, it collects the entire heap. (In practice, if the nursery size drops below some small fixed threshold, the heap is considered full.) This collector has better throughput than the fixed-nursery generational collector [8]. Its write barrier records old-to-nursery pointers by testing if the source is beyond the current nursery limit. Its throughput is even better than FGEN because it utilizes the heap more fully, but its average pause times suffer because of the nursery variability. Maximum pause times are similar to FGEN.

MS: A mark-sweep collector organizes the heap space using a free list. Our implementation partitions blocks of memory into size classes [17, 25], and allocates an object into the first available slot in the block of the smallest size class in which it fits. MS allocates a new block of the requested size class if no memory is available. It recycles blocks to different size classes only if the block becomes completely free. It triggers collection when the heap is full. The collector scans the reachable objects, marks them as reachable by setting their mark bit in a block table, and reclaims all unreachable objects. It then flips the sense of the mark bit. The time to collect is proportional to the number of live objects. The space requirements include the live objects and fragmentation due to both mismatches between object sizes and size classes (internal fragmentation), and distribution of live objects among different size classes (external fragmentation). Since MS is a whole heap collector, its maximum pause time is poor and its performance suffers from repeatedly tracing old objects.

MS-hybrid: This hybrid generational collector uses a fixed-size nursery and a mark-sweep policy for the older generation. When the nursery fills up, it triggers a nursery collection. If after a nurs-

[1]This build-time configuration is known as *Fast*.

	allocation				write barrier			RC increments			RC decrements		
benchmark	alloc	SS min	MS min	alloc:SS	fast	slow	ref cnt	pure	hybrid	%	pure	hybrid	%
_201_compress	130M	18M	10MB	7:1	118K	6.2%	3.2%	73K	773	1.0%	173K	9K	5.3%
_202_jess	296M	18M	10MB	16:1	26M	0.1%	0.4%	26M	38K	0.1%	34M	428K	1.2%
_205_raytrace	168M	18M	10MB	9:1	2M	0.3%	2.6%	1M	59K	3.6%	8M	161K	2.0%
_209_db	94M	23M	14MB	4:1	33M	0.1%	9.1%	33M	2M	9.0%	36M	3M	8.3%
_213_javac	531M	28M	19MB	19:1	21M	2.5%	12.1%	18M	1M	8.3%	29M	2M	9.1%
_227_mtrt	182M	21M	21MB	9:1	3M	0.3%	2.3%	1M	59K	3.0%	8M	159K	1.8%
_228_jack	650M	16M	18MB	41:1	13M	6.3%	1.6%	10M	80K	0.7%	25M	735K	2.9%
pseudojbb	622M	28M	30MB	35:1	22M	2.3%	18.8%	16M	1M	11.2%	35M	3M	10.2%

Table 1: Benchmark Characteristics

ery collection the entire heap is full, it performs a full heap MS collection over the old space. The write barrier thus only remembers pointers from the old space to the nursery. By exploiting the generational hypothesis, MS-hybrid mitigates the drawbacks of MS for throughput and average pause times, but full heap collections drive up maximum pause times. It uses space more efficiently than FGEN and VGEN because it has a non-copying old space.

RC: The pure deferred reference-counting collector organizes the heap using the same free-list allocator as MS. The write barrier remembers all new pointers to an object in an increment buffer, and over-written pointers to objects in a decrement buffer. Collection time is proportional to number of dead objects, but the mutator load is significantly higher than the generational collectors since it records one or two entries for every heap pointer store.

4.4 JMTk

Together with researchers at IBM Watson, we recently developed a new composable memory management framework (JMTk) for exploring garbage collection and memory management algorithms. JMTk separates allocation and collection policies, then mixes and matches them. It also provides a number of mechanisms shared between algorithms such as write barriers, pointer enumeration, and sequential store buffers for storing remembered sets, increments, etc. The heap it manages includes all dynamically allocated objects, inclusive of the program, compiler, and itself (e.g., the collector meta data such as remembered sets). It contains implementations of all of the collectors we study, and hybrids share the non-hybrid components. Because of the shared mechanisms and code base, our experiments truly compare policies.

For our free-list allocators, we use a range of size classes similar to but smaller than the Lea allocator [17]. We selected size classes with the goal of worst case internal fragmentation of 1/8. The size classes are arranged in 4, 8, 16, 32, 256, and 1024 byte steps, so small, word-aligned objects get an exact fit. All objects 8KB or larger get their own block.

JMTk implements the large object space (LOS) as follows. For pure free-list allocators (MS and RC), we just allocate large objects directly. For hybrid collectors with a free-list in the older generation, we allocate (*pretenure*) objects that are 64K or larger directly onto the free-list in the old space. For SS, FGEN, and VGEN, we add a MS space only for these objects. During full heap collections, we scan and collect the large objects.

4.5 Boot Image

There is one limitation in our implementations of RC and RC-hybrid with respect to the Jikes RVM boot image. The boot image contains various objects and precompiled classes necessary for booting Jikes RVM, including the compiler, classloader and other essential elements of the virtual machine. None of the Jikes RVM collectors *collect* the boot image objects. Jikes RVM's tracing collectors (including SS, VGEN, FGEN, MS, and MS-Hybrid) trace through the boot image objects whenever they perform a full heap collection. We have not implemented reference counting of boot image objects.[2] Since none of the reference counting collectors ever scan the entire heap, they do not scan the boot image, and thus dead boot image objects that point into the heap cause excess retention (our write barrier records increments from those pointers). Bacon et al. have this same limitation in their reference counting algorithms, which we discovered by examining the publicly available source code [6].

4.6 Benchmarks

Table 1 shows key characteristics of each of our benchmarks. We use seven taken from the SPEC JVM benchmarks, and pseudojbb, a slightly modified variant of SPEC JBB2000 [21, 22]. Rather than running for a fixed time and measuring transaction throughput, pseudojbb executes a fixed number of transactions. This modification made it possible to compare running times under a fixed garbage collection load.

For these results, we compile each benchmark with the Jikes RVM adaptive compiler. For all of the generational collectors, we inline the write barrier *fast path* which filters out stores to nursery objects and thus does not record between 93.7% to 99.9% of pointer updates. Depending on the collector, the *slow-path* makes the appropriate entries into the remembered set, increment buffer, and/or decrement buffer. The semispace and pure mark-sweep collectors have no write barrier. Since the write barrier for the pure reference counter is unconditional, it is fully inlined.

The 'alloc' column in Table 1 indicates the total number of bytes allocated for each benchmark. The next two columns indicate the minimum heaps in which the benchmarks can run with the SS and MS collectors respectively (this heap size is inclusive of the memory requirements of the adaptive compiler compiling the benchmark). The fourth column indicates the ratio between total allocation and SS minimum heap size, giving some indication of the garbage collection load for each benchmark.

The write barrier columns show for RC-hybrid with a 4MB nursery: the number of times the write barrier is invoked ('fast'), the frequency with which the slow-path is taken in order to remember pointers into the nursery ('slow', line 11 of Figure 3), and the frequency with which RC-hybrid executes the reference counting portion of its write barrier ('ref cnt', lines 13–17). The final six columns indicate the number of increments and decrements performed by RC and RC-hybrid respectively for each of the benchmarks.

These results show that the nursery effectively reduces the load on the reference counter, and that their overall contribution is low with respect to all pointer stores.

5. Results

[2]This implementation would require statically establishing correct reference counts for all boot image objects, and then writing appropriate initial values into the headers of those objects at boot image writing time.

We now compare RC-hybrid with other collectors. We discuss throughput and pause time. For pause time, we present average, maximum, and minimum mutator utilization results. We study in detail the influence of cycle garbage detection on _213_javac, which contains a lot of cyclic job. For two sample programs, we explore many variations in heap size in Section 5.4. As our default RC-hybrid configuration, we use a nursery size of 4MB, a pause time guideline of 40ms, and a meta-data limit of 512K (see Section 3.5). Section 5.5 shows the sensitivity of total time and maximum pause times to variations in these parameters.

We begin by comparing RC-hybrid and MS-hybrid. Figure 4 compares throughput, average pause time, and maximum pause times for the heap sizes shown in Figure **??**c). We choose a program specific, relatively large heap size in which MS-hybrid would execute at least one full heap collection. For _205_raytrace and _227_mtrt this was not possible.

We then compare with VGEN, MS, RC as well as MS-hybrid. These experiments were done at different heap sizes because several of the collectors, which are not as space efficient as MS-hybrid, cannot complete in the same heap sizes. Table 2 shows throughput and pause times for the VGEN, MS, RC, MS-hybrid, and RC-hybrid collectors, along with the heap sizes we used in that experiment. A number of the collectors that must by design perform full heap collections do not, which misrepresents a typical maximum pause time for them.

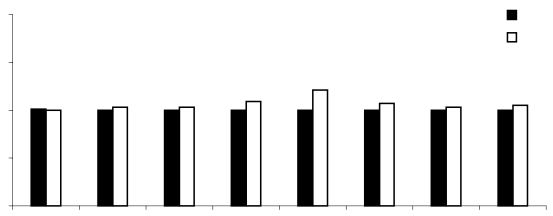
5.1 Throughput

RC-hybrid gives throughput comparable to MS-hybrid and VGEN, the high throughput collectors. Figure 4 shows that with the exception of _213_javac (discussed in Section 5.3), RC-hybrid is within 10% of MS-hybrid, and often within 3%. Comparing against pure MS and RC, Table 2 shows that RC-hybrid matches or outperforms MS and RC on all benchmarks, outperforming each of them by more than 50% in some benchmarks.

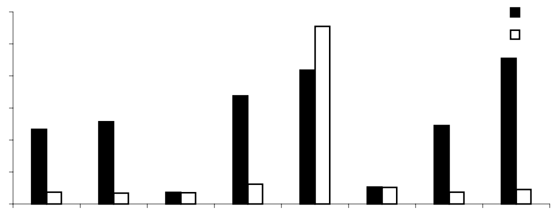
This result is not surprising, because generational collectors such as MS-hybrid and VGEN are also designed to exploit the typical space and time behavior of young and old objects. The space and allocation-time advantages of a bump-pointer / freelist hybrid benefit both MS-hybrid and RC-hybrid. The collection-time advantage of generational collection also benefits both hybrids, as well as VGEN. A comparison of MS and RC in Table 2 confirms the conventional wisdom that the trade-off between lower collection time and higher mutator overhead inherent in reference counting leads to an overall reduction in throughput. However, RC-hybrid dramatically limits its exposure to this trade-off by using reference counting only with the low-maintenance older objects (Table 1). The performance of RC-hybrid is also impacted by a 4 byte per object overhead required by the reference counter. We have not yet implemented the obvious optimization of excluding that header from nursery objects.

5.2 Pause Times

RC-hybrid has very good average and maximum pause times, with the exception of _213_javac (discussed below). In fact, Table 2 shows that it has better average and maximum pause times than pure RC because it is exposed to less load. Figure 4 shows that MS-hybrid performs full heap collections for most of the benchmarks, and attains the expected poor maximum pause time behavior due it its mark-sweep of the full older generation. On two of the benchmarks, _205_raytrace and _227_mtrt, MS-hybrid did not perform a full heap collection at the selected heap size and could not run to completion at the next smaller heap size. Hence for these benchmarks MS-hybrid's maximum pause time reflects the modest cost



(a) Throughput.



(b) Pause time.

benchmark	heap used MB	best time sec	MS-hybrid norm time	MS-hybrid pause ms avg	MS-hybrid pause ms max	RC-hybrid norm time	RC-hybrid pause ms avg	RC-hybrid pause ms max
_201_compress	27M	95.4	1.01	84	233	1.00	12	37
_202_jess	7M	21.3	1.00	4	257	1.03	5	34
_205_raytrace	8M	28.6	1.00	4	36	1.03	5	35
_209_db	12M	38.4	1.00	18	338	1.09	7	62
_213_javac	60M	26.9	1.00	20	418	1.21	38	555
_227_mtrt	11M	29.1	1.00	4	53	1.07	4	52
_228_jack	18M	30.6	1.00	6	245	1.03	6	37
pseudojbb	11M	47.5	1.00	20	455	1.05	27	45

(c) Tabulated results.

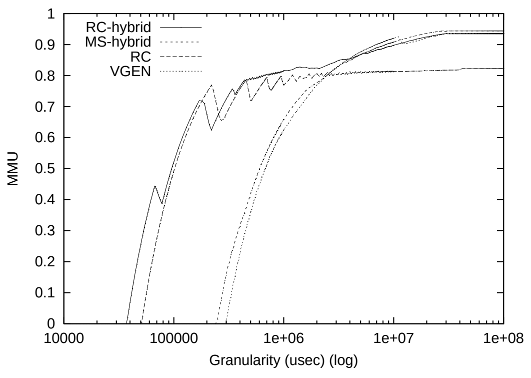
Figure 4: A comparison of MS-hybrid and RC-hybrid, showing normalized execution time, average pause time and max pause times.

of a nursery collection. In Table 2 some of the heap sizes are larger, so MS-hybrid is less exposed to full heap collections. However, the substantial cost of full heap collections is clear in the pause time results for MS, and to a lesser extent in VGEN, which performs full heap collections in a number of the benchmarks, and suffers the maximum pause time consequence.

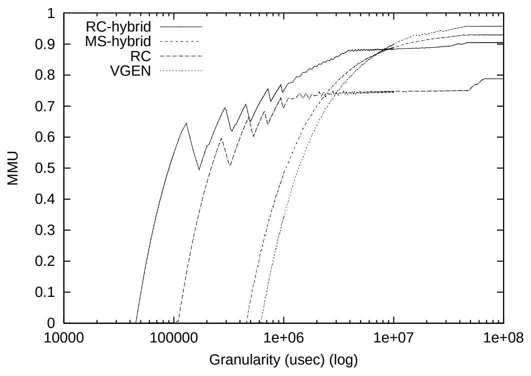
Although maximum pause time is an important measure of responsiveness, a tight cluster of short pauses may be just as damaging to an application's progress as a single longer pause. Minimum mutator utilization (MMU) characterizes pause time behavior with respect to the percentage of time in an interval in which the mutator does useful work. The intuition behind MMU is to consider responsiveness with respect to a certain minimum CPU utilization requirement of the application. The MMU curve then identifies the maximum period that the application's requirement will not be satisfied. For example, in Figure 5a), an application that required at least 50% of the CPU (MMU $= 0.5$) would experience a pause of around 10msec from RC and RC-hybrid, and a pause of around 40msec from MS-Hybrid and VGEN. The *x*-intercept of an MMU curve reflects maximum pause time while the asymptotic *y*-value reflects total mutator utilization. These MMU graphs illustrate that RC-hybrid performs very well, both in terms of responsiveness,

benchmark	heap used MB	best time sec	MS norm time	MS pause ms avg	MS pause ms max	RC norm time	RC pause ms avg	RC pause ms max	VGEN norm time	VGEN pause ms avg	VGEN pause ms max	MS-hybrid norm time	MS-hybrid pause ms avg	MS-hybrid pause ms max	RC-hybrid norm time	RC-hybrid pause ms avg	RC-hybrid pause ms max
_201_compress	27	95.4	1.00	167	394	1.00	3	14	1.01	102	265	1.01	84	233	1.00	12	37
_202_jess	10	20.5	1.77	151	394	1.62	53	59	1.05	5	314	1.00	5	37	1.02	6	37
_205_raytrace	13	28.2	1.21	174	395	1.17	51	161	1.00	4	37	1.00	7	57	1.02	7	58
_209_db	20	37.2	1.09	207	395	1.37	46	99	1.00	50	419	1.00	16	78	1.07	28	74
_213_javac	78	25.8	1.19	289	395	1.88	57	478	1.00	41	492	1.02	19	39	1.19	42	555
_227_mtrt	23	28.7	1.15	212	395	1.20	42	52	1.00	7	53	1.01	10	56	1.02	11	59
_228_jack	18	30.5	1.35	156	395	1.35	45	50	1.01	7	298	1.00	6	246	1.03	6	37
pseudojbb	110	46.1	1.13	431	609	1.44	56	59	1.00	43	618	1.03	20	457	1.08	27	45

Table 2: A comparison of MS, RC, VGEN, MS-hybrid and RC-hybrid, showing normalized execution time, average pause time and max pause times.



(a) _228_jack



(b) pseudojbb

Figure 5: Minimum mutator utlization (MMU).

where it matches or exceeds RC, and in terms of throughput, where it is competitive with MS-hybrid and VGEN. This mixing of the best of throughput and responsiveness is quite clear in Figure 5a), where the RC-hybrid curve follows RC's good response time curve and then 'jumps' to the MS-hybrid/VGEN curve to achieve good throughput.

5.3 Cycle Detection

In our initial implementation of RC-hybrid we implement a simple, synchronous cycle detection algorithm [7] which we describe in Section 3.4. Table 3 shows that for _213_javac, this algorithm slows RC-hybrid down and leads to very large pauses. _213_javac generates a large amount of cyclic garbage [6]. The very large pause is a result of the non-incremental nature of the synchronous cycle detection algorithm we use (any graph of potentially cyclic garbage must be traversed *completely* before the mutator can resume).

_213_javac	time	avg p	max p
With cycle detection	1.17	43	557
Without cycle detection	1.05	23	40

Table 3: Performance of RC-hybrid with and without cycle detection.

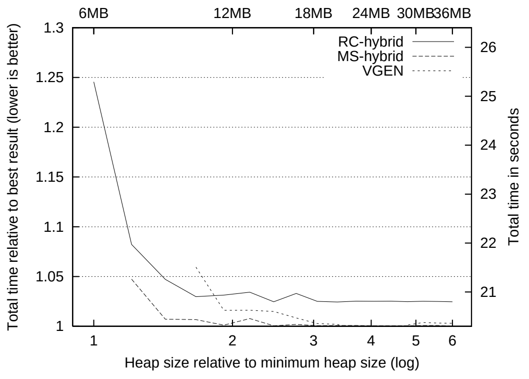
As we point out in Section 3.5, this problem could be addressed either through the implementation of an incremental or concurrent cycle detection [6]. Because RC-hybrid reduces the load on the reference counting system by 89%–99% (see Table 1) we expect concurrent cycle collection mechanism to impact the performance of RC-hybrid much less than in a pure reference counting system. RC-hybrid's 58% performance advantage over RC on _213_javac confirms this view (see Table 2).

5.4 Sensitivity to Heap Sizes

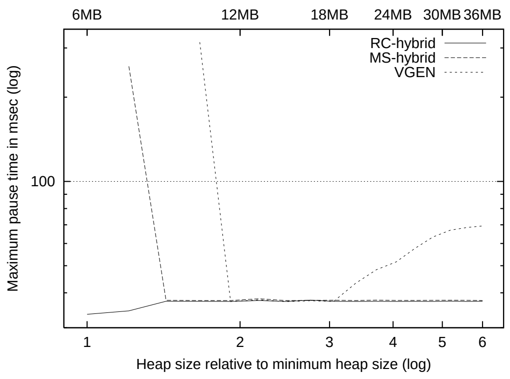
In Figure 6 we see the impact of heap size on both throughput and responsiveness for RC-hybrid, MS-hybrid and VGEN. The impact on throughput is unsurprising—as heap space diminishes, each of the collectors must do more work and so the total throughput degrades until they are unable to satisfy the application's requests. Both of the hybrids are able to operate in smaller heaps, most likely because their free-list older generation is more space efficient that the copying older generation of VGEN. We speculate that RC-hybrid is able to run in a smaller heap that MS-hybrid on this benchmark because the continuous reclamation of the reference counter leads to less fragmentation in the free list, as compared to long periods of allocation followed by occasional freeing in a full heap collect.

The impact of heap size on maximum pause time is more interesting. VGEN and MS-hybrid suffer sudden and massive degradations in pause time behavior when they perform full heap collections. With small, short running, benchmarks such as those in the SPEC JVM suite, full heap collections can be avoided by simply making the heap bigger. The pause time behavior of VGEN gradually degrades as the heap becomes much larger. This is because VGEN has a variable size nursery, which can be as large as half of the heap. Although much lower survival rates mean that full-heap nursery collections are much cheaper than full heap older generation collections, the maximum pause for VGEN will nonetheless grow as the heap size grows. Interestingly RC-hybrid's maximum pause times shrink as the heap gets smaller. This is because the nursery gets smaller, and as it does so, reference counting collections become more frequent, but shorter. So although maximum pauses are better, more frequent, shorter, collections degrade overall mutator utilization.

5.5 Collection Triggers



(a) Throughput



(b) Pause time

Figure 6: Varying heap sizes for _202_jess.

For Table 4, we experimented with settings for the collection triggers described in Section 3.5 to explore their effects on total time and the maximum pause time. Our base line configuration uses a nursery size of 4MB, a pause time guideline of 40ms, and a meta-data limit of 512K.

Except for pseudojbb, smaller nursery sizes change total execution time by very little, and even speed it up by 1 or 2% in a few cases. In pseudojbb, a 1MB nursery, which is below 1% of the total heapsize degrades total performance by 9%. The small nursery is the most effective mechanism for reducing maximum pause times (and average). _213_javac does not run in small nurseries, and we believe the large number of promoted cycles with the pause time guideline prevents reference counting from making progress.

The pause time guideline trigger and the meta-data limit hardly change total time (except for _213_javac). Since we apply the pause guideline after the nursery collection and the first set of decrements, a small value enables only a small set of deletions, and very little cycle detection. For _213_javac with a small trigger value, it again fails to find enough free memory to execute. Its relative performance actually increases when we disable the trigger because it reclaims dead cycles more efficiently in larger increments. pseudojbb shows the most value from this trigger, and its maximum pause times are directly correlated with the guideline. The maximum pause times of the other programs remains exactly the same. Varying the meta data limit also changes the pause times for _213_javac and pseudojbb. _213_javac runs out of control without this limit; its total time increases by 30% and its maximum pause time by a factor of 10 as it reclaims cycles and tries to reclaim cycles that are not yet dead. The smaller 256K meta data limit actually increases the max pause time for pseudojbb. We hypothesis that it triggers collections too often and promotes more nursery survivors that die in the reference counting space. s

6. Conclusion

The tension between responsiveness and throughput has been longstanding in the garbage collection literature. Until now, collectors have either exhibited good throughput performance or good responsiveness, but not both.

We describe RC-hybrid, a new garbage collector that by carefully matching allocation and collection policies to the behaviors of older and younger object demographics, delivers both good throughput and good responsiveness. Key to our algorithm is a generalization of deferred reference counting which allows mutations to nursery objects to be safely ignored, reducing the reference counting load by around 90%.

7. Acknowledgements

We want to thank IBM Research and in particular Perry Cheng with whom we designed and implemented the JMTk. Without JMTk and the Jikes RVM, this work would not have been possible. We also thank Michael Hind and David Grove for supporting this work.

8. REFERENCES

[1] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA '99, Denver, Colorado, November 1-5, 1999*, volume 34(10) of *ACM SIGPLAN Notices*, pages 314–324. ACM Press, Oct. 1999.

[2] B. Alpern, D. 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 System Journal*, 39(1):211–238, February 2000.

[3] A. W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[4] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *OOPSLA'00 ACM Conference on Object-Oriented Systems, Languages and Applications, Minneapolis, MN, USA, October 15-19, 2000*, volume 35(10) of *ACM SIGPLAN Notices*, pages 47–65. ACM Press, October 2000.

[5] C. R. Attanasio, D. F. Bacon, A. Cocchi, and S. Smith. A comparative evaluation of parallel garbage collectors. In *Languages and Compilers for Parallel Computing, 14th International Workshop, LCPC 2001, Cumberland Falls, KY, USA, August 1-3, 2001*, Lecture Notes in Computer Science. Springer-Verlag, 2001.

[6] 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 *Proceedings of the ACM SIGPLAN'01 Conference on Programming Languages Design and Implementation (PLDI), Snowbird, Utah, May, 2001*, volume 36(5) of *ACM SIGPLAN Notices*. ACM Press, June 2001.

[7] D. F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In *Proceedings of 15th European*

			Nursery Size						Pause Time Guideline						MetaData Limit					
	default		1MB		2MB		∞		20ms		80ms		∞		256K		2MB		∞	
benchmark	time	max	time	max	time	max	time	max	time	max	time	max	time	max	time	max	time	max	time	max
_201_compress	1.01	37	1.00	18	1.00	29	1.00	43	1.01	37	1.01	37	1.01	37	1.01	37	1.01	37	1.01	37
_202_jess	1.01	34	1.00	18	1.02	29	1.01	34	1.01	34	1.01	34	1.01	34	1.01	34	1.01	34	1.01	34
_205_raytrace	1.01	35	1.00	19	1.00	32	1.01	35	1.01	35	1.01	35	1.01	35	1.01	35	1.01	35	1.01	35
_209_db	1.00	62	1.01	41	1.02	38	1.00	70	1.00	62	1.00	62	1.01	62	1.00	62	1.00	62	1.01	62
_213_javac	1.12	556					1.00	363			1.12	358	1.08	235	1.12	299	1.42	3062	1.32	883
_227_mtrt	1.00	52	1.02	18	1.00	32	1.00	54	1.00	52	1.00	52	1.00	52	1.00	52	1.00	52	1.00	52
_228_jack	1.02	37	1.05	18	1.02	29	1.00	43	1.02	37	1.02	37	1.02	37	1.02	37	1.02	37	1.02	37
pseudojbb	1.04	45	1.13	45	1.07	44	1.00	259	1.04	37	1.04	87	1.04	115	1.03	218	1.03	44	1.04	67

Table 4: Sensitivity to variations in collection triggers (defaults are 4MB nursery, 40ms pause time guideline, and 1M meta data limit).

Conference on Object-Oriented Programming, ECOOP 2001, Budapest, Hungary, June 18-22, 2001.

[8] S. M. Blackburn, R. E. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation, PLDI'02, Berlin, June, 2002*, volume 37(5) of *ACM SIGPLAN Notices*, Berlin, Germany, June 2002. ACM Press.

[9] S. M. Blackburn and K. S. McKinley. In or out? Putting write barriers in their place. In *Proceedings of the Third International Symposium on Memory Management, ISMM '02, Berlin, Germany*, volume 37 of *ACM SIGPLAN Notices*. ACM Press, June 2002.

[10] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

[11] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, Nov. 1970.

[12] P. Cheng and G. Belloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Languages Design and Implementation (PLDI), Snowbird, Utah, May, 2001*, volume 36(5) of *ACM SIGPLAN Notices*, pages 125–136. ACM Press, June 2001.

[13] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, Dec. 1960.

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

[15] R. L. Hudson and J. E. B. Moss. Incremental garbage collection for mature objects. In Y. Bekkers and J. Cohen, editors, *Proceedings of the First International Workshop on Memory Management, IWMM'92, St. Malo, France, Sep, 1992*, volume 637 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

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

[17] D. Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html, 1997.

[18] R. D. Lins. Cyclic reference counting with lazy mark-scan. *Information Processing Letters*, 44(4):215–220, 1992. Also Computing Laboratory Technical Report 75, University of Kent, July 1990.

[19] S. Nettles and J. W. O'Toole. Real-time replication garbage collection. In *Proceedings of SIGPLAN 1993 Conference on Programming Languages Design and Implementation*, pages 217–226, Albuquerque, NM, June 1993.

[20] T. Printezis and D. Detlefs. A generational mostly-concurrent garbage collector. In *Proceedings of the International Symposium On Memory Management (ISMM), Minneapolis, U.S.A, 15-16 October, 2000.* ACM Press, 2000.

[21] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

[22] Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 edition, 2001.

[23] D. Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. PhD thesis, University of Massachusetts, 1999.

[24] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984.

[25] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In H. Baker, editor, *Proceedings of International Workshop on Memory Management, IWMM'95, Kinross, Scotland*, volume 986 of *Lecture Notes in Computer Science*. Springer-Verlag, Sept. 1995.