Ulterior Reference Counting: Fast Garbage Collection without a Long Wait

Stephen M Blackburn
Department of Computer Science
Australian National University
Canberra, ACT, 0200, Australia
Steve.Blackburn@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 high throughput. For example, generational collectors can achieve high throughput. They have modest average pause times, but occasionally collect the whole heap and consequently incur long pauses. At the other extreme, concurrent collectors, including reference counting, attain short pause times but with significant performance penalties. This paper introduces a new hybrid collector that combines copying generational collection for the young objects and reference counting the old objects to achieve both goals. It restricts copying and reference counting to the object demographics for which they perform well. Key to our algorithm is a generalization of deferred reference counting we call *Ulterior Reference Counting*. Ulterior reference counting safely ignores mutations to select heap objects. We compare a generational reference counting hybrid with pure reference counting, pure mark-sweep, and hybrid generational mark-sweep collectors. This new collector combines excellent throughput, matching a high performance generational mark-sweep hybrid, with low maximum pause times.

Categories and Subject Descriptors

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

General Terms

Design, Performance, Algorithms

Keywords

Ulterior Reference Counting, 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. This goal is especially important for large server and interactive applications. Table 1 illustrates the state of this trade-off with the throughput (running time), and responsiveness (maximum pause) of a highly responsive reference counting (RC) collector and a high throughput generational collector (BG-MS) on two Java benchmarks. Generational (G) collectors separate new objects from old ones since they tend to have different demographics. BG-MS uses a *bounded* (B) copying *nursery* for newly allocated objects whose size varies between an upper and lower bound. It uses a mark-sweep (MS) free-list for the old objects. This type of collector is a popular high performance choice in modern virtual machines. BG-MS significantly improves total throughput compared with RC due to the pointer tracking costs in RC. BG-MS has much higher maximum pause times than RC due to full heap collections.

	Total Time (sec)		Max Pause Time (ms)	
benchmark	BG-MS	RC	BG-MS	RC
_228_jack	7.2	12.7	185	72
_209_db	19.2	21.3	238	43

Table 1: Throughput and Responsiveness of High Throughput (BG-MS) versus Highly Responsive (RC) Collectors

To resolve this dichotomy, we introduce a new generational collector with a copying nursery and reference counting mature space that combines high throughput and low pause times. The key to our algorithm is a generalization of deferred reference counting [19], called *Ulterior Reference Counting*[1]. A reference counting (RC) collector computes the number of references to an object, and when it falls to zero, RC reclaims the object. To improve performance, deferred RC ignores frequent pointer mutations to the stacks and registers which eliminates most of their reference counting load. Deferred RC must thus postpone all reclamation until it periodically enumerates the stacks and registers.

Ulterior reference counting (URC) extends deferral to select heap objects and object fields by periodically enumerating the pointers within them. URC divides the heap into logical partitions of RC and non-RC objects. It uses an RC collection policy on the RC objects, and selects other policies for the non-RC objects. URC can either enumerate the deferred pointers by tracing them, or by using a write barrier to remember deferred mutated pointers. To build an efficient collector with this mechanism, we want to (1) defer the fields of highly mutated objects and enumerate them quickly, and (2) reference count only infrequently mutated fields.

[1]Dictionary definitions of ulterior include occurring later and lying beyond what is evident.

Object lifetime and mutation demographics combine well to fit these requirements. Young objects mutate frequently [3, 29] and die at a high rate (the weak generational hypothesis [24, 30]). Old objects mutate infrequently [29] and die at a slower rate. These demographics favor generational collection, with a copying algorithm for the young objects. Copying uses fast contiguous (*bump pointer*) allocation, ignores pointers among young objects, and is proportional only to the number of live objects. The less frequent mutation and high survival rate of older objects favor (1) a space efficient free-list allocator, and (2) a reference counting collection algorithm which is proportional to the number of dead objects and pointer mutations.

We implement BG-RC, a *hybrid generational* collector that partitions the objects into a bounded copying nursery (BG) for newly allocated objects, and an RC mature space for objects that survive a nursery collection. BG-RC is a hybrid because it combines different garbage collection algorithms into one collector. It continually reference counts the mature space. It ignores nursery pointer mutations. BG-RC efficiently piggy backs reference counting on to the enumeration of live pointers during the tracing and copying of surviving nursery objects into the RC space. After every nursery collection, it computes reference counts for the mature space and reclaims objects with no references. It uses a variant of Bacon and Rajan's trial deletion to detect cyclic garbage in the RC space [9].

We compare BG-RC with pure deferred reference counting (RC), pure mark-sweep (MS), and a high throughput generational mark-sweep hybrid (BG-MS), implementing them all in Jikes RVM [1, 2] using JMTk [11], a new memory management toolkit for Java. The hybrid collectors share implementations with their non-hybrid counterparts. All collectors share common mechanisms and are efficient [11]. Our experiments thus compare policies rather than implementation subtleties. The results demonstrate that BG-RC matches throughput with the high performance BG-MS (on average 2% better in moderate heap sizes) but has dramatically lower maximum pause times (a factor of 4 lower on average) on the SPEC JVM Benchmarks.

To our knowledge, Azatchi and Petrank implement the only other generational reference counting algorithm [6]. (Our work is independent and concurrent with theirs [13].) Their concurrent non-moving collector uses free-lists in both generations. It thus must store every nursery object on a list, and then enumerate them all, rather than just the live ones. It achieves excellent pause times, but consequently has high space and time overheads which significantly degrade total performance [6, 23]. Our contributions are the generalization of deferred reference counting (*ulterior reference counting*), a generational copying reference counting hybrid that combines high throughput and low pause times, and experimental evaluation of the effectiveness of the collector and its components.

2. Background

This section presents the prior work on which this paper builds. We first discuss generational collection and then reference counting (RC). Section 6 discusses related work in concurrent, incremental, and generational reference counting with a mark-sweep nursery.

2.1 Generational Collection

This section discusses the motivation for generational collection, the copying and mark-sweep algorithms, and defines three types of copying nursery organizations.

Generational algorithms exploit the low rate of object survival for new *nursery* objects using tracing [21, 30]. *Tracing* identifies dead objects indirectly—by tracing the live objects and excluding those that it did not trace. The two broad approaches to tracing are copying and mark-sweep. A copying collector copies all live objects into another space. Its cost is thus proportional to the number of live objects and pointers to them, and works well when few objects survive. A mark-sweep collector marks live objects, identifies all unmarked objects, and frees them. Typically it will perform the sweep lazily during allocation, so its cost is proportional to the number of live objects and pointers to them, and the number of allocations. Copying collectors use monotonic (*bump-pointer*) allocation, and mark-sweep collectors use *free-list* allocation. Bump-pointer allocation is faster, but requires copying collection which incurs a space penalty. It must hold space in reserve for copying. Free-list allocation is slower, but needs no copy reserve. Free-list allocation without compaction can lead to poor memory utilization by scattering live objects sparsely through memory and increasing page residency.

Because of the high mortality of nursery objects [30], generational copying collectors copy nursery survivors to an older generation [3, 5, 12, 21]. Generational organizations repeatedly collect the nursery, and only collect the older generation when it is full. A *flexible-sized* nursery consumes all the usable heap space, and is reduced by the size of the survivors at each collection until the heap is full [3]. Since the nursery can grow to the size of the heap, nursery collection time is highly variable. A *fixed-size* nursery always fills the nursery, but never lets it grow bigger or smaller than a set size. It thus reduces the variability of the time to collect the nursery. A *bounded* nursery uses both an upper and lower bound. When the heap is almost full, the collector reduces the nursery using the variable-nursery policy until the nursery size reaches the lower bound. This strategy reduces the number of full heap collections compared to a fixed-size nursery, but has the same upper-bound on nursery collection time. Our experiments show that a bounded nursery has higher throughput than a fixed-size nursery, but not as good as a variable nursery. We use a bounded nursery for all the generational collectors in this paper since we target both throughput and pause times.

Generational collectors perform well because they are incremental and concentrate on the nursery which tends to have the highest object mortality. A copying nursery exploits low survival rates and ignores the frequent mutations of new objects. In current generational collectors, the old space is either a copied space (classic generational), or a mark-sweep collected space.

2.2 Reference Counting

To lay out the RC design space, this section overviews the existing mechanisms and optimizations from the literature. We describe methods for implementing deferral, buffering, and coalescing of reference counts. Previous work does not identify these choices as orthogonal. To clarify the subsequent ulterior reference counting discussion and summarize previous work, we follow with formal definitions of the reference counting actions.

Reference counting garbage collectors track the number of pointers to each object by continuously monitoring mutations [18, 19]. Each time a mutation overwrites a reference to p_{before} with a reference to p_{after}, the collector increments the reference count for p_{after}, and decrements p_{before}. If an object's reference count becomes zero, the collector reclaims it. The collection work is proportional to the number of object mutations and dead objects. Reference counting is attractive because the work of garbage detection is spread out over every mutation, and is thus very incremental. However since one mutation can cause many objects to become unreachable, its incrementality suffers unless the collector bounds the number of objects it collects by buffering some of the processing for the next collection [9, 19], or performs collection concurrently.

Reference counting has two disadvantages. Since counts never go to zero in a dead cycle, reference counting is not *complete* and an additional algorithm must reclaim cycles. (Section 3.2.4 describes the solution we use for this problem.) Furthermore, tracking every pointer mutation is expensive and seriously degrades mutator performance.

2.2.1 Mechanisms

Deferral. Deutsch and Bobrow introduce *deferred* reference counting which only examines certain heavily mutated pointers periodically, such as register and stack variables [19]. All deferred RC has a *deferral phase* in which reference counts are not correct, and an *RC phase* in which they are. The deferral phase typically corresponds to a *mutation* phase, i.e., program execution. Deferred RC trades incrementality for efficiency; it finds garbage later by ignoring intermediate updates.

Two approaches to deferring stacks and registers are: the zero count table [19] and temporary increments [7, 9]. In the deferral phase of a zero count table (ZCT), the collector applies reference counts for undeferred pointers and records any whose count goes to zero in the ZCT. In the RC phase, the collector scans the registers and stack. It then frees any object recorded in the ZCT that is not reachable from the registers and stack. In the temporary increment approach, the collector applies increments at the beginning of the RC phase for every stack and register before examining object reference counts. At the conclusion of the RC phase, it then inserts a corresponding decrement for each of these increments which it applies at the beginning of the next RC phase.

Buffering. RC algorithms need not perform RC increments and decrements immediately, but can buffer and process them later. Buffering only effects *when* an increment or decrement is applied, not *whether* it is applied. Because deferral introduces periodicity, the RC phase is a natural point to apply buffered counts. Deutsch and Bobrow suggest placing all reference counts in a single buffer and then processing the buffer in the RC phase [19]. Bacon et al. place increments and decrements in separate buffers [7, 9]. Their approach avoids race conditions in a concurrent setting. The RC phase applies all increments (buffered and temporary) before any decrements, thereby ensuring that no live object ever has a reference count of zero. This mechanism replaces the ZCT.

Coalescing. Levanoni and Petrank [23] observe that the periodicity of deferred RC implies that only the initial o_i and final o_f values of pointer fields of heap objects are relevant; the collector can safely ignore intermediate mutations $o_{i'} \dots o_{i''}$. We call this optimization *coalescing*. Levanoni and Petrank describe coalescing with respect to remembering mutated pointer fields (*slots*) [23], but the implementation remembers mutated objects [6, 25].

In their implementation, coalescing uses the differences between before and after images of mutated objects and uses the differences to generate increments and decrements. It records pointer fields of each mutated object just prior to its first mutation, and then at collection time, compares values with the current state, introducing a decrement and increment for each changed fields. For large but sparsely mutated objects, this mechanism imposes a large space overhead for which they propose a mechanism similar to card marking to remember regions of memory rather than objects.

2.2.2 RC Formal Definitions

The following three definitions provide a concise formalism for RC actions.

Mutation event. A mutation event $RCM(p)$ generates an increment and decrement for the before and after values of a mutated pointer, i.e., $RC(p_{before})$--, $RC(p_{after})$++. An $RCM(p)$ may be buffered or performed immediately. An $RCM(p)$ series to the same pointer p may be coalesced, yielding a single $RCM(p)$ for the initial p_i and final p_f values.

Retain event. A retain event $RCR(p)$ for p_o temporarily retains o regardless of its reference count. $RCR(p)$ can be implemented through a zero count table (ZCT), or by generating a *temporary* increment for o.

Deferral. A pointer p is *deferred* if no mutation event to p generates an $RCM(p)$. The correctness of reference counting is only guaranteed if for all deferred pointers p_o, the collector issues a retain event $RCR(p)$ preserving o.

Ulterior reference counting is independent of particulars of the reference counting mechanisms for deferral, buffering, and coalescing, and is compatible with both Bacon et al. and Levanoni and Petrank's implementations. For our stop-the-world garbage collector implementation, we use Bacon et al.'s increment and decrement buffers, coalesce with a new object remembering mechanism that performs decrements for original mutated field targets and increments for their final targets, and retain deferred stack and registers using temporary increments and corresponding decrements.

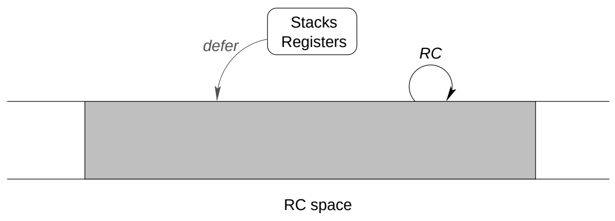
3. Ulterior Reference Counting

This section describes the two primary contributions of the paper: 1) a generalization of deferred reference counting for heap objects, which we call ulterior reference counting (URC), and 2) a concrete instance of a generational URC collector that achieves both high throughput and low pause times.

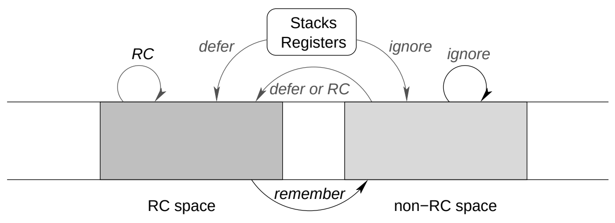
3.1 Generalizing Deferral

This section separates and defines deferral and collection policies in a URC heap. It introduces three URC deferral mechanisms for heap pointers. We define a new RC event, called an *integrate event*, that transitions heap pointers from deferred to non-deferred. We begin with an abstract example URC heap organization, illustrated in Figure 1, to clarify the basic design choices.

Figure 1(a) shows an abstract view of deferred reference counting in which all stacks and registers are deferred, and the heap objects are not. Figure 1(b) illustrates a simple URC configurations



(a) Classic deferred reference counting



(b) A simple instance of URC

Figure 1: Deferred and Ulterior Reference Counting

that generalizes over basic deferral by splitting the heap into RC and non-RC spaces. In this example, the collection policy for the non-RC space is unspecified. The collector can defer or not the pointers from non-RC to RC space, and it must remember the pointers in the other direction (RC to non-RC space) for correctness.

As with many garbage collected systems, a URC collector has mutator phase and a collection phase. The URC mutator phase includes a URC deferral phase in which mutations to non-deferred pointers cause mutation events and mutations to deferred pointers do not. The URC collection phase includes a URC phase and other collection activities. These definitions mirror the ones for the deferral and RC phases in Section 2 for a classic whole heap RC system.

The figure does not depict an important URC feature in which the collector may transition an object from deferred to non-deferred. For example, our generational URC collector moves surviving nursery objects into the RC space and triggers the appropriate future mutation events. An *integrate event* performs this transition during collection as follows.

Integrate event. An RC integrate event $RCI(p)$ changes a deferred pointer p_o to *not-deferred* by generating an increment for o, and for all future mutation events to p, the collector generates a mutation event $RCM(p)$.

There also may be circumstances in which the collector wants to transition a highly mutated object in RC space to non-RC space. In this case, the collector signals to the deferral phase to stop issuing mutation events. In either case, the collector can move the object or not. This choice dictates how to identify RC and non-RC objects and pointer fields. Physically partitioned spaces, as depicted in Figure 1, can use addresses, whereas other organizations require object tags.

3.1.1 Deferral Policies

A deferral policy determines for *each pointer* whether or not to perform mutation events. The collection policy determines for *each object* which collection algorithm is responsible for collecting it. In practice only synergistic choices will attain good performance, but in principle they are distinct.

The URC deferral phase ignores pointer updates to certain pointer fields within heap objects. The design challenge is to enumerate the deferral set efficiently during the collection phase. We present three approaches to this enumeration (1) trace all deferred fields, (2) maintain a list of deferred mutated pointer *fields*, and (3) maintain a list of *objects* containing one or more deferred mutated pointer fields.

Trace Deferred Fields. URC may piggy back *retain* or *integrate* events onto the enumeration of live objects by a tracing collector, essentially for free. Our URC generational implementation uses this mechanism for a copying nursery, promoting survivors into the RC space, and issuing *integrate* events for each promoted pointer. Alternate organizations could issue *retain* events for survivors and continue to defer these pointers. These organizations require that the collector trace all live deferred pointer fields just prior to every reference counting phase.

Record Mutated Deferred Fields. The URC deferral phase can record all deferred pointer fields that point to a reference counted object. Similar to stacks and registers, the collector then enumerates each field, issuing a *retain* event for each.

Record Mutated Deferred Objects. The URC deferral phase can record the object containing a mutated field instead of the field. The collector then enumerates each field in a remembered object and issues *retain* events for pointers to reference counted objects.

Field and object remembering make different trade-offs. If updates to a given object's fields are sparse and/or the object is large, field remembering is more efficient. Both mechanisms can prevent duplicates, and thus absorb multiple updates to the same field. Keeping track of remembered objects takes one bit per object, and is thus more space efficient than tracking fields.

3.1.2 Collection Policies

The collection policy choses which algorithms to apply to which *collection set*. Collection sets may be defined as spatial regions of the heap (as illustrated in Figure 1) or as logical state associated with each object. Section 2 enumerates the policy choices for the RC components. The key factor in the choice of non-RC algorithm is the extent to which it will accommodate efficient deferral. A copying nursery collector easily accommodates the enumeration of surviving objects and the piggy backing of integrate events. With multi-generational collectors or Beltway [12], an ulterior reference counting algorithm on the oldest generation could defer mutations within all lower generations with a mixture of tracing and object/field recording.

3.2 A Generational RC Hybrid Collector

In this section, we describe a specific instance of an ulterior reference counting collector that achieves high throughput and low pause times. We call this collector BG-RC because it is Generational with a Bounded copying nursery and uses Reference Counting in the mature space. (Section 2.1 defines bounded.) The mechanics and implementation of this collector work for any copying nursery (fixed-size, variable, or bounded), but for simplicity we limit the discussion to BG-RC.

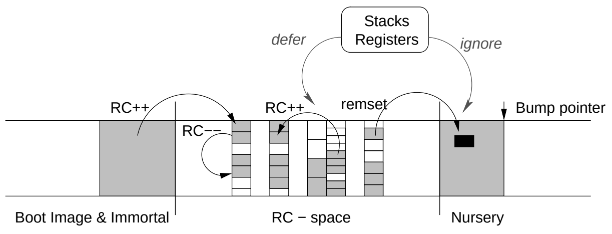
BG-RC divides young and old objects and is thus generational. It is a hybrid because it uses different policies for the generations. In the nursery, it uses bump-pointer allocation and copying collection. Copying performs well on nursery objects, since most are short-lived. A bounded nursery size controls the worst case nursery collection time and the copy reserve space overhead. In the old generation, BG-RC uses a free-list allocator with a reference counting collector that has 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. BG-RC defers mutations to the nursery, stacks and registers.

We now present our collector organization, mechanics, and write barrier in more detail. To control worst case pause times, we present a number of techniques based on work estimates to limit the time for any particular collection.

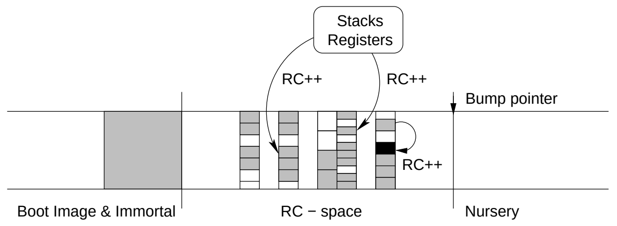
3.2.1 Organization and Mechanics

BG-RC has two generations: the young nursery space and the mature RC space. During mutation, a write barrier a) generates RC mutation events, and b) records pointers into the nursery from other portions of the heap. For mutation events, a *mutated object buffer* records each mutated non-nursery objects. The write barrier is described in more detail below. The mutator allocates into the bounded nursery using a bump pointer and copies nursery survivors into the mature RC space. The mature space allocator uses a free list segregated by size class [22, 31]. (See Section 4.3 for more details.)

A collection is triggered whenever the nursery is full. The nursery phase includes scanning roots, processing the modified object



(a) During mutator activity



(b) During GC

Figure 2: The Organization of a Hybrid Generational URC Collector

buffer, tracing the nursery, and copying and integrating nursery survivors into the RC space. For each object in the modified object buffer, the nursery phase enumerates the objects to which it points. It generates an increment for each referent object, and marks as live referent nursery objects. The nursery scan traces from these objects and the roots, finding and copying all live nursery objects using breadth-first order [16].

When the collector encounters an un-forwarded (not yet copied) nursery object, it copies the object to the mature space and enqueues it for scanning. As it scans each surviving object o, it performs an *integrate event* for each of its pointer fields o_f, generating an increment for the referent. Each time BG-RC encounters an old or young live object directly from the root set, it *retains* the object by generating an increment, which it offsets by a decrement during the next collection, making the increment temporary. When the nursery collection is complete, all objects reside in the mature space. In the RC phase, BG-RC then applies all of the buffered increments, followed by the decrements. It frees objects with reference count zero, and recursively decrements the counts of their children. During the decrement phase, BG-RC identifies decrements whose target did not go to zero as cyclic garbage candidates and periodically collects them using a variant of Bacon et al.'s trial deletion algorithm [7, 9], which we summarize in Section 3.2.4.

We arrange virtual memory such that the nursery is in high memory, the mature space in middle memory, and the Jikes RVM boot image and immortal space in low memory. Figure 2(a) illustrates our heap organization during mutator activity. Notice the counts in the mature space which are buffered, and that there are no reference counting operations on the stack or registers or the nursery during mutation. The boot image and immortal space are not deferred, so BG-RC logs mutated objects in these spaces. It could defer these objects with the mechanisms described in Section 3.1.1. (See Section 4.3 for more discussion of the boot image and immortal space.)

Figure 2(b) shows the heap just after a nursery collection when all objects are in the mature space and all reference counts in or into the old space are buffered. For example, Figure 2(a), shows a pointer from the top right old space object to a young object. This pointer causes the collector to promote and integrate the object into the fourth slot in the right most mature space block, as shown in Figure 2(b). The integrate event includes the RC increment. The RC phase of the collector may now commence and correctly frees any object in RC space with a zero reference count.

3.2.2 Write Barrier

The *write-barrier* remembers pointers into the nursery from the non-nursery spaces (RC, immortal and boot image spaces) and generates *mutation events* (*RCM(p)*) for mutations to pointer fields within the non-nursery spaces. Our barrier is an object remembering coalescing barrier.

Each time the program mutates a pointer field within an object that is not yet *logged*, the barrier logs it. Logging records the address of the mutated object in a modified object buffer and buffers a *decrement* for each field with a referent object in the RC space. At collection time, the collector processes the modified object buffer and for each object, generates an *increment* for each referent object in the RC space.

Because the barrier generates a decrement for the initial values and the collection phase generates an increment for the final values, this mechanism achieves coalescing. If, while processing a modified object, the collector finds that a final value points into the nursery, it retains the referent object. This logging mechanism performs a fixed amount work that is proportional the number of fields in the mutated object. It performs unnecessary work for sparsely modified objects. In contrast, Levanoni and Petrank perform coalescing by using a snapshot of the initial value of the object during mutation and at GC time compare the initial and final values of each field, only generating a mutation event if the values differ [23]. Their approach can generate fewer mutation events, but requires a copy of the unmutated object.

Figure 3 shows the Java code for our barrier that is correct for concurrent mutators. (The collector itself is not concurrent.) By ensuring that nursery objects are initialized so they appear to have been logged at birth, we avoid logging these heavily mutated objects. We use zero to denote the LOGGED state to avoid explicitly initializing nursery objects.

In Figure 3, the *fast path* (lines 1–8) simply performs the store (line 7) if the source object has already been logged. Since most

```
 1 private void writeBarrier(VM_Address srcObj,
 2                           VM_Address srcSlot,
 3                           VM_Address tgtObj)
 4   throws VM_PragmaInline {
 5   if (getLogState(srcObj) != LOGGED)
 6     writeBarrierSlow(srcObj);
 7   VM_Magic.setMemoryAddress(srcSlot, tgtObj);
 8 }
 9
10 private void writeBarrierSlow(VM_Address srcObj)
11   throws VM_PragmaNoInline {
12   if (attemptToLog(srcObj)) {
13     modifiedBuffer.push(srcObj);
14     enumeratePointersToDecBuffer(srcObj);
15     setLogState(srcObj, LOGGED);
16   }
17 }
18
19 private boolean attemptToLog(VM_Address object)
20   throws VM_PragmaInline {
21   int oldState;
22   do {
23     oldState = prepare(object);
24     if (oldState == LOGGED) return false;
25   } while (!attempt(object, oldState, BEING_LOGGED));
26   return true;
27 }
```

Figure 3: BG-RC Write Barrier

mutated objects are mutated many times, the *slow path* is infrequently taken. For this reason, we inline the fast path and force the slow path out of line [14].

The test on line 5 of the fast path performs an unsynchronized check of two bits in the object header to see if the state is not `LOGGED`. Line 12 of the slow path eliminates the potential race generated by the unsynchronized check. This approach avoids expensive synchronization in the fast path while maintaining correctness. The code in `attemptToLog()` (lines 19–27) returns true if it successfully changes the object's state from `UNLOGGED` to `BEING_LOGGED`, otherwise it returns false. It spins while the state is `BEING_LOGGED`, and proceeds only once the object's state is `LOGGED`. We use a conditional store implemented with Jikes RVM's prepare and attempt idiom (lines 23 and 25). If the barrier is successful in attempting to log, it performs the logging operations (lines 13 and 14) before setting the log state to `LOGGED`.

3.2.3 Controlling Pause Times

Nursery collection and reference counting times combine to determine pause times. Nursery collection time is proportional to the number of surviving nursery objects and pointers in to the nursery. In the worst and pathological case, the pause could include all nursery objects surviving. A large nursery size can thus degrade pause times. The nursery size should not be too small either since this will lead to more frequent collections, which diminishes the effect of coalescing in RC and gives nursery objects less time to die. If too many objects survive, it will place unnecessary load on the old generation. We measure these effects and see both in our programs in Section 5.4.

A program may mutate heavily and thus generate large amounts of RC work while performing little allocation. In order to bound the accumulation of RC work between collections, we limit the growth of *meta data*. The meta data includes the modified object buffer and the decrement buffer, which grow each time an object is logged. However, a write barrier is not typically a GC safe point, and thus is not an opportunity to trigger collection. We extended Jikes RVM with an asynchronous GC trigger and set it when a meta data allocation at a non-GC-safe point fills the heap or exceeds the meta data limit. The next scheduling time slice then triggers a collection. We found that the collectors were very robust to this trigger although the pure RC needs a much more generous allowance (4MB) than BG-RC (512K) because it generates a lot more meta data since it does not defer the nursery.

When a reference counted object is freed, the collector decrements the counts of each of its descendants, and if their count drops to zero it frees them. In the worst case, the whole heap could die at once. The trial deletion cycle detection algorithm we use performs a similar transitive closure which could be considerable in the worst case even without a cycle. To limit the time spent performing this work, a parameter specifies a *time cap* which curtails the recursive decrement or trial deletion when the time cap is reached. We found that if the time cap is too low, some programs may be unable to reclaim cyclic garbage. We experiment with these triggers in Section 5.4.

3.2.4 Cycle Detection

The current BG-RC implementation uses a variant of the synchronous trial deletion algorithm from Bacon and Rajan [9]. On every collection, their algorithm creates a candidate set of potential cycle roots from all the decrements which do not go to zero. It colors these objects *purple* and puts them on a list. At the end of a RC phase, it examines elements of this list. If a purple object still has a non-zero reference count, it computes a transitive closure coloring the object and objects reachable from it gray and decrements their reference counts. At the end, all of the gray roots with reference count zero are cyclic garbage. It then recursively frees these objects and their children with zero reference counts. For non-garbage objects, it restores the reference counts and color.

In our RC implementation, we do not perform cycle detection at the end of every RC phase. We instead add a *cycle detection trigger* to decide when to perform cycle detection. As the available heap space falls toward a user-defined limit, we perform cycle detection with increasing probability. We use a limit of 512KB, performing cycle detection with probability 0.125 when available heap space falls below 4MB, 0.25 at 2MB, 0.5 at 1MB, and 1.0 at 512KB. Alternative implementations could express this limit as a fraction of the heap size.

4. Methodology

This section first briefly describes Jikes RVM and JMTk which are publicly available[2] and include BG-RC and all other collectors evaluated in this paper. We then overview the additional collectors, how they work, and a few implementation details. A more thorough explanation of the implementation is available elsewhere [11]. As we point out in the previous section, all of these collectors share a common infrastructure, and reuse shared components. We then present the characteristics of the machine on which we do all experiments, and some features of our benchmarks.

4.1 Jikes RVM and JMTk

We use Jikes RVM (formerly known as Jalapeño) for our experiments with a new memory management tool kit JMTk. Jikes RVM is a high performance VM written in Java with an aggressive optimizing compiler [1, 2]. We use the *Fast* build-time configuration which precompiles key libraries and the optimizing compiler, and turn off assertion checking in the VM. We use the adaptive compiler which uses a baseline compiler and based on samples, recompiles hot methods with an aggressive optimizing compiler [4]. It places the most realistic load on the system, but suffers from variability. A consequence of Jikes RVM's Java in Java design is that the VM's optimizing compiler uses the allocator and write-barrier specified by the collector, and thus the collector changes the VM.

Together with Perry Cheng at IBM Watson, we recently developed a new composable memory management framework (JMTk) for exploring efficient garbage collection and memory management algorithms [11]. JMTk separates allocation and collection policies, then mixes and matches them. It shares mechanisms such as write barriers, pointer enumeration, sequential store buffers for remembered sets, and RC increments between algorithms. The heap includes all dynamically allocated objects, inclusive of the program, compiler, and itself (e.g., the collector meta data such as remembered sets). It contains efficient implementations of all the collectors we study [11], and hybrids share the non-hybrid components. Because of the shared mechanisms and code base, our experiments truly compare policies.

4.2 Collectors

We compare BG-RC to several other collectors. We first summarize and categorize each collector, then justify our selection, and discuss each in more detail.

RC: The coalescing, deferred reference-counting collector uses one policy on the whole heap: a free-list allocator and a collector that periodically processes the modified object buffer and

[2] `http://www.ibm.com/developerworks/oss/jikesrvm/`.

coalesced increment and decrement buffers, deleting objects with a reference count of zero.

BG-RC: The generational reference counting hybrid uses a bounded copying nursery and promotion into a RC mature space, as described in the previous section.

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 lazily reclaims unmarked objects.

BG-MS: The generational mark-sweep hybrid has a bounded copying nursery, and promotes survivors to a MS mature space.

We categorize these collectors as follows. The *generational* collectors divide the heap into a nursery and old generation and collect each independently (BG-MS and BG-RC). The *whole heap* collectors (MS and RC) scavenge the entire heap on every collection with one policy. The *hybrid* collectors use multiple policies. BG-RC and BG-MS use the same copying nursery. All the collectors use the same free-list for either the mature space (BG-RC and BG-MS) or the entire heap (MS and RC) with either reference-counting or mark-sweep collection. Section 4.3 describes the segregated fit free-list implementation in detail.

We choose BG-MS as our high performance comparison point because in our experiments, it is comparable to pure copying generational and better than whole heap collectors [11]. This result is consistent with use of similar collectors in a variety of high performance JVMs. We include the full heap MS collector to reveal the uniform benefits of generational collection. We fix the heap size in our implementation to ensure fair comparisons. For each of the comparison collectors, we now describe their collection mechanisms, collection trigger, write barrier, and space and time overheads.

4.2.1 RC: Reference Counting

The pure deferred reference-counting collector organizes the entire heap with the free-list allocator. The RC algorithm uses all of the same mechanisms as BG-RC. It however only defers counting the registers, stacks, and class variables (statics). The write barrier generates mutation events for mutations to heap objects using the logging mechanism. Our experiments trigger a collection after each 1MB of allocation, or due to a pause time trigger. Collection time is proportional to the number of dead objects, but the mutator load is significantly higher than the generational collectors since it logs all mutated objects.

4.2.2 MS: Mark Sweep

The mark-sweep collector organizes the entire heap with a segregated free list. MS triggers collection when the heap is full. For each block, MS keeps a *mark* bit map. During a collection, it scans the reachable objects, and marks them as reachable by setting a corresponding bit in its *mark* bit map. If any block contains no marked objects, the collector frees the whole block. It sweeps *lazily*. The first time the allocator uses a block after a collection, it uses the mark bit map to construct a new free list for the block (unmarked objects are free).

Tracing is proportional to the number of live objects, and since it is performed lazily, reclamation is proportional to the number of allocations. The space requirements include the live objects, bit maps, 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). Some MS implementations occasionally perform mark-sweep-compaction to limit external fragmentation, but we do not explore that option here. Since MS is a whole heap collector, its maximum pause time is poor and its performance suffers from repeatedly tracing objects that survive many collections.

4.2.3 BG-MS: Generational Copying/Mark-Sweep

This hybrid generational collector uses a bounded copying nursery and the above mark-sweep policy for the older generation. BG-MS allocates using a bump pointer and when the nursery fills up, it triggers a nursery collection. It scans and copies all of the reachable nursery objects into the mature space. Classic copying semi-space collectors divide the heap into two equal size parts: *to-space* and *from-space* [16]. They allocate into to-space, and copy into from-space. From and to-space are equal in size since all objects could survive. For the same reason, BG-MS reserves a copy space of free blocks at least equal to the nursery size in the MS space. When this size drops below the nursery upper bound, the collector reduces the nursery size as does BG-RC. BG-MS triggers a full heap collection when the bounded nursery drops below its lower bound (256KB) or when the application explicitly requests a GC (through `System.gc()`). The write barrier only remembers pointers from the mature space to the nursery. By exploiting the generational hypothesis, BG-MS mitigates the drawbacks of MS for throughput and average pause times, but occasional full heap collections drive up maximum pause times.

4.3 Implementation Details

This section includes a few key implementation details about the free-list allocator, the large-object space, object headers, the bounded nursery, inlining write barriers and allocation sequences, the boot image, and immortal space.

All of the collectors use a segregated fit free-list allocator [22, 31] either on the whole heap or on the mature space. The heap is divided in to *blocks* of contiguous memory. Each block has a free list and contains only one size class. The allocator assigns a new block to a size class when no block contains a free object of the right size. It changes the size class of a block only when the block is completely free. We use a range of size classes similar to but smaller than the Lea allocator [22]. We select 40 size classes with the goal of worst case internal fragmentation of 1/8. The size classes are 4 bytes apart from 8 to 63, 8 bytes apart from 64 to 127, 16 bytes apart from 128 to 255, 32 bytes apart from 256 to 511, 256 bytes apart from 512 to 2047, and 1024 bytes apart from 2048 to 8192. A word is 4 bytes, so small, word-aligned objects get an exact fit. Blocks contain a single size class and range in size from 512 bytes for the smallest size class to 32KB for the largest.

All objects 8KB or larger are separately allocated into a large object space (LOS) using an integral number of pages. The generational collectors allocate large objects directly into this space. During full heap collections, MS and BG-MS scan and collect the large objects. RC and BG-RC reference count the large objects at each collection.

The standard Jikes RVM object header size is 2 words, with an additional word (4 bytes) for reference counting. Thus RC and BG-RC need additional space in the RC free-list. As an optimization, BG-RC eliminates the header in the nursery, and adds it only when performing integration of nursery objects into the RC space.

As our default BG-RC and BG-MS configurations, we use a nursery upper bound of 4MB, and a lower bound of 256KB. For RC, we trigger a collection every 1MB, or when one of the collection triggers applies. For BG-RC, we use a time cap of 60ms, a meta-data limit of 512KB, and a cycle detection limit of 512KB. JMTk measures free space in completely free pages. In BG-MS and BG-RC, the bounded nursery size starts at its upper bound. At

	Allocation			Write barrier		Modifi ed objects			RC increments			RC decrements		
benchmark	alloc	MS min	alloc:MS	total	rem set	RC	BG-RC	%	RC	BG-RC	%	RC	BG-RC	%
_202_jess	403MB	16MB	25:1	28.63	0.16%	8.58	0.01	0.09%	25.68	0.23	0.91%	25.07	0.40	1.59%
_213_javac	593MB	26MB	23:1	20.78	2.41%	8.96	0.71	7.97%	13.52	2.26	16.74%	10.25	4.32	42.12%
_228_jack	307MB	14MB	22:1	10.44	7.23%	8.75	0.01	0.13%	5.21	0.09	1.67%	4.84	0.29	6.01%
_205_raytrace	215MB	18MB	12:1	7.35	0.98%	7.04	0.03	0.47%	3.10	0.11	3.61%	2.09	0.16	7.78%
_227_mtrt	224MB	21MB	11:1	8.49	1.00%	7.32	0.04	0.59%	3.43	0.15	4.41%	2.22	0.22	9.78%
_201_compress	138MB	17MB	8:1	1.53	0.71%	0.25	0.00	0.49%	0.41	0.01	2.67%	0.21	0.01	5.77%
pseudojbb	339MB	46MB	7:1	23.31	3.66%	8.45	0.24	2.80%	11.12	0.98	8.81%	9.40	3.36	35.72%
_209_db	119MB	20MB	6:1	35.03	0.52%	3.85	0.00	0.06%	10.92	5.50	50.33%	10.11	5.52	54.58%
_222_mpegaudio	51MB	12MB	4:1	9.79	0.23%	0.89	0.01	0.89%	1.56	0.03	2.00%	0.73	0.04	5.15%
mean	**265MB**	**21MB**	**13:1**	**16.15**	**1.74%**	**6.01**	**0.12**	**1.96%**	**8.33**	**1.04**	**12.50%**	**7.21**	**1.59**	**22.05%**
geometric mean	**216MB**	**20MB**	**11:1**	**11.89**	**0.98%**	**3.98**	**0.02**	**0.50%**	**4.86**	**0.22**	**4.62%**	**3.53**	**0.38**	**10.66%**

Table 2: Benchmark Allocation Characteristics and Write Barrier Events (in millions)

the end of each collection, it sets aside half of the free space as copy reserve and the other half is available to the nursery. The collectors set the nursery size to be the smaller of the nursery bound (4MB) and the free mature space. At the start of each collection, BG-MS estimates nursery survival using a conservative survival estimate (80%). If the collection may cause the next nursery size to fall below the lower bound (256KB), BG-MS performs a full heap collection.

For all of the generational collectors, we inline the write-barrier fast path which filters out mutations to nursery objects and thus does not record most pointer updates [14]. In the reference counting collectors, the slow path generates decrements and modified object buffer entries. In BG-MS, the slow path inserts remembered set entries. The pure mark-sweep collector has no write barrier. We inline the allocation fast path for all collectors.

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. Similarly, JMTk has an immortal space that the VM uses to allocate certain immortal objects and which must not be moved. None of the JMTk collectors *collect* the boot image or immortal objects. MS and BG-MS trace through the boot image and immortal objects during full heap collection. RC and BG-RC just assume the all boot images are live which is essentially true. (The source code for Bacon et al.'s RC implementations reveals the same limitation [7]).

4.4 Experimental Platform

We perform all of our experiments on 2 GHz Intel Xeon, with 16KB L1 data cache, a 16K L1 instruction cache, a 512KB unified L2 on-chip cache, and 1GB of memory running Linux 2.4.20. We run each benchmark at a particular parameter setting six times and use the second fastest of these. The variation between runs is low, and we believe this number is the least likely disturbed by other system factors and the natural variability of the adaptive compiler.

4.5 Benchmarks

Table 2 shows key characteristics of each of our benchmarks. We use the eight SPEC JVM benchmarks, and pseudojbb, a slightly modified variant of SPEC JBB2000 [27, 28]. Rather than running for a fixed time and measuring transaction throughput, pseudojbb executes a fixed number of transactions (70000) to generate a fixed garbage collection load. The SPEC JVM benchmarks are run at the default size of 100.

The *alloc* column in Table 2 indicates the total number of bytes allocated by each benchmark. The next column indicates the minimum heaps in which the benchmarks can run using MS collector. The 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 MS minimum heap size, giving an indication of the garbage collection load for each benchmark. This ratio shows these are reasonable, but not great benchmarks for garbage collection experiments. We order the benchmarks according to this ratio.

The write-barrier columns show for a 4MB nursery: *total*–the number of times the write barrier is invoked, i.e., all instrumented stores; and *rem set*–the fraction of those stores that point from mature space into the nursery. These counts are all expressed in *millions*.

The final nine columns indicate the number of entries made to the modified object, increment, and decrement buffers by RC and BG-RC for each of the benchmarks. Section 3.2.2 explains how the collectors generate these entries. These results show that by excluding nursery objects from the reference counter, we reduce its load dramatically, a factor of 50 with respect to the modified objects on average.

These measurements indicate part of both the time and memory overheads of the different collectors for their meta data which includes the remembered sets and RC buffers. Most programs do not mutate the old objects heavily, and thus reference count fewer than 10% of writes. However, both _209_db and _213_javac write many pointers in the mature space, and thus put a relatively high load on the reference counter in BG-RC.

5. Results

This section compares mark-sweep (MS), reference counting (RC) and their generational hybrids (BG-MS & BG-RC) with respect to throughput and responsiveness. The results show BG-RC has much better responsiveness, and matches and can sometimes beat BG-MS on throughput in moderate heaps. We include a limit study in which we measure the cost of reference counting over nursery collection, and find it is very low. In addition to maximum pause time, we present bounded mutator utilization [12, 17] to reveal whether many short pauses group together to reduce mutator efficiency, and find this problem does not occur for BG-RC. Section 5.3 explores the effect of the heap size on throughput. In moderate and large heaps, BG-RC matches BG-MS. In small heaps, BG-MS performs better. Section 5.4 shows the sensitivity of BG-RC due to variations in the nursery size, time cap, and cycle collection trigger.

Table 3 compares throughput and responsiveness of the four collectors in a moderate heap. The default collection triggers for these results are a nursery of 4MB for BG-RC and a time cap of 60ms and cycle detection limit of 512KB for RC and BG-RC. We use a more frequent trigger of 1MB of allocation for RC to make it more responsive. We relax the time cap on a number of benchmarks for RC so that it will complete some cycle detection and run to completion. These results are printed in italics in Table 3. Column two states the heap size, which is 1.5 $\times$ minimum heap size for all benchmarks except pseudojbb (1.6 $\times$) and _213_javac (2.6 $\times$).

	heap	BG-MS	MS		BG-MS		BG-RC		RC	
	used	time	norm	max	norm	max	norm	max	norm	max
benchmark	MB	sec	time	pause	time	pause	time	pause	time	pause
_202_jess	24	6.2	1.91	182	1.00	181	0.99	44	*2.36*	*131*
_213_javac	68	13.4	1.01	268	1.00	285	1.00	68	*1.78*	*580*
_228_jack	21	7.7	1.52	184	1.00	185	0.94	44	1.66	72
_205_raytrace	27	7.5	1.31	203	1.00	184	1.03	49	*1.71*	*133*
_227_mtrt	32	8.3	1.29	241	1.00	180	1.04	49	*1.75*	*130*
_201_compress	25	11.6	0.98	160	1.00	175	0.88	68	0.93	72
pseudojbb	74	20.0	1.00	264	1.00	281	1.00	53	*1.33*	*297*
_209_db	30	19.2	1.01	238	1.00	244	1.01	59	1.11	43
_222_mpegaudio	18	10.3	1.05	185	1.00	178	0.96	43	*1.14*	*121*
mean	**35**	**11.3**	**1.23**	**214**	**1.00**	**210**	**0.98**	**53**	**1.53**	**175**
geometric mean	**31**	**10.4**	**1.20**	**211**	**1.00**	**206**	**0.98**	**52**	**1.47**	**130**

Table 3: Throughput and Responsiveness of MS, BG-MS, BG-RC, and RC at a Moderate Heap Size

These benchmarks contain very large cycles and require a larger heap to prevent trial deletion in cycle detection from executing too frequently in combination with cycle detection preemption due to the time cap. This artifact is due to our synchronous cycle detection algorithm. Bacon et al. point out the difficulty of collecting cycles in _213_javac as well [7].

Column three in Table 3 contains the total execution time of BG-MS, and columns four, six, eight, and ten normalize to this time. Columns five, seven, nine, and eleven give the maximum pause times. Some of the benchmarks can run to completion at these heap sizes without requiring BG-MS to perform a mature space collection, however the SPEC benchmarks use `System.gc()` to clear the heap at the start and end of each benchmark, at which point the collectors perform a whole heap collection. BG-MS therefore performs at least one full heap collection in each of these benchmarks. If, instead BG-MS performs only a nursery collection for each `System.gc()`, it improves by 2.3% on average, with a peak improvement of 7% (_202_jess) and in the worst case a degradation of 7% (_205_raytrace). The improvement is largely due to the absense of full heap collections, which is unrealistic. BG-RC *continuously* collects the mature space, as do the full heap collectors, RC and MS.

5.1 Throughput

Table 3 shows that BG-RC delivers excellent throughput in a moderate heap, at best 12% faster (_201_compress), at worst 4% slower (_227_mtrt), and on average around 2% faster than BG-MS. The full heap collectors perform worse, and often much worse than their hybrid generational counterparts. BG-MS improves over MS by more than 50% on two of the benchmarks. The generational collectors are designed to exploit the typical space and time behavior of young and old objects. The space and allocation-time advantages of a bump-pointer free-list hybrid benefit both BG-MS and BG-RC. The collection-time advantage of a copying nursery also benefits both hybrids. The fact that older objects are usually mutated much less frequently benefits BG-RC.

A comparison of MS and RC in Table 3 confirms the conventional wisdom that the trade-off between lower collection time and higher mutator overhead inherent in reference counting leads on average to a substantial reduction in throughput. However, RC performs better than MS and BG-MS on _201_compress because it couples a low allocation to live ratio with infrequent mutations to its data as shown in Table 3. Mutator time measurements confirm this explanation on a variety of heap sizes in Section 5.3. On average however, the absence of a concurrent cycle detector often causes RC to be unresponsive. BG-RC dramatically limits its exposure to this trade-off by using reference counting only with the low-maintenance older objects (Table 2), even though it has an additional space overhead of a word for each object.

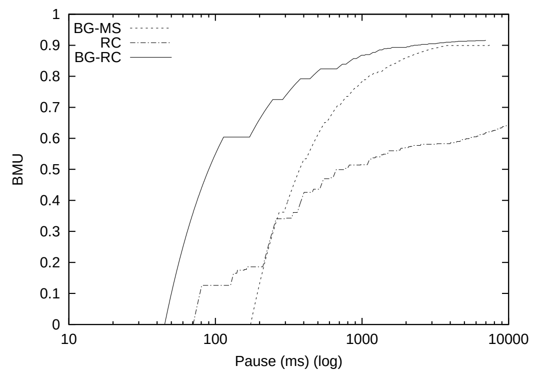
Table 4 illustrates the throughput of the two generational collectors in a large 512MB heap. At this heap size, BG-MS does not scavenge the mature space in any of the benchmarks, and in this experiment, BG-MS only collects the nursery on calls to `System.gc()`. The performance of BG-MS thus *only* includes the cost of nursery collections. By contrast, BG-RC continuously collects the mature space, regardless of heap size. The difference in performance between BG-MS and BG-RC in Table 4 therefore quantifies the additional cost of continuous RC collection over nursery only collection as very modest. The overhead is at worst 8%, and on average only 3%.

5.2 Pause Times

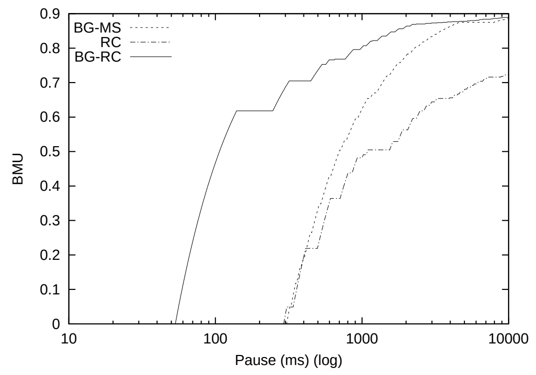
BG-RC has low maximum pause times. In fact, Table 3 shows that it has much better maximum pause times than pure RC because it is exposed to less load. In particular, cycle detection causes problems as evidenced by the _213_javac result. Without cycle detection, RC pause times go down. However, since we trigger RC more frequently than BG-RC (every 1MB of allocation rather than every 4MB), RC's higher pause times also reveal that a copying nursery is a good idea—much better than using RC or MS on the young objects. Since BG-MS performs full heap collections for all of the benchmarks, it attains the expected poor maximum pause time behavior due to the mark-sweep of the full older generation. MS achieves similarly poor max pause times. BG-RC always achieves a much lower maximum pause time than BG-MS, a factor of 4 on average, and at worst 8ms above its 60ms time cap.

	best	BG-MS	BG-RC
	time	norm	norm
benchmark	sec	time	time
_202_jess	5.7	1.00	1.07
_213_javac	12.2	1.00	1.08
_228_jack	7.1	1.00	1.02
_205_raytrace	8.0	1.00	0.97
_227_mtrt	8.1	1.00	1.05
_201_compress	9.9	1.00	1.01
pseudojbb	19.1	1.00	1.05
_209_db	18.6	1.00	1.04
_222_mpegaudio	9.8	1.00	1.00
mean	**10.9**	**1.00**	**1.03**
geometric mean	**10.1**	**1.00**	**1.03**

Table 4: Limit Study of Throughput Using 512MB Heap for BG-MS and BG-RC



(a) _228_jack



(b) pseudojbb

Figure 4: Bounded Mutator Utilization (BMU)

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. To measure this effect, we modify Cheng and Belloch's minimum mutator utilization (MMU) methodology which measures the fraction of time in which the mutator does useful work in a given period, but is not monotonic [17]. We use bounded mutator utilization (BMU) which plots a point (w, m) on a BMU curve if, for all intervals (windows) of length w or more that lie entirely within the program's execution, the mutator utilization is at least m. BMU curves are monotonically increasing, where the x-intercept is the maximum GC pause for the run, and the asymptotic y-value is the overall throughput (fraction of time spent in the mutator). The BMU curve then identifies the maximum period that the application's mutator fraction requirement will not be satisfied.

For example, in Figure 4(a), if _228_jack requires at least 20% of the CPU at all times (BMU $= 0.2$), it will experience a pause of around 50msec from BG-RC, around 190msec from BG-MS, and about 75msec from RC. These BMU graphs illustrate that BG-RC performs very well, both in terms of responsiveness, where it exceeds RC and BG-MS, and in terms of throughput, where it matches BG-MS at the y-intercept. The BMU graphs for the other benchmarks are very similar. RC would undoubtedly be more responsive if it were a concurrent implementation, and its throughput would improve with extra CPU resources dedicated to the task of RC collection (which is common practice in RC implementations).

5.3 Sensitivity to Heap Sizes

Figures 5, 6, and 7 show the impact of heap size on GC time, mutator time, and total time for all four collectors. As the total heap size increases, BG-RC and BG-MS both increase their mature space size. BG-RC triggers young and mature space collections due to a full nursery, full heap, pause-time control, or user triggered GC. Whereas BG-MS triggers a nursery collection when the nursery is full, and a full-heap collection when the mature space is full or when the user triggers a GC. Thus as the heap gets bigger, BG-MS does fewer collections of the mature space.

These graphs indicate the trade-offs the respective collectors make between GC and mutator efficiency. Unsurprisingly, the generational collectors spend less time in GC on average. Both also have better average mutator times, which reflects the benefit of allocating with a bump pointer to the nursery which is usually about 70 bytes of IA32 instructions as opposed to allocating to a free list which is 140 bytes of IA32 instructions in our implementation. This difference is similar in other implementations [5, 10, 22]. In addition, copying can have a positive impact on locality by compacting survivors into the mature space. This effect is most dramatically evidenced in _227_mtrt in Figures 6(e) and 7(e), where at about 1.2 $\times$ minimum heap size, BG-MS performs around 25% better than any other collector. This result is repeatable and examination of the collection log shows a single full-heap collection after the main data structure is built must lead to significant locality improvements in the mutator.

As the heap shrinks, each of the collectors must do more work and total throughput degrades until they are unable to satisfy the application's requests. BG-RC tends to degrade more rapidly than BG-MS in very small heaps because the pause-time guidelines prevent it from reclaiming cyclic garbage promptly. _213_javac is most sensitive to this effect, and does not execute to completion until the heap grows to almost 2.6 $\times$ the maximum live size, as explained above.

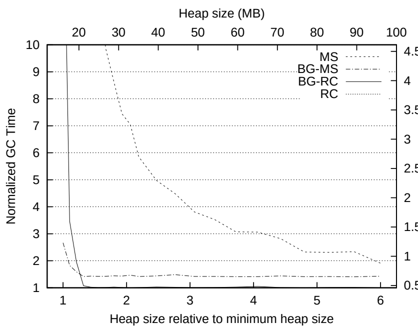
With respect to mutator time, MS performs best and BG-MS often performs somewhat better than BG-RC. Although BG-RC does match BG-MS frequently, and they are usually within 5% of each other. RC mutator time is often worse than the others, and is much worse for benchmarks such as _205_raytrace, _228_jack, and _213_javac. With respect to GC time, BG-RC performs significantly better than BG-MS on large and moderate heaps, except for pseudojbb which performs a large number of mutations. In this case, the low ratio of maximum live size to total allocation means that MS or BG-MS perform few full heap collections.

5.4 Collection Triggers

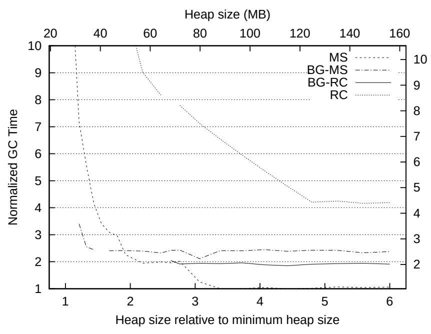
Table 5 varies the collection triggers described in Sections 3.2.3 and 3.2.4 to explore their effects on total and maximum pause time. Our base line configuration uses a nursery size of 4MB, a time cap of 60ms, and a cycle detection trigger of 512KB. We execute the benchmarks in the same heap sizes listed in Table 3.

Generally, smaller nursery sizes lead to lower throughput, which is unsurprising. Reducing the nursery size to 2MB tends to *degrade* responsiveness compared with a 4MB nursery. Because a smaller nursery filters fewer mutations and less garbage, it increases the pressure on the RC mature space. Two of the benchmarks were unable to run to completion with this configuration (_213_javac and pseudojbb). Increasing the nursery size to 8MB also degrades responsiveness compared to 4MB, due to the increase in nursery collection time. For example, the poor maximum pause time for _209_db is due to a high nursery survival rate after _209_db constructs a large long-lived data structure.

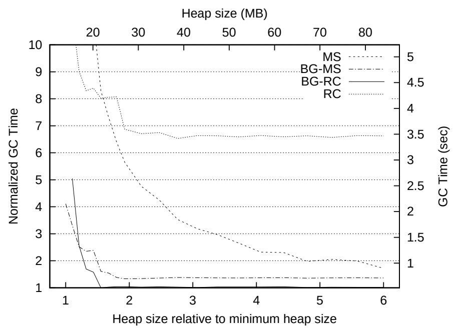
The impact of the time cap is as expected. A small time cap



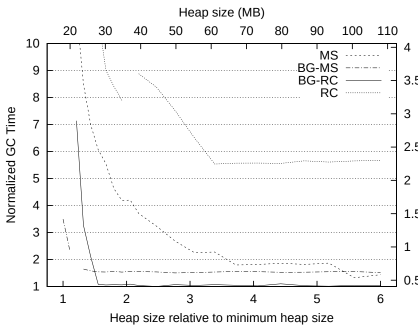
(a) _202_jess



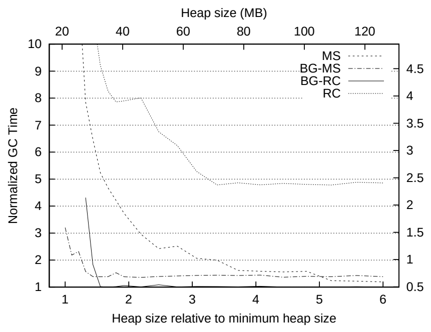
(d) _213_javac



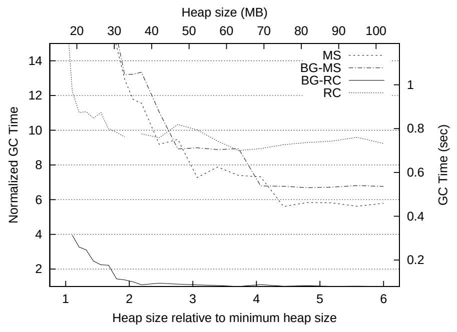
(g) _228_jack



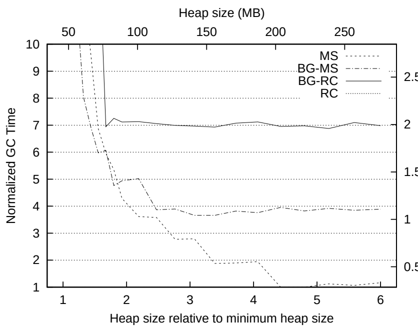
(b) _205_raytrace



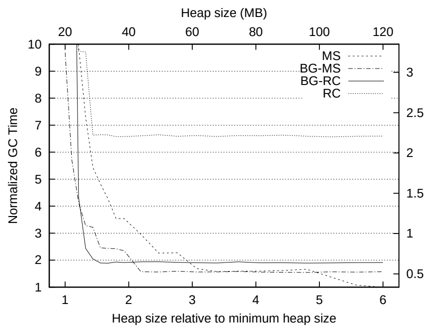
(e) _227_mtrt



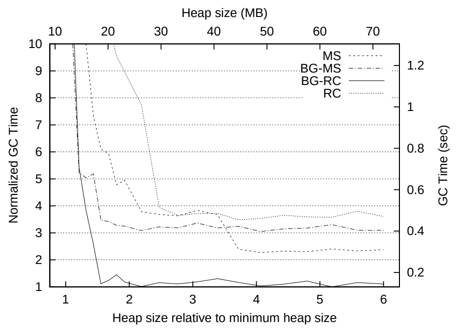
(h) _201_compress



(c) pseudojbb



(f) _209_db



(i) _222_mpegaudio

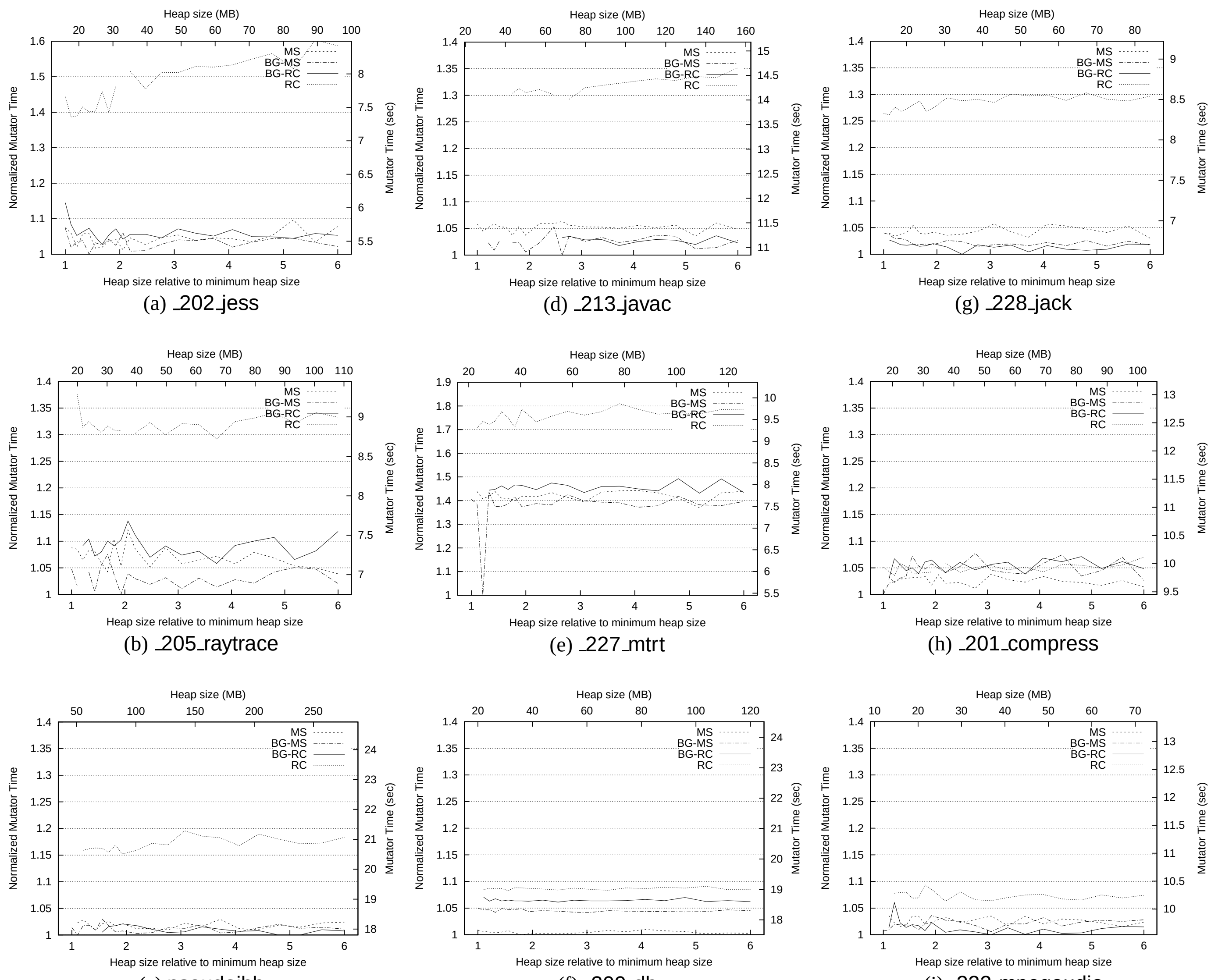
Figure 5: GC Time as a Function of Heap Size

	default		Nursery Size				Time Cap				Cycle Detection Trigger			
	(4,60,512)		2MB		8MB		30ms		120ms		256KB		1MB	
benchmark	time	max	time	max	time	max	time	max	time	max	time	max	time	max
_202_jess	1.00	44	1.05	41	1.01	43	1.00	45	1.01	44	1.00	43	1.02	67
_213_javac	1.03	69			1.00	79	1.03	46	1.05	130	1.02	44	1.04	63
_228_jack	1.01	45	1.08	72	1.01	44	1.01	44	1.00	44	1.00	45	1.06	75
_205_raytrace	1.02	69	1.09	71	1.00	62	1.02	48	1.05	120	1.00	50	1.07	67
_227_mtrt	1.03	51	1.06	76	1.00	79	1.01	48	1.01	50	1.02	50	1.05	75
_201_compress	1.01	61	1.02	67	1.01	59	1.02	44	1.00	80	1.00	59	1.01	62
pseudojbb	1.03	53			1.00	72	1.05	51	1.03	76	1.02	53	1.04	64
_209_db	1.00	59	1.01	47	1.00	115	1.00	59	1.00	60	1.00	59	1.00	59
_222_mpegaudio	1.01	65	1.01	68	1.01	63	1.00	44	1.01	114	1.00	44	1.01	70
mean	**1.02**	**57**	**1.05**	**63**	**1.00**	**68**	**1.02**	**48**	**1.02**	**80**	**1.01**	**50**	**1.03**	**67**
geometric mean	**1.02**	**57**	**1.05**	**62**	**1.00**	**66**	**1.02**	**47**	**1.02**	**74**	**1.01**	**49**	**1.03**	**67**

Table 5: BG-RC Sensitivity to Variations in Collection Triggers (defaults are 4MB nursery, 60ms time cap, and 512KB cycle trigger)

reduces the pause time, and does not degrade performance. However since some of the collection (such as the nursery collection) is compulsory, the collector could not honor the 30ms time cap, but lowering the cap did reduce pauses times on most benchmarks.

Lowering the cycle detection meta-data limit leads to better throughput and responsiveness. However, since this experiment was conducted at a moderate heap size, it does not expose the fact that reducing opportunities for the cycle detector to function will exhaust the heap faster. We chose our default setting (512KB) to allow BG-RC to operate in a similar range of heap sizes as BG-MS. Increasing opportunities for cycle detection increases the probability that the trial deletion algorithm will try to reclaim large structures and produce long pauses, which is why we see maximum pause time growing with the cycle detection limit.



(a) _202_jess (d) _213_javac (g) _228_jack

(b) _205_raytrace (e) _227_mtrt (h) _201_compress

(c) pseudojbb (f) _209_db (i) _222_mpegaudio

Figure 6: Mutator Time as a Function of Heap Size

6. Related Work

This section describes related work in concurrent and incremental collection. Concurrent collectors sacrifice throughput for low pause times, where as ulterior reference counting can achieve both. This section also compares our work with Azatchi and Petrank's concurrent generational reference counting collector.

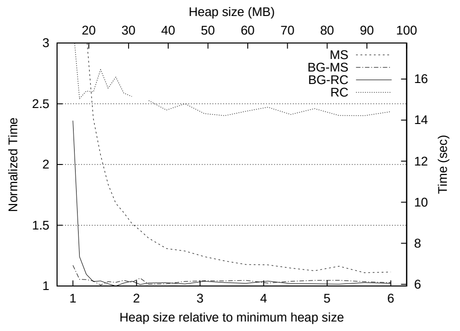
6.1 Incremental and Concurrent Collection

Other incremental approaches include MOS, real-time, and concurrent collectors. The mature object space (MOS) collector traces and copies objects, incrementally packing connected objects together [20]. It achieves completeness without full heap collections and can be configured to be highly incremental, yielding low pause times. However, completeness comes at a performance cost since it potentially copies objects numerous times before it identifies them as garbage.

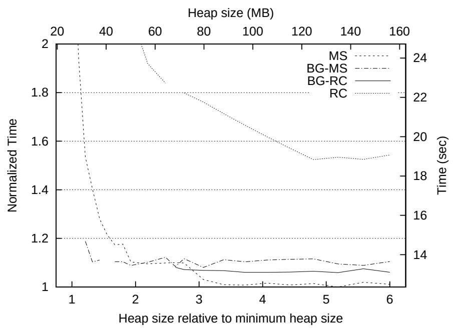
Concurrent tracing collectors [15] use a special write barrier to accommodate interference by the mutator in the tracing phase. Bacon, Cheng, and Rajan claim the best utilization to date and very short pause times for a concurrent real-time collector [8]. Their collector is mostly non-moving and incremental. Dedicating a separate CPU to the task of collection can mitigate such significant overheads but hurts total throughput [7, 9, 26]. In contrast, we do not target real-time applications and do not collect concurrently with the mutator, but achieve throughput matching a high performance collector. We consider solutions that do not require additional CPUs.

6.2 Generational Reference Counting

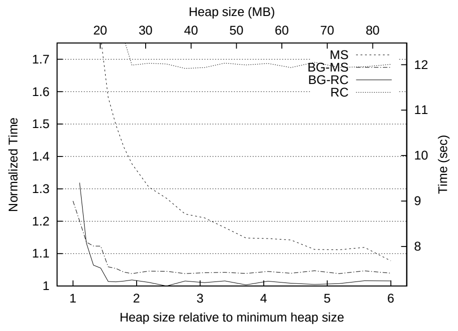
In parallel with our work [14], Azatchi and Petrank add generations to a *sliding-view* concurrent reference counting collector [6, 23]. Their generational collector uses a free list for all objects. They snapshot all mutated objects between collections. Since nursery objects are scattered throughput the heap, the algorithm must keep a list of *all* of them to collect them separately. During a nursery collection, the collector marks all live nursery objects, and sweeps



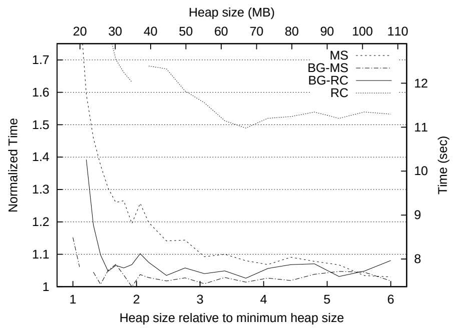
(a) _202_jess



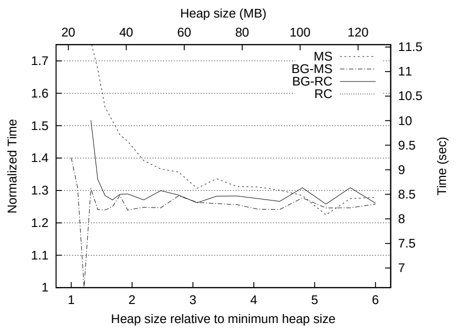
(d) _213_javac



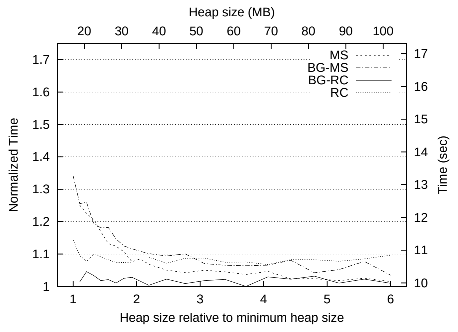
(g) _228_jack



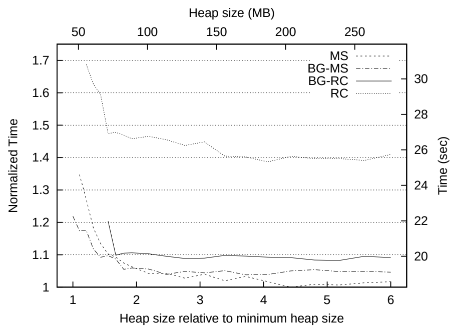
(b) _205_raytrace



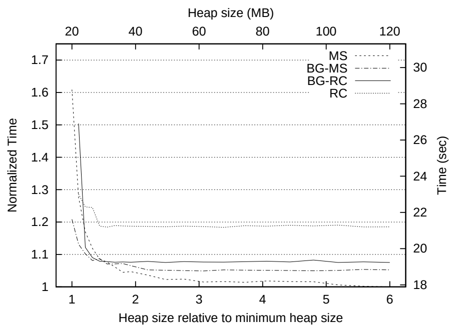
(e) _227_mtrt



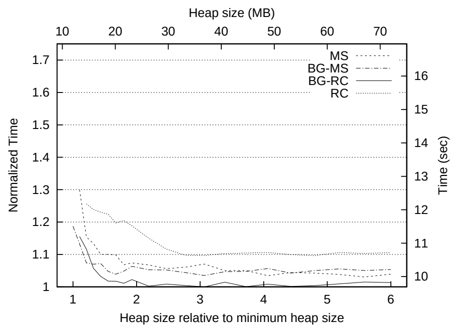
(h) _201_compress



(c) pseudojbb



(f) _209_db



(i) _222_mpegaudio

Figure 7: Total Time as a Function of Heap Size

the remainder. Their reclamation is thus proportional to the entire nursery size, rather than the survivors as in a copying nursery. A copying nursery performs better. While processing the nursery log, the collector reference counts survivors as it promotes them. The basic concurrent design attains excellent pause times, better than Bacon et al. [6, 7]. The sliding view reference counting collector performs occasional concurrent mark-sweep collections to collect cycles.

Our work introduces the tracing deferral mechanism for heap objects. Both papers discuss slot and object logging deferral and measure variants of object logging. The major quantitative advantage of our approach is the combination of space and time efficiency, yielding a much higher throughput collector coupled with low pause times. For example, since we use a copying nursery, we achieve the fastest possible allocation time, and combine scanning and reclamation time proportional to survivors. We also have significantly less space overhead.

In this paper, we compare performance with BG-MS which combines the best of copying and mark sweep, using bump-pointer allocation and a space efficient mark-sweep free list for long-lived objects. Previous [5] and concurrent [11] work shows that BG-MS performs better than a variety of other collection algorithms, including other generational collectors. MS, which we also include, is a widely used algorithm but it performs poorly on throughput and pause time. 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 entire, full heap.

Beltway collectors [12] generalize over classic copying generational collectors by adding incremental collection on independent *belts* which are analogous to generations. Beltway configurations outperform generational copying collectors, but have not been directly compared to BG-MS [12]. We believe that a generalization of ulterior reference counting as the last belt is a Beltway configuration that could perform better than the results here.

7. Conclusion

The tension between responsiveness and throughput is a longstanding problem in the garbage collection literature. Until now, collectors either exhibited good throughput performance or good responsiveness, but not both. BG-RC carefully matches allocation and collection policies to the behaviors of older and younger object demographics, and thus delivers both excellent throughput and good responsiveness. The key to this result is an algorithm that generalizes deferred reference counting to safely ignore mutations to nursery objects and thus significantly reduces the reference counting load. Future collectors could improve on these results with incremental or concurrent dead cycle detection for large structures, and an adaptive algorithm that selects a more appropriate collector for highly-mutated but long-lived objects.

8. Acknowledgements

We want to thank IBM Research and in particular Perry Cheng with whom we designed and implemented JMTk. Without JMTk and Jikes RVM, this work would not have been possible. We also thank Michael Hind, David Grove, and David Bacon for their support. Finally, we thank our respective research groups for their feedback and encouragement.

9. 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] H. Azatchi and E. Petrank. Integrating generations with advanced reference counting garbage collectors. In *International Conference on Compiler Construction*, Warsaw, Poland, Apr. 2003. To Appear.

[7] 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), June 2001.

[8] D. F. Bacon, P. Cheng, and V. T. Rajan. A realtime garbage collector with low overhead and consistent utilization. In *POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, January 15-17, 2003*, volume 38(1) of *ACM SIGPLAN Notices*. ACM Press, Jan. 2003.

[9] D. F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In J. L. Knudsen, editor, *Proceedings of 15th European Conference on Object-Oriented Programming, ECOOP 2001, Budapest, Hungary, June 18-22*, volume 2072 of *Lecture Notes in Computer Science*, pages 207–235. Springer-Verlag, 2001.

[10] E. D. Berger, B. G. Zorn, and K. S. McKinley. Building high-performance custom and general-purpose memory allocators. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementati on*, pages 114–124, Salt Lake City, UT, June 2001.

[11] S. M. Blackburn, P. Cheng, and K. S. McKinley. A garbage collection bakeoff in a Java memory management toolkit (JMTk). Technical Report TR-CS-03-02, ANU, Mar. 2003.

[12] 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*. ACM Press, June 2002.

[13] S. M. Blackburn and K. S. McKinley. Fast garbage collection without a long wait. Technical Report TR-CS-02-06, Dept. of Computer Science, Austrailian National University, Nov. 2002.

[14] 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.

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

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

[17] 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.

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

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

[20] 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.

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

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

[23] Y. Levanoni and E. Petrank. An on-the-fly reference counting garbage collector for Java. In *ACM Conference Proceedings on Object–Oriented Programming Systems, Languages, and Applications*, pages 367–380, Tampa, FL, Oct. 2001.

[24] H. Lieberman and C. E. Hewitt. A real time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[25] E. Petrank. Private communication, July 2003.

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

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

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

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

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

[31] 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.