Beltway: Getting Around Garbage Collection Gridlock

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

Richard Jones
Computing Laboratory
University of Kent
Canterbury, Kent, CT2 7NF, UK
R.E.Jones@ukc.ac.uk

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

J Eliot B Moss
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003-4610, USA
moss@cs.umass.edu

ABSTRACT

We present the design and implementation of a new garbage collection framework that significantly generalizes existing copying collectors. The *Beltway* framework exploits and separates object age and incrementality. It groups objects in one or more increments on queues called *belts*, collects belts independently, and collects increments on a belt in first-in-first-out order. We show that Beltway configurations, selected by command line options, act and perform the same as semi-space, generational, and older-first collectors, and encompass all previous copying collectors of which we are aware.

The increasing reliance on garbage collected languages such as Java requires that the collector perform well. We show that the generality of Beltway enables us to design and implement new collectors that are robust to variations in heap size and improve total execution time over the best generational copying collectors of which we are aware by up to 40%, and on average by 5 to 10%, for small to moderate heap sizes. New garbage collection algorithms are rare, and yet we define not just one, but a new family of collectors that subsumes previous work. This generality enables us to explore a larger design space and build better collectors.

Categories and Subject Descriptors

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

General Terms

Design, Performance, Algorithms

Keywords

Beltway, copying collection, generational collection, Java

1. Introduction

Programmers are turning more and more to garbage-collected object-oriented languages that automate the reclamation of memory that the program can no longer access. Garbage collection (GC) improves programming productivity by reducing errors that result from explicit memory deallocation, and underpins sound software engineering principles of abstraction and modularity. Current GC algorithms, however, still have a performance overhead. Figure 1(a) shows the fraction of time that six SPEC Java programs spend in GC as a function of heap size, using a high performance Appel-style generational copying collector [3] in the Jikes RVM [1, 2]. When heap space is tight, GC can take up to 35% of execution time. Applications may reduce this cost simply by using a larger heap in which GC typically comprises less of total execution time. However, as shown by Figure 1(b) and by other research [10], the best total execution time is not always achieved when GC time is minimized by large heap sizes. Application cache, memory, and TLB locality may degrade with large heaps. For example, paging degrades pseudojbb's performance at the large heap sizes in Figure 1(b). Thus, achieving high performance remains a challenge, especially for programs and workloads with large memory requirements.

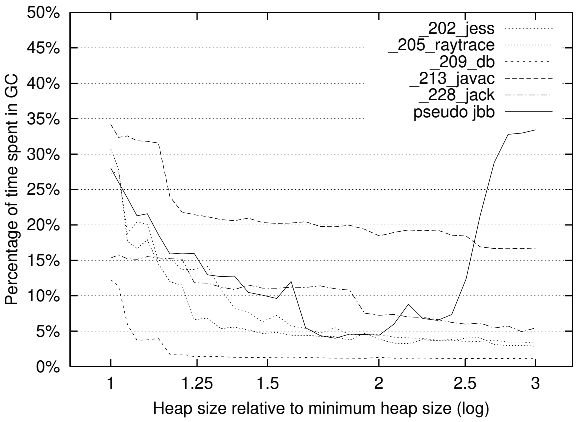
In more than forty years of research, a few key insights have shaped copying garbage collection. (1) The *weak generational hypothesis* that 'most objects die young' underpins generational garbage collectors, which preferentially collect the youngest objects [34]. (2) As a corollary to this observation, generational collectors avoid collecting old objects. (3) Using incrementality to improve response time has led to the use of small nursery generations and to incremental algorithms [13, 24]. (4) Researchers also use small nurseries and copying collectors to improve data locality [25, 38]. (5) More recently, Stefanović et al. demonstrate that giving the very youngest objects time to die can improve collector performance [32].

The *Beltway* collection framework is the first to combine and exploit all five insights flexibly and efficiently. In addition, Beltway generalizes over previous work: we can configure Beltway to behave as every other region-based copying collectors of which we are aware. A Beltway collector uses *increments* and *belts* as shown in Figure 2. An increment is the unit of collection. A belt groups one or more increments into a first-in-first-out (FIFO) queue. Beltway collects each increment on a belt independently in FIFO order, and also collects each belt independently. The promotion policy determines where to copy surviving objects, whether to the same or to another belt. Increments make belts more general than generations since all objects within a generation must be collected *en masse*, but we collect increments independently and there may be multiple increments on a belt. We demonstrate Beltway configurations (from command line parameters) that behave as semi-space collectors, traditional generational copying collectors [34, 3], and older-first collectors [32]. To our knowledge, Beltway configurations match all previous copying collector organizations.

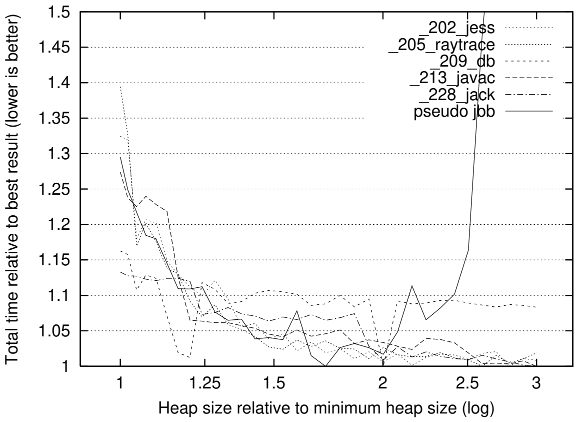
We further show that this generality enables us to combine all

*This author did this work while at the University of Massachusetts. This work is supported by NSF ITR grant CCR-0085792, NSF grant ACI-9982028, DARPA grants F30602-98-1-0101 and F33615-01-C-1892, EPSRC grant GR/R42252, and IBM. Any opinions, findings, conclusions, or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

PLDI'02, June 17–19, 2002, Berlin, Germany.



(a) Percentage of time spent in GC.



(b) Total application performance.

Figure 1: The impact of heap size on the performance of six SPEC benchmarks using the Appel-style generational collector. Optimal performance is not always attained at the largest heap size.

the above ideas within a single collector. We present the design and implementation of a range of copying collectors that exploit the high mortality of young objects, can avoid collecting the very youngest objects, avoid collecting previously copied objects, and perform collection incrementally. Our generality increases pointer tracking costs but we develop several novel mechanisms to bound these costs. For instance, we maintain the fewest possible cross-increment pointers, and we can trigger collections when the number of cross-increment pointers exceeds a threshold.

We show several configurations that reduce garbage collection costs via reduced copying and better heap utilization compared to the best generational copying collector of which we are aware [3]. These reductions improve total execution time over generational collectors by an average of 5 to 10%, and up to 35% on tight heaps, for 6 Java SPEC programs. Our new collectors thus use resources more effectively than generational collectors. We also present evidence that our framework enables us to explore the tradeoff between responsiveness and throughput, but we leave a more thorough investigation of responsiveness to future work.

The remainder of the paper is organized as follows. We first present the Beltway framework for exploring copying collection and the program characteristics that it exploits. Section 3.1 shows how to configure Beltway to implement a variety of copying collectors. We then present two new collectors designed to reduce total execution time. Section 3.3 introduces several novel mechanisms that make our collectors efficient. Section 4 compares these collectors to generational collectors and demonstrates that they reduce GC time and total execution time on 6 Java programs. Section 4.3 offers some sample responsiveness results. We find that reduced total execution time can also be combined with improved responsiveness, but this area needs further exploration. We then present related work and conclude.

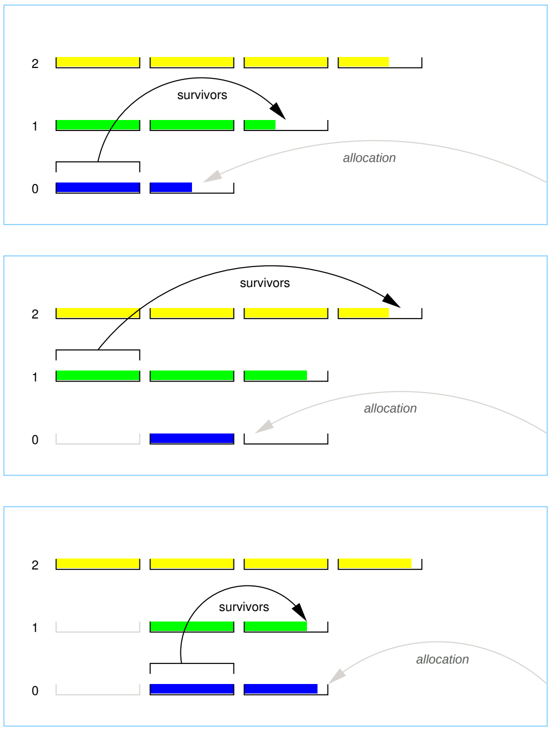


Figure 2: A Beltway configuration with three belts, showing three successive collections (top to bottom). The light arrows show allocations going into the nursery, and darker arrows represent copying surviving objects from the target to the source increments. Darker objects are younger.

2. Generalizing Copying Garbage Collection

The novelty of the Beltway framework is that it generalizes over copying collection by combining all the key ideas of copying GC in a single collector. This section first outlines these ideas. We then describe the Beltway framework and how it exploits the ideas.

2.1 Key Ideas in Copying GC

Most objects die young. The *weak generational hypothesis* is the basis for generational garbage collection [34]. A generational collector divides the heap into regions called *generations* that contain progressively more mature objects. The youngest generation, called the *nursery*, is likely to contain a large fraction of dead objects and so is most frequently targeted for collections. The collector *promotes* objects that survive by copying them into the next generation.

Avoid collecting old objects. A corollary is that those objects that do not die young tend to be long lived, suggesting that older generations should be collected less frequently.

Give objects time to die. A further observation is that while most objects die young, all objects require some time to die. The older-first collector [32] exploits this observation by avoiding collecting the very youngest objects.

Incrementality improves responsiveness. Length and frequency of collection limits the responsiveness of garbage collected programs. Generational collectors reduce average pause time by repeatedly collecting the nursery, and occasionally collecting the whole heap. They therefore tend not to improve on worst case pause time. Other algorithms are more aggressively incremental. For example, the Mature Object Space collector [24] collects only one increment at a time and never collects the whole heap.

Copying GC can improve locality. Programs often access objects of a similar age together [20]. Copying collectors exploit this pattern to improve locality with consequent benefits for cache and TLB behavior [38]. Collectors copy older objects near to each other in the heap. This clustering reduces the incidence of pointers that span regions of the heap, and thus avoids retaining dead objects simply because they are referenced by a dead object in an uncollected region [35]. The nursery attains locality by keeping the youngest and most frequently accessed objects near each other.

Previously, no collector has exploited all of these ideas together. Simple semi-space collectors improve locality through copying, but do not exploit any of the other ideas. Generational collectors do not give the very youngest objects time to die and must occasionally collect the whole heap, so are not fully incremental. The older-first collector cannot always avoid frequent copying of old objects, and because it does not collect the whole heap it is not *complete*, i.e., it cannot guarantee it will collect all garbage.

2.2 Beltway Collectors

Beltway collectors depend on two very simple organizational principles. An *increment* is an independently collectible region of memory. A *belt* is a grouping of one or more increments, collected in strict FIFO order (analogous to conveyor belts). By selecting different increment sizes, belt organizations, and promotion policies, a collector in the Beltway framework can be configured to implement any of the well-known copying collection algorithms.

Figure 2 shows a Beltway collector with three belts, each with one or more increments. This collector promotes survivors from each increment in to the next higher belt. It copies survivors in the highest belt to the end of that belt. New allocations go to the last increment in the lowest, nursery belt.

A Beltway collector can exploit each of the five ideas outlined above as follows. Belts generalize over generations by decoupling incrementality from the generation size. Thus, the lowest belt is analogous to the nursery in a generational collector. By preferentially collecting increments from the nursery belt, we exploit the weak generational hypothesis and avoid collecting old objects. Because Beltway decouples collection and belts, it can be arbitrarily incremental. Because the oldest increment on a belt will always be collected first (FIFO order), Beltway can give objects time to die. Finally, the copying, generational, and incremental aspects of Beltway improve the locality of surviving objects, and provide locality for the youngest objects by allocating them together in the nursery.

3. Concrete Instances of Beltway Collectors

To demonstrate the generality of the Beltway framework, we describe Beltway configurations that correspond exactly to well-known copying collectors. We then describe two new collectors, and the novel mechanisms that are key to implementing these new collectors efficiently.

3.1 Modeling Existing Copying Collectors

One factor in common among all copying collectors is that they must hold in reserve sufficient memory to accommodate a collection of the largest possible increment. This *copy reserve* space must be large enough to accommodate the worst case survival for a collection, i.e. when all objects survive.[1] If the copy reserve is fixed at half the heap, as it is in the semi-space collector and generational collector implementations, heap utilization and efficiency can suffer. In the remainder of this section we use the term *usable memory* to refer to the total heap space less the appropriate copy reserve for that collector.

Semi-space collectors are the simplest copying collectors [12]. They correspond to a trivial Beltway configuration: a single belt containing a single increment, as large as the usable memory, collected whenever it is full, as shown in Figure 3(a). We call this configuration *BSS*, Beltway Semi-Space. *BSS* copies survivors into a new increment on the same belt.

Appel-style generational collectors have two generations [3]. They make efficient use of memory by allowing the nursery to grow to consume all usable memory not consumed by the higher generation. Consequently, they collect the nursery only when both generations consume all usable memory. Appel corresponds to Beltway configured with two belts, each with one increment capable of accommodating all usable memory, as depicted in Figure 3(b). We call this configuration *BA2*, Beltway Appel with two generations. Whenever the two increments consume all usable memory, *BA2* collects the nursery increment, copying survivors to the higher belt. When the higher increment consumes all usable memory, *BA2* collects it, copying survivors to a new increment on the same belt. (In practice, if the nursery size drops below some small fixed threshold, the heap is considered full.)

Older-First Mix algorithms are an incremental variation on the semi-space collector [19, 31]. They are called older-first mix because they mix copies and newly allocated objects in memory. This Beltway configuration, *BOFM*, shown in Figure 3(c), has one belt and multiple increments. *BOFM* both allocates and copies survivors to the last increment on the belt, triggering collection when the increments consume all usable memory.

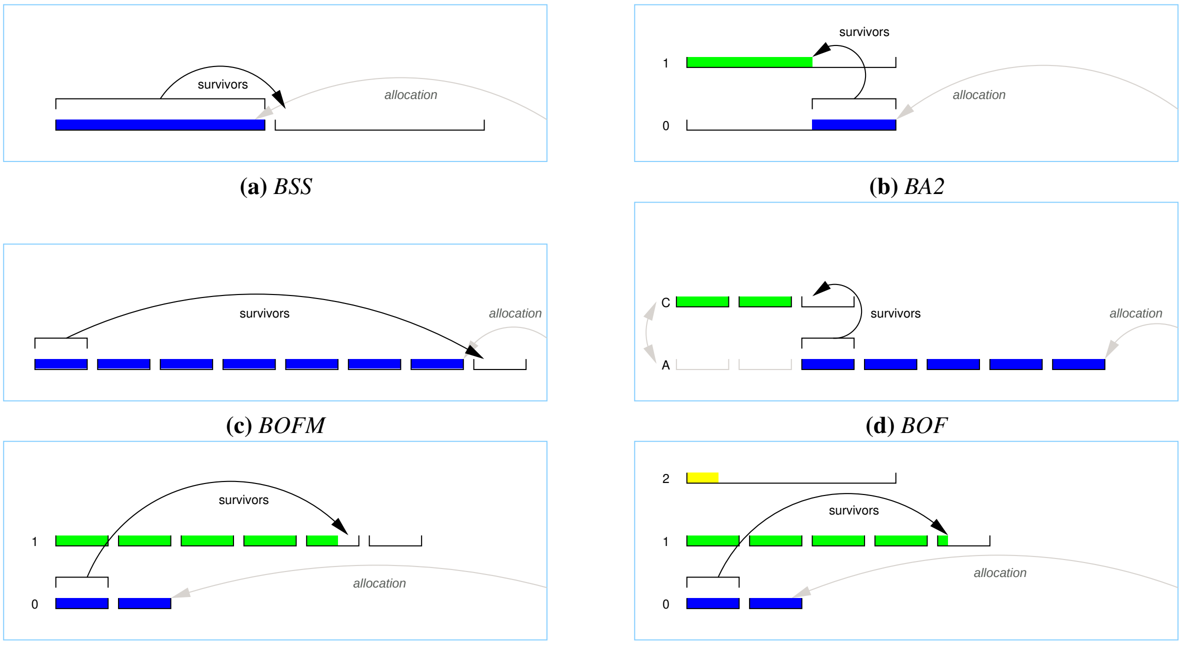
Older-First collectors organize the heap by object age [32]. They collect a fixed-size window that slides through the heap from older to younger objects. When the heap is full, OF collects the window, returns any free space to the nursery, and then positions the window for the next collection over objects just younger than those that survived. If it bumps into the allocation point, it resets the window to the oldest end of the heap. This Beltway configuration, *BOF*, illustrated in Figure 3(d), has an allocation belt *A* and a copy belt *C* where increments are the size of the collection window. *BOF* allocates to the back of belt *A*. Whenever all usable memory is consumed, *BOF* collects the first increment in belt *A*, copying survivors to the back of belt *C*. If all usable space is consumed and *A* is empty, then *BOF* 'flips' the belts, collects the first increment in the new belt *A*, and copies its survivors to the last increment in the new, now empty, belt *C*. *BOF* then continues to allocate to the back of the new belt *A*.

3.2 New Beltway Configurations

We now go beyond existing copying collectors and describe two new collector configurations, Beltway *X.X* and Beltway *X.X.100*.

Beltway *X.X* collectors add incrementality to Appel-style generational collection. They have two belts, and each belt contains increments of maximum size *X*, conventionally expressed as a percentage of usable memory. The two belts correspond to generations, and *X* reflects the degree of incrementality. As with Appel's collector, the lower, nursery belt grows until it consumes all remaining

[1] In fact the copy reserve must be slightly more generous because the copied data may not pack as well as the original data, as an artifact of object alignment and copying order.



(a) *BSS* **(b)** *BA2* **(c)** *BOFM* **(d)** *BOF* **(e)** Beltway *X.X* **(f)** Beltway *X.X.100*

Figure 3: The Beltway configurations described in Sections 3.1 and 3.2. Each diagram shows a configuration of belts and increments during the copying of survivors. An arrow indicates the allocation that triggered the collection.

usable space, at which point Beltway *X.X* collects the oldest increment in the nursery. It promotes survivors to the youngest increment in the higher belt. When the higher belt becomes full, it collects the oldest increment in the higher belt, and copies survivors to the youngest increment in the same belt. Similar to generational collectors, Beltway *X.X* collects the higher belt only when the higher belt is full and the nursery belt is thus empty. However, it collects only a fixed size increment, rather than the entire belt.

Beltway *X.X* combines most of the features of Appel's generational collector and the Older-First collector, and exploits all five ideas outlined in Section 2.1. In fact, *BA2* is a special case of Beltway *X.X* where X is set to 100. When $X < 100$, the steady state differs from *BA2* as follows. The nursery typically contains one increment that varies in size up to size X, and the older belt contains the other increments. Together they can occupy no more than the usable heap space. For example with $X = 33$, we can have four increments, one partially full and two completely full increments on belt 1, and one partially full increment on belt 0. Our framework and implementation also supports Beltway *X.Y* collectors where $X \neq Y$, but we do not explore these configurations here. Unfortunately, when $X < 100$, Beltway *X.X* lacks *completeness*: it does not guarantee the eventual collection of all garbage, because it fails to collect garbage cycles that span more than one increment.

Beltway *X.X.100* collectors address the failure of Beltway *X.X* with respect to completeness by retaining the two lower belts with increments of size $X \leq 100$, and adding a third, highest belt with a single increment that may grow as large as the usable memory. Any objects Beltway *X.X.100* does not reclaim in the lower belts, it promotes to the third belt, which it will collect in its entirety only once it has grown to consume all usable memory. It thus guarantees eventual collection of large dead structures. This configuration achieves completeness at the expense of incrementality (the worst case collection increment is the same as the classic semi-space and generational collectors). Note that when X is equal to 100, Beltway *X.X.100* implements a three-generational Appel-style collector.

Section 4 shows that Beltway *X.X.100* almost always outperforms the Appel collector, sometimes by as much as 35% of total benchmark running time. An alternative approach to lack of completeness in the Beltway *X.X* collector is to use a complete, incremental collector (such as the Mature Object Space collector [24]) in place of the third belt, but that investigation is beyond the scope of this work.

3.3 Realizing Efficient Beltway Collectors

Three broad implementation issues are key to the viability of our approach. (1) Incrementality depends heavily on the use of *write barriers*, so the efficiency of write barriers and their associated data structures is critical. (2) We found that the best time to collect is not always when the heap is full. To this end, Beltway collectors use *collection triggers* to preempt identifiable performance problems in later collections. (3) Finally, we exploit a dynamically sized *copy reserve* based on the increment size and heap occupancy, which leads to better heap utilization and ultimately better performance.

3.3.1 Frames and Write Barriers

In order to collect increments independently and efficiently, garbage collectors must remember references into collected increments from the rest of the heap. A simple semi-space collector avoids the need for any such mechanism by always collecting all usable memory. Generational collectors very cheaply notice and remember the creation of any references into the nursery from the rest of the heap. For example, they place the nursery in high memory and observe the creation of any pointers that cross a boundary between high and low memory [7]. One also must instrument the *mutator*, (i.e., the application) with a write barrier to remember relevant pointers.

Beltway collectors implement increments by using *frames*. A frame is an aligned contiguous region of virtual memory that can accommodate an increment. Frames reduce the cost of incremental collection in two ways. First, frames are power-of-two aligned in the address space, and we can distinguish inter-frame pointers from intra-frame pointers using a shift and compare. Second, we maintain a number associated with each frame that indicates the frame's relative collection order.

When we encounter an inter-frame reference, we need to remember it only if we might collect the target frame sooner than the source frame. We therefore do not record all inter-frame references. Further, if an increment spans multiple frames marked with the same collection time, we do not store pointers between the constituent frames. Although the barrier is not address-ordered, it is uni-directional with respect to frames. For example, in the *BOFM* collector (Figure 3), only cross-increment pointers in the right-to-left direction must be remembered. Figure 4 shows a basic implementation of this frame-based, unidirectional write-barrier. We partially inline the write barrier [7].

```
public static final void writeBarrier(ADDRESS source,
                                      ADDRESS target) {
  int s = (source>>>FRAME_SIZE_LOG);
  int t = (target>>>FRAME_SIZE_LOG);
  if ((s != t)          // pointer is inter-frame
      && (Belt.collect_[t] < Belt.collect_[s])) {
    // target will be collected before source
    int rsidx = (s<<REMSET_SHIFT) | t;
    GCTk_RememberedSet.insert(rsidx, source);
  }
}
```

Figure 4: The basic Beltway frame-based write-barrier.

3.3.2 *Remembered Sets*

The total number of frames in a configuration is limited by the increment size, the number of belts, and the total usable memory, and is tightly bounded. We can thus maintain distinct remembered sets, *remsets*, for each target-source frame pair. At run time, we enter each inter-frame reference in the appropriate set. An advantage of this approach is that we can trivially delete all remsets relating to a frame. We also ignore remsets between two increments in the occasional case when we collect them together. For example, given sufficient copy reserve, if we are about to empty a lower belt and fill the next higher belt, we will collect the increment on the lower belt together with the first increment on the next belt. This optimization performs a single collection, rather than two in immediate succession.

Because of the high write-barrier activity in the nursery, we limited Beltway *X.X* and Beltway *X.X.100* to a single, bounded nursery increment, which minimizes write-barrier activity. Jikes RVM introduces substantial write barrier overhead due to the initializing of each object's type ('TIB') pointer. The type object is older, usually much older, than the object. To eliminate this overhead, we use a single nursery increment and extend the basic Beltway barrier to filter any pointers where the source is in the nursery. This optimization foregoes older-first behavior within the nursery. Systems without this overhead should be able to benefit from multiple nursery increments.

3.3.3 *Collection Triggers*

For a variety of reasons, it is not always best to collect only when the heap is completely full. For instance, fixed-size nursery collectors collect when the nursery is full rather than when the heap is full. In this section, we describe a number of *collection triggers* that define a range of additional conditions that can initiate a garbage collection. We explored three mechanisms, *nursery*, *remset*, and *time-to-die* triggers with multiple nursery increments, and believe that configurations of Beltway will benefit from one or more of them. For the Beltway *X.X* and Beltway *X.X.100* configurations we report below, only the nursery trigger that limits the nursery to a single increment proved useful.

Nursery Trigger. The size of the nursery belt may be bounded to ensure that we frequently collect the young objects since many die quickly. An obvious example of this trigger is the classic fixed-size nursery generational collector which limits the nursery to one increment, where the size of that increment is always equal the maximum nursery size. In Beltway *X.X* and Beltway *X.X.100*, we use this trigger to limit the maximum size, but not the minimum size of a single nursery increment.

Remset Trigger. Because remembered set entries are collection roots, as the number of remset entries grows, the survival rate for an increment goes up as well as the time to scan the remset itself. A simple and very effective solution to this problem is to trigger collection whenever remembered sets grow to some threshold.

Time-to-Die Trigger. We may want to collect a nursery increment before it reaches its maximum capacity. For example, an Appel-style nursery increment can accommodate all the usable memory, but we often collect the nursery when it is only partially full. By using two increments on the youngest belt rather than one, we can avoid collecting the *very* youngest objects, which would not yet have had time to die. The time-to-die trigger ensures that all objects will have at least *TTD* time to die before we collect them (time is measured in bytes of allocation). When the heap is within *TTD* bytes of being full, we can use the time-to-die trigger to ensure that all new allocations go into the second frame. If the system is allocating into the first increment, it then starts allocating objects into the second frame. Subsequently, when the heap fills, it collects the first frame, which may not be full. This trigger prevents the collectors from collecting the objects allocated in the last *TTD* bytes of allocation (i.e., when they are too young).

We believe future configurations of Beltway will be able to exploit all these triggers to improve collector performance further, but we use only the nursery trigger in our results.

3.3.4 *Dynamic Conservative Copy Reserve*

As we pointed out in Section 3.1, all copying collectors need a *copy reserve* space into which to copy survivors. In order to avoid failure in the worst case, the copy reserve must be slightly larger than the largest possible collection increment. Since the usable memory is the heap space less the copy reserve, it is obviously advantageous to minimize the copy reserve. Finer-grained incremental collectors, such as Mature Object Space (MOS) collectors or Beltway *X.X*, where X$\ll$100, have a distinct memory utilization advantage because they require only a small copy reserve. Classical generational and semi-space collectors must reserve half the heap.

Beltway collectors dynamically calculate a conservative minimal copy reserve that will always accommodate survivors from the worst case collection sequence. The copy reserve is either the largest increment size, or the largest potential increment occupancy at the next collection. We determine maximum potential occupancy for each increment by adding its current occupancy plus the maximum occupancy of any other increment from which the collector could copy into this one.

The dynamic conservative copy reserve is particularly effective in the Beltway *X.X.100* collector where the third belt is rarely full. The copy reserve is thus usually determined by the smaller increment size. As the third belt fills, the copy reserve grows until it is finally half of the heap (so that the third belt occupancy and the copy reserve are equal in size). After we collect the third belt, the copy reserve automatically falls back to a smaller size, thereby continuously maximizing usable memory.

Benchmark	Description	Min. heap size	Total allocation	GCs
_202_jess	An expert system shell	12MB	301MB	24–337
_205_raytrace	A ray tracing program	15MB	127MB	9–139
_209_db	Simulates a database management system	22MB	102MB	5–115
_213_javac	The Sun JDK 1.02 Java compiler compiling jess	32MB	266MB	10–100
_228_jack	Generates a parser repeatedly	20MB	320MB	16–135
pseudojbb	Emulates a 3-tier transaction processing system	70MB	381MB	4–126

Table 1: Benchmark characteristics: minimum heap size, total bytes allocated, and the number of GCs performed by an Appel-style collector at large and small heaps respectively.

4. Results

This section describes our experimental setting, including our garbage collector environment, hardware, and benchmarks. We then present GC time and total execution time for Beltway and generational collectors with a variety of configuration parameters. Finally, we show some sample responsiveness results.

4.1 Experimental Setting

Jikes RVM and GCTk. We use Jikes RVM version 2.0.2 for our implementation study. Jikes RVM (formerly Jalapeño) is a high performance VM written in Java that includes an aggressive optimizing compiler [1, 2]. We used the Jikes RVM *adaptive* compiler and its *fast* build-time configuration (which omits assertion checking and pre-compiles as much as possible into the Jikes RVM boot image).

We have recently developed a new GC toolkit for Jikes RVM called GCTk, which includes Beltway as well as implementations of previous generational collectors. GCTk is an efficient and flexible platform for GC experimentation that exploits the object-orientation of Java and the VM-in-Java property of Jikes RVM. Prior to developing Beltway, we implemented a number of GC algorithms in GCTk. These collectors include Appel-style and fixed-nursery generational collectors whose performance we report below. We found their performance to be similar to existing Jikes RVM GC implementations. Existing Jikes RVM collectors all statically partition their heap into small and large object spaces, and unconditionally utilize the large object space. Unfortunately, GCTk currently does not yet implement a large object space. Direct comparisons between GCTk and the native Jikes RVM collectors are therefore not possible without significant changes to one of the systems.

After developing the generational collectors, we tuned them over an eighteen-month period of heavy use in several contexts [7, 8]. For example, they use a very fast address-order write barrier [7]. We compare Beltway against these collectors. These collectors are not limited in any way by the generalizations we employ in Beltway. We did however design GCTk using object-oriented techniques which enables the reuse of key GC infrastructure. Of the 26 classes in Beltway and in the generational collectors, 23 are common to both. We implemented the Beltway collectors in GCTk as a single collector with command-line options to specify the configuration.

Benchmarks. We use six SPEC benchmarks, five drawn from the SPEC JVM98 suite, and pseudojbb, a slightly modified variant of SPEC JBB2000 [28, 29]. 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 reasonably. Dieckman and Hölzle present a thorough analysis of SPEC JVM98 [17]. Table 1 shows some characteristics in our system: the minimum heap size in which an Appel-style collector does not fail, the bytes allocated in this system, and the number of GCs at large and small heap sizes.

We ran each program 5 times for each collector configuration and picked the best execution time (i.e., the one least disturbed by other effects in the system). We separately performed a statistics gathering run for each configuration to measure the rate at which write barrier fast and slow paths were taken. We ran these programs on 33 heap sizes, ranging from the smallest one in which the program completes up to 3 times that size.

Hardware. Our experimental timing runs were performed on a Macintosh PowerMac G4, with a 733MHz processor, 32KB on-chip L1 data and instruction caches, a 256KB unified L2 cache, a 1MB L3 off-chip cache, and 128MB of memory, running PPC Linux 2.4.10.

4.2 Throughput

This section examines the GC and total application performance for a range of generational, Beltway *X.X.100*, and Beltway *X.X* collectors. We begin by comparing Beltway configurations that match Appel-style generational collectors and show they perform similarly. We then turn to the choice of generational collector; we compare fixed-nursery collectors with a range of sizes to the flexible-nursery Appel generational collector. Our experiments show that Appel improves performance, typically by about 50%, regardless of nursery size. We therefore use it as our main comparison point.

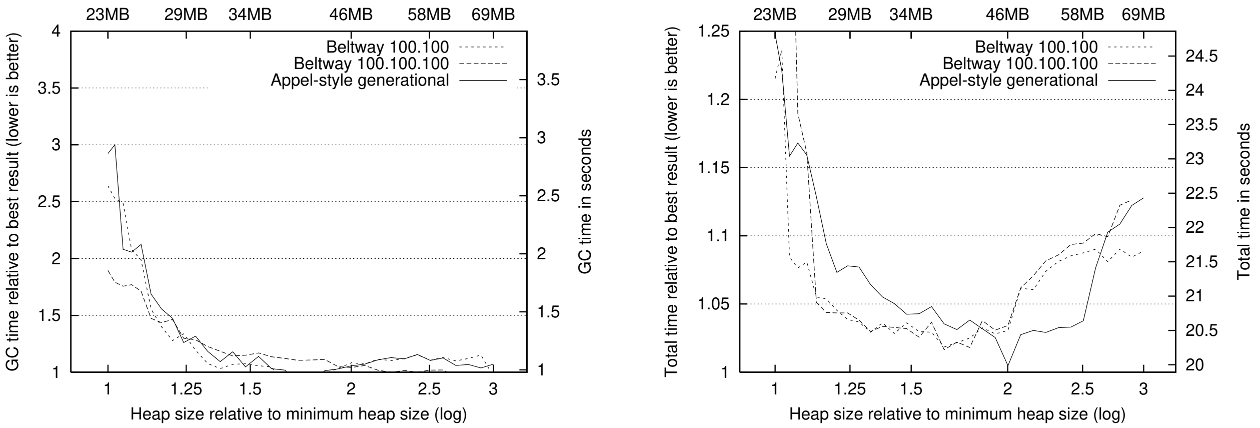
We then explore the effect of increment sizes on Beltway *X.X.100* and find that as long as the size is not too small, Beltway *X.X.100* is not very sensitive to increment size. We also compare Beltway *X.X.100* to Beltway *X.X* for one increment size, and find their performance comparable. We finally present execution and collection times for Beltway *X.X.100*, Appel, and a fixed-size nursery collector, which show that Beltway *X.X.100* generally performs much better than generational collectors.

4.2.1 Beltway as Appel

Figure 5(a) compares GC time, and (b) compares total application time. In all the performance graphs, the left y-axis is the performance relative to the best in the figure, the right y-axis is the actual time, the bottom x-axis is the heap size relative to the minimum, and the top x-axis is the actual heap size.

Figure 5 compares Appel, Beltway *100.100* (the Appel configuration of Beltway), and Beltway *100.100.100*, using the geometric mean of our 6 benchmarks. These collectors adapt the nursery size to occupy all available space not consumed by the higher generation. Garbage collection time is virtually the same for Appel and Beltway *100.100*. Beltway *100.100.100*, the logical generalization of Appel to 3 generations, enjoys a collection time advantage at the smallest heap size, but has the same performance as Appel and Beltway *100.100* at most heap sizes. There is more variation in the total time results because two programs, _209_db and pseudojbb, are very sensitive to locality effects, which are magnified here by small variations in the collectors. For example, Appel uses a simple boundary crossing write barrier and thus must scan the boot image on each collection [3]. Beltway *100.100* uses the general write barrier described in Section 3.3.1, which records as needed pointers from the boot image.

Most importantly, these results show that Beltway *X.X.100* does



(a) GC time **(b)** Total time

Figure 5: Comparison of the Appel-style collector with Beltway two- and three-generation Appel configurations.

not obtain performance improvements over Appel by simply adding a third generation.

4.2.2 *Incrementality in Generational Collectors*

Incrementality is a key parameter for both generational and Beltway *X.X.100* collectors. This section measures the performance of generational collectors, both with a range of fixed nursery sizes and also with a flexible nursery. It then explores the impact of incrementality on the Beltway *X.X.100* collector.

Figure 6 compares the performance of four configurations of a fixed-size nursery generational collector and an Appel-style collector, with respect to both GC time and total time, using a geometric mean of performance of all 6 benchmarks. The Appel collector's superior space utilization naturally makes good trade-offs between frequency of collection and space utilization, which results in its good total and collector performance.

In contrast, a small fixed-size nursery increases the frequency of GCs and limits time-to-die in the nursery. Both degrade overall performance. A large fixed-size nursery reduces the available space for the higher generation and thus leads to more frequent full heap collections and worse overall performance. Furthermore, the reservation of a fixed proportion of the heap for the nursery significantly impacts the collector's capacity to perform in tight heaps. The lack of results for small heap sizes in Figure 6 illustrates the failure of the generational collector to perform at all in small heap sizes.

These results show that the Appel-style collector is the best performing generational configuration, a result that to our knowledge has not previously appeared in the literature. On the basis of these results, we use the Appel-style configuration, and the best performing fixed-size nursery collector with nursery size of 25% in subsequent comparisons.

4.2.3 *Incrementality in Beltway*

Figure 7 compares the peformance of Beltway *X.X.100* with four different increment sizes. We can see that Beltway *X.X.100* is fairly robust across increment sizes, although the small increment size of 10 degrades performance. This degradation could be attributed to more frequent nursery collections and less time-to-die, or to a diminished capacity to collect large cycles in the second generation. The latter would increase the load on the third generation, and lead to a reduction in heap utilization because it also will increase the copy reserve. We use the *25.25.100* configuration in the remainder of the results section as it appears to perform well, and it is a natural point of comparison with the 25% fixed-size nursery generational collector.

4.2.4 *Beltway* X.X *versus* X.X.100

Figure 8 compares Beltway *25.25* to Beltway *25.25.100* to explore if sacrificing completeness improves performance. However, the geometric means for these two configurations are the same. A few programs do improve slightly using Beltway *25.25*, but _213_javac performance actually degrades because Beltway *25.25* never reclaims a large cyclic garbage structure.

4.2.5 *Garbage Collection Time*

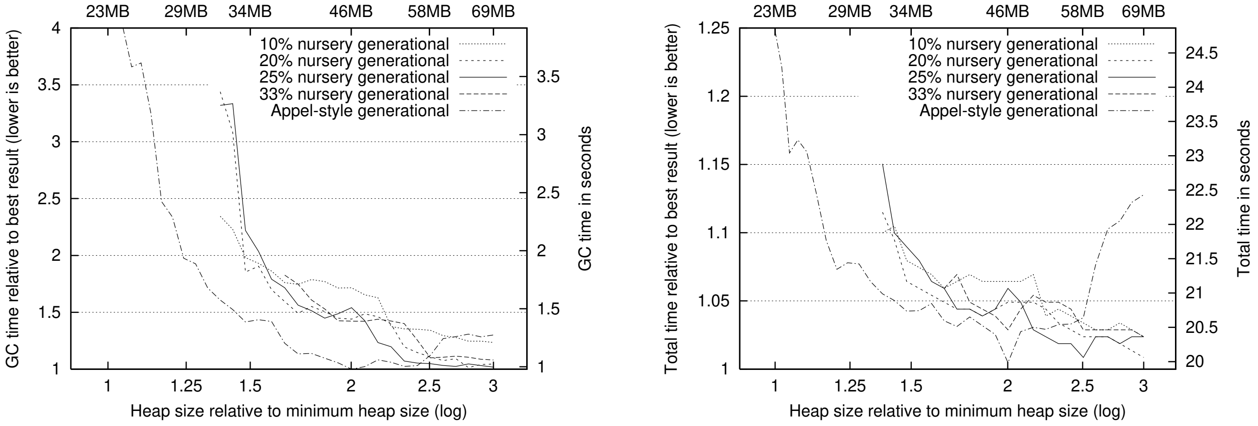
Figure 9(a) shows the geometric mean of the time spent in GC for Beltway *25.25.100*, a fixed-size 25% nursery generational collector, and an Appel-style collector. The robustness of Beltway with respect to heap size is clear. The Appel configuration allows the higher generation to grow as large as possible and so performs better than the fixed nursery configuration, whereas Beltway *X.X.100* exploits the small increment size, dynamic copy reserve, and FIFO behavior in the higher generations to reduce GC overhead substantially in small heaps.

4.2.6 *Total Time*

Figure 9(b) presents the geometric mean of the program execution times for Beltway *25.25.100*, Appel-style generational, and a fixed-size 25% nursery generational collector. Figure 10 shows the results for each benchmark. In general, Beltway improves performance significantly in small to moderate heaps, and performs excellently across all heap sizes.

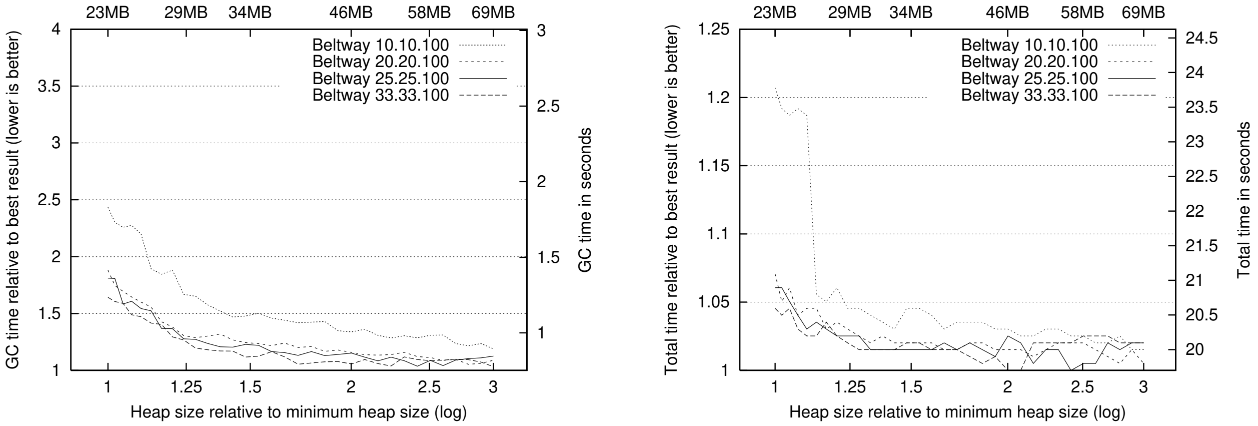
Two interesting results standout. First, Appel performs very poorly in large heaps for pseudojbb because the program thrashes when its nursery becomes too large and spreads out live data too much. Second in _209_db, garbage collection is not a dominant factor. Again, locality effects cause the variations in performance across different heap sizes on all collectors. With the exception of _209_db and jbb, Appel outperforms the fixed nursery collector at all heap sizes and all programs.

Comparing Appel to Beltway *25.25.100* Appel's performance does not match Beltway *X.X.100* until the heap grows to at least 1.5 times the minimum heap size (except for _209_db). In addition, Appel needs at least 2.5 times the minimum heap size for _213_javac and 2 for _228_jack to match Beltway *X.X.100*'s performance. Beltway *X.X.100* uses small and moderate heaps more effectively than Appel and the fixed nursery collector, achieving good performance even when memory is constrained. In _202_jess, _205_raytrace, _213_javac, and _228_jack, Beltway achieves within 5% of the best performance for virtually every configuration. It thus performs well



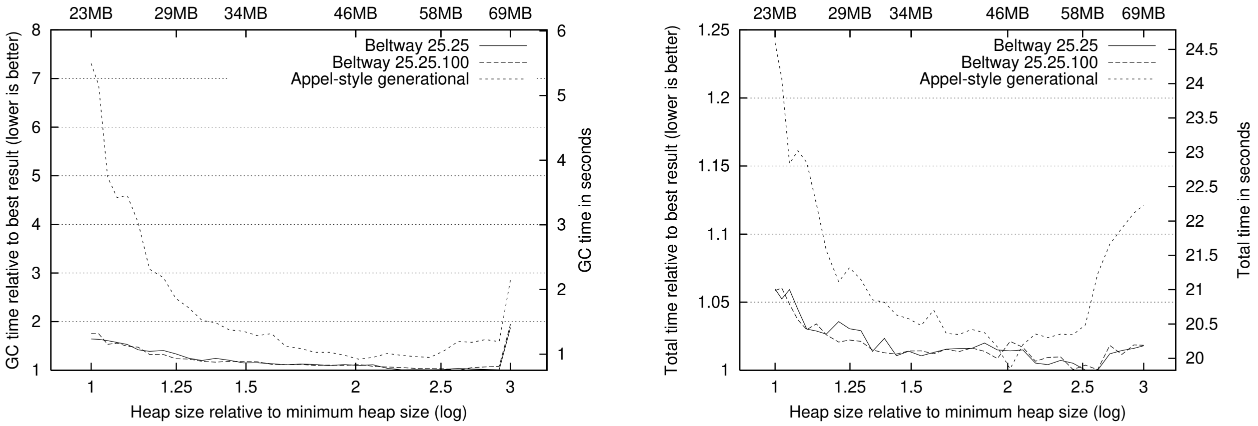
(a) GC time **(b)** Total time

Figure 6: Impact of nursery size on performance of a two-generation collector.



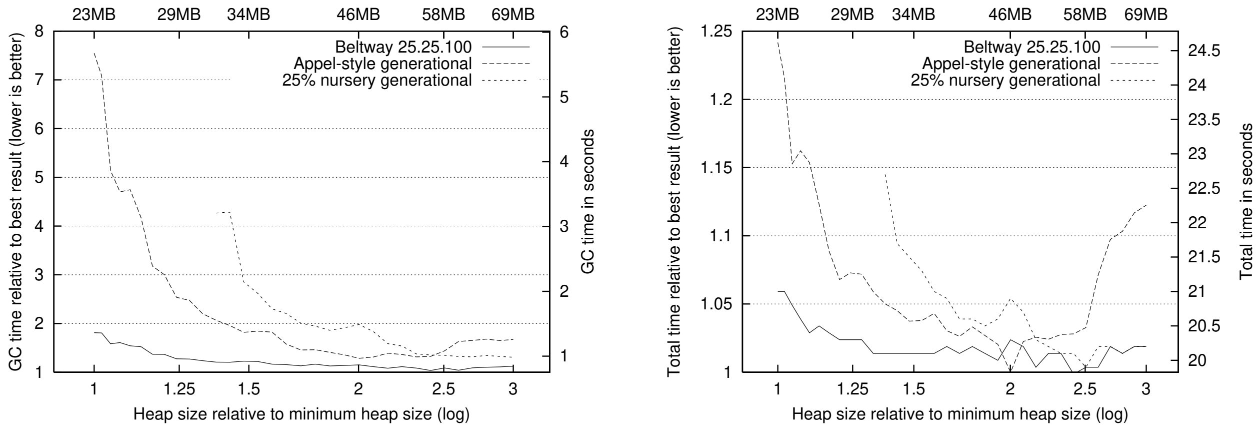
(a) GC time **(b)** Total time

Figure 7: Impact of increment size on performance of Beltway *X.X.100*.



(a) GC time. **(b)** Total time.

Figure 8: Comparisons of Beltway *25.25*, Beltway *25.25.100*, and Appel-style generational relative to best.



(a) GC time. **(b)** Total time.

Figure 9: Performance of Beltway *25.25.100*, Appel-style, and fixed-size 25% nursery generational collectors relative to best.

in a wide variety of circumstances.

4.3 Responsiveness

We did not design Beltway to provide hard real-time performance. However, we did expect that some configurations would offer better responsiveness than other collectors, such as Appel. Simple measures, such as the length of the longest GC pause or a distribution of pause times, do not take into account clustering of GCs, which might prevent mutator progress over a longer period. To our knowledge this issue was first considered by Hölzle and Ungar [22], with respect to pauses caused by dynamic compilation. We follow the methodology of Cheng and Blelloch [13].

To incorporate such possibilities, we measure *mutator utilization* (MU). We define MU to be the fraction of time the mutator runs within an interval $[t_0,t_1)$. Clearly, MU ranges over $[0,1]$ and higher MU means the GC is running less. In our graphs, we present *minimum mutator utilization* (MMU). A point (w,m) lies on an MMU 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. MMU curves are monotonically increasing, with the x-intercept being the maximum GC pause for the run, and the asymptotic y-value being overall throughput (fraction of time spent in the mutator).

Figure 11 shows graphs of MMU curves for runs of _213_javac at two heap sizes. In each graph there are two groups of curves. Those to the left indicate better responsiveness: higher MMU over the same or smaller intervals. On both graphs, Beltway *10.10* and *10.10.100* behave similarly, and offer better responsiveness (and throughput) than other configurations. In the second graph, we see that at larger heap sizes, the maximum pause time is larger (because the increment size is larger, being 10% of the usable heap, etc.), and that Beltway *33.33* and *33.33.100* offer behavior intermediate between *10.10*/*10.10.100* and Appel.

It is clear that some configurations of Beltway offer better responsiveness than others, including Appel. Thus Beltway *can* be adjusted to provide better responsiveness, though we have not yet explored the configuration space fully, or related it to characteristics of various benchmarks, to offer a tuning strategy.

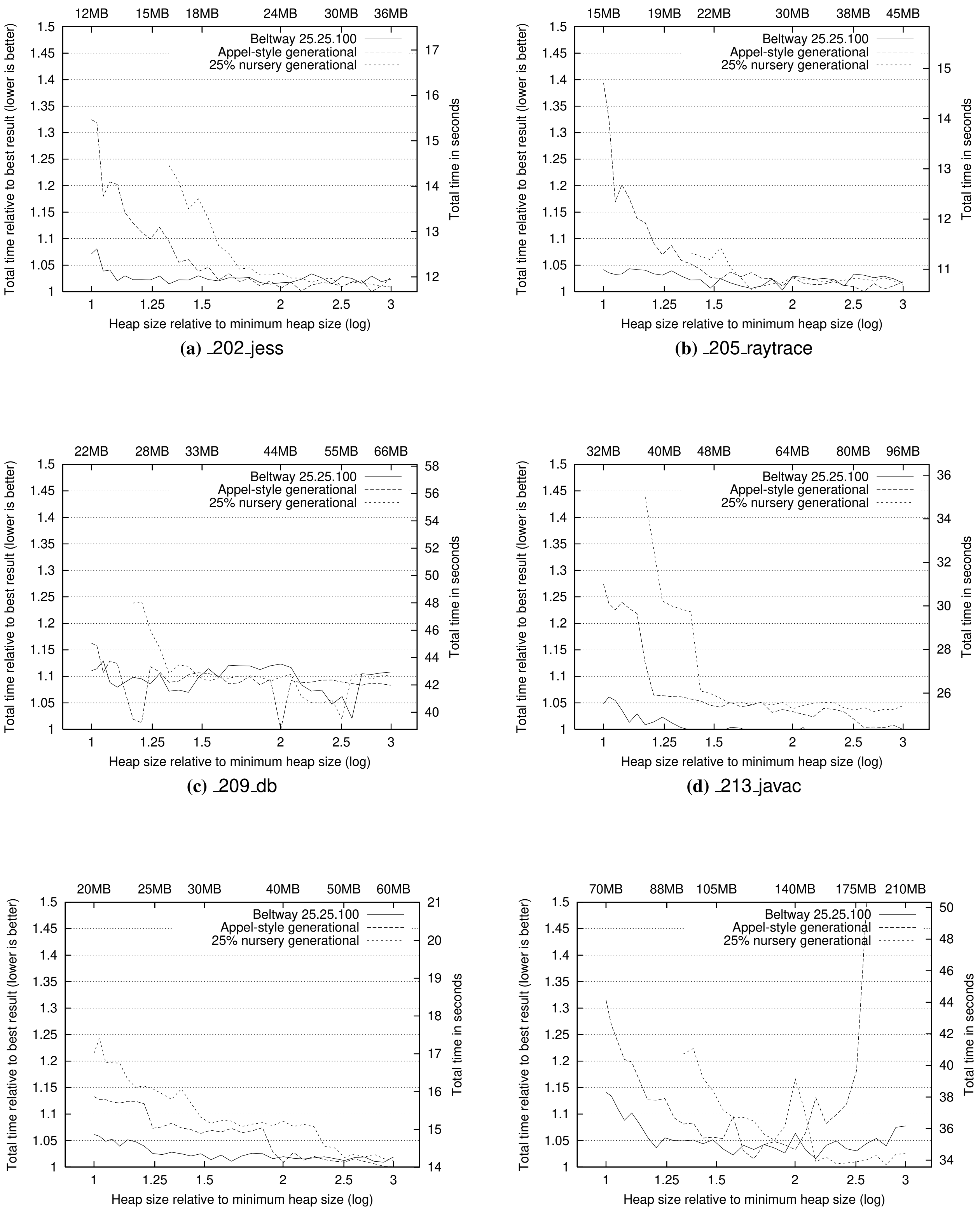
5. Related Work

The Beltway framework combines and exploits key insights of incremental garbage collection [6], segregating objects to different physical regions of the heap in order to improve collector (and sometimes mutator) performance. This section compares Beltway to other collectors with respect to object segregation, pointer tracking, promotion policies, incrementality, completeness, and hybrids.

The most common segregation policy is by age: two or three age-based regions are common, but some collectors use more [27]. Generational age-based collectors exploit the weak generational hypothesis [34]. Older-first collection [32] and renewal older-first (here called older-first mix) [15, 19] are premised upon the just allocated (Older-First) or copied (renewal older-first) objects being likely to stay reachable for a while. Beltway configurations exploit these characteristics with multiple increments on FIFO belts. Beltway *X.X.100* is complete, unlike older-first. More importantly, Beltway generalizes over all these previous collector organizations, and, in addition, supports segregation by object characteristics such as size [21], type [27], or allocation-site (e.g., segregation of long-lived, immortal, or immutable objects) [8, 14], although we do not explore this type of segregation in this paper.

Any references into an increment must be tracked if that increment is to be collected independently. Pointer tracking may use remembered sets [34], card marking [39], hardware support [4, 9, 16, 26], or a combination of techniques. *Card tables* [39] are a common alternative to the remsets we use in Beltway. Card tables trade a fast write-barrier (typically two or three machine instructions) for increased work scanning at collection time. A marked entry in the card table means that one or more pointers were written to some address within the heap range (the *card*) corresponding to this mark; the collector must scan the card to find such pointers and test each one to discover whether it is 'interesting'. Beltway collectors do not use card tables for two reasons. First, Jikes RVM lays out array and scalar objects in different directions in the heap. Thus, the start of one object cannot be determined from the previous object. Second, the performance of card tables or remsets depends strongly on application behavior, and in particular on the relative frequency of writes and remset/card table scanning. Earlier experience [23] suggests that remsets are generally faster.

To give objects more time to die, generational collectors may vary the size of the nursery [3], use an allocation threshold rather than a capacity [40] to trigger collection, or move the boundary between generations to reflect demographic changes [35]. Beltway can use these techniques, but not the 'threatening boundary' technique [5]. To prevent less frequently collected increments from filling prematurely, some collectors further segregate surviving objects by age to mitigate early promotion [40]. For example, collectors can control promotion by recording object ages [36], or by organizing generations into creation and survivor spaces or into bucket brigades [34, 37]. At collection time, these methods must access *every* object in



(a) _202_jess

(b) _205_raytrace

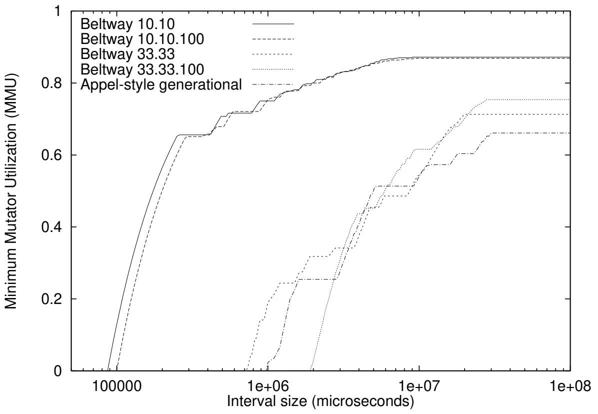
(c) _209_db

(d) _213_javac

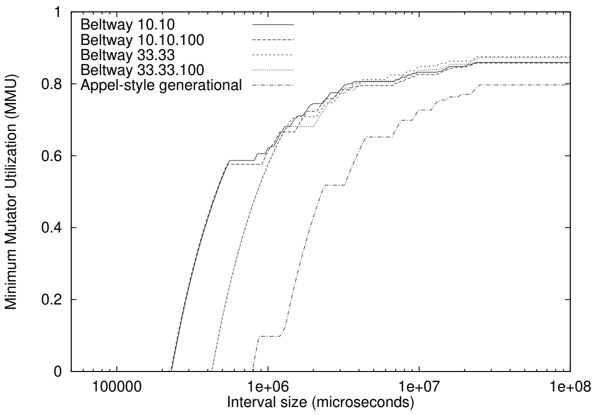
(e) _228_jack

(f) pseudojbb

Figure 10: Execution times for Beltway *25.25.100*, Appel-style, and fixed-nursery generational collectors relative to best.



(a) _213_javac, heap size 83 Mb



(b) _213_javac, heap size 115 Mb

Figure 11: MMU plots for _213_javac at two heap sizes.

the generation, whether to promote it, copy it within the generation, or increment its age. Beltway subsumes and improves on these techniques through multiple increments and time-to-die triggers in a belt. Not only does it prevent premature promotion, it also does not touch objects prematurely.

By collecting one region at a time, region collectors provide incrementality [24]. Generational collectors offer improved expected pause-times, but their need for occasional full-heap collections prevents any worst-case guarantee. The Beltway framework allows investigation of throughput and pause-time tradeoffs. Beltway *X.X* offers incrementality at the expense of completeness, and Beltway *X.X.100* provides completeness at the cost of occasional full-heap collections. One possibility that we leave to future work is adding Mature Object Space [24] copying rules to Beltway so as to obtain completeness without full-heap collections.

It is common for region collectors to manage different regions with different policies or through different managers. For example, large object areas or the oldest generation of a generational collector may be managed by a non-moving collector. A Mature Object Space (MOS) collector handles older generations specially, bounding the volume copied at any collection and offering (eventual) completeness [24]. Some regions may not be managed by a collector at all, either remaining uncollected [8], handled by static analysis [33], or via a stack-like discipline [11, 18, 30]. We could combine Beltway with other collectors, but such exploration is beyond the scope of this paper.

6. Conclusion

We present a new collector design, Beltway, that subsumes previous work on copying collectors. The generality of the Beltway framework enables the implementation of new copying collectors that combine key ideas in the garbage collection literature. We illustrate how Beltway encompasses all of the previous generational and region copying collectors of which we are aware, and identify two new collectors, Beltway *X.X* and Beltway *X.X.100*. The design and implementation of these collectors introduces the need for a number of new mechanisms, including the use of frames to minimize write barrier costs, collection triggers to preempt future collection problems, and a dynamic conservative copy reserve to make the most efficient use of heap space. Using these mechanisms, our results show that Beltway *X.X.100* outperforms both a state-of-the-art Appel-style collector and a fixed-size nursery generational collector. The Beltway framework provides a novel, very general, and efficient design and implementation that results in better collectors, but more importantly opens to further exploration a large design space for copying region collectors.

7. Acknowledgments

We acknowledge with gratitude IBM Research for making the Jikes RVM system available to us, and especially the the generosity Mark Wegman, Vivek Sarkar, Mike Hind, Mark Mergen, and their research teams who built the Jikes RVM and worked with us. Thanks to Darko Stefanović who modified SPEC JBB to produce pseudojbb, and to Emery Berger, Matthew Hertz, Tony Hosking, and Zhenlin Wang for their contributions to this paper.

8. REFERENCES

[1] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeño in Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA '99*, volume 34(10) of *ACM SIGPLAN Notices*, pages 314–324, Denver, Colorado, USA, October 1999. ACM Press.

[2] Bowen Alpern, Dick Attanasio, John J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn Hummel, D. Lieber, V. Litvinov, Mark Mergen, Ton Ngo, J. R. Russell, Vivek Sarkar, Manuel J. Serrano, Janice Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1), February 2000.

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

[4] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

[5] David A. Barrett and Benjamin Zorn. Garbage collection using a dynamic threatening boundary. Computer Science Technical Report CU-CS-659-93, University of Colorado, July 1993.

[6] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, MIT Laboratory for Computer Science, May 1977. Technical report MIT/LCS/TR–178.

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

[8] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinley, and J. Eliot B. Moss. Pretenuring for Java. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001*, volume 36(11) of *ACM SIGPLAN Notices*, pages 342–352, Tampa, Florida, USA, November 2001. ACM Press.

[9] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

[10] Tim Brecht, Eshrat Arjomandi, Chang Li, and Hang Pham. Controlling garbage collection and heap growth to reduce the execution time of Java applications. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001*, volume 36(11) of *ACM SIGPLAN Notices*, Tampa, Florida, USA, November 2001. ACM Press.

[11] Dante Cannarozzi, Michael Plezbert, and Ron Cytron. Contaminated garbage collection. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2001*, volume 35(5) of *ACM SIGPLAN Notices*, Vancouver, Canada, May 2000. ACM Press.

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

[13] Perry Cheng and Guy Belloch. A parallel, real-time garbage collector. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2001*, volume 36(5) of *ACM SIGPLAN Notices*, pages 125–136, Snowbird, Utah, USA, May 2001. ACM Press.

[14] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '98*, volume 34(5) of *ACM SIGPLAN Notices*, Montreal, Canada, May 1998. ACM Press.

[15] William D. Clinger and Lars T. Hansen. Generational garbage collection and the Radioactive Decay model. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '97*, volume 32(5) of *ACM SIGPLAN Notices*, pages 97–108, Las Vegas, Nevada, USA, May 1997. ACM Press.

[16] Robert Courts. Improving locality of reference in a garbage-collecting memory management-system. *Communications of the ACM*, 31(9):1128–1138, 1988.

[17] Sylvia Dieckman and Urs Hölzle. A study of the allocation behaviour of the SPECjvm98 Java benchmarks. In Erik Jul, editor, *Proceedings of the 12th European Conference on Object-Oriented Programming, ECOOP'98*, volume 1445 of *Lecture Notes in Computer Science*, pages 92–115, Brussels, Belgium, 1998. Springer-Verlag.

[18] David Gay and Alex Aiken. Memory management with explicit regions. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'98*, volume 33(5) of *ACM SIGPLAN Notices*, pages 313–323, Montreal, Canada, May 1998. ACM Press.

[19] Lars Thomas Hansen. *Older-first garbage collection in practice*. PhD thesis, North-eastern University, November 2000.

[20] Barry Hayes. Using key object opportunism to collect old objects. In Andreas Paepcke, editor, *Proceedings of the Sixth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA'91*, volume 26(11) of *ACM SIGPLAN Notices*, pages 33–46, Phoenix, Arizona, USA, October 1991. ACM Press.

[21] Michael Hicks, Luke Hornof, Jonathan T. Moore, and Scott Nettles. A study of Large Object Spaces. In Richard Jones, editor, *Proceedings of the First International Symposium on Memory Management, ISMM '98*, volume 34(3) of *ACM SIGPLAN Notices*, pages 138–145, Vancouver, Canada, March 1999. ACM Press.

[22] Urs Hölzle and David Ungar. A third-generation Self implementation: Reconciling responsiveness with performance. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA'94*, volume 29(10) of *ACM SIGPLAN Notices*, pages 229–243, Portland, Oregon, USA, October 1994. ACM Press.

[23] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In Andreas Paepcke, editor, *Proceedings of the Seventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA'92*, volume 27(10) of *ACM SIGPLAN Notices*, pages 92–109, Vancouver, Canada, October 1992. ACM Press.

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

[25] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[26] David A. Moon. Garbage collection in a large LISP system. In Guy L. Steele, editor, *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, Austin, Texas, USA, August 1984. ACM Press.

[27] John H. Reppy. A high-performance garbage collector for Standard ML. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ, December 1993.

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

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

[30] Bjarne Steensgaard. Thread-specific heaps for multi-threaded programs. In Antony L. Hosking, editor, *Proceedings of the Second International Symposium on Memory Management, ISMM 2000*, volume 36(1) of *ACM SIGPLAN Notices*, Minneapolis, Minnesota, USA, January 2001. ACM Press.

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

[32] Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. Age-based garbage collection. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA'99*, volume 34(10) of *ACM SIGPLAN Notices*, pages 370–381, Denver, Colorado, USA, October 1999. ACM Press.

[33] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, February 1997.

[34] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

[35] David M. Ungar and Frank Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, 1992.

[36] Derek White and Alex Garthwaite. The GC interface in the EVM. Technical Report SML TR–98–67, Sun Microsystems Laboratories, December 1998.

[37] Paul R. Wilson. A simple bucket-brigade advancement mechanism for generation-based garbage collection. *ACM SIGPLAN Notices*, 24(5):38–46, May 1989.

[38] Paul R. Wilson. Garbage collection and memory hierarchy. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St Malo, France, September 1992. Springer-Verlag.

[39] Paul R. Wilson and Thomas G. Moher. A card-marking scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *ACM SIGPLAN Notices*, 24(5):87–92, 1989.

[40] Benjamin G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, March 1989. Technical Report UCB/CSD 89/544.