Comparison of Garbage Collectors Operating in a Large Address Space

Sergiy Kyrylkov*
SA Consulting
Khmelnytsky, Ukraine 29000
Email: mail@sergiy.kyrylkov.name

Darko Stefanović
Department of Computer Science
MSC01 1130
1 University of New Mexico
Albuquerque, NM 87131–0001
Email: darko@cs.unm.edu

April 1, 2005

Abstract

We analyze the performance of several copying garbage collection algorithms in a large address space offered by modern architectures. In particular, we describe the design and implementation of the *RealOF* garbage collector, an algorithm explicitly designed to exploit the features of 64-bit environments. This collector maintains a correspondence between object age and object placement in the address space of the heap. It allocates and copies objects within designated regions of memory called *zones* and performs garbage collection incrementally by collecting one or more ranges of memory called *windows*. The windows are managed so as to collect middle-aged objects, rather than almost always collecting young objects, as with a generational collector. The address-ordered heap allows us to use the same inexpensive write barrier that works for generational collectors. We show that for server applications this algorithm improves throughput and reduces heap size requirements over the best-throughput generational copying algorithms such as the Appel-style generational collector.

1 Introduction

Server-side 64-bit computing today is characterized by very large physical memory support, very large application virtual address spaces, and 64-bit integer computation using 64-bit general-purpose registers. In such systems, an application's virtual address space is measured in terabytes and an increasing number of programs can exploit this opportunity. Database servers use a large address space for scalability, maintaining buffer pools, caches, and sort heaps in memory to reduce

*Work done while the first author was at the University of New Mexico.

the volume of I/O they perform. Simulation and other computationally intensive programs benefit from keeping much larger arrays of data entirely in memory. Finally, another large group of programs, application servers, has been deployed on 64-bit platforms for some time now.

Some of these applications heavily rely on Java technology, and this has forced leading companies like IBM, Sun, and BEA to introduce 64-bit versions of their Java Virtual Machines.[1] As a result, 64-bit computing introduces a new set of research opportunities in the field of virtual machines related both to evaluating previously existing 32-bit solutions in the 64-bit world and inventing brand-new approaches that specifically exploit the benefits of 64-bit architectures.

Server-side applications tend to have a very high heap object allocation rate. When the heap is full, garbage collection must free some space in it to allow the application to continue running. Concurrent collectors can be used for the old generation of generational collectors. However, employing a concurrent collector as the only collector, in order to completely remove garbage collection pauses, may not be acceptable under the prevailing circumstances today, viz., server systems with just one or two processors, as such collectors tend to reduce throughput significantly. Our experimental results are obtained on a system of this kind, an Apple G5. For such systems "stop-the-world" garbage collection remains a viable option as long as the collection pauses are reasonably short.

Previously, we proposed an older-first garbage collector [SMM99], which differs from generational collectors in that it does not always collect the youngest data along with the older data. Similar to generational collectors, it relies only on relative object age, deduced from object position in the heap, to make decisions about which sets of objects to collect. As described, and as implemented in the present paper, it does not take advantage of static analysis [HDH03] or profiling-based heuristics [BSH+01, ISF03], though it could. In our earlier work, we demonstrated that an emulation of this algorithm in a 32-bit address space can have good performance [SHB+02]. Here for the first time we have an implementation of the algorithm as originally envisaged, in a large address space (except that special treatment of permanent data ("pretenuring") is still lacking). As the results will show, excellent throughput results are achieved with this algorithm, especially in tight heaps where it matters the most.

2 Background

The conceptual design of the RealOF collector has been described fully elsewhere [SMM99, Ste99]; here for completeness we sketch the main points. A traditional generational garbage collector always collects a youngest subset of heap objects (i.e., some number of youngest generations). An *older-first* collector, on the other hand, chooses a middle-aged subset of heap objects. Imagine a heap *logically* laid out with objects in the order of their age: this is depicted in Figure 1, with the oldest objects on the right, and the most recently allocated objects on the left. The older-first collector chooses to collect a subset C, called the collection window, that is immediately to the left of the survivors of the previous collection. The current collection's survivors S are left in place (logically). After a collection, an amount of free space $|C - S|$ is available for new allocation. In

[1] We survey commercial 64-bit JVM platforms elsewhere [Kyr05].

this logical view, the heap remains laid out in age order, so the free space shows up on the far left, where it is available for new allocation (green arrows). Thus, the window sweeps the heap from older to younger, hence the name older-first. Initially, objects fill the entire heap and the window is positioned at the old end of the heap. Eventually the window reaches the young end; after collecting the young end of the heap, the window is reset to the old end.

Figure 1 shows a series of eight collections, and indicates how the window moves across the heap when the collector is performing well. If the window is in a position that results in small survivor set sizes $|S|$ (Collections 4–8), the window moves by only that small amount from one collection to the next. As the window moves slowly, it remains for a long time in the same logical region. In a copying-collector implementation, this means that a great deal of allocation, $|C - S|$, takes place with little copying work, $|S|$; in other words, that the performance is good. The reason why this behavior might be expected to arise in some programs is that the position of the window in the heap corresponds to object age, and it has been observed that object lifetimes tend to cluster around a few dominant values. Whilst a generational collector takes advantage of the cluster around zero ("most objects die young"), the older-first collector may take advantage of middle-aged lifetime clusters.

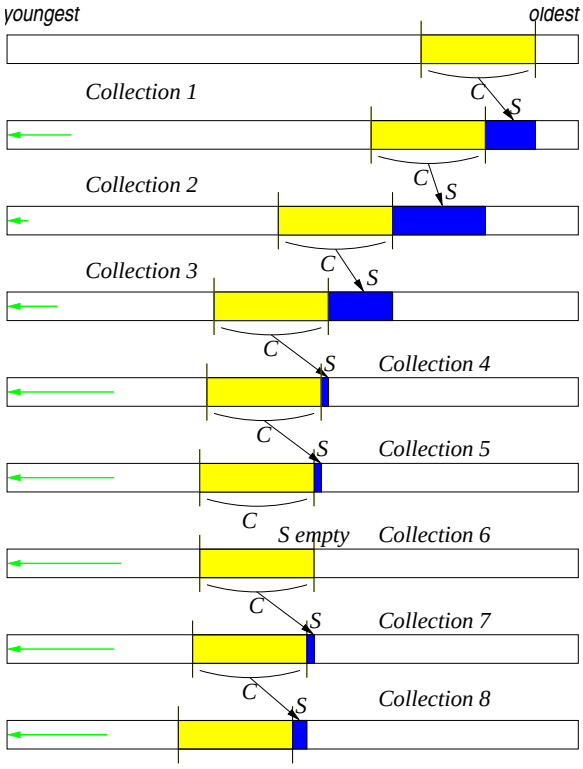

Figure 1: Older-first window motion example.

Like generational collectors, the older-first collector collects less than the entire heap each time, and thus it must maintain a write barrier and remember certain pointer updates. The general rule is that when a store creates a reference $p \rightarrow q$, we need to remember it only if q might be

collected before *p*. Figure 2 illustrates this rule applied to the older-first collector. The crossed-out pointers need not be remembered. It might seem complicated to apply this rule in a write barrier. However, if a large address space is available, objects can be laid out in age order, as we detail in Section 3. The allocation starts into highest addresses in an allocation zone, and copying is into lower addresses in another zone. Once the allocation zone is exhausted, the copying zone becomes the new allocation zone, and another chunk of address space is made into the new copying zone. In a large address space we can do this for a very long time. Now the rule for the write barrier filtering is little more than an address comparison, as shown in Figure 3, the same as in efficient write barriers of generational collectors. Here for the first time we present a complete implementation of the older-first collector in a large address space, which we now label RealOF. Previously we reported a 32-bit implementation that, in the absence of a large address space, resorted to indirection through an age lookup table in order to resolve write barriers [SHB+02]; we label it OF in the present paper.

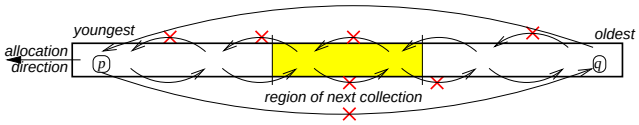

Figure 2: Directional filtering of pointer stores: crossed-out pointers need not be remembered.

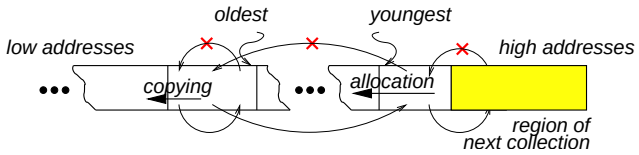

Figure 3: Directional filtering with an address-ordered heap.

3 Implementation

3.1 Infrastructure

Our implementation framework is the Jikes Research Virtual Machine (Jikes RVM), developed by IBM Research [AAB+00, AAB+99], an open-source virtual machine capable of running a wide variety of Java programs. It offers two compilers, baseline and optimizing, but uses no interpreter. We built our infrastructure in stages, as follows. We ported Jikes RVM version 2.0.3, with the baseline compiler alone, to the 64-bit PowerPC/AIX platform [Kyr03, KSM04], and then we extended this port to the PowerPC/Linux architecture and specifically the Apple G5. This gave us the only 64-bit open-source virtual machine (at the time) that provided a flexible test-bed for implementing new memory managements algorithms, owing to the presence in Jikes RVM of the easily pluggable Garbage Collection Toolkit GCTk (now MMTk [BCM04]).[2] The GCTk already

[2] http://www.cs.umass.edu/~gctk/

contained fast implementations of generational copying collection algorithms, as well as of the OF and Beltway collectors; to this we added our implementation of the RealOF collector (and allocator). Using this system, we obtained encouraging preliminary performance results [KS05]. We then ported the Jikes RVM optimizing compiler to the 64-bit PowerPC/Linux architecture. It is in this system that the results we report below were obtained. (In the meantime, we have contributed our work to the effort, led by Kris Venstermans of the University of Ghent and David Grove of IBM, to port the newest version of Jikes RVM to 64-bit PowerPC/Linux.)

3.2 Collector and allocator

The implementation of the RealOF algorithm supports the notions of zones and windows as described above, but they are of necessity discretized in size and tied into the functioning of the allocator. A *zone* is a contiguous region of memory and the largest logical memory unit of the RealOF collector. All zones are of equal, power-of-2 size (in our experiments, 8 GB), and are allocated from higher to lower addresses in order to maintain the address-order heap. At any moment in time the algorithm has two zones: the *allocation zone* and the *copy zone*. Newly created objects are placed in the allocation zone, from higher addresses to lower. During a garbage collection, survivors are placed in the copy zone, from higher addresses to lower.

A zone consists of a number of *windows*. A window is a contiguous, power-of-2 size, region of memory, smaller than a zone, allocated within a particular zone from higher to lower addresses. In our implementation a window is the smallest unit of memory allocation and deallocation. Thus every garbage collection increment collects exactly one window. The size of a window is limited from below by the minimum size of mappable virtual memory, which in our operating system is 4 MB.

The RealOF allocator is a relatively simple and fast bump-pointer allocator, attached to the current allocation window. The allocator actually implements allocation in either direction, but our experiments have shown, somewhat surprisingly, that there is no performance difference between the two. If a particular architecture supports hardware data prefetching such as within cache lines, consistent with a lower-to-higher access order, we might expect to suffer a performance hit allocating objects from higher to lower addresses. In reality, there is none (on our PowerPC system). Note that with either direction of allocation the object layout remains the same, with array objects laid out from lower to higher addresses and scalar objects laid out from higher to lower addresses [AAB+99] and the address access pattern of the object initialization sequence thus remains unaffected. There is apparently no observable memory system effect spanning multiple consecutively allocated objects.

We illustrate the progress of the algorithm in Figure 4. At the onset of virtual machine execution, both the allocation and the copy zone are empty. We allocate the very first window inside the allocation zone and start placing newly created objects inside this window using our simple bump pointer allocator. When the first window fills up, we allocate another window inside the allocation zone and proceed without garbage collection (1). The first garbage collection happens when the number of windows in the heap becomes equal to the maximum allowed number of windows:

$$windows = \frac{heap\_size}{window\_size} - 1,$$

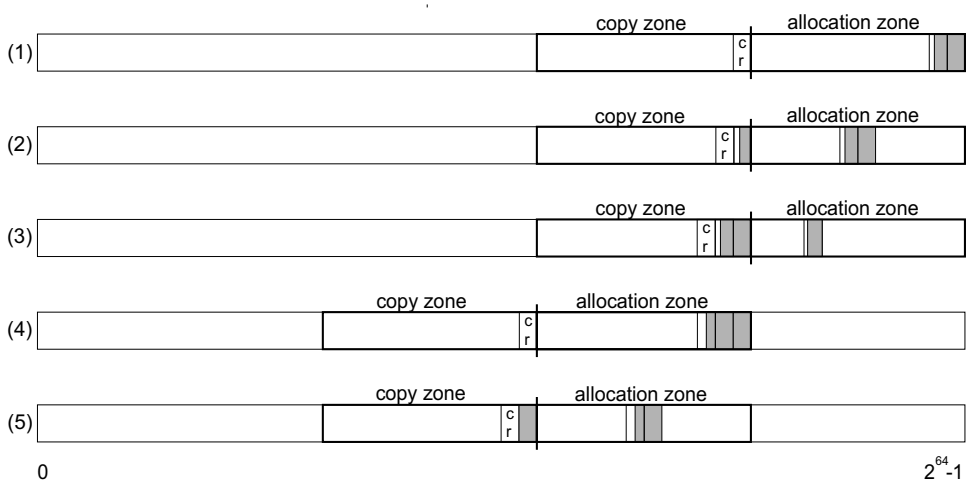

Figure 4: The RealOF algorithm shown operating with three windows in the heap (completely or partially gray rectangles) and one copy reserve window (cr). Allocation proceeds from higher addresses to lower (right to left); similarly, copying into the copy zone proceeds from higher addresses to lower. Snapshots, top to bottom: (1) before any GC occurred; (2) after several GCs, with most windows residing in the allocation zone; (3) after several more GCs, with most windows residing in the copy zone; (4) right after a zone reset; (5) after several GCs following a zone reset, with most windows residing in the allocation zone.

where *heap_size* is rounded down to fit an integer number of windows and 1 accounts for the copy reserve window. The maximum number of windows in the heap is maintained as an invariant of the algorithm. At every garbage collection increment, we collect the highest (rightmost) window in the heap, copy its survivors to a window allocated in the copy zone, and deallocate the collected window (2). If the ratio of surviving objects is relatively high, in order to satisfy the allocation request we may need to perform several garbage collection increments and collect more than one window in the allocation zone, creating additional windows in the copy zone.

At some point, the number of windows in the copy zone may become larger than the number of windows in the allocation zone (3) and eventually all allowed windows may end up in the copy zone. This situation is called *zone reset*. When the zone reset occurs we rebind the current copy zone to function as the new allocation zone and create a new copy zone right below the old one (4). Note that here we have a situation similar to the one before the first garbage collection, when all the windows reside in the allocation zone. After the zone reset we can proceed as described before (5).

Another possible situation is the exhaustion of the allocation zone. When this situation is detected we perform several garbage collection increments to deallocate all windows from the allocation zone, which in turn triggers the zone reset mechanism.

In the unlikely event that we reach the bottom of available address space we perform a full-heap garbage collection and move all live data back to the highest end of the address space. We describe full-heap collection below.

3.2.1 Write Barrier

By using the heap in address order we are able to use the same inexpensive write barrier as the one used in traditional generational collectors [BM02], conceptually defined by this code:

```
public static final void
writeBarrier
(ADDRESS source, ADDRESS target) {
  if (source < ((target >>> WINDOW_SIZE_LOG)
                        << WINDOW_SIZE_LOG)) {
    GCTk_WriteBufferSlot.insert(source);
  }
}
```

The optimizing compiler translates this Java phrase into an efficient three-instruction sequence on the PowerPC, a mask, a compare, and a conditional branch [SHB+02]. As we show later, the address-order write barrier gives a consistent performance improvement over the indirect write barrier used in the previous implementation of the older-first algorithm [SHB+02], which performed table lookups to map object addresses in a small address space to logical object ages.

3.2.2 Remembered Sets

We map windows to their corresponding remembered sets by extracting the remembered set number directly from the target address:

```
public static final void
conditionalRemsetInsert
(ADDRESS source, ADDRESS target) {
  if (source < ((target >>> WINDOW_SIZE_LOG)
                        << WINDOW_SIZE_LOG)) {
    GCTk_RememberedSet.insert((int)((target
      << (BITS_IN_ADDRESS - ZONE_SIZE_LOG - 1))
     >>> (BITS_IN_ADDRESS - ZONE_SIZE_LOG - 1 +
          WINDOW_SIZE_LOG)), source);
  }
}
```

In order for this approach to work, the number of remembered sets must be equal to:

$$remsets = 2 \times \frac{zone\_size}{window\_size}$$

In order to keep the overhead of remembered sets relatively low, it is beneficial to have as few remembered sets as possible. This can be achieved by always keeping the window size as large as

possible for a particular heap size. On the other hand, having large windows hurts incrementality; hence a large window may not always be a good solution. Another way to keep the overhead of the remembered sets relatively low may be remembered-set triggered GC, wherein a GC increment is performed if the size of the remembered sets reaches some upper threshold.

3.3 Full Heap Collection

We implemented a full-heap collection mechanism in order to make the RealOF algorithm complete. The mechanism is in principle necessary when the algorithm runs out of address space and needs to move all data from the lowest-addressed zone back to the top of the address space; this, however, will happen only in *extremely* long-running programs. The mechanism can also be invoked to honor `System.gc()` hints from the application. Lastly, it may be invoked adaptively, if high survival rates suggest the presence of garbage cycles spanning multiple windows. However, we do not have adaptive heuristics worked out yet, none of the tested programs comes close to exhausting the address space, and, for fair comparison, we ignore `System.gc()` hints, as they are ignored by the remaining collectors implemented in GCTk. (If we do honor them, the minimum heap size requirement for the RealOF algorithm executing the _213_javac benchmark decreases from 88 MB to 56 MB, compared with 64 MB for both Appel-style generational and Beltway collectors.)

In a system with the full-heap collection mechanism we use a third type of zone called the *reserved zone*, Figure 5. The reserved zone serves as a place-holder for a temporary copy zone in the event of full-heap collection, and following full-heap garbage collection it becomes the new allocation zone. At other times it is neither used nor mapped into the address space of the process.

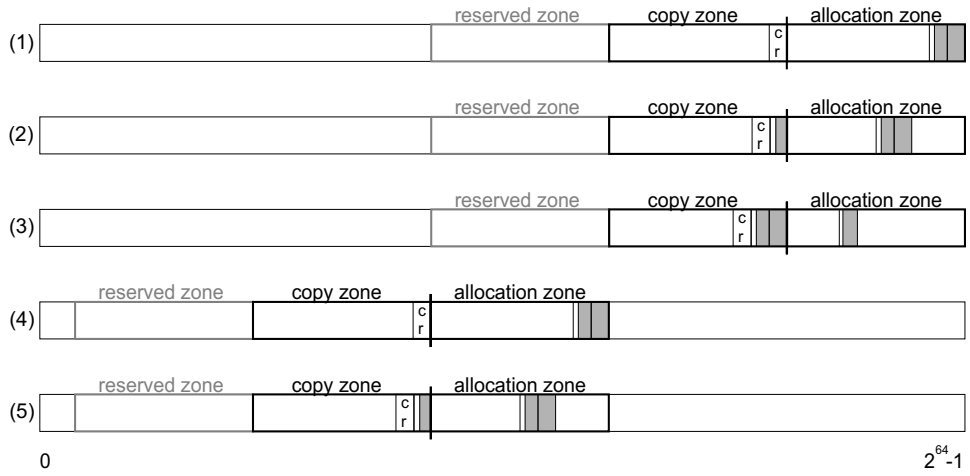

Figure 5: Full heap collection in the RealOF algorithm with three windows in the heap (completely or partially gray rectangles) and one copy reserve window (cr). Snapshots, top to bottom: (1) before any GC occurred; (2) after several GCs, with most windows residing in the allocation zone; (3) after several more GCs, with most windows residing in the copy zone; (4) right after a full heap collection; (5) after several GCs following a full heap collection, with most windows residing in the allocation zone.

4 Results

4.1 Experimental Setting

We use the Jikes RVM optimizing compiler both to build the boot image (the virtual machine itself) and to compile the application code at run-time. (Note that run-time compilation activity is included in the reported results.) We use the so-called *Fast* configuration, which skips assertions checks and pre-compiles *all* the classes of the virtual machine into the boot image. Our hardware platform is an Apple G5, with a PowerPC 970 processor at 2 GHz and 2 GB of memory, running an early-beta version of the 64-bit Yellow Dog Linux 3.0.1 for G5 with the 2.6.1 kernel.

Our benchmark programs are the GC-relevant programs from SPECjvm98 [Sta99, DH98], and SPECjbb2000 [Sta01]. Some characteristics of the benchmarks are summarized in Table 1. We assume that SPECjvm98 programs are representative of short-running client applications, whereas SPECjbb2000 is representative of server applications.

Benchmark	Description	Minimum Heap		Maximum Heap		Total Allocation, MB
		Size, MB	GCs	Size, MB	GCs	
SPECjvm98 jess	Java Expert System Shell	24	443	144	17	956
SPECjvm98 db	A database simulation program	40	150	96	16	386
SPECjvm98 javac	Java compiler from JDK 1.0.2	64	227	256	15	1365
SPECjvm98 jack	A Java parser generator	32	347	160	16	930
SPECjbb2000 - 1	Emulates a 3-tier system with 1 warehouse	96	862	640	28	4928–6292
SPECjbb2000 - 2	Emulates a 3-tier system with 2 warehouses	128	926	640	42	4011–6039
SPECjbb2000 - 4	Emulates a 3-tier system with 4 warehouses	196	783	640	49	3732–5707
SPECjbb2000 - 8	Emulates a 3-tier system with 8 warehouses	352	543	640	83	3785–5324

Table 1: Benchmark information including the number of garbage collections performed by the Appel-style collector.

For the SPECjvm98 benchmarks, our performance metric is the running time of the program. For SPECjbb2000, however, the SPEC benchmarking procedure fixes the running time, and the benchmark itself reports the measured throughput as the number of transactions per second, so this is our performance metric. (Because the amount of useful work varies with the efficiency of collection and in turn on heap size, the Total Allocation column of Table 1 contains ranges of values in the SPECjbb2000 rows.)

The garbage collectors compared are an Appel-style two-generation collector [App89] (labelled 2G in plots); the Beltway collector [BJMM02] in its default 25.25.100 configuration; the older-first collector, with the indirect write barrier and window size of 25% of the heap [SHB+02] (labelled OF in plots); and the RealOF collector, described in Section 3.

We ran each benchmark and each collector with a wide range of heap sizes, careful to include the smallest heaps in which the programs are able to complete. Each such configuration was run three times, and for final results we report the best run. (We intend to repeat the experiments and get a proper characterization of variance; it appears to be small.)

4.2 Running Time and Throughput

In Figure 6 we show the total execution time of the several garbage collection algorithms as the heap size is varied. The Appel-style collector is known to provide excellent throughput (i.e., to have low GC overhead measured as fraction of total execution time), and therefore we also provide Figure 7 in which the total execution times of each collector are divided by the total execution time of the Appel-style collector.

Consistent with our expectations, using a fast write barrier makes RealOF uniformly faster than OF across all benchmarks. We now examine how RealOF behaves for different benchmarks. Somewhat surprisingly, in some cases (notably SPECjvm98 jess) after the heap size becomes relatively large for a particular benchmark, the performance of RealOF begins to slightly decrease. We have determined that this happens because after some point the cost of processing increasingly large numbers of remembered pointers outweighs the benefits of a larger heap (and, with a fixed window size, the total number of pointers remembered for all windows grows in rough proportion to the number of windows, i.e., heap size).

From the measurements of RealOF with different window sizes in SPECjvm98 , we conclude that, in general, larger window size leads to better performance, as soon as the larger window size is feasible. There are two reasons for this. First, for smaller window sizes we have to invoke garbage collection more frequently and the total cost of invoking several smaller garbage collections is at least as high or higher than the cost of invoking one larger collection.[3] Second, collecting a bigger window we are able to free more space. Since one bigger window encompasses two, four, etc. smaller windows, some inter-window pointers (a burden on remembered sets) turn into intra-window pointers (no cost), resulting in diminished pointer processing time and a reduction of garbage unnecessarily retained. Indeed, we find that having four or five windows in the heap gives the best results, consistent with the 15–25% estimates for the optimal window to heap ratio from our previous work.

Thus, it appears that in some cases it would be beneficial to have a simple adaptive window resizing mechanism. It would be responsible for setting an initial window size for a given heap size, and for switching to the largest possible window size during execution when the heap is resized dynamically. This incurs the one-time cost of reorganizing the remembered sets, which could be piggybacked onto a garbage collection, and the cost of adaptively changing the write barrier code.[4]

Overall, other than for SPECjvm98 jess, there are configurations of RealOF that are either better or about as good as the Appel-style generational collector. We have separately carried out

[3] Here we are not concerned with pause times but only with collector throughput performance.

[4] In the Jikes RVM baseline compiler, this is simply a matter of recompiling one method; in the optimizing compiler, which normally inlines the write barrier, we must either recompile all methods (unwieldy and expensive) or code the write barrier using variable-amount shift instructions instead of fixed-amount shift instructions (both available on the PowerPC) and dedicate a register to hold the shift amount, i.e., the logarithm of window size.

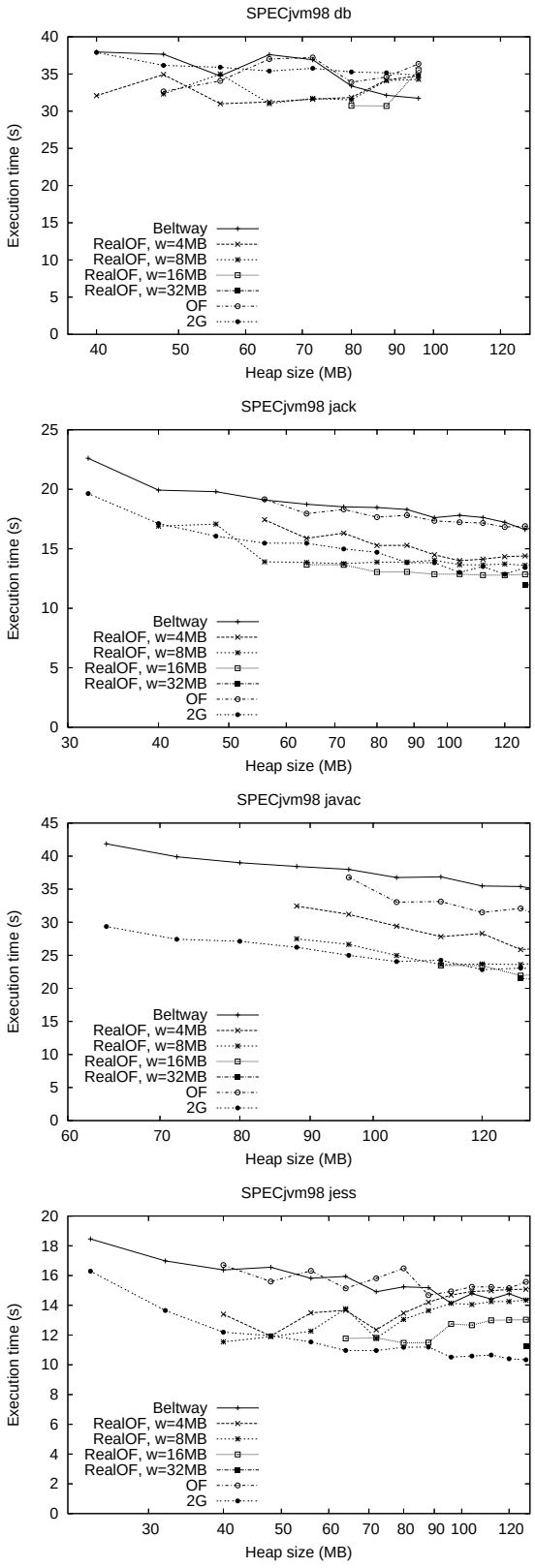

Figure 6: Absolute execution time for SPECjvm98 benchmarks for different garbage collectors. Lower is better.

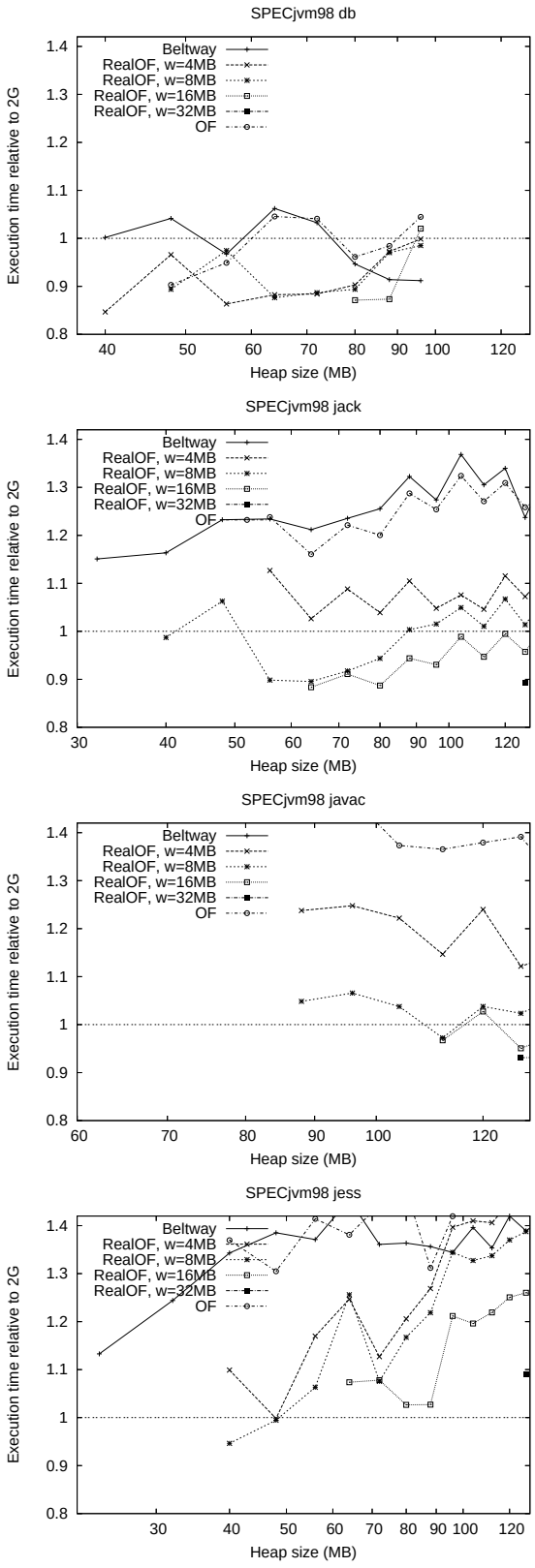

Figure 7: Relative execution time for SPECjvm98 benchmarks for different garbage collectors (shown relative to the Appel-style collector, 2G). Lower is better.

heap profiling studies of SPECjvm98 jess using exact tracing [HBM+02], and we know that the large amounts of permanent data allocated in that program hurt the performance of RealOF.

Turning to the SPECjbb2000 runs, Figure 8 and Figure 9, we see that in tight heaps the RealOF algorithm tends to have the highest throughput overall among all collectors tested. Not only RealOF significantly decreases the minimum heap requirements (Figure 8), especially in the 8 "warehouses" case (224 MB with RealOF vs 352 MB with Appel-style and Beltway), but also provides very good improvement in throughput (Figure 9), up to 2.4 times in the 4 "warehouses" configuration.

The overall performance benefit of the RealOF collector over other collectors tends to slightly increase with the number of "warehouses", so that in the SPEC-standard configuration with 8 warehouses RealOF collector performs better than Appel-style in the whole range of heaps.

In summary, RealOF shows performance competitive with generational collection on observed client-side programs, and significantly better on server-side programs.

5 Concluding remarks

Our results demonstrate that for Java server applications, a large address space with an equitable, fast write barrier confers a clear performance advantage on the older-first algorithm over traditional generational collectors. Importantly, the advantage is most pronounced for small heap sizes.

In previous work we showed that the distribution of pause times incurred by the OF collector is favorable [SHB+02]. The differences introduced in the RealOF implementation should not affect the pause time distribution, but this remains to be confirmed experimentally.

However, remembered set maintenance and permanent data remain a potential weak spot of the algorithm that can hurt its performance on some programs. Therefore in our current and future work (using the successor to GCTk, MMTk, included in newer releases of JikesRVM) we are investigating remembered set triggers and hybrid models in which the basic idea of the algorithm will be combined with recent advances in object lifetime prediction and allocation-time or collection-time pretenuring.

6 Acknowledgments

We are grateful to IBM Research for developing Jikes RVM and making it available as an open-source product. We are also very grateful to Eliot Moss for his advice and help during the implementation of the 64-bit PowerPC/AIX port of Jikes RVM 2.0.3.

This material is based upon work supported by the National Science Foundation (grants CCR-0219587, CCR-0085792, EIA-0218262, EIA-0238027, and EIA-0324845), the Defense Advanced Research Projects Agency (grant F30602-02-1-0146), Microsoft Research, and Hewlett-Packard (gift 88425.1). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsors.

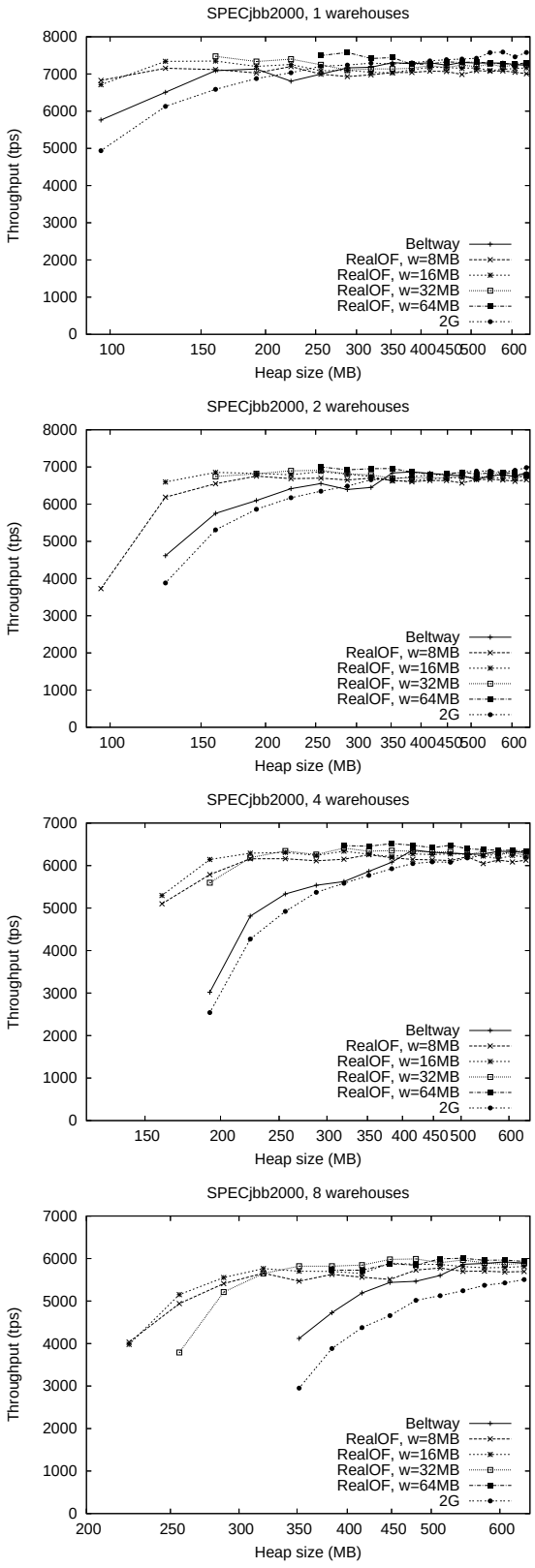

Figure 8: Absolute throughput for the SPECjbb2000 benchmark for different garbage collectors. Higher is better. The four plots correspond to benchmark scale: 1, 2, 4, and 8 “warehouses”.

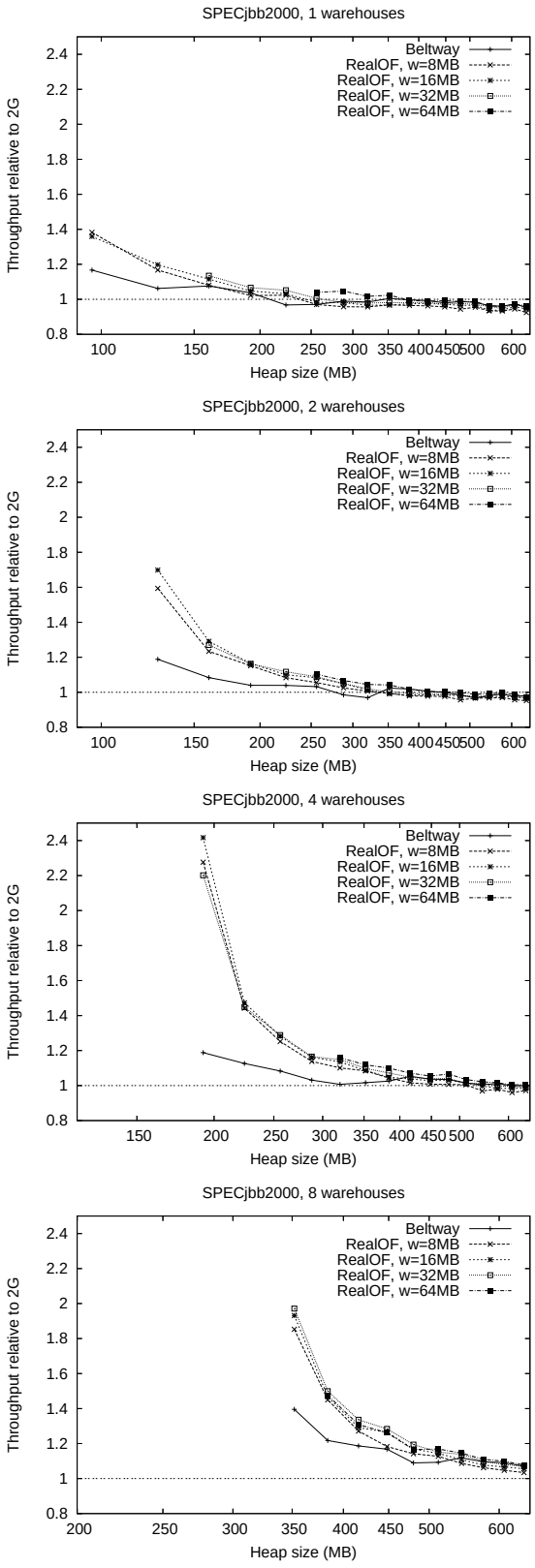

Figure 9: Relative throughput for the SPECjbb2000 benchmark for different garbage collectors (shown relative to the Appel-style collector). Higher is better. The four plots correspond to benchmark scale: 1, 2, 4, and 8 "warehouses".

References

[AAB+99] Bowen Alpern, Dick Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Mark Mergen, Ton Ngo, Janice Shepherd, and Stephen Smith. Implementing Jalapeño in Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, 1999.

[AAB+00] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[App89] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[BCM04] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? high performance garbage collection in Java with JMTk. In *Proceedings of the 26th International Conference on Software Engineering*, 2004.

[BJMM02] Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: getting around garbage collection gridlock. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2002)*, 2002.

[BM02] 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.

[BSH+01] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinley, and J. Eliot B. Moss. Pretenuring for Java. In *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*, 2001.

[DH98] 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.

[HBM+02] Matthew Hertz, Stephen M. Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, and Darko Stefanović. Error-free garbage collection traces: How to cheat and not get caught. In *SIGMETRICS*, 2002.

[HDH03] Martin Hirzel, Amer Diwan, and Matthew Hertz. Connectivity-based garbage collection. In *Proceedings of the 2003 ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003)*, 2003.

[ISF03] Hajime Inoue, Darko Stefanović, and Stephanie Forrest. Object lifetime prediction in Java. Technical Report TR-CS-2003-28, Department of Computer Science, University of New Mexico, May 2003.

[KS05] Sergiy Kyrylkov and Darko Stefanović. A study of garbage collection with a large address space for server applications. Technical Report TR-CS-2005-1, Department of Computer Science, University of New Mexico, February 2005.

[KSM04] Sergiy Kyrylkov, Darko Stefanović, and Eliot Moss. Design and implementation of a 64-bit PowerPC port of Jikes RVM 2.0.3. In *2nd Workshop on Managed Runtime Environments (MRE'04)*, 2004.

[Kyr03] Sergiy Kyrylkov. Jikes Research Virtual Machine - design and implementation of a 64-bit PowerPC port. Master's thesis, University of New Mexico, 2003.

[Kyr05] Sergiy Kyrylkov. 64-bit computing & JVM performance. *Dr. Dobb's Journal*, 30(370):24–27, March 2005.

[SHB+02] Darko Stefanović, Matthew Hertz, Stephen M. Blackburn, Kathryn S. McKinley, and J. Eliot B. Moss. Older-first garbage collection in practice: Evaluation in a Java virtual machine. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance (MSP 2002)*, 2002.

[SMM99] Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. Age-based garbage collection. In *Proceedings of the 1999 ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, 1999.

[Sta99] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

[Sta01] Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 edition, 2001.

[Ste99] Darko Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. PhD thesis, University of Massachusetts Amherst, 1999.