A Study of Garbage Collection With a Large Address Space for Server Applications

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
Phone: +1 505 2776561
Fax: +1 505 2776927

1 February 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 address-ordered heap allows us to use the same inexpensive write barrier that works for traditional 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 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

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

[1]We survey commercial 64-bit JVM platforms elsewehere [12].

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 the heap 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 one or two processors, as such collectors tend to reduce throughput significantly. Our experimental results are obtained on a system of this kind. For such systems "stop-the-world" garbage collection remains a good option as long as the collection pauses are reasonably short.

Previosly, we proposed an older-first garbage collector [17], 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 [9] or profiling-based heuristics [7, 10], though it could. We demonstrated that an emulation of this algorithm in a 32-bit address space can have good performance [16]. 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 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 Implementation

2.1 Infrastructure

Our implementation framework is the Jikes Research Virtual Machine (Jikes RVM), developed by IBM Research [1,2], an open-source virtual machine capable of running a wide variety of Java programs. It offers two compilers, baseline and optimizing, but has no interpreter. We ported Jikes RVM version 2.0.3 to the 64-bit PowerPC/AIX platform [11, 13], and then we extended this port to the PowerPC/Linux architecture and specifically the Apple G5. When we began our work, this was the only 64-bit open-source virtual machine that provided a flexible testbed for implementing new memory managements algorithms, owing to its easily pluggable Garbage Collection Toolkit GCTk (now MMTk [4]).[2] We implemented the RealOF collector and allocator within GCTk; other collectors we use in our study were already provided in GCTk.

2.2 Collector and Allocator

The conceptual design of the RealOF collector has been described fully elsewhere [17, 15]; 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 the picture shown in Figure 1, with 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, which 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

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

new allocation. In this logical view of the heap, it remains laid out in age order, so the free space shows up on the far left (green arrows indicating the space now available for new allocation). 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. The rest of 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 sets (Collections 4–8), the window moves by only that small amount from one collection to the next. As the window continues to move slowly, it remains for a long time in the same logical region, corresponding to the same age of objects. In a copying-collector implementation, this means that a great deal of allocation takes place without much copying work, 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.

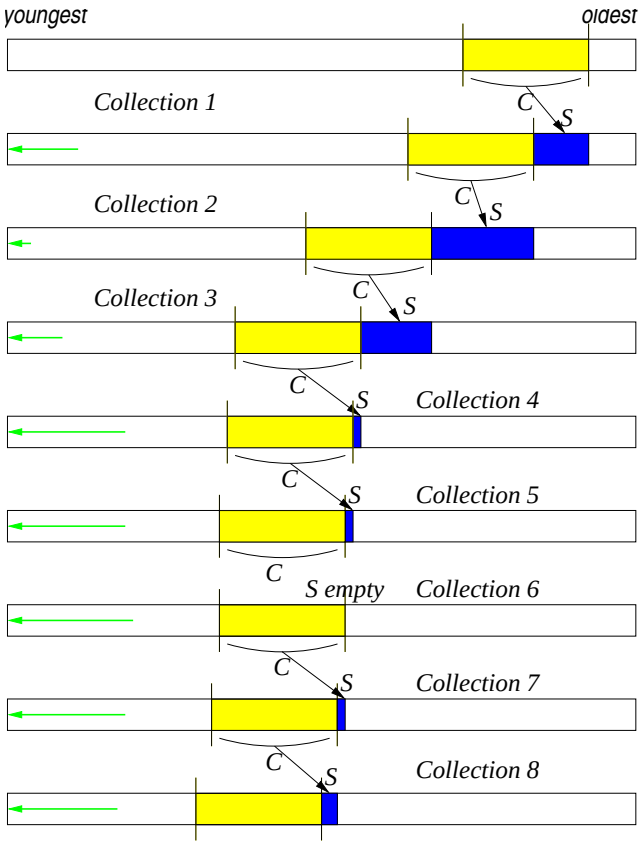

Figure 1: Older-first window motion example.

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 $p \rightarrow q$, then 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 below. 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 had to resort to indirection through an age lookup table to resolve write barriers [16]; we label it OF in the results section of the present paper.

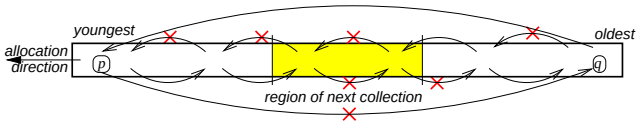

Figure 2: Directional filtering of pointer stores: crossed-out pointers need not be remembered.

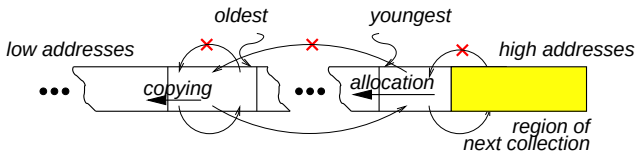

Figure 3: Directional filtering with an address-ordered heap.

The implementation of the RealOF algorithm supports the notions of *zones* and *windows*, but they are of necessity discretized 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 system is 4 MB.

The RealOF allocator is a relatively simple and fast bump-pointer allocator, attached to the current allocation window.[3]

We illustrate the progress of the algorithm in Figure 4. At the onset of virtual machine execution, both

[3]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 and relies on the fact that allocation traditionally goes from lower to higher addresses, we might suffer a performance hit allocating objects from higher to lower addresses. In reality, there is none. Note that with either directions 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 [2] 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.

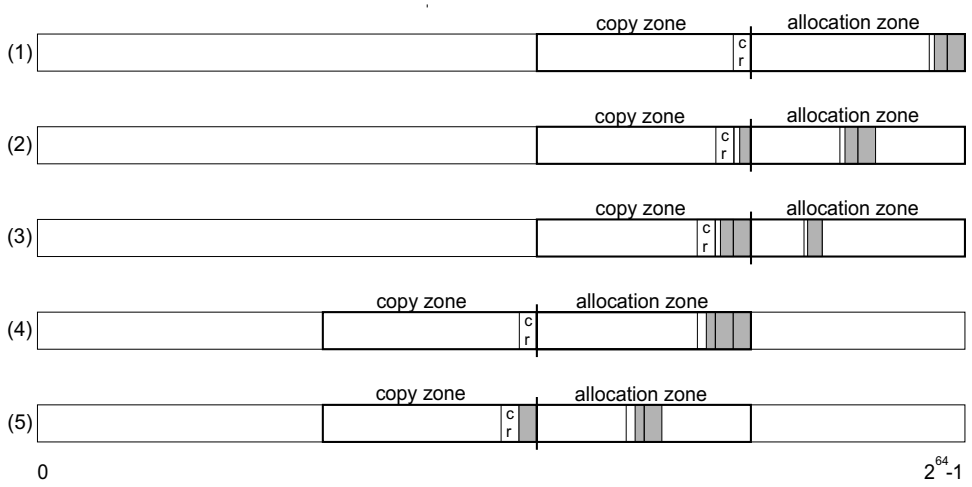

Figure 4: The RealOF algorithm shown operating with three windows in the heap and one copy reserve window (CR). Allocation proceeds from higher addresses to lower (right to left); similarly copying into the copy zone proceeds from higher addresses to lower. Snapshots, top to bottom: (1) before any GC 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.

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:

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

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 always maintained as an invariant during virtual machine execution. 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 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 case 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. Our implementation goes beyond the original description of the OF algorithm in that it provides a mechanism for full-heap collections, which makes RealOF a *complete* GC algorithm in the sense that all garbage is guaranteed to be found eventually. Full-heap collections are triggered if the address space is exhausted by a particularly poorly-behaved application; however no programs we have tested cause this to happen, so we omit a detailed description of the mechanism.

2.2.1 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 [6], 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);
}
```

As we show later, it gives a consistent performance improvement over the indirect write barrier used in the previous implementation of the Older-First algorithm [16], which performed table lookups to map object addresses in a small address space to logical object ages.

2.2.2 Remembered Sets

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 Results

3.1 Experimental Setting

We used the baseline compiler for both the boot image and the application code. In addition, we built Jikes RVM in the *Fast* configuration, which skips assertions checks and pre-compiles all the classes of the virtual machine into the boot image. Our hardware platform was an Apple G5 with a single PowerPC 970 processor at 1.8 GHz, with 1 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. The machine was run in single-user mode detached from the network; this minimized variance between runs. We ran each configuration three times and we report the best run.

The collectors compared are a simple semispace collector [8]; an Appel-style two-generation collector [3]; the Beltway collector [5] in its default 25.25.100 configuration; the older-first collector (OF) with the indirect write barrier and window size of 25% of the heap [16]; and RealOF. For clarity, in Figure 6 we separately show the running times of RealOF as the window size is varied from 4 to 32 MB.

Here we report the results for the Java server application performance benchmark, SPECjbb2000 [14].[4] Table 1 indicates the general behavior of SPECjbb2000 in the two versions we measured: with one "warehouse" (single-threaded), and with four "warehouses" (multi-threaded). The minimum heap size is the experimentally determined smallest heap in which the program can run. The number of garbage collections needed by the semispace collector provides a crude measure of the load placed by the application on the collector. The maximum heap size used in our experiments is chosen so that the semispace collector needs at least 10 collections. Because the benchmark runs for a constant time, the amount of useful work varies depending on the efficiency of the collector, and thus the total amount allocated varies as well.

Benchmark	Description	Minimum Heap		Maximum Heap		Total Allocation, MB
		Size, MB	GCs	Size, MB	GCs	
SPEC jbb2000 - 1	Emulates a 3-tier system with 1 warehouse	72	38	192	10	203–896
SPEC jbb2000 - 4	Emulates a 3-tier system with 4 warehouses	176	22	248	12	301–757

Table 1: Benchmark information including the number of garbage collections performed by the semispace collector.

3.2 Measured Throughput

In the SPECjbb2000 benchmark, the running time is fixed and the benchmark itself reports the measured throughput as the number of transactions per second.[5] We show the reported transaction throughput, thus in Figures 5 and 6 higher is better.

Consistent with our expectations, using a fast write barrier makes RealOF uniformly faster than OF. We now examine how RealOF behaves for different configurations. In all cases after the heap size becomes relatively large for a particular benchmark, the performance of RealOF begins to decrease gradually. 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). This problem could be alleviated using garbage collection triggered by excessive remembered set growth.

From the measurements of RealOF with different window sizes, Figure 6, we conclude that, for the most part, 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.[6] 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

[4] We also obtained results for the SPECjvm98 benchmark suite, which is intended to be representative of client applications, but those results are beyond the scope of this paper.

[5] The benchmark cycles through three phases: first it constructs the data structures for the "warehouses", then it executes transactions for a predefined warm-up period of 30 s, and then it executes transactions for 120 s, reporting the throughout; it repeats this cycle incrementing the number of warehouses from one to a designated maximum.

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

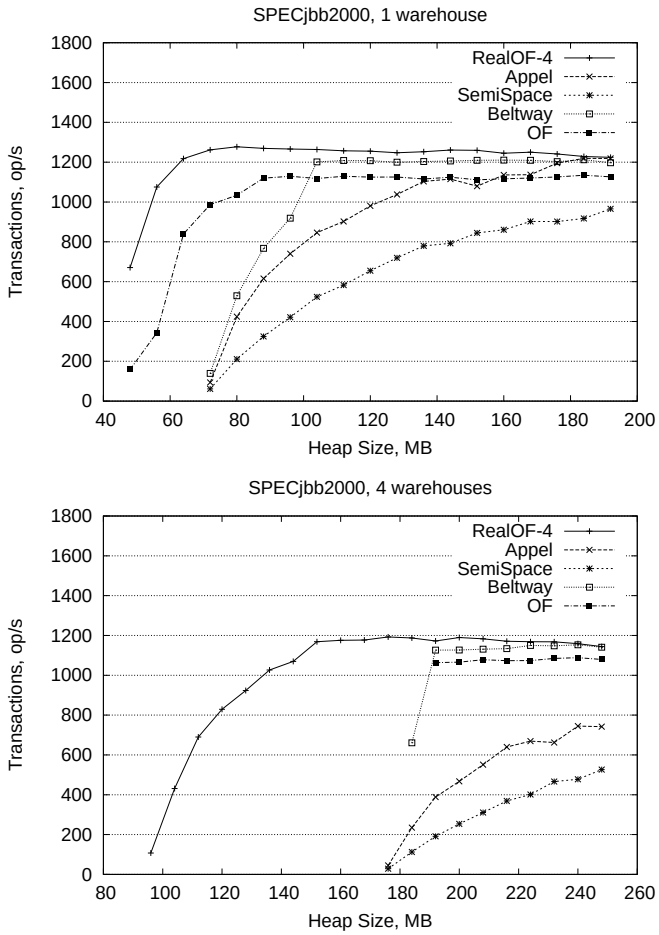

Figure 5: Transaction throughput for the SPECjbb2000 benchmark: comparison of different garbage collectors.

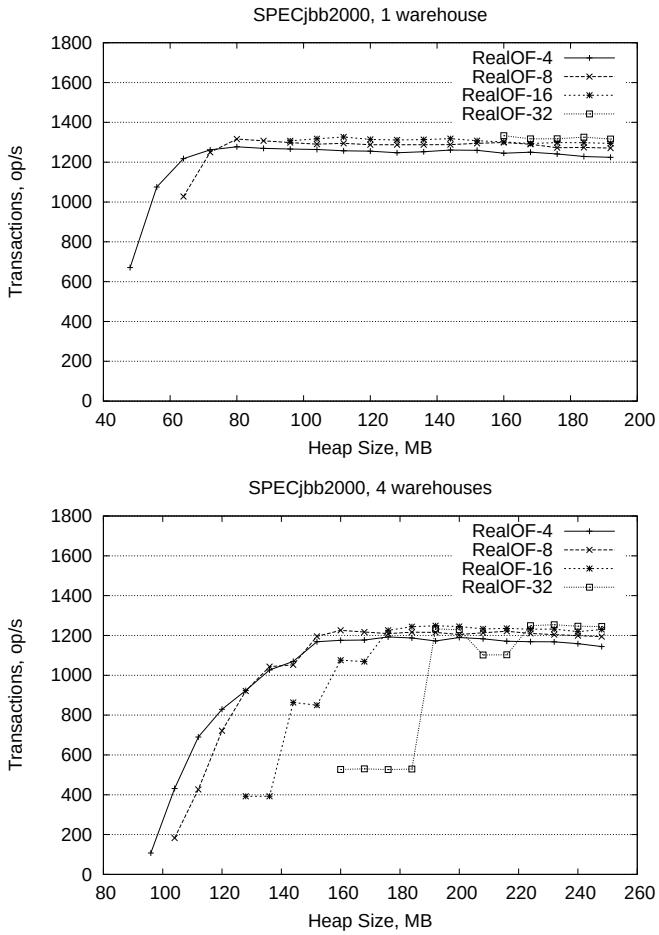

Figure 6: Transaction throughput for the SPECjbb2000 benchmark: comparison of different window sizes for the RealOF collector.

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 it would be beneficial to have a simple adaptive window resizing mechanism.

Finally, the salient result seen in these graphs is that the RealOF algorithm has the *highest throughput overall* among all collectors tested. This is particularly relevant with respect to the Appel-style collector, which has long been recognized as having exceptionally good throughput.

4 Concluding remarks

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

However, remembered set maintenance remains a weak point of the algorithm that can hurt its performance on some programs. Therefore we intend to investigate 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. In our current work, we are revisiting the results reported here in the context of the optimizing compiler (to gauge the relative influence of garbage collection algorithm differences more realistically) and the successor to GCTk, MMTk (to account for garbage collector metadata memory usage), included in newer releases of JikesRVM.

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 constant help and support 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.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. 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, Feb. 2000.

[2] B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, 1999.

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

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

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

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

[7] S. M. Blackburn, S. Singhai, M. Hertz, K. S. McKinley, and J. E. B. Moss. Pretenuring for Java. In *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*, 2001.

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

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

[10] H. Inoue, D. Stefanović, and S. Forrest. Object lifetime prediction in Java. Technical Report TR-CS-2003-28, Department of Computer Science, University of New Mexico, May 2003.

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

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

[13] S. Kyrylkov, D. Stefanović, and E. 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.

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

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

[16] D. Stefanović, M. Hertz, K. S. M. Stephen M. Blackburn, and J. E. 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.

[17] D. Stefanović, K. S. McKinley, and J. E. 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.