A Memory-Efficient Real-Time Non-Copying Garbage Collector

Tian F. Lim Przemysław Pardyak Brian N. Bershad
Department of Computer Science and Engineering
University of Washington, Seattle, WA, USA
{tian,pardy,bershad}@cs.washington.edu

1 Abstract

Garbage collectors used in embedded systems such as Java and Inferno and operating systems such as SPIN must operate with limited resources and minimize their impact on application performance. Consequently, they must maintain short real-time pauses, low overhead, and a small memory footprint. Most garbage collectors are inadequate because they are either not real-time or they trade space for time. For example, Treadmill uses segregated free lists to allocate and reclaim memory in constant time but at the cost of under-utilizing memory. We have implemented a new Treadmill variant and deployed it in the SPIN *extensible operating system kernel. We chose Treadmill for its excellent real-time latencies and low overhead. We improve its poor memory utilization by using real-time page-level management techniques that reduce the fragmentation caused by segregated free lists. We use a page-wise collection to locate pages of free memory and dynamically reassign them between free lists as needed. We use virtual memory to dynamically remap free pages into continuous ranges. Finally, we allocate objects on under-utilized pages when fragmentation is high. Our experiments demonstrate that we have substantially improved memory utilization without compromising latency or overhead, and that the new collector performs very well for* SPIN*'s workloads and regular Modula-3 programs.*

2 Introduction

Garbage collection is gaining mainstream acceptance and is entering new domains of systems software. Currently, it is used to manage memory in Java [1], in embedded systems such as Inferno [2] and PersonalJava [3], and in extensible operating systems such as *SPIN* [4]. These systems are demanding environments in which a wide variety of workloads, including interactive and soft real-time applications, must perform well with limited resources. Consequently, their GC implementations must balance good end-to-end performance with short pauses and low memory consumption. Many existing collectors are geared towards user-level programs and make trade-offs, such as sacrificing space for time, that make them inadequate for systems software.

In this paper, we present a new garbage collector based on Treadmill [5] that is optimized for both time and space, and provides excellent performance for the *SPIN* operating system kernel and regular user-space Modula-3 [6] programs. Our main contributions are the descriptions of efficient techniques that reduce external and internal fragmentation in segregated free-list allocators, and the demonstration of their effectiveness in our workloads.

2.1 Motivation

We have built an extensible operating system kernel, *SPIN*, that allows untrusted applications to link extensions directly into the kernel. The kernel is protected by a type-safe language, Modula-3 [6], that uses garbage collection to manage its heap, allowing untrusted extensions to share memory safely. One of the goals of the *SPIN* project was to achieve performance comparable to commercial operating systems such as Digital Unix or Windows NT. Consequently, *SPIN*'s garbage collector must not sacrifice the performance of system services. Delays must be short because many services, such as GUI or I/O systems, are time-sensitive. Collection overhead must be low, because GC affects the end-to-end performance of the applications in the system. Finally, the collector must minimize wasted memory to reduce the kernel's memory footprint, which competes with applications for memory. Ultimately, garbage collection must deliver good performance in terms of both time and space if it is to become a viable alternative to manual memory management in operating systems or embedded runtimes.

The performance of a garbage collector can be measured by its latency, overhead, and memory utilization. It is relatively easy to optimize a collector for any one of these metrics, but usually by trading off the others. For instance, incremental collection can yield real-time latencies, but increases overhead [7]; compaction improves memory utilization by increasing overhead [8]. Most collectors trade memory utilization for lower latency or overhead. This is reasonable when memory is ample, as is the case for most user-level applications.

In this paper, we observe that memory utilization is as important as latency and overhead for operating system workloads. Building an adequate garbage collector for these workloads therefore requires techniques that balance time and space. We have found that the Treadmill collector provides excellent real-time behavior but has poor memory utilization because of the fragmentation introduced by

segregated free lists. We address Treadmill's fragmentation problems by adding a set of real-time page-level memory management techniques that preserve its compelling time characteristics.

2.2 The new collector

We have built a new garbage collector and deployed it both in the *SPIN* operating system kernel and in user-level Modula-3 programs that run on Digital Unix. The collector is based on Baker's Treadmill real-time non-copying collector [5], which allocates and deallocates in constant time by sacrificing memory utilization. Obvious improvements to memory utilization, such as compaction or a different allocator, would compromise its low overhead or real-time latencies. Instead, we use less aggressive page-level memory management to improve memory utilization:

- large objects are allocated from a separate pool and aligned to pages to limit internal fragmentation,
- free pages are located via a page-wise collection to enable fast coalescing, migration and remapping
- free pages are migrated between lists to eliminate page-level external fragmentation,
- free pages are remapped into continuous ranges for large allocations.
- page filling allocation directs allocations to under-utilized pages.

Our techniques improve overall memory utilization by performing local operations at the grain of pages (large uniform blocks of memory). Pages provide a convenient scope for such operations because they are larger than most objects and break memory into manageable blocks. Thus, determining if a page is free and coalescing the objects on it can be performed efficiently and in bounded time. Because these operations consume constant time, they do not compromise real-time latencies.

We further improve memory utilization by admitting arbitrary free-list sizes, and by reducing header size.

We evaluate these techniques in the context of the *SPIN* operating system and some user-level applications. We show that our new collector allows all of the *SPIN* benchmarks to be executed in 25% to 60% less memory than the unimproved Treadmill collector. In addition, our collector limits application pauses to 15ms.

2.3 The rest of the paper

In Section 3 we provide some background. We describe our techniques in Sections 4, 5, and 6 and their performance in Section 7. We compare our research to related work in Section 8 and conclude in Section 9.

3 Background

Three metrics determine the performance of a collector: latency, overhead, and memory utilization. Latency is the length of pauses incurred by the allocator and the collector. Overhead is the ratio of the time spent allocating and collecting to the total execution time. Memory utilization is the ratio of useful memory (i.e., all potentially live data, excluding space devoted to object headers, padding, etc.) to total size of the heap. The three metrics are closely interrelated, making it hard to optimize one without sacrificing one or both of the others.

3.1 Treadmill

Treadmill is a real-time, non-copying collector that offers bounded time allocation and memory reclamation. All objects, whether free or allocated, are kept in circular doubly-linked lists. All object operations are performed by updating pointers within the lists. Treadmill uses a four-color collection scheme (Figure 1). In addition to "white" (free), "gray" (live, but not scanned), and "black" (live and scanned) objects, a fourth color "ecru" is used to denote objects that were live or allocated since the last collection, but have not yet been marked or scanned. Objects are allocated in constant time by moving them from the white list to the ecru list. Collection proceeds by removing live objects out of the ecru list until it contains only dead objects. These are reclaimed in constant time by appending them to the free list, which requires changing just 2 pointers. The reclamation is implicit, i.e., none of the reclaimed objects are touched.

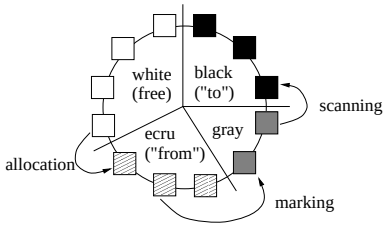

Figure 1. *The Treadmill collector supports constant time allocation and reclamation by using a four-color marking scheme and by keeping all objects in a doubly linked list.*

Allocation time is constant because all objects are the same size, thus avoiding the need to search for a free object large enough. Wilson [9] relaxes this restriction by using segregated free lists for differently sized objects, with sizes increasing in powers of two. Each list is managed separately by the Treadmill algorithm. Allocation requests are rounded up to the nearest power of two and sent to the appropriate list. A collection occurs if any list is exhausted. If no memory can be returned to that list, more is requested from the operating system.

Wilson notes that memory utilization deteriorates rapidly under programs with large varieties of allocation sizes. The worst case internal fragmentation is 50%, and since free memory is neither coalesced nor moved from list to list, external fragmentation can be extremely high. The extra list pointer overhead also increases internal fragmentation, especially for small objects.

We chose the segregated Treadmill collector as the starting point for the development of our new collector due

to its excellent time characteristics. We implemented Treadmill from scratch in Modula-3 and built our improvements on top of it.

4 Improving memory utilization

In this section, we describe our enhancements to the Treadmill collector that improve memory utilization and reduce memory footprint while preserving short latencies and low overhead.

4.1 Our approach

Memory utilization is poor in the Treadmill collector because of fragmentation. *Internal fragmentation* is memory that has been wasted by the allocator and cannot be used by *any* allocation. *External fragmentation* is memory that is free, but cannot be used for *all* allocations. In Treadmill, internal fragmentation comes from the linked list pointers and from rounding object sizes up to powers of two. External fragmentation arises because memory committed to a free list of a given size cannot be used for allocations from a different list.

One measure of fragmentation is *peak memory utilization*, which is the amount of memory that a collector can make available for allocation before needing to reclaim garbage. For a given workload, collectors with higher peak memory utilization require smaller heaps, and for a given heap size they incur fewer collections.

Improving Treadmill's peak memory utilization requires minimizing wasted space and coalescing memory. However, typical smart allocation [10] and coalescing techniques [25] incur unbounded time and may increase overhead, which is unacceptable for real-time applications. In this paper, we use less aggressive operations that reduce fragmentation within smaller ranges of memory but execute in constant time.

One reason fragmentation is high in Treadmill is that implicit collection makes identifying free objects impossible. Thus, it is hard to compute global decisions that could decrease fragmentation, such as locating objects that can be coalesced. We solve this problem by maintaining page-level information about live memory, which allows us to make decisions on a page by page basis. For example, if all objects on a page are free then the page can be used in other lists. All of our techniques operate on a limited number of pages, which guarantees that they complete in bounded time.

Improving memory utilization can increase overhead and latencies, but it also reduces the frequency of collections. Since the costs of collecting are substantially larger than the costs of the improvements, increasing memory utilization is one way to drastically improve overhead.

The rest of this section provides detailed descriptions of our page-level management techniques.

4.2 Big object list

Large objects introduce substantial fragmentation in the segregated Treadmill collector. Internal fragmentation can reach 50% and external fragmentation can be very high because large objects that become free cannot be used for allocations of other sizes. *SPIN*'s workloads often involve many large allocations, further motivating the need to deal with large objects specially.

We reduce the fragmentation caused by objects larger than a page by allocating them from a *big object list* (BOL). Segregated free lists are only used for smaller allocations (Figure 2). The BOL is a bank of memory which is broken up into chunks of the required size. All objects in the BOL are page-aligned to make coalescing and allocation easier. All memory in the BOL is managed by the standard Treadmill algorithm, although allocations require a search since blocks of different sizes reside on the list. Allocation cannot be done in bounded time, but section 4.4 shows how page-level coalescing enables fast allocations from the BOL, and section 5.3 specifies when the real-time bounds will be violated.

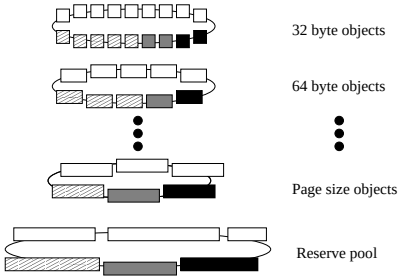

Figure 2. *Segregated Treadmills are used for objects of page size and smaller. The BOL is broken up into chunks of the required size*

The BOL limits the internal fragmentation incurred per object to the size of a page. In the worst case, where all large objects are just bigger than a page, internal fragmentation is still 50%. However, internal fragmentation drops sharply for larger objects.

4.3 Page migration and collection

The segregated Treadmill collector does not coalesce objects or move them from list to list, so when a list is exhausted, more memory is requested from the operating system. This approach, although reasonable when memory is abundant, results in poor memory utilization. We solve the problem by re-assigning free pages in bounded time.

Each Treadmill list has a doubly linked list of pages associated with it. These pages contain that list's objects. When objects are allocated, the pages they reside on are moved from the free-page list to an allocated-page list. During collections, when an object is marked, the page it resides on is also marked. When collections end, unmarked pages are moved into their respective free-page lists.

Page migration (Figure 3) re-assigns free pages to lists that have been exhausted. When a free list is empty, a free page is removed from another list and its objects unlinked.

It is then split into objects of the new size. The page and its new objects are then added to the exhausted list.

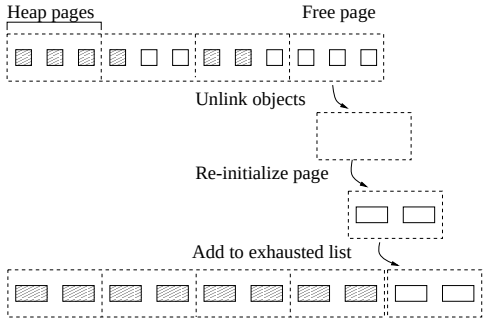

Figure 3. *Page migration moves free pages (with only free objects on them) to exhausted lists.*

Page migration improves memory utilization and overhead by deferring collections until a free list is exhausted *and* no other list has free pages.

Page migration takes constant time. The time to locate a free page is bounded by the number of free lists, and the time to re-initialize the page is bounded by the maximum number of objects that fit on a page. In our current implementation, we sweep pages when they are reclaimed in order to clean their dirty bits. Since the heap size is relatively small (typically 20MB), the amount of work is also small. Reclaiming an arbitrary number of free pages can be done in constant time by using a Treadmill list to manage the pages.

4.4 Page remapping

The BOL suffers from page-level external fragmentation because longer-lived BOL objects break up contiguous free memory. Page migration does not help because it does not coalesce pages. In addition, pages should not be moved from the BOL back to the segregated free lists since this may further fragment the BOL. Compacting objects can create contiguous regions, but requires copying.

We create contiguous memory by virtually remapping free pages into contiguous blocks on demand: when a large allocation cannot be serviced because of fragmentation, the collector finds enough free pages and remaps them into one block.

Since page-level fragmentation is eliminated, pages can now be moved from the BOL back to the free lists. As a result, any page can be used to service any allocation, and collections are incurred only if there are insufficient free pages in the heap.

Page remapping eliminates external fragmentation for large allocations. Since it acts on free memory, mutator pointers need not be altered. Page remapping can also be used to eliminate the need to search the BOL at allocation time. All free memory in the BOL can be remapped into a single contiguous block and allocation would involve incrementing a pointer. However, remapping all memory eagerly can take unbounded time. We propose a new technique, *lazy remapping*, which is analogous to lazy copying. Instead of remapping all pages eagerly, we reserve the pages and remap them incrementally in the background or on demand at fault time. Even if the memory must be initialized, the runtime can do so as the pages are remapped. Since remapping and BOL allocations are fast and rare, we did not implement lazy remapping.

4.5 Page-filling allocation

Page migration introduces sub-page external fragmentation. When an object is allocated from a free page, that page is committed to its list and is wasted if there are no further allocations from that page or list. We will show that sub-page external fragmentation does not manifest itself in *SPIN*'s workloads. We attribute this to long-lived objects that are allocated during initialization stages, as well as the fact that *SPIN* workloads tend to allocate a large number of a wide variety of sizes, ensuring that almost all idle memory gets used. For workloads that incur large sub-page external fragmentation, we have designed a *page-filling allocator,* which allocates from the gaps on underutilized pages.

The allocator maintains information about continuous chunks of memory within pages. When utilization is low, the allocator tries to allocate objects from under-utilized pages. This allocator allows objects of different sizes to reside on the same page and reduces sub-page external fragmentation.

Since page filling can be done on demand, it imposes overhead only when necessary. The required statistics can be maintained for only highly fragmented pages and can be regenerated from scratch during collection when memory becomes scarce.

4.6 Small object internal fragmentation

Internal fragmentation in segregated free lists that use power of two sizes can reach 50%. The BOL reduces the problem for large objects, but small objects can still introduce large internal fragmentation. *SPIN*'s workloads allocate many mid-size and large objects, amortizing the fragmentation caused by smaller objects. We have implemented two optimizations for workloads that use many small objects.

First, we reduce the fragmentation caused by headers. Treadmill lists require two pointers per object in addition to any required header information. We reduce header size by using fewer bits to represent the pointers and calculating their full value on demand. This allows us to reduce the total header size from 3 words to 1 or 2 words, depending on how important internal fragmentation is and how much overhead we are willing to pay to reduce it. We use 2 word headers for *SPIN*, and 1 word headers for some user-space programs.

Second, we reduce the padding used to round object sizes to powers of two. We use free lists whose sizes do not increase in powers of two [24] for user-space programs.

4.7 Discussion

Summarizing, our techniques eliminate most of the external fragmentation in the Treadmill collector by imposing an additional level of memory management. Page-wise collection and migration remove page-level external fragmentation, page remapping coalesces arbitrary pages, and page-filling allocation reduces sub-page external fragmentation. We reduce internal fragmentation by reducing memory wastage. The BOL limits the internal fragmentation of large objects to a page. Free lists that manage arbitrary sizes reduce the internal fragmentation incurred by small objects. Finally, packed headers reduce the memory cost of maintaining Treadmill lists.

The following two sections will discuss latency and overhead in our collector.

5 Latency

To avoid the long pauses incurred by collecting in a single step, an incremental collector breaks collection work into smaller chunks and performs them during allocations or when the mutator is synchronized with the collector. A trade-off is made between the frequency of pauses, their duration, and overhead. Higher frequencies shorten pauses by splitting work into smaller increments but they increase overhead because of the added cost of initiating each collection increment. In our collector, all collection work is done during allocation pauses.

5.1 Write barrier

Our collector uses a compiler generated write barrier to ensure no traced references are lost. Stack writes are not tracked because stacks are scanned at the end of collections. This reduces conservatism and barrier overhead since stack writes occur very often.

The write barrier must ensure that if a reference is stored into a "black" (already scanned) object then the referent, if not already marked, is marked "gray" for future scanning. However, to determine whether the destination object is black, its header must be examined, which is an expensive operation. First, accessing the color information in the header would cost several additional instructions per each assignment. Secondly, locating an object header generally requires scanning the page that contains the address to find the enclosing object. Although the compiler can generate code that avoids this overhead for regular assignments, it cannot do so for assignments into locations passed by reference. We opted to always gray the referent if there is a collection in progress. Since the referent is never an internal pointer, this is a faster but slightly more conservative approach.

5.2 Incremental collection

Incremental collection in our collector proceeds through three main phases:

- *collect-start* – initializes the collection, all global variables are scanned and their referents grayed,
- *collect-some* – at each allocation, some bounded amount of scanning and marking is performed,
- *collect-finish* – completes the collection, stacks are scanned, newly marked objects are scanned, and free memory is reclaimed.

The collect-start phase scans global roots, and the collect-some phase locates all objects reachable from them. Thread stacks are scanned last, and once objects reachable from the stacks are located, all other memory can be reclaimed.

5.3 Real-time properties

Our objective for the Alpha implementation of our collector was to limit latencies to 15ms or less. Unfortunately, enforcing strict real-time bounds on every operation is infeasible. For every GC system there exists an allocation request (any allocation that exhausts all memory in a single request) that will break any latency bounds. Instead we establish the conditions under which real-time latencies are maintained and we describe these conditions below.

Collect-start

The amount of time spent in this phase is proportional to the number of global pointers. In the Alpha implementation, it would take 1500 live objects[1] reachable from globals in order to break the 15ms bounds. Since this situation never arises in our workloads, we simply scan the globals atomically. This phase can, however, be performed incrementally.

Collect-some

The latency associated with each increment of this phase is proportional to the number of pointers scanned. However, to bound the number of pointers scanned would involve making the scanning code more expensive. Instead, we limit the number of objects marked to 1500.

Collect-finish

This phase requires the most amount of work to be done atomically. It must scan the stacks, gray and scan all newly reachable objects, and reclaim garbage. Reclaiming memory in Treadmill is cheap and real-time, but because writes to stacks are not detected by the barrier code, stack scanning should occur atomically. Consequently, the finish stage can take an unbounded amount of time if there are many threads that point to many otherwise unreachable objects. We have added scheduler support to allow stack scanning to proceed incrementally. No more than 80KB of stack memory is scanned at a time. If scanning is interrupted, the scheduler ensures only recently run thread stacks are re-scanned.

Write barrier

Whenever a traced write occurs, a test is done to see if there is a collection in progress. If so, the referent is grayed

[1] This and other values in this section are derived experimentally.

if it is not already marked as live. Thus, the write barrier takes bounded time. In the worst case it takes 20us in the Alpha implementation.

Page migration
As mentioned, page migration takes constant time. In the worst case, it can take 1.5ms.

Page remapping
Pages are remapped on demand at allocation time, which is not real-time since the number of pages to be remapped is proportional to the size of the allocation. However, it would require remapping at least 80 pages (640k) at once in the worst case to break the 15ms bounds. Lazy remapping can perform remapping incrementally, but since no allocations in *SPIN*'s workloads require remapping this many pages, we elected not to implement it.

5.4 Collection policy

The collection policy determines when a collection should start and how much work should be done at each increment. If a collection starts too soon, it may end too early and reduce peak memory utilization, thus forcing more collections. If the collector starts too late or does too little work per increment, the mutator may exhaust the heap before any garbage is recycled. This forces the collector to complete the collection in a single step that may exceed the time bounds.

In our collector, an allocation–mark ratio (*gcRatio*) is used to determine when collections are triggered and how much memory to mark at each increment. For each byte allocated, *gcRatio* bytes should be marked. Figure 4 shows the variables used to determine when to start a collection. We distinguish between "truly" free memory (memory in free pages, which can be migrated), and free memory, F_x, committed to a list *x*. A collection should be initiated when there is just enough free memory to last the entire collection. We anticipate that each list *x* will allocate A_x / *gcRatio* bytes during the collection. Each list already has some free memory F_x, but B_x extra bytes may have to be taken from the "truly" free pool, where

$$B_x = A_x / gcRatio - F_x.$$

Thus, collections must be started as soon as

$$SUM(B_x) >= TrulyFreeBytes.$$

Since there is a bound on the number of objects marked per increment, certain allocation patterns may cause the collector to fall permanently behind the marking ratio. This may happen if the average allocation size is too large, and there are too many small objects to mark. The relationship between these factors is shown in the following inequality. Assume we have faithfully maintained the marking ratio thus far. Let *A* be the average allocation size from now until the end of the collection, and *s* be the average size of the unmarked live objects (data that must be marked). The collector will fall permanently behind the ratio if $A > s*gcRatio$. In practice, we find this does not happen because there is a wide distribution of allocation sizes in our workloads.

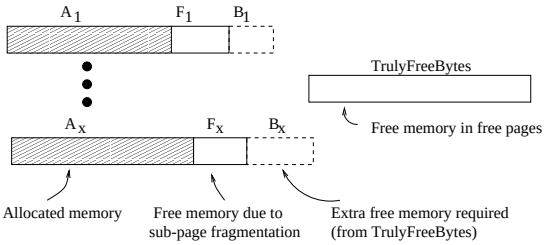

Figure 4. *The variables used to describe free lists*

6 Overhead

The following list summarizes the main sources of overhead in our collector:

- *Incremental collection* – every increment of collection work is accompanied by some initialization.
- *Write barrier* – barriers increase the cost of traced writes by three times in the common case, when no collection is in progress, in addition to increasing code size. Overhead is reduced by using a write instead of a read barrier, leaving stack writes unprotected, and minimizing barrier code instruction counts.
- *Doubly linked lists* – doubly linked list management is more expensive than singly linked lists and the extra pointers waste memory. However, the use of these lists enables implicit memory reclamation. Pointer packing reduces the wasted memory, but increases overhead.
- *Page-level management* – page-level management reduces fragmentation but increases the costs of allocation and deallocation as pages must be moved from list to list. Furthermore, page colors must be reset at the end of collections, as we do not currently reclaim pages implicitly. Extra memory must also be used to store page structures.

In general, we trade some overhead for real-time latencies and good memory utilization. However, the improved memory utilization will reduce the number of collections, and thus, the total overhead incurred.

7 Performance

In this section we describe the effect of our techniques on several workloads. We demonstrate:

- the improvement in memory utilization over the original Treadmill collector,
- the impact of improved memory utilization—smaller memory requirements, lower collection frequencies, lower overheads, and shorter end-to-end running times,
- that our techniques do not compromise latency, and can be used in real-time systems.

7.1 Experimental environment

Four versions of the segregated Treadmill collector are used to evaluate the effects of our techniques on *SPIN* workloads:

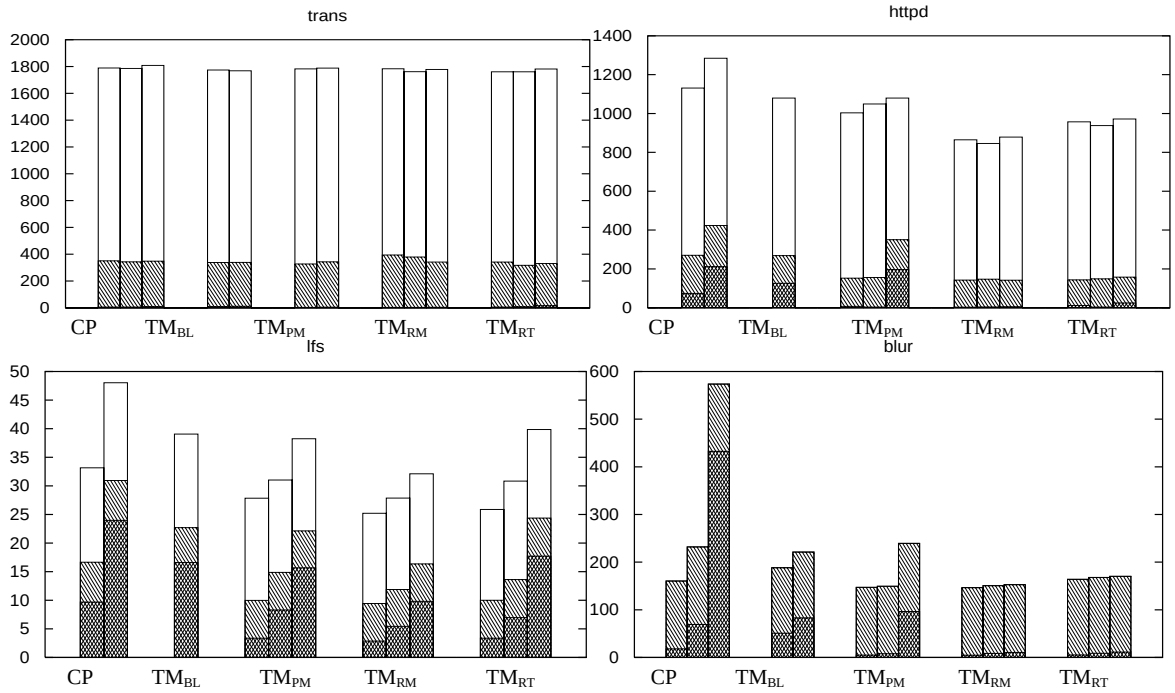

Figure 5. *End-to-end performance. The bars represent, from the bottom: allocation and GC overhead (dark), mutator time(light), and idle time (white). The y-axis is in seconds. Each workload is run three times with decreasing heap size (Table 3). Missing bars represent an attempt that ran out of memory. These figures show that our collectors' performance degrades slower than collectors with poorer memory utilization.*

- **TM_{BL}** – the base-line, non-incremental approximation of Wilson's segregated Treadmill collector.
- **TM_{PM}** – TM_{BL} with BOL and page migration.
- **TM_{RM}** – TM_{PM} with page remapping enabled.
- **TM_{RT}** – the real-time, incremental version of TM_{RM}.

For comparison, we use:

- **CP** – an improved SRC Modula-3 VM-based mostly-copying collector [11]. It is configured to run incrementally but without generations.

For our user-space experiments, we use the following collectors:

- **FLS** – the TM_{RM} collector, which uses fixed list sizes.
- **FLS-PH** – FLS with packed headers.
- **ALS** – FLS with arbitrary list sizes.

We do not use TM_{RT} for the user-space experiments, because we want to isolate the results from the impact of the incremental collection policy.

All collectors run in both the SPIN kernel and user-space Modula-3 applications. The benchmarks were measured on a 266 MHz DEC AlphaStation 250/66, with the SPIN workloads running in the SPIN v1.33 kernel and the user space workloads on Digital Unix 3.2. The page size is 8kB and the heap size is 22.7MB unless otherwise specified.

Four OS benchmarks are used to evaluate the performance of our techniques in the *SPIN* kernel:

- **TRANS** – the RVMbench [12] benchmark exercises *SPIN*'s in-kernel transaction services.
- **HTTPD** – *SPIN*'s web tree (65 files, 5MB) is copied 14 times by 10 concurrent, scalable web clients [13] via the *SPIN* in-kernel web server.
- **LFS** – a subset of the Andrew file system benchmark [14] is run on the *SPIN* log structured file system [15].
- **BLUR** – a 2.5MB image is blurred while maintaining copies of modified pixels, similar to the way a paint program would operate.

Some numbers characterizing the benchmarks are included in Table 1. BLUR exhibits a high object turnover rate, while HTTPD maintains a large amount of live memory at all times. The live memory for LFS is constantly increasing due to a poor caching policy.

Benchmark	Allocated [MB]	Average Live [MB]
TRANS	43.7	3.2
HTTPD	16.1	9.7
LFS	96.7	4.6
BLUR	212.1	4.5

Table 1. *Megabytes allocated per benchmark. The average amount of live memory was sampled at collection points.*

Because the key page-level management improvements were implemented with *SPIN* in mind, we evaluate those first (Sections 7.2 to 7.7) and demonstrate that they are adequate for the *SPIN* workloads. We then show that user space workloads have different characteristics and evaluate the improvements made specifically for them in a separate subsection (7.8).

7.2 Minimum heap requirements

The improved memory utilization of our collectors is succinctly demonstrated in the minimum heap sizes required to run each workload (Table 2).

	TM_{BL}	TM_{PM}	TM_{RM}	TM_{RT}	CP
TRANS	15.6	7.8	6.6	6.6	10.9
HTTPD	20.3	15.6	14.1	14.1	19.5
LFS	19.0	13.3	11.7	11.7	14.8
BLUR	10.8	10.2	6.3	6.2	9.4

Table 2. *Minimum heap size (in MB) necessary to execute benchmarks using our collectors. TM_{RM} and TM_{RT} have the lowest requirements.*

The workloads will fail to run on any heap size smaller than the ones indicated in the table because the allocator cannot supply enough memory. For example, given a 15 MB heap, only TM_{RM} and TM_{RT} would be able to run all the workloads. Table 2 shows that our techniques reduce memory footprint and that those benefits are preserved in the case of incremental collection.

7.3 End-to-end performance

Figure 5 shows the end-to-end performance for the benchmarks. Each benchmark was run three times on each collector (Table 3): once with ample memory (22.7 MB), once again with less memory (close to the minimum necessary for all collectors to successfully run that workload), and finally with even less memory (TM_{BL}, TM_{PM} and CP sometimes fail to complete).

	Ample memory	Less memory	Even less memory
TRANS	22.7	11.0	10.9
HTTPD	22.7	19.5	16.4
LFS	22.7	11.7	8.6
BLUR	22.7	19.5	9.5

Table 3. *Heap sizes (in MB) used to generate end-to-end graphs.*

With ample memory, the end-to-end running times do not vary by much, which shows that our techniques do not adversely affect performance when they are not necessary.

Performance degrades for all collectors when there is less memory available, but the performance of our Treadmill variants degrades slower than the other collectors. The TM_{RM} and TM_{RT} collectors exhibit the best performance over the range of heap sizes because of their memory utilization. TM_{RT} performs slightly worse than TM_{RM} because of the overheads associated with incremental collection.

TRANS is a long running benchmark with much I/O and relatively little allocation. Its mutator and idle time fluctuate with changes in disk performance. As a result, collector impact on TRANS is hard to gauge from these graphs alone.

7.4 Memory utilization

Figure 7 shows the amount of potentially useful memory, header space, padding (from rounding allocation sizes up), and external fragmentation plotted against an allocation clock. Figure 6 shows a legend. Usage increases as memory is allocated and the sudden drops correspond to collections. The ideal collector minimizes internal fragmentation (the area between the lines is small), and exhibits no external fragmentation (the peaks hit 100%).

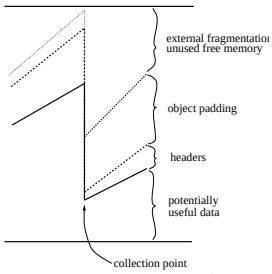

Figure 6.*Legend for Figure 7.The regions of the graph, from the bottom, are : potentially live memory, header space, object padding, and external fragmentation/free memory.*

TM_{BL} does poorly on all benchmarks because of internal fragmentation in large objects and external fragmentation caused by the inability to move free memory. TM_{PM} shows improvement due to the BOL and page migration, but external fragmentation is still high because memory cannot move between the BOL and free lists. TM_{RM} does very well and, including internal fragmentation, almost reaches 100% utilization. It does not actually hit 100% because of sub-page external fragmentation, which is minimal in *SPIN*'s workloads. TM_{RT} reclaims memory earlier than TM_{RM} because it starts collections earlier. The CP collector is generally restricted to 50% of the heap to reserve enough space for copying. CP sees spikes in internal fragmentation due to the space wasted for copying.

HTTPD maintains a large amount of live memory, so TMBL collects constantly due to external fragmentation and CP thrashes because about 50% of the heap is live. The amount of live memory for the LFS workload is slowly increasing, and this increases the collection frequency (the density of spikes) during the run. BLUR allocates a wide range of sizes, resulting in large external fragmentation (more than 50% of the heap) for TMBL. TRANS clearly shows how collectors with better peak memory utilization collect less often.

In summary, these figures show that page migration and remapping improve memory utilization substantially, reducing the frequency of collections.

7.5 Collection frequency

To measure how frequent collections are, we computed the average number collections per megabyte allocated (Table 4) for the case when heap memory is abundant (22.7MB).

The table shows that the frequency of collections drops significantly when our techniques are applied. Note that as heap size decreases, the disparity in frequency increases, as evidenced by some of the end-to-end graphs, in particular HTTPD and BLUR.

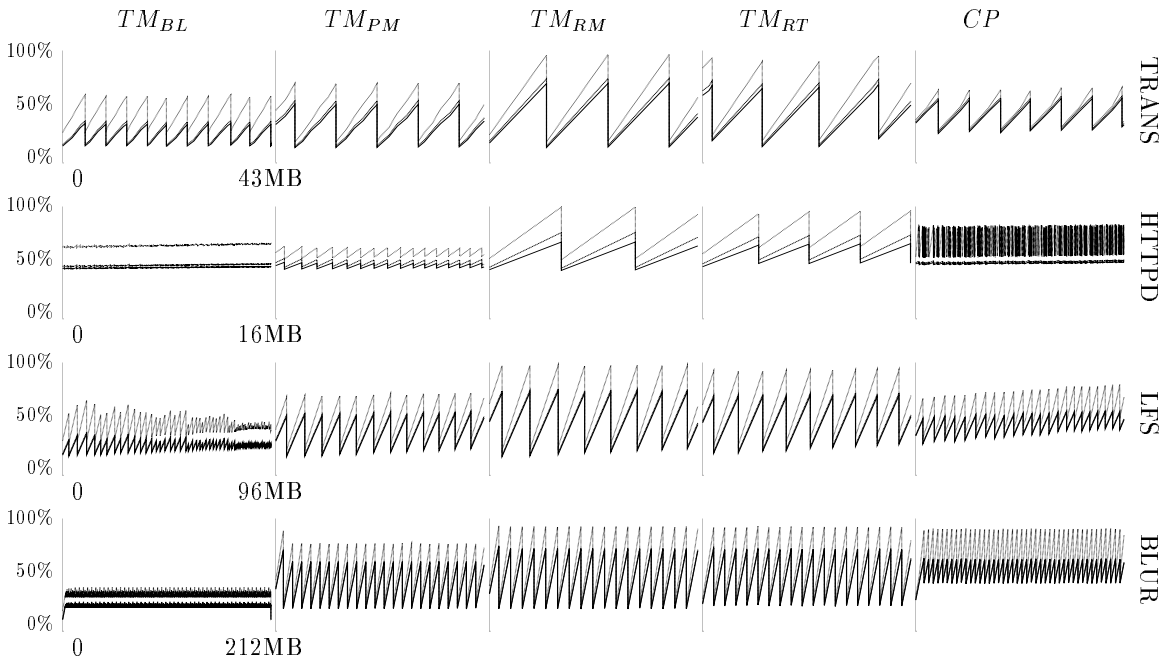

Figure 7. *Memory utilization using the five collectors plotted against an allocation clock. The lower lines are the amount of useful data. The upper lines are the total amount of memory allocated. The middle lines are internal fragmentation due to headers. The y-axis is percentage of heap used. Note that the x-axis differs across benchmarks and goes from 0 to the total amount of memory allocated for that workload.*

	TM_{BL}	TM_{PM}	TM_{RM}	TM_{RT}	CP
TRANS	0.28	0.12	0.08	0.11	0.18
LFS	0.82	0.13	0.08	0.10	0.25
HTTPD	15.94	0.99	0.18	0.32	13.87
BLUR	1.88	0.13	0.08	0.09	0.22

Table 4. *Collection frequency measured as the number of collections incurred per megabyte of allocated memory.*

7.6 Overhead

Figure 8 shows the overheads imposed by the collectors on all benchmarks in the case where there is abundant memory. Overhead is calculated as a percentage of the time spent computing, excluding idle time (e.g., due to I/O). The overheads vary widely among benchmarks because each has different allocation rates. However, in all cases, TM_{RM} performs better than TM_{PM}, which does better than CP and TM_{BL}. TM_{RT} imposes more overhead per collection than TM_{RM} because each collection increment computes how much memory should be marked.

Little time (<1%) is spent performing page migration and remapping, despite their large impact on memory utilization. This shows that even an order of magnitude increase in the costs of remapping will not substantially impact overheads, which is important because remapping is more expensive in user-space.

To summarize, although our techniques impose some small overhead, the costs are hidden by the drastic reduction in collection overhead.

7.7 Latency

Although TM_{RT} has higher overheads than TM_{RM}, it has significantly shorter collection pauses. Table 5 summarizes the longest pause times incurred by each collector. It shows that TM_{RT} satisfies its 15ms real-time bound.

	TM_{BL}	TM_{PM}	TM_{RM}	TM_{RT}	CP
TRANS	491	155	170	*14*	55
HTTPD	660	344	441	*14*	270
LFS	240	140	137	*11*	80
BLUR	132	93	95	*12*	78

Table 5. *Longest allocation pauses observed (all times in ms).*

7.8 Preliminary user-space experiments

Our page-level improvements to memory utilization have made Treadmill the collector of choice for the *SPIN* kernel. However, two problems remain although they do not manifest in *SPIN*'s workloads: internal fragmentation can be high for small objects and sub-page external fragmentation can be high for certain workloads. Figure 9 demonstrates these effects and how they can be addressed in two user-level workloads. CUBE is a simple demonstration of the Modula-3 GUI, and M3PP is a Modula-3 source code pretty-printer.

Each benchmark is run under three different collectors with a heap size of 1.6MB. FLS is identical to TM_{RM} with power of two list sizes; FLS-PH is FLS modified with the header size halved; ALS is FLS modified to admit arbitrary, non power-of-two list sizes.

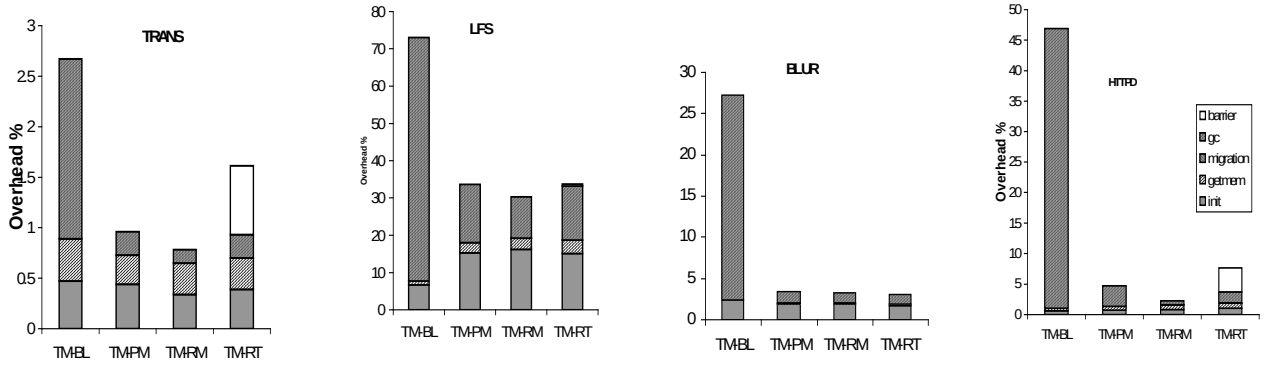

Figure 8. *Allocation and collection overhead as a percentage of mutator time.* Init *is initialization costs,* getmem *is the time to dequeue free memory,* migration *is the time spent migrating or remapping, and* barrier *is the work done in the write barrier. This chart shows migration costs are negligible despite their impact on overall overhead. Note that these were measured with ample memory. As heap size decreases, the results differentiate more.*

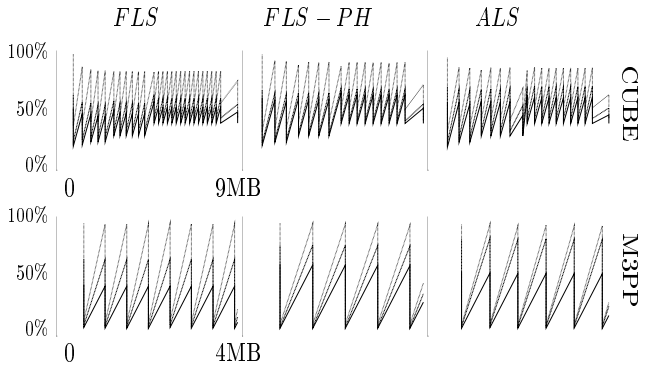

Figure 9. *Memory utilization of two user-level Modula-3 application under three user-level collectors. The leftmost collector is TM_{RM}, followed by the header packing collector and the collector with arbitrary list sizes. All runs are with a 1.6MB heap.*

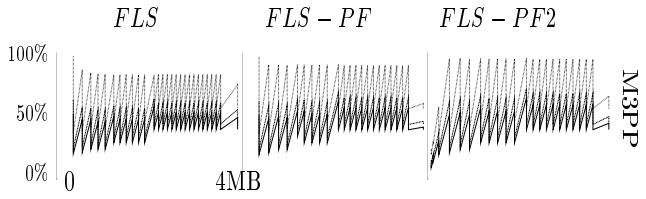

Figure 100. *Sub-page external fragmentation is reduced by the page-filling allocator (FLS-PF), and further reduced by limiting the number of pages committed to a list (FLS-PF2).*

The FLS runs show how both workloads exhibit substantial internal fragmentation, primarily from small object header wastage and padding. M3PP incurs 55% internal fragmentation, of which 43% is headers. CUBE incurs 45% internal fragmentation, of which 25% is headers.

FLS-PH reduces header waste by 50%. Since the distribution of object sizes changes, this has the side effect of reducing padding fragmentation by 5% in these workloads. ALS manages to halve padding by 50% for both workloads. These graphs demonstrate that both forms of internal fragmentation can easily be reduced.

CUBE exhibits about 15% sub-page external fragmentation. Figure 10 shows the results of a preliminary implementation of our page-filling allocator. The FLS-PF run shows that the page-filling allocator halves the sub-page external fragmentation to about 8%. Further analysis indicates that the remaining waste arises because too much memory is committed to a list that is only used at initialization time. By incurring collections earlier before too much memory is committed, we arrived at the third run (FLS-PF2), which further reduces external fragmentation to about 3-4%.

In summary, although our user-level workloads exhibit more header wastage, internal fragmentation and sub-page external fragmentation, each can be addressed with simple solutions.

Note to reviewers: the implementation of these techniques is still preliminary and we expect to integrate these improvements together and report results in the final version of the paper.

8 Related work

Treadmill [5] is a non-copying extension to the original Baker real-time collector [16]. Wilson's improvements [9] reduce overhead and support bounded allocations for objects of any size but they do not address memory utilization. Johnstone and Wilson [17] also confirm fragmentation problems when multiple segregated lists are used.

Page migration has been implemented before [18], but has typically relied on per-page object counts to reclaim pages. Since Treadmill deallocates objects implicitly, we use a page-wise collection to reclaim pages, which has the advantage of lower overhead. In addition, we have extended the idea of migration with remapping to further reduce external fragmentation. Virtual memory remapping has been used to re-allocate mapped memory (Doug Lea's allocator) or to remap large objects instead of copying them [19]. We use remapping to create contiguous free regions,

instead of moving live objects. This obviates the need to modify mutator pointers

There are many examples of techniques for improving memory utilization in the form of smart allocation techniques [20,21], but none of these bound allocation times and some require copying. We have focussed on simplifying memory management by managing pages which is simpler and faster than buddy schemes, and has the advantage of enabling us to remap memory.

Some of our techniques have been independently developed for the Great Circle collector by Geodesic Systems [22]. Both use a large object area to reduce fragmentation, and remap free memory into contiguous extents. The Great Circle collector uses deallocation and allocation to relocate memory. This is a viable technique for OS-es that do not export an interface to remap virtual pages. The key difference is our focus on real-time latencies and performance for *SPIN* OS workloads.

9 Conclusions

We have analyzed the relationship between memory utilization, collection frequency, and overhead in a Treadmill collector. We have built a new version of Treadmill that uses page-level memory management to substantially improve memory utilization without imposing unbounded latencies. The big object list limits internal fragmentation, page migration eliminates page-level external fragmentation in multiple free-list, while page remapping cheaply creates contiguous free memory. Finally, page filling allocation reduces sub-page external fragmentation caused by bursty allocation patterns. Together, the techniques reduce overhead by reducing collection frequency. They enable our collector to run efficiently in small heaps and give it performance adequate for a high performance operating system kernel.

References

1 Java, http://java.sun.com/, 1996

2 Inferno, http://inferno.lucent.com/, 1996

3 PersonalJava, http://java.sun.com/, 1997

4 B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers, Extensibility, Safety and Performance in the SPIN Operating System, In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, 1995.

5 H. G. Baker, The Treadmill, Real-time Garbage Collection without Motion Sickness", SIGPLAN, ACM, 27, 3, 1992.s

6 G. Nelson, ed., *Systems Programming in Modula-3*, Prentice Hall, 1991.

7 E. W. Dijkstra, L. Lamport, A.J. Martin, C. S. Scholten, E. F. M. Steffens, On-The-Fly Garbage Collection: An Exercise in Cooperation, Communications of the ACM, 21 (11): 965–975, November 1978.

8 R. A. Saunders, The LISP system for the Q–32 computer, in E. C. Berkeley, D. G. Bobrow (ed.), *The Programming Language LISP*, 1964

9 P. R. Wilson and M. S. Johnstone, Truly Real-Time Non-Copying Garbage Collection, in E. Moss, P. R. Wilson, B. Zorn, editors, OOPSLA/ECOOP'93 Workshop *on Garbage Collection in Object-Oriented Systems*, October 1993.

10 D. S. Wise, The double buddy system, TR 79, CS Dept. Indiana University, December 1978.

11 J. Bartlett, Compacting Garbage Collection with Ambiguous Roots, Technical Report, DEC WRL 88/2, February 1988.

12 M. Satyanarayanan, H. Mashburn, P. Kumar, D. Steere and J. Kistler, Lightweight Recoverable Virtual Memory, ACM *Transactions on Computer Systems*, 12(1): 33–57, 1994.

13 G. Banga and P. Druschel, Measuring the Capacity of a Web Server, in *Proceedings of the USEXIX Symposium on Internet Technologies and Systems*, December 1997.

14 J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, Scale and Performance in a Distributed File System, *ACM Transactions on Computer System*, 6(1), February 1988.

15 M. Rosemblum and J.K. Ousterhout, The Design and Implementation of a Log-Structured File System, In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, 1991.

16 H. G. Baker, List Processing in Real-Time on a Serial Computer, Communications of ACM, 21(4), pp. 280--94, 1978.

17 M. S. Johnstone and P. R. Wilson, The Memory Fragmentation Problem: Solved?, OOPSLA'97 Workshop *on Garbage Collection in Object-Oriented Systems*, October 1997

18 C. Weinstock, Dynamic Storage Allocation Techniques, PhD thesis, Carnegie-Mellon University, April 1976.

19 P. T. Withington, How Real is "Real Time"' Garbage Collection?, OOPSLA'91 *Workshop on Garbage Collection in Object-Oriented Systems*, 1991.

20 B. Baker, E. G. Coffman, and D. E. Willard, Algorithms for Resolving Conflicts in Dynamic Storage Allocation, JACM, 32(2), pp. 327--343, April 1985.

21 J. L. White, Address/Memory Management for a Gigantic Lisp Environment, or, GC Considered Harmful, *Conference Record of the 1980 Lisp Conference*, Redwood Estates, CA, pp. 119–127, August 1980.

22 M. Spertus and G. Rodriguez-Rivera, "A Non-Fragmenting, Non-Copying, Garbage Collector", submitted for publication, 1998.

23 H. G. Baker, List Processing in Real-Time on a Serial Computer, Communications of ACM, 21(4), pp. 280--94, 1978.

24 M. S. Johnstone and P. R. Wilson, The Memory Fragmentation Problem: Solved?, OOPSLA'97 Workshop *on Garbage Collection in Object-Oriented Systems*, October 1997

25 C. Weinstock, Dynamic Storage Allocation Techniques, PhD thesis, Carnegie-Mellon University, April 1976.

26 P. T. Withington, How Real is "Real Time"' Garbage Collection?, OOPSLA'91 *Workshop on Garbage Collection in Object-Oriented Systems*, 1991.

27 B. Baker, E. G. Coffman, and D. E. Willard, Algorithms for Resolving Conflicts in Dynamic Storage Allocation, JACM, 32(2), pp. 327--343, April 1985.

28 J. L. White, Address/Memory Management for a Gigantic Lisp Environment, or, GC Considered Harmful, *Conference Record of the 1980 Lisp Conference*, Redwood Estates, CA, pp. 119–127, August 1980.

29 M. Spertus and G. Rodriguez-Rivera, "A Non-Fragmenting, Non-Copying, Garbage Collector", submitted for publication, 1998.