A Non-Fragmenting Non-Moving, Garbage Collector

Gustavo Rodriguez-Rivera Michael Spertus Charles Fiterman
Geodesic Systems
414 North Orleans Street, Suite 410
Chicago, IL 60610
Tel. (312) 832-1221
{grr, mps, cef}@geodesic.com

1. ABSTRACT

One of the biggest disadvantages of non-moving collectors compared to moving collectors has been their limited ability to deal with memory fragmentation. In this paper, we describe two techniques to reduce fragmentation without the need for moving live data. The first technique reduces internal fragmentation in BiBoP (Big-Bag-of-Pages) like allocators. The second technique reduces external fragmentation using virtual memory calls available in most modern operating systems. It can also reduce the size of the heap after periods of great activity in long lived applications. These techniques have been successfully used in Geodesic Systems' Great Circle, a commercially-available conservative garbage collector. This paper describes these techniques, their implementation, and some experimental results.

1.1 Keywords

Memory allocation, garbage collection, fragmentation, conservative garbage collection, non-copying garbage collection

2. INTRODUCTION

Moving garbage collectors (i.e. copying and compacting garbage collectors [5, 8]) eliminate fragmentation by moving live data and leaving all free space in consecutive locations. Non-moving collectors are unable to move live data, and therefore free memory is interleaved with holes of live memory. This causes that some requests for free memory will not be satisfied with the available free memory because it is not in consecutive locations, even though the total free memory could be greater than the amount requested.

Moving-collectors can solve the fragmentation problem [12]. Unfortunately, moving collectors can not be used in environments where no accurate pointer information is available, since pointers in live objects have to be updated when the objects they point to, are moved. For example, off-the-shelf C and C++ compilers do not give accurate pointer information. Some work has been done to obtain pointer information of C programs from the debugging information [13], however their use is limited since C and C++ can store pointer values in non-pointer types, and keeping track of pointer information at execution time is difficult.

Conservative garbage collection [4] also has some problems when combined with moving garbage collection. Conservative garbage collection considers every valid memory address that is word-aligned and that points to a valid object as a real pointer. Conservative garbage collectors are safe because a superset of the pointers in live objects is always used during the collections and no live memory is ever reclaimed. However, an arbitrary sequence of bytes that happens to be word-aligned and that points to a valid object can be erroneously interpreted as a pointer causing memory retention. This sequence of bytes, called a false pointer, is not detrimental, since retention of garbage is benign as long as it is not excessive. A moving garbage collector needs to update the pointers in live objects to reflect the new location of the moved objects they point to. However, a moving garbage collector can not move live objects detected by conservative garbage collection scanning because false pointers may be erroneously updated causing unpredictable results in the program. Some moving collectors deal with this problem by *pinning* objects that are pointed by pointers that were found using conservative pointer finding [1, 6], i.e., objects found by the conservative garbage collector are not moved, creating fragmentation problems.

ISMM '98 10/98 Vancouver, B.C.

In this paper, we describe two techniques to reduce fragmentation in non-moving garbage collectors. The first technique is intended to reduce internal fragmentation in allocators that use a BiBoP scheme (Big-Bag-of-Pages) [4,12]. In a BiBoP allocator, the objects found in the same page have the same characteristics. The second technique is intended to reduce external fragmentation and uses virtual memory primitives that can be found in most modern operating systems.

Allocators such as the one used in Boehm's collector [4] divide the objects in two classes: small and large objects. Small objects are obtained by dividing a single page in small pieces, and large objects are the ones that use one or more consecutive pages. In this paper, we show that by dividing the objects in only two classes, the objects with sizes around the page size have a large internal fragmentation. To solve this problem, the first technique introduces a third class called medium-objects. This class of objects can be managed by dividing more than one consecutive pages into small pieces, in the same way as Boehm divides a single page, in order to reduce the internal fragmentation. We have called this class of objects *medium-objects*.

Most modern operating systems have virtual memory operations that allow mapping of uncommitted memory that only reserves address space, but do not reserve swap space, and allow committing or de-committing swap space at run-time.

The basic idea of the second technique is to de-commit swap space for those sections of the free list that are too fragmented to be used for allocations, and to reuse this swap space in a consecutive area in the address space. In effect, this technique provides many of the benefits of moving garbage collection by virtually moving the free data rather than the live data.

In addition, this technique allows returning to the operating systems parts of the heap that were used during periods of heavy allocation and that are no longer used. We will refer to this technique as *footprint reduction*.

Both medium objects and footprint reduction have been implemented in a variation of Boehm's conservative garbage collection library [4] called Great Circle. Great Circle is a commercially-available garbage collection library from Geodesic Systems.

The ideas presented here are explained in the context of garbage collection. However, they can also be applied to any general-purpose memory allocator that does not use garbage collection.

The paper is organized as follows. The first section introduces Great Circle and medium objects. The second section explains the footprint reduction mechanism. Finally, some experimental results are shown.

3. MEDIUM OBJECTS

The first technique, called medium objects, is designed to reduce internal fragmentation. It is based on the observation that small objects that do not fit evenly into a page have high internal fragmentation, while large objects that are slightly larger than a page boundary lead to significant internal fragmentation in the page-based large object allocator as well. The high internal fragmentation associated with both large small objects and small large objects suggest that one should create an intermediate class of medium objects.

The memory allocator of Great Circle divides the objects in three classes: small, medium, and large objects. Small objects are objects that are obtained by dividing a single page in small pieces. Medium objects are objects that are allocated by dividing multiple consecutive pages in pieces. Large objects are objects that are a multiple of a page-size.

The small and medium object allocator consists of buckets of objects for each size, called segregated free lists [14]. Great Circle pre-computes the sizes of small and medium objects and makes sure that the maximum fragmentation for each object size is less than some pre-specified constant. The large object allocator is a single list of objects ordered by address to allow coalescing.

When a small or medium object is requested, an object is returned from the corresponding bucket. If the corresponding bucket is empty, the allocator will request one or more pages from the large object allocator, divide them into pieces and put the objects in the corresponding bucket. The allocator satisfies requests for large objects by returning the first block available in the free list. A search for a block in the free list starts where the previous one ended.

The allocator uses a BiBoP scheme to get the information of the objects. Each page has a page information structure that contains the size of the objects stored in that page. A page stores only objects of the same size, and therefore, given a pointer to the object, it is possible to find out the starting address and the size of the object.

In Boehm's allocator, objects are only divided in small and large with no medium objects. The main drawback of this approach is that the maximum internal fragmentation of objects a few bytes larger than a page size, or a few bytes larger than half a page size is very large. Since we found that these allocation sizes happen to be the common case for many applications, we decided to add medium objects that result from dividing multiple consecutive pages in small pieces.

For example, let us assume that there is a request to allocate an object of 2050 bytes in a system that has a page size of 4096. An allocator that uses only small and large objects will return an object of 4096 bytes because it is the next object size after 2048 bytes. An allocator that uses

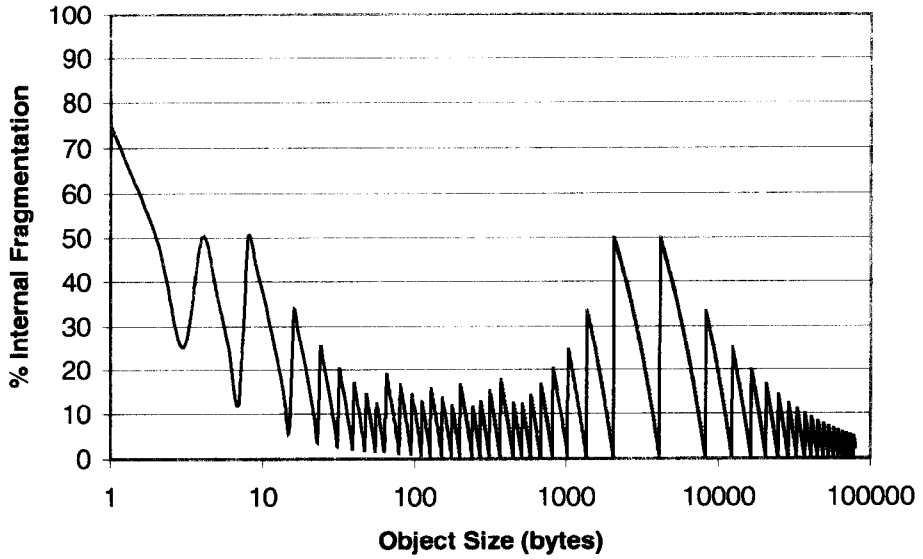

Figure 1. Fragmentation of allocator with only small and large objects

medium objects can divide two consecutive pages in 3 objects of size 2730 bytes (plus 2 spare bytes that cannot be divided), and return one of these objects. In the first case, 2046 bytes are wasted, and in the second case only 680 bytes are wasted.

Figure 1 and 2 show the internal fragmentation for different object sizes. Figure 1 shows the fragmentation when only large and small objects are used, and figure 2 graph shows the fragmentation when besides large and small objects, medium objects are also used. In both cases, the page size is 4096 bytes. The object size axis uses logarithmic scale. The internal fragmentation for small objects is about the same in both graphs. The initial peaks for the smallest objects are caused by the initial object sizes that are four and eight bytes. These sizes are chosen because of alignment limitations in the architecture used. In the Sparc architecture, objects larger than 8 bytes have to be aligned to 8 byte boundaries to make sure that they can store double types. In the Sparc architecture, double

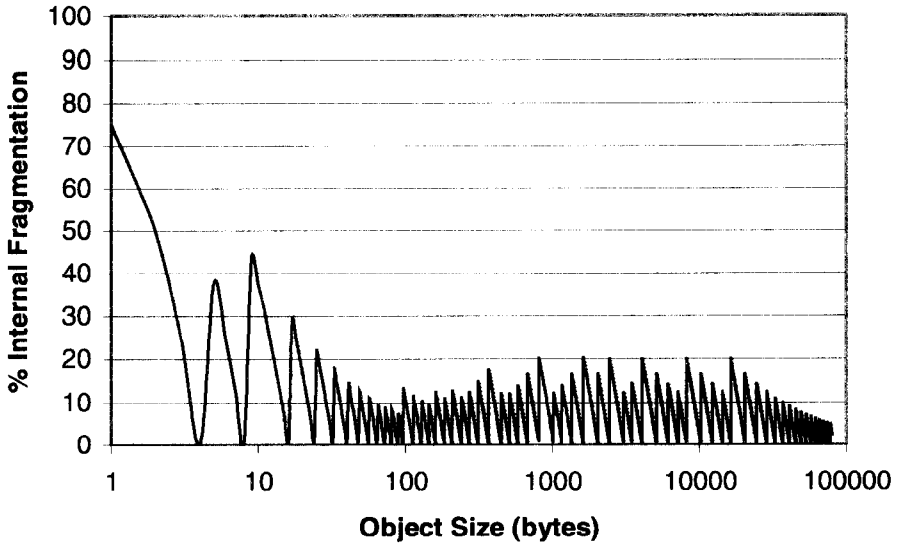

Figure 2. Fragmentation of allocator with small, medium, and large objects

The internal fragmentation is the percentage of wasted space for a given object size. It can be computed by subtracting the requested object size from the real size of the object returned by the allocator and dividing the result by the real size.

$$\%\ Fragmentation = 100.0 * (realSize - requestedSize) / realSize$$

types are aligned to 8 byte boundaries, and pointer types are aligned to 4 byte boundaries. Nothing can be done to reduce the fragmentation for these sizes.

For sizes around a page size, the fragmentation of the allocator without medium objects is excessive and reaches 50%. Especially at sizes 2049 and 4097 bytes (one byte after half the page size and one byte after the page size) the fragmentation reaches 50%. With medium objects however, the internal fragmentation for these sizes is bounded to 20%.

The savings in space that medium-objects give is important for applications that allocate objects around these sizes. We have seen that this is the common case for many applications.

Footprint reduction is based on the observation that moving collectors make the free data contiguous by moving the live data. However, one can more easily make the free data contiguous by moving the free data. This provides a major defragmentation benefit of moving collection without the complexity, restrictions, or expense of moving data and updating pointers.

In footprint reduction, the swap space of the pages that make the free list unusable is decommitted. Later this swap space can be recommitted somewhere else in the address space where these pages can be consecutive.

Figure 3 shows how footprint reduction works. On the left side, there is a representation of the heap in virtual memory and in physical memory. The heap consists of four pages with live objects, three pages with free objects,

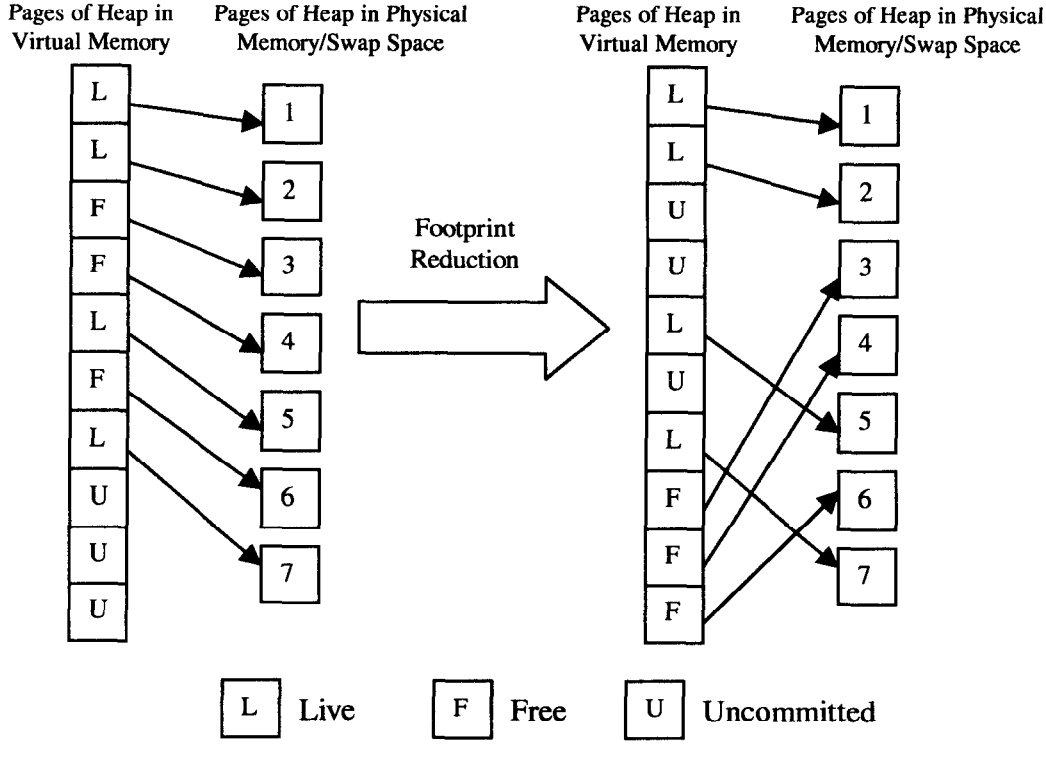

Figure 3. Footprint reduction example

4. FOOTPRINT REDUCTION

The second technique, called footprint reduction, is designed to reduce external fragmentation. It is used to reduce the fragmentation in the free list of large objects. Notice that the small and medium object allocators are relatively unaffected by external fragmentation, because the free elements in a small or medium object allocator are always large enough to satisfy an allocation request. In addition, footprint reduction can decommit pages in the free list, allowing long-running programs to reduce their swap-space requirements when their memory requirements decrease.

and three pages that are uncommitted, i.e., that do not have swap space or physical memory assigned. The three free pages are not consecutive, and therefore a request to the allocator for a three-page object will not be able to be satisfied with the available free pages. Footprint reduction de-commits the three free non-consecutive pages and commits the three uncommitted consecutive pages at the end of the heap. In this way, the request can be satisfied using the same amount of swap space. This is shown in the right side of the figure.

To implement footprint reduction, the pages of the heap are represented by an array of bytes called *page-flags*,

where each byte represents a page. Every bit in a byte represents a different characteristic of the page. For footprint reduction, only three bits in each byte are used: *committed-bit*, *free-bit*, and *recently used-bit*.

At initialization time, the allocator memory maps a large sequence of pages, called *arena*, of uncommitted memory, i.e. only address space in the arena is reserved but no swap space is committed. This means that no other memory map operation will return a page in this range since address space is reserved. However, since no swap space is reserved, a memory read/write operation to a word in this range of pages at this point in the execution of the program may result in a segmentation violation. Finally, *page-flags* are cleared.

During a memory request of a large object, if the request cannot be satisfied with the existing objects in the free list, the allocator will do the following. It will search in *page-flags* for a range of uncommitted pages large enough in the arena to satisfy the request or some larger amount if the requested size is too small. Then it will call a virtual memory operation to commit swap space for this range of pages. The reason a larger amount is committed if the requested size is too small is to amortize the cost of the virtual memory operation. This range of pages is returned to the free list and the corresponding committed-bits and free-bits updated. Finally, the request is satisfied. If the number of consecutive uncommitted pages in the arena is not enough, another large group of uncommitted memory is mapped and added to the arena.

Whenever a group of pages is returned to the free-list, the corresponding recently used-bits and the free-bits are set. The recently used-bits tell the allocator that the corresponding page has been recently used and that it is not a good candidate for footprint reduction.

During a footprint reduction, the pages that have the recently used-bit cleared are uncommitted, i.e., the swap space they are using is returned to the operating system and the heap shrinks. Finally, the recently used-bit is cleared for all the pages.

The frequency footprint reduction is executed is linked to the activity of the allocator. In our implementation, a footprint reduction is performed after a pre-specified number of garbage collections. A page that has not been used during this pre-specified number of garbage collections is uncommitted and returned to the operating system. Programs that explicitly manage their memory will run a footprint reduction after a pre-specified number of bytes have been explicitly returned to the free-list. Alternatively Great Circle supplies a footprint reduction procedure that the program can explicitly call after periods of heavy allocation.

Alternatives to the commit/uncommit operations are the map/unmap operations. The difference is that the uncommit memory operations return the associated swap space to the operating system, however the address space range is kept. The unmap operation returns to the operating system both the address space and the swap space. We have decided to use the commit/uncommit operations over a contiguous uncommitted arena because it allows recycling address space. The map operation can return memory mappings that are not contiguous making the heap have holes.

Another modification for footprint reduction is that during the allocation of large objects every search in the free list always starts from the first block in the list. This will result in reusing the same large objects most of the time

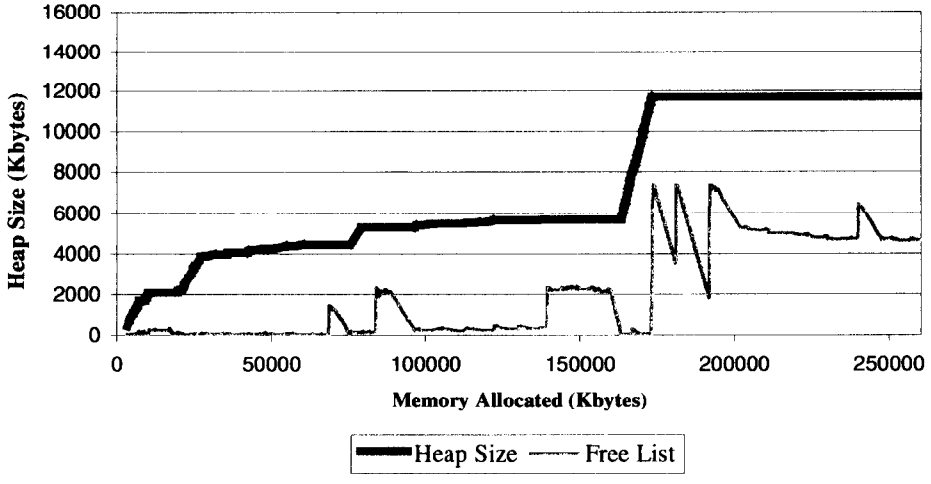

Figure 4. Netscape heap size without footprint reduction

and leaving the least used objects at the end of the list. If a new search started where the previous one ended it would reuse all the objects in the list and would not give the opportunity for footprint reduction.

A good side effect of footprint reduction is that pages that are black-listed [3], and therefore cannot be used because they are being pointed by false pointers, are unmapped if they continue black-listed for several consecutive allocations.

5. EXPERIMENTS

In this section, we show the advantages of using footprint reduction in a long-lived program that is subject to changing loads.

The program used to show the advantages of the techniques mentioned before is a web-browser called Netscape. Netscape is a long-lived program that may run for hours at a time and is subject to multiple loads. For instance, HTML documents come in very different sizes and may contain pictures with sizes that go from small icons to very large GIF files. The allocation pattern of Netscape depends of the documents requested by the user, and the interaction with the browser.

The experiments shown here run in a Sparc Station 10 running the Solaris Operating System. The Great Circle library is linked into Netscape and the X-server at run-time using the injection technique described in [10], and therefore no recompiling or linking is necessary. Great Circle provides a substitute for the malloc/free operations. In the experiments shown, explicit memory management is enabled, and therefore the free operations really return memory back to the free lists. The garbage collector only runs to guarantee that the program will not run out of memory because of memory-leaks.

Figure 4 shows the allocator behavior during a session of 5 minutes with Netscape. During this time, about 40 documents were loaded. The graph shows how the sizes of the heap and the free list change. Without footprint reduction, the heap increases monotonically after bursts of allocation. However, it does not go back to its original size when the periods of heavy allocation are over.

Figure 5 shows the behavior of the allocator in an equivalent session with Netscape but now with footprint reduction. The size of the heap now adapts to the requirements of the application, returning free memory back to the operating system when the memory is no longer needed by the application, or when the memory is too fragmented to satisfy requests for large objects. A footprint reduction phase runs every time 100 Kbytes of memory are returned to the free-list. This parameter can be fine-tuned by the application. Additionally, Great Circle gives to the application the possibility to call the footprint reduction function after periods of heavy allocation.

6. CONCLUSIONS

Footprint reduction is useful for long-lived applications because it reduces fragmentation and therefore reduces the sizes of the heap. This is especially true for applications that have changing workloads such as Netscape.

The advantages of medium objects are not obvious in the previous experiments. Further experimentation with real applications is necessary to show the impact of using medium objects.

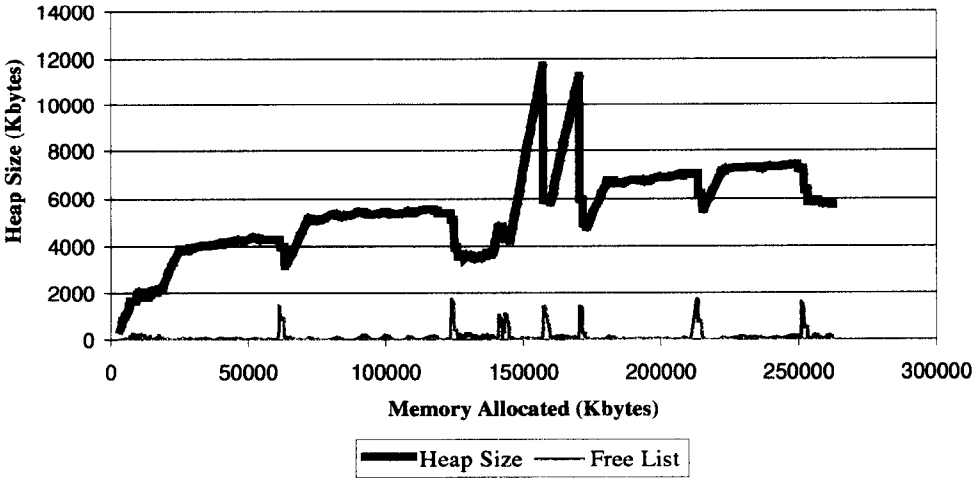

Figure 5. Netcape heap size with footprint reduction

At least in Netscape, footprint reduction does not seem to have a great impact in the execution overhead. However, if the execution overhead is a problem, the frequency footprint reduction runs could be controlled in a way that footprint reduction does not cause a great execution overhead.

One drawback of footprint reduction is its coarse granularity that may not seem adequate for applications that use mostly small objects. More experimentation with other real applications is necessary to see when that is the case. Also the use of medium objects may conflict with footprint reduction, especially when free memory used by medium objects can not be uncommitted because there are still live objects in the pages they share. More experimentation is necessary to explore this problem. We believe that every application has its own allocation patterns and not all applications may benefit from the techniques described. However, both these strategies have performed well for us in a commercial garbage collection implementation used in a wide variety of applications.

7. ACKNOWLEDGMENTS

We would like to give thanks to Hans Boehm for his comments and for writing his conservative garbage collection library. We also would like to give thanks to the organizers of this event and to the reviewers for their invaluable comments.

8. REFERENCES

[1] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Equipment Corporation Western Research Laboratory, Palo Alto, California, February 1988.

[2] Yves Bekkers and Jacques Cohen, editors. International Workshop on Memory Management, number 637 in Lecture Notes in Computer Science, St. Malo, France, September 1992. Springer-Verlag.

[3] Hans-Juergen Boehm, Space-efficient conservative garbage collection. In Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation [9], pages 197-206.

[4] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. Software Practice and Experience, 18(9):807-820, September 1988.

[5] Jacques Cohen and Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. ACM Transactions on Programming Languages and Systems, 5(4):532-553, October 1983.

[6] David L. Detlefs. Concurrent, Atomic Garbage Collection. PhD thesis, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1991. Technical Report CMU-CS-90-177.

[7] OOPSLA '93 Workshop on Memory Management and Garbage Collection, October 1993. Available for anonymous FTP from cs.utexas.edu in /pub/garbage/GC93.

[8] Richard E. Jones and R. Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. 1996. Wiley.

[9] Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation, Albuquerque, New Mexico, June 1993. ACM Press.

[10] Gustavo Rodriguez-Rivera and Vincent Russo. Non-intrusive cloning garbage collection with stock operating system support. Software Practice and Experience, 27(8), August 1997.

[11] Paul R. Wilson. Uniprocessor garbage collection techniques. In Bekkers and Cohen [2], pages 1-42.

[12] Paul R. Wilson. Garbage Collection. Computing Surveys, 1995. Expanded version of [11]. Draft available via anonymous internet FTP from cs.utexas.edu as /pub/garbage/bigsurv.ps.

[13] Paul R. Wilson and Mark S. Johnstone. Truly real-time non-copying garbage collection. In OOPSLA '93 Workshop on Memory Management and Garbage Collection [7]. Expanded version workshop position paper submitted for publication.

[14] Paul R. Wilson, Mark S. Johnstone. Michael Neely, and David Boles. Dynamic Storage Allocation: A survey and Critical Review. Available for anonymous FTP from cs.utexas.edu in /pub/garbage/.