Presented at GC'91 Workshop on Garbage Collection in Object-Oriented Systems at OOPSLA'91.

Cache-Conscious Copying Collectors

Henry G. Baker

Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436
(818) 986-1436 (818) 986-1360 (FAX)

Abstract

Garbage collectors must minimize the scarce resources of cache space and off-chip communications bandwidth to optimize performance on modern single-chip computer architectures. Strategies for achieving these goals in the context of copying garbage collection are discussed. A multi-processor mutator/collector system is analyzed. Finally, the Intel 80860XP architecture is studied.

1. Introduction.

The ubiquity of on-chip caches in modern processor architectures is forced by the increasing latency of on-chip memory relative to off-chip memory. Unfortunately, this latency is also accompanied by the restricted bandwidth of small pin-outs. As a result, the management of on-chip memory space and off-chip communications traffic has become the major problem in gaining fast execution times on these processors. Lam *et al* [Lam91] graphically demonstrate this issue for the problem of multiplying large matrices, where the measured performance of one processor was increased from 0.9 MFLOPS to 4 MFLOPS through the more careful management of these scarce resources. In other words, modern processor chips are no longer "CPU-bound", but "I/O-bound".

Symbolic processing has always been more "I/O-bound" than "CPU-bound" because very few symbolic processing algorithms involve the heavy bit-twiddling of floating-point arithmetic. Numeric processing with special-purpose floating-point hardware can also be "I/O-bound" [Lam91], and the traditional remedy is increased bandwidth to memory. Since the control structures of numeric programs tend to be quite oblivious to the actual data values, a programmer or compiler can utilize quite sophisticated prefetching strategies (e.g., vector registers) to ensure that the data is available when it is needed. This traditional strategy is no longer appropriate for today's limited bandwidth processors, hence the great interest in SIMD architectures in which the required bandwidth is spread over many memories and processors.

Unfortunately, the control structures of symbolic processing programs tend to be extremely data-dependent—e.g., run-time method determination in object-oriented programming languages—and hence the potential for programmer-directed and compiler-directed prefetching strategies is limited. While the general problem of optimizing high-speed symbolic processing with modern chip technology is quite difficult, at least one portion of symbolic processing seems regular enough to be amenable to more sophisticated memory space and bandwidth management strategies—garbage collection.

2. Copying Garbage Collection

Minsky is credited with the first copying garbage collector [Minsky63], Cheney with an elegant 2-space model [Cheney70], and Baker with a real-time version of Cheney's algorithm [Baker78]. Because the first garbage collector did not copy, and because a large fraction of implemented garbage collectors do not copy, it is important to review the pro's and con's of copying garbage collection.

A non-copying garbage collector (NCGC) has the advantage of requiring less address space, because objects do not ever occupy two different locations at the same time. NCGC does not move objects behind the compiler's back, and so does not invalidate certain pointer register caching compiler optimizations [Chase87]. NCGC can be used in a "conservative" garbage collection environment because it is necessary to find only *one* pointer to each active object, instead of *all* pointers to all active objects [Boehm88]. Finally, the smaller address space and less object motion would seem to optimize performance on cache-based architectures [Zorn91].

A copying garbage collector (CGC) has the advantage of trivial allocation of non-homogeneous objects due to its compact free area, which advantage is essential in a "real-time" environment [Baker78]. CGC can also be used to improve "locality" for virtual memory or cache purposes, although NCGC does better than might be expected, due to its linear scan for "sweeping" [Clark77]. CGC is a "single-phase" algorithm, which makes it easier to understand and analyze than the two-phase "mark/sweep" NCGC algorithm.[1] CGC can more easily expand, and more importantly contract, the amount of space under management than can NCGC [Baker78]. Of all of these issues, the simplicity of storage management under widely varying demands is probably the biggest single benefit of CGC.

3. Overcoming the Drawbacks of Copying Garbage Collection

Many of the apparent drawbacks of copying garbage collectors can be overcome. For list pair objects, the apparent doubling of address space requirements is offset through the use of CDR-coding, which allocates the successive elements of lists in adjacent locations, so that no actual "CDR" pointer is required.[2] For large objects, one can finesse the need to copy by managing only small "headers"; unfortunately, this brings back the storage fragmentation problem that copying eliminates. A better approach is to avoid copying large objects by modifying the page map instead;[3] while this scheme does not save any virtual address space, it does save physical address space and copying overhead. Of course, the system must support page maps which alias different virtual addresses; this requirement has significant implications in cache design, translation lookaside buffer design and operating system design. In a properly designed system, however, the aliased cache locations are accessible without reloading, thus avoiding unnecessary overhead.

Although the forwarding machinery of a copying garbage collector involves some complexity, it can also be used in support of "lazy allocation" [Baker92CONS]. Lazy allocation is a degenerate form of generational garbage collection in which the each generation is a stack frame. All allocation is initially performed in the current stack frame, and an object is tenured and copied when a reference is installed in a higher stack frame or in the global environment. Since CONS is a subprogram, its arguments are already (conceptually) "on the stack", and so lazy allocation involves trivially returning a pointer to this block of arguments—i.e., C's `alloca`. Since a substantial fraction of allocations either already conform to stack discipline, or can be made to conform through proper programming style, lazy allocation can use the stack to filter out these allocations before the garbage collector ever becomes aware of their existence. We believe that lazy allocation, by removing the load of these extremely short-lived objects, reduces much of the need for more traditional generational systems, and allows the garbage collector to spend its efforts on more permanent objects.

It has long been known that the Cheney list copying algorithm has the potential for substantial optimization in a virtual memory and/or cache environment [Barbacci71]. The allocate and scan pointers both step sequentially through memory, offering opportunities for optimizations such as prefetching. In a "stop-and-copy" GC implementation using the Cheney algorithm, the storage accessed by the allocate pointer is "write-only", and therefore should never be "paged in" (or

[1] A single-phase NCGC has recently been described [Baker92Tread].

[2] Although CDR-coding is currently out of fashion due to not being required by Tak and 8Queens, the inexorable pressure to reduce off-chip bandwidth will force its revival.

[3] This "map-diddling" scheme requires that large objects not share pages with other objects; this requirement is easily arranged, and does not significantly increase overhead, so long as the objects are larger than one page.

"write-allocated" in the cache). While the storage accessed by the scan pointer is read and updated in place, its page/cache line can be marked for replacement immediately after being completely processed, because it will not be referenced again during the garbage collection. Of course, copying of any kind in a direct-mapped cache can be disastrously inefficient in the unlucky case where both the "from" and "to" locations map to the same cache line; obviously, this situation must be specially handled.

The most troublesome references in the Cheney algorithm are its references to "fromspace" to check for and obtain a forwarding pointer. Consider the situation in which we have the most information—the Cheney algorithm is run twice in succession on exactly the same data. The data under the scan pointer is used to access the fromspace, to see whether it has a forwarding pointer yet. If the object in fromspace has not yet been moved, then there is no forwarding pointer, and the object must be copied to "tospace" at the location of the allocate pointer. *But since this is the second running of the algorithm on the same data, the object accessed must have been at the same relative location in "fromspace" as the allocate pointer is in "tospace"!* In other words, there is a ghost or shadow in "fromspace" of the allocate pointer in "tospace". As a result of this ghosting/shadowing, the page/cache line in the shadow of the allocate pointer will almost certainly already be in memory, and therefore the first access to an object will probably not cause a page fault/cache miss! Since the objects being accessed in the shadow of the allocate pointer are also updated with forwarding pointers, this updating will also likely occur in the cache. In other words, if the graph structure being copied is really a tree—i.e., no shared nodes—then Cheney will copy in a highly local fashion, so long as no rearrangement of the objects in address space is actually being done [Baker80].

The ghosting/shadowing argument works equally well for other copying algorithms, including those using depth-first orders, so long as the same strategy is used every time. Since depth-first ordering is more likely to be preserved in the presence of mutator activity, ghosting/shadowing is also likely to be a factor in depth-first incremental/real-time copying garbage collectors. Of course, "out-of-order" copying caused by the mutator should be performed in a separate region from "in-order" copying caused by the collector itself.

References which do obtain a forwarding pointer—i.e., references which are shared and whose reference count exceeds 1—are much less predictable. The fact that there are relatively few shared objects whose reference count exceeds one does not help very much, unless the garbage collector knows which objects are shared. It is therefore important to understand why forwarding pointers are important. Forwarding pointers are used to preserve the semantics of shared objects, so that the sharing relationships of the copy are isomorphic to the original. If a garbage collector does not use forwarding pointers, then it will "unshare" all objects during copying, leaving all objects with exactly one reference, and it will diverge on cyclic list structures. In other words, forwarding pointers are used to preserve "object identity" [Baker93ER].

Based on our intuition and experience, we feel that a large fraction of the unshared objects in Clark's measurements [Clark77] are really "functional objects" (e.g., CONS cells), which are never directly modified (e.g., by RPLACA/D). The usual reason for preserving sharing is so that a modification using one path to an object will be visible to all other paths. In the case of functional objects, however, the object cannot be modified, and therefore such an operational definition of "object identity" will not work. So long as any predicates for comparing object identity (e.g., EQ/EQL) suitably distinguish between functional and non-functional objects [Baker93ER], then a program will have no way of distinguishing copies of a functional object from the "original". Thus, for functional objects, a forwarding pointer is not strictly necessary, although without a forwarding pointer, we may do more copying than is strictly necessary, and thereby increase the amount of storage required. However, we can also efficiently and incrementally calculate the "extended size" of a functional object while it is being created, so that the garbage collector can quickly estimate the cost of copying versus sharing the object [Baker93ER]. Therefore, if a significant fraction of objects are functional, then a copying garbage collector which distinguishes functional from non-functional[4] objects can outperform one which does not.

Shared non-functional objects still comprise a significant fraction of references, however. We can further eliminate the copying and forwarding of non-functional objects which for some reason cannot become garbage—e.g., objects referenced from "outside", or otherwise permanent objects.

Finally, we must deal with the remainder of shared temporary non-functional objects. The forwarding pointers for these objects will be referenced again at some point during the copy, so if this forwarding pointer has fallen out of the cache/main memory, then accessing it will cause a cache miss/page fault. If we have already implemented the other optimizations which make the best use of the cache/main memory, then almost all of the misses/faults will be of the forwarding pointer reference type. In other words, the cache/main memory becomes a cache for storing mainly forwarding pointers. Obviously, in order to make the best use of this space, we should make sure that the density of forwarding pointers is at a maximum. In this case, it might prove beneficial to keep forwarding pointers in a separate table, instead of storing them in the fromspace. This is certainly the case with large objects, but the cost/benefit for smaller objects requires additional study.

The latency of pointer forwarding is a limit to copying collector performance. The traditional Cheney GC must wait until the forwarding pointer is available before it can move the scan pointer. However, if latency—not bandwidth—is the bottleneck, one can perform a number of forwarding pointer lookups in a pipelined fashion, so that while the first lookup may have significant latency, the additional forwarding pointers will arrive very quickly, and the time for the whole sequence is significantly smaller than the product of the number of lookups and the random latency. In fact, Appel uses this scheme in his vectorized copying garbage collector [Appel89].

Unfortunately, this pipelining scheme does not reduce the communication bandwidth, since every pointer must be traced, even though most of them will not be forwarded. The scheme does have the advantage, however, of not tying up space in the cache, which might be used more profitably by the mutator instead of the collector. For example, the original implementation of Baker's RTGC on the MIT Lisp Machine (not generational) required special paging code to keep the GC from pushing out all of the mutator pages. Even a "stop-and-copy" GC will push out all mutator pages, and recent data indicate that the cost of restoring pages/lines after a context switch may be much higher than previously thought [Mogul91].

If we can reliably distinguish forwarding references with a simple calculation before performing the tracing a reference, then we can pipeline a large number of forwarding lookups to run in parallel, not only with the mutator, but also with the copying/allocation portion of the garbage collector. All that is needed is a presence/scoreboard/I-structure/future bit, like that found in the Denelcor HEP, which indicates that the contents of the requested memory reference have arrived. The pipeline of forwarding lookups is then loaded with a series of memory-memory moves, and it will drain itself automatically.

Consider a scheme for distinguishing first (copying) from non-first (forwarding lookup) references. Suppose that every pointer is tagged at the time it is stored by the garbage collector with a bit indicating whether it is the first reference to its target object; this information is trivially computed by the garbage collector while copying. We must now make sure that the mutator does not destroy this information. If all new conses are in a separate region, then the garbage collector can treat these differently from previously copied conses, and thus new conses do not have to contain this information. The only problem occurs when the mutator changes a pointer in an old cons using RPLACA/D. RPLACA/D of an old pointer into an old cons is the difficult case. There are two

---

[4]Advocates of functional objects have always felt that the antonym of "functional" was "non-functional"! Perhaps now the politically correct term should be "other-functioned" or "differently-functioned" objects.

possible solutions—either have the mutator install an invisible pointer to a newly consed hidden cell which the GC will recognize as special, or have the mutator do additional work. If a back-pointer is stored in every object which points back to its first reference, then RPLACA/D can compare this back-pointer to its first argument. If RPLACA/D is storing into a location whose address is smaller than the back-pointer, then RPLACA/D can update the back-pointer to point to its first argument, which is now earlier than the previous first reference.

Unfortunately, schemes for flagging the first reference (in GC order) provide only "hints". If a pointer is encountered which indicates that it is the first reference, then it will be. On the other hand, the first reference pointer itself may become inaccessible, and the GC will encounter a non-first reference before it has copied the target object. In other words, the GC must be prepared to handle the case in which a forwarding pointer is expected, but it is not there.

4. Parallel Garbage Collection

Since copying garbage collection increases the traffic to memory and pushes out mutator information from the cache/main memory, a copying garbage collector implemented by means of a second processor chip looks promising. So long as cache coherency between the mutator and collector chips can be assured, and so long as the two caches do not fight for control of the various cache lines, then a 2-chip system could provide a relatively inexpensive speedup for symbolic processing tasks.

The first line of defense in a parallel symbolic system should be *lazy allocation* [Baker92CONS]. Because lazy allocation efficiently allocates and deallocates very short-lived objects as part of the normal operation of the call stack, it meshes very well with normal mutator cache behavior. Furthermore, in a parallel system with multiple mutators, each processor can lazily allocate in its own private stack without any interference from the other processors.

We now consider a 2-processor system using the original RTGC scheme [Baker78]. The object of the game is to fool the mutator into thinking that the collection was finished completely during the "flip". We first suppose that the mutator is stopped during a "flip", so that its TLB and data cache can be invalidated. Then, when it resumes processing, it starts accessing only the tospace. While several schemes have been proposed which utilize the mutator's virtual memory page map to protect the mutator from unscanned objects [Shaw87] [Appel88], we will consider a scheme involving more cooperation from the mutator.

Suppose that the mutator attempts to perform a CAR/CDR and finds a fromspace pointer. Suppose the mutator attempts to handle the "move" itself. Then it will access fromspace to find a forwarding pointer; if a forwarding pointer is not found, then the object must be transported into tospace. If the mutator accesses fromspace, then it will have a miss on its TLB, as well as a miss on its cache, which will cause some mutator information to be removed from these caches. If the mutator then finds no forwarding pointer, then it must copy the information into tospace, using either its own allocate pointer, or somehow gaining access to the collector's allocate pointer. Accessing the portion of tospace where the object is being copied will also likely cause cache misses and replacements, although any tospace information brought into the mutator cache in this way will very likely be referenced, and we may have performed a beneficial prefetch.

Suppose that the architecture offers a way to read bypassing the data cache. Then the fromspace can be accessed without affecting the contents of the cache, and the transport can also be performed without affecting the contents of the mutator cache. Unfortunately, the TLB cannot usually be bypassed, so its contents may be replaced.

If the mutator does some of its own transportation, then some synchronization is required between the mutator and the collector. The mutator can look for a forwarding pointer without synchronizing with the collector, but if one is not found, then it must synchronize with the collector. If synchronization is inevitable, then perhaps the collector should do the actual copying itself. This requires that the mutator tell the collector which object to move. The mutator puts the address of the object in a mailbox which the collector checks on every cycle, and starts spinning on the object in fromspace waiting for a forwarding pointer to appear. The collector finds the object address in its mailbox, removes it and "transports" the object, installs the forwarding pointer, and then goes back to its scanning. No special synchronization hardware is required, because only the collector is allowed to modify fromspace with forwarding pointers. Furthermore, if there are only the two processors in the system, then the mailbox consists of a single memory location.

Assuming modern MESI or MEOSI cache consistency protocols (e.g., IEEE Futurebus), the mutator and the collector will not fight over the fromspace forwarding address cache line because both processors have a copy—i.e., the line has status "Shared". Only when the collector stores the forwarding address will it invalidate the mutator's cache line, which will cause it to get the updated information.

When the collector has finished, it sends a message in a special mailbox which the mutator checks from time to time inside its allocation routine, and the collector then goes to sleep. When the mutator decides the time is ripe, it "flips". To flip, it sets up certain registers and wakes up the garbage collector, which starts by scanning the mutator stack from top to bottom. The mutator spins on the collector scanner, waiting for it to finish the top stack frame. Once the top stack frame has been scanned, the mutator can resume processing. From then on, every time the mutator wants to pop the stack, it must spin until that stack frame has been scanned.

Should the mutator invalidate its TLB and flush its cache on a flip? The mere action of the collector installing forwarding pointers during the flip will flush and/or invalidate most mutator cache lines, so an explicit flush should not be necessary. Furthermore, it is likely that right after a flip the mutator will have to do a significant amount of pointer forwarding itself, in which case the old TLB and/or cache entries could still be useful.

We have argued that objects larger than one page should be page-aligned and not be shared with other objects. This allows the object to be "copied" by merely changing the page map instead of physically moving the object. In this case, older mutator cache entries may still be valid once their new virtual address has been updated, and it is therefore important that these entries not be flushed during a flip. When a pointer is found which points to a large object in fromspace, the mutator puts it into the mailbox and spins on its forwarding address, as before. When the forwarding address is obtained, however, the mutator tospace page map is updated to point to this forwarding address, and the mutator can continue.

In summary, modern cache consistency protocols should allow most garbage collection overhead to be removed to a second processor. The vast majority of traffic would flow through the GC processor instead of the mutator, allowing the mutator to keep its cache locality and make its mutating more efficient.

5. A Case Study—The Intel 80860XP Architecture

The Intel 80860 is a modern 64-bit RISC architecture with on-chip instruction and data caches. It has a standard 32-bit byte-addressed virtual address space which is mapped onto a 32-bit physical address space. The page map has the standard read-write, read-only and no access capabilities, as well as reference and dirty bits. There is an on-chip translation lookaside buffer (TLB) which caches page table entries. The 80860 has single instructions which can read and write quad-word (16-byte) blocks, although these instructions do not finish in a single cycle.

The 80860XP 16Kbyte on-chip data cache is 4-way set-associative, with a line size of 32 bytes. This on-chip cache can be easily extended into an off-chip 256Kbyte/512Kbyte SRAM cache. The 80860 architecture has "pipelined" load instructions which normally bypass the data cache, although for consistency they load data from the cache if there is a cache hit. The 80860 store instructions write back to the cache if there is a cache hit, or write directly back to memory if there is not a cache hit; they do not "write-allocate" in the cache. In other words, only normal reads can be used to load the data cache.[5]

The 80860XP utilizes a MESI cache coherency protocol based on a global bus and "snooping". The caches are "write-back" by default, but can be made "write-through" on a per-page basis, or even "non-cachable".

The 64-bit bus and quad-word load/store capabilities give the 80860 substantial bandwidth to support a copying garbage collector, although the bandwidth of the 80860 bus will only support a load every other cycle [Moyer91]. The cache-bypassing capabilities of the "pfld" instruction allow the collector to operate without damaging the cache locality of the mutator. Unfortunately, "pfld" loads a floating-point register with latency of at least 6 cycles, and the information must then be transferred to an integer unit with a latency of 3 before before it can be used as an address. The set-associativity of the cache avoids the copy-thrashing behavior of a direct-mapped cache. Finally, the advanced cache coherency protocol should allow a dedicated garbage collector processor to reduce the load on a main mutator processor.

6. Conclusions

We have examined the details of the interaction of a copying garbage collector with the memory system of modern RISC architectures. In most cases, the drawbacks of a copying collector can be easily ameliorated, especially if mechanisms for bypassing the cache are available. We also found that modern cache coherency protocols support parallel garbage collection rather nicely. Finally, we studied the Intel 80860XP architecture, and found that it is very suitable for copying garbage collection, in both single-processor and multi-processor configurations.

We are in the process of constructing a simple experiment to measure the improvement in copying collector performance due to the optimizations discussed above. We are particularly interested in the improvements due to bypassing the cache.

References

Appel, Andrew W. "Garbage Collection Can Be Faster Than Stack Allocation". *Info. Proc. Let. 25* (1987),275-279.

Appel, Andrew W.; Ellis, John R.; and Li, Kai. "Real-time concurrent garbage collection on stock multiprocessors". *ACM Prog. Lang. Des. and Impl.*, June 1988,11-20.

Appel, Andrew W. "Simple Generational Garbage Collection and Fast Allocation". *Soft. Prac. & Exper. 19*,2 (Feb. 1989),171-183.

Appel, A.W., and Bendiksen, A. "Vectorized Garbage Collection". *J. Supercomputing 3* (1989),151-160.

Baker, Henry G. "List Processing in Real Time on a Serial Computer". *CACM 21,4* (April 1978), 280-294.

Baker, Henry G. "The Paging Behavior of the Cheney List Copying Algorithm". Tech. Rept., CS Dept., U. of Rochester, NY, 1980; copies available from the author.

Baker, Henry G. "Unify and Conquer (Garbage, Updating, Aliasing, ...) in Functional Languages". *Proc. 1990 ACM Conf. on Lisp and Functional Progr.*, June 1990, 218-226.

Baker, Henry G. "CONS Should not CONS its Arguments, or, A Lazy Alloc is a Smart Alloc". ACM *Sigplan Not. 27*,3 (March 1992), 24-34.

Baker, Henry G. "Equal Rights for Functional Objects or, The More Things Change, The More They Are the Same". ACM *OOPS Messenger 4*,4 (Oct. 1993), 2-27.

Baker, Henry G. "The Treadmill: Real-Time Garbage Collection Without Motion Sickness". ACM *Sigplan Not. 27*,3 (March 1992), 66-70.

Barbacci, M. "A LISP Processor for C.ai". Memo CMU-CS-71-103, CMU, Pittsburgh, 1971.

Bishop, P.B. *Computer Systems with a very large address space and garbage collection*. Ph.D. Thesis, TR-178, MIT Lab. for Comp. Sci., Camb., MA, May 1977.

Boehm, Hans-J., and Demers, Alan. "Garbage Collection in an Uncooperative Environment". *Soft. Pract. & Exper. 18*,9 (Sept. 1988),807-820.

[5]This cache behavior means that high-level language compilers must not "optimize" away certain apparently redundant loads, which are used only to load the cache.

Chase, David. "Garbage Collection and Other Optimizations". PhD Thesis, Rice University Comp. Sci. Dept., Nov. 1987.

Cheney, C.J. "A Nonrecursive List Compacting Algorithm". *CACM 13*,11 (Nov. 1970),677-678.

Clark, D.W., and Green, C.C. "An Empirical Study of List Structure in LISP". *CACM 20*,2 (Feb. 1977),78-87.

Fisher, D.A. "Bounded Workspace Garbage Collection in an Address-Order Preserving List Processing Environment". *Inf.Proc.Lett. 3*,1 (July 1974),29-32.

Gelernter, H., *et al*. "A Fortran-Compiled List-Processing Language". *J. ACM 7*,2 (Sept. 1960),87-101.

Hederman, Lucy. "Compile Time Garbage Collection". MS Thesis, Rice University Computer Science Dept., Sept. 1988.

Lam, Monica S., *et al*. "The Cache Performance and Optimizations of Blocked Algorithms". *ACM ASPLOS-IV, Sigplan Not. 26*,4 (April 1991),63-74.

Lieberman, H., and Hewitt, C. "A Real-Time Garbage Collector Based on the Lifetimes of Objects". *CACM 26*, 6 (June 1983),419-429.

MacLennan, B.J. "Values and Objects in Programming Languages". *Sigplan Not. 17*,12 (Dec. 1982),70-79.

McDermott, D. "An Efficient Environment Allocation Scheme in an Interpreter for a Lexically-scoped LISP". *1980 Lisp Conference*, Stanford, CA, Aug. 1980,154-162.

Minsky, M.L. "A LISP garbage collector algorithm using serial secondary storage". MIT AI Memo 58, Oct. 1963.

Mogul, J.C., and Borg, A. "The Effect of Context Switches on Cache Performance". *ACM ASPLOS-IV, Sigplan Not. 26*,4 (April 1991),75-84.

Moon, D. "Garbage Collection in a Large Lisp System". *ACM Symp. on Lisp and Functional Prog.*, Austin, TX, 1984, 235-246.

Moss, J.E.B. "Managing Stack Frames in Smalltalk". *SIGPLAN '87 Symp. on Interpreters and Interpretive Techniques*, in *Sigplan Notices 22*,7 (July 1987), 229-240.

Moyer, Steven A. "Performance of the iPSC/860 Node Architecture". IPC-TR-91-007, Inst. for Parallel Comp., Eng. & Applied Sci., U. of Va., May 1991.

Rees, J. and Clinger, W., et al. "Revised Report on the Algorithmic Language Scheme". *Sigplan Notices 21*,12 (Dec. 1986),37-79.

Ruggieri, Cristina; and Murtagh, Thomas P. "Lifetime analysis of dynamically allocated objects". *ACM POPL '88*,285-293.

Shaw, Robert A. "Improving Garbage Collector Performance in Virtual Memory". Stanford CSL-TR-87-323, March 1987.

Terashima, M., and Goto, E. "Genetic Order and Compactifying Garbage Collectors". *IPL 7*,1 (Jan. 1978),27-32.

Unger, D. "Generation Scavenging: A non-disruptive, high performance storage reclamation algorithm". *ACM Soft. Eng. Symp. on Prac. Software Dev. Envs., Sigplan Notices 19*,6 (June 1984),157-167.

Weizenbaum, J. "Knotted List Structures". *CACM 5*,3 (March 1962),161-165.

Wilson, Paul R. "Some Issues and Strategies in Heap Management and Memory Hierarchies". *ACM Sigplan Not. 26*,3 (March 1991),45-52.

Wise, D.S., and Friedman, D.P. "The One-Bit Reference Count". *BIT 17* (1977),351-359.

Zorn, Benjamin. "The Effect of Garbage Collection on Cache Performance". TR CU-CS-528-91, U. Colorado, Boulder, May 1991,41p.