ACM Sigplan Notices **27**,3 (March 1992),66-70.

The Treadmill:
Real-Time Garbage Collection Without Motion Sickness

Henry G. Baker

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

A simple real-time garbage collection algorithm is presented which does not copy, thereby avoiding some of the problems caused by the asynchronous motion of objects. This in-place "treadmill" garbage collection scheme has approximately the same complexity as other non-moving garbage collectors, thus making it usable in a high-level language implementation where some pointers cannot be traced. The treadmill is currently being used in a Lisp system built in Ada.

INTRODUCTION

In 1978, we presented a relatively simple storage management algorithm using garbage collection ("GC") which was "real-time" ("RT"), in the sense that all of its operations could be bounded by a small constant, except for allocation, which was bounded by a small constant times the size of the object being allocated [Baker78]. Since initialization requires time proportional to the size of the new object, this algorithm was optimum, to within a constant factor. The key ideas of the paper were the tricolor marking scheme and the use of the allocation pointer as a clock to measure the "time" until the garbage collection must be finished.

Our 1978 paper had two major goals—to show that garbage collection could be done in real time, and to show a relatively practical algorithm. After the discovery of the proof shown in that paper, and before the discovery of the particular algorithm shown in that paper, we considered a number of different strategies for implementing a real-time garbage collector. The search space included a number of different dimensions, including copying v. non-copying, breadth-first v. depth-first, mark-sweep v. non-mark-sweep. The copying GC which appeared in the paper was chosen because 1) it was space-efficient, which appeared to be important for an embedded computer with all "real" (non-virtual) memory; 2) it compacted by copying, which allowed for the simplest allocation strategy—pointer incrementation; and 3) it had a single phase, unlike the 2-phase mark-sweep algorithm.

We have since learned that compile-time garbage collection should be used whenever possible [Chase87] [Chase88] [Hederman88] [Baker90]; that stack allocation should be used more often [Baker91b]; that functional objects should be treated differently from non-functional objects [Baker93]; that depth-first copying often causes fewer faults in a virtual memory and/or caching environment [Moon84] [Andre86] [Wilson91]; that the asynchronous movement of objects is detrimental to compiler optimization [Chase87] [Chase88]; and that more efficient allocation strategies exist [Brent89] [White90]. A "conservative" garbage collector [Boehm88] works much better without copying, since it can never be sure that all pointers to an object have been found and updated. Due to the greater perceived costs of copying and due to the greater perceived benefits of not copying, it now seems worthwhile to revisit an algorithm which lost the initial real-time GC face-off.

TRICOLOR MARKING

We use 3 colors to mark the nodes of a rooted directed graph—white, grey and black.[1] At the commencement of marking, all the nodes are white. We then mark the root nodes grey. At any

[1]Given the date of this real-time GC algorithm (1976), we now wish that we had used the colors white, red and blue.

point during the marking, we find a grey node, darken all of the nodes it points to, and then blacken it. Marking terminates when there are no more grey nodes.

Real-time garbage collection overlaps marking and mutating (user program execution). The mutator is never allowed to see a white node. If this policy is ever in danger of being violated,[2] the mutator marks the white node due to be accessed and continues. Since a black node can never point directly to a white node without a grey node intervening, and since the mutator only sees grey or black nodes, this marking by the mutator is not harmful. When marking is done, we interchange the interpretation of the colors white and black (at this point there are no grey cells), mark the roots grey, and then restart the algorithm.

The copying algorithm in the 1978 paper mapped white nodes into "fromspace" nodes, grey nodes into "tospace" nodes which had been copied but not yet scanned, and black nodes into "tospace" nodes which had been both copied and scanned. This is not the only possible mapping, however.

IN-PLACE GARBAGE COLLECTION

The only requirements of tricolor marking are: 1) it is easy to enumerate free cells for allocation; 2) it is easy to enumerate grey cells; 3) it is easy to determine the color of a cell; 4) it is easy to change the color of a cell; and 5) it is easy to interchange the interpretation of the colors white and black.

Doubly-linked lists [Knuth73,2.2.5] satisfy these requirements. Consider a system of Lisp-like pairs which have two extra "hidden" pointer components. Initially, the free-list is a doubly-linked list through these hidden pointer components. When a cell is allocated ("consed"), it is removed from the free-list doubly-linked list, and inserted into the non-free-list doubly-linked list.

In a non-real-time system, marking begins when the free-list becomes empty. All cells are on the non-free-list at this point; i.e., all cells are white. Marking begins by making the root cells grey; i.e., transferring the root cells from the white list to a grey list, which is done by unsnapping the cells from the one list and snapping them into the other list. Marking proceeds by unsnapping cells from the grey list and snapping them into an initially empty black list. Since an empty grey list is easy to detect, the algorithm will terminate with all accessible cells on the black (non-free) list and all inaccessible cells on the white (free) list. The mutator continues after the interpretation of white and black has been interchanged.

A real-time collection system is obtained by overlapping marking and mutating, as in the 1978 paper. If the mutator attempts to access a white cell, it first darkens it by unsnapping it from the white list and snapping it into the grey list for the marker to process; i.e., the algorithm utilizes a "read barrier". The real-time system will require *four* colors, however, since unmarked white cells must be distinguished from cells on the free list;[3] i.e., unmarked white cells must use "off-white" (ecru) instead of the "dead white" of the free-list. At the end of marking, the ecru cells are converted to dead white cells to form the new free list.[4]

It is easy to see that this "in-place" algorithm is real-time, since the basic operations of determining a cell's color, changing a cell's color (including unsnapping and snapping its links), etc., are all constant-time operations. If we incrementally update the simple statistics required to calculate the appropriate "cons/mark" ratio [Baker78], then this in-place system is real-time if and only if the original copying system is real-time.

---

[2]A "read barrier" checks for this violation.

[3][Kung77] also uses the same color scheme for his *parallel* garbage collector, which uses the two phases mark and sweep; that collector does not meet our definition of real-time, however.

[4]This color change to dead white can be accomplished by brute force, as in "Ecru, Brutè!"

ACM Sigplan Notices **27**,3 (March 1992),66-70.

THE TREADMILL OPTIMIZATION

A new (1991) optimization for this algorithm is obtained by linking *all* the cells into into the same large cyclic doubly-linked list (the treadmill "tread"), while keeping the various colors in contiguous subsequences of this list. The four segments—white, grey, black and ecru—are delimited by four pointers—`bottom`, `top`, `free` and `scan`. We use the hidden links "forward" and "backward" to orient this cyclic doubly-linked list. We thus have the pointers and segments in the following cyclic order: `bottom`, ecru, `top`, grey, `scan`, black, `free`, white, `bottom`. When the mutator allocates a cell, the `free` pointer is moved one cell "forward", thus changing a white (free) cell directly into a black (allocated) cell. When the cell under the `scan` pointer has been scanned, the `scan` pointer is moved "backward", thus changing a grey cell into a black cell. To scan a grey cell, its visible pointers must be checked. If a scanned pointer is black or grey, we do nothing, but if the scanned pointer is ecru, it is unlinked from the ecru segment between `bottom` and `top`, and relinked into the grey area, either at the `scan` pointer—for depth-first ordering, or at the `top` pointer—for breadth-first ordering. Notice that only one bit of color distinction must be stored in the cell—whether or not it is ecru.

When the `scan` pointer meets the `top` pointer, the collection cycle has finished, and when the `free` pointer meets the `bottom` pointer, we must "flip". At this point, we have cells of only two colors—black and ecru. To flip, we make ecru into white and black into ecru; `bottom` and `top` are then exchanged. The root pointers are now "scanned" by making them grey; the cells they point to are unlinked from the ecru region and linked into the grey region.(between `scan` and `top`). We can restart the collector, as it now has grey cells to scan.

The "treadmill" optimization eliminates the need to resnap links during mutator allocation[5] and when changing from grey to black. However, we must still resnap links when changing from ecru to grey, since we need to separate the accessible ecru cells from the garbage ecru cells.

In the exposition above, we explicitly "moved" cells from the ecru list to the grey list. On a multi-processor system, however, we might rather move the cell directly from the white list to the black list, but also put it onto a marker stack. In this modified scheme, a grey cell is a black cell which is also on the marker stack. This optimization may be useful in reducing the latency in the mutator's read barrier.[6]

COSTS

We now compare the costs of the treadmill "in-place" algorithm to the 1978 "copying" algorithm. The in-place algorithm requires 2 additional pointers per CONS pair, but it does not require the additional "tospace". Therefore, for CONS pairs, the space requirements are identical, and the in-place algorithm requires less space for larger objects.[7] The cost of resnapping links is probably larger than that of copying for CONS pairs, but for larger objects the in-place algorithm should require less effort.[8]

---

[5] A real-time system may sometimes find it advantageous to increase the number of cells under management by allocating a cell external to the "tread" and snapping it in during a CONS.

[6] [Kung77] uses a queue with two ends ("dequeue") for the same purpose; the second end reduces conflicts between the mutator and the collector. If cells greyed by the mutator were resnapped at the top pointer, while cells greyed by the collector were resnapped at the scan pointer, then we would have a close approximation to Kung's dequeue.

[7] We ignore here fragmentation, which has been called "storage erosion" in real-time systems. Storage erosion is analogous to land erosion—the land is still there, but has become so eroded as to be useless for cultivation.

[8] On a virtual memory system, however, one does not have to physically copy large objects even when one is using a "copying" garbage collector algorithm. This is possible if large objects are always located on their own set of pages, so that the algorithm need only diddle the page map instead of physically copying these objects. This optimization is especially valuable for large structures of raw bits, such as color bitmaps. Therefore, one cost

If one uses a modern RISC architecture with a cache, and if both hidden links of an object occupy the same cache line, then link resnapping may not be nearly as expensive as a count of memory references would indicate. To enhance locality in a dual (mutator-collector) processor system, one could separate the object into two pieces—one piece holding the links visible to the mutator and the other holding the links visible to the collector; this separation would keep the cache consistency protocol from thrashing.

The biggest potential cost of an in-place algorithm, and the factor which lost it a place in the 1978 paper, is the fact that the free-list must be searched if objects of different sizes are managed [Baker89]. In other words, allocation is no longer a simple pointer-increment operation, but a search of a free-list for an amount of storage big enough to satisfy the allocation. Thus, the in-place algorithm appears to be most useful when managing a homogeneous collection of objects.

There are several possible solutions to this allocation problem. Brent [Brent89] showed a first-fit technique for managing storage in which allocation could be performed in O(log(`w`)) time, where `w` is the maximum number of words allocated dynamically. Jon L. White's technique [White90] utilizes a hierarchy of bit-vectors which achieves O(log(`s`)) allocation time, where `s` is the amount of storage under management.

Of course, these techniques only put off the inevitable fragmentation caused by immobile objects of different sizes [Robson74]. Bounds on the sizes of available memory blocks can be obtained by combinatorial arguments; these bounds are not good, but are worst-case, not average bounds. We therefore have a situation with poor worst-case bounds, but good average-case behavior; this may be a trap for real-time systems designers, who should be preparing for the worst, not the average, case [Baker89].

Brenda Baker [Baker85] has shown how a "buddy" storage system can be modified to "make space" by moving objects when a large allocation request cannot be fulfilled normally. Her algorithm operates so long as memory is not already full, and the time to allocate a block is proportional to the size of the block. All of these characteristics would make her scheme seem ideal, except that the original goal of a motionless garbage collector has not been achieved!

Thus, it would appear that if the allocation problem of large immobile objects of different sizes could be solved, then our in-place real-time variant would be an attractive way to collect garbage.

CONCLUSIONS

We have shown an elegant technique based on doubly-linked lists for in-place real-time memory management which is isomorphic to our original copying algorithm. We are using this in-place real-time garbage collector in a Lisp system built on top of the Ada programming language [Baker91a]. Other authors [Yuasa90] [Beaudoing91] have shown in-place real-time algorithms based on the 2-phase mark-sweep algorithm, although [Queinnec89] has shown how to integrate the sweep phase with allocation.[9] [Moss87] uses a similar doubly-linked list to manage the stack frames of Smalltalk. Doubly-linked lists can also be used to convert generational copying garbage collectors [Lieberman83] into in-place algorithms.

ACKNOWLEDGEMENTS

Many thanks to Hans Boehm, Eliot Moss, Paul Wilson and others at the GC'91 Workshop of Garbage Collection in Object-Oriented Systems for their suggestions and feedback on this paper.

---

associated with a relocating collector is saved; other costs remain, however, such as the costs of updating all pointers and foregoing some compiler optimizations.

[9]It is rumored that this idea goes back at least to Fitch and/or Norman at the U. of Bath???

REFERENCES

Andre, David L. *Paging in Lisp Programs*. M.S. Thesis, U. of Maryland, 1986.
Appel, Andrew W., Ellis, John R., and Li, Kai. "Real-time concurrent garbage collection on stock multiprocessors". *Proc. ACM PLDI*, June 1988,11-20.
Baker, Brenda, *et al*. "Algorithms for Resolving Conflicts in Dynamic Storage Allocation". *J. ACM 32*,2 (April 1985),327-343.
Baker, Henry. "List processing in real time on a serial computer". *CACM 21*,4 (April 1978),280-294.
Baker, Henry. "Garbage Collection in Ada". Ada-9X Revision Request#643, Ada Joint Program Office, Oct., 1989.
Baker, Henry. "Unify and Conquer (Garbage, Updating, Aliasing ...) in Functional Languages". *Proc. 1990 ACM Conf. on Lisp and Functional Programming*, Nice, France, June, 1990,218-226.
Baker, Henry. "Structured Programming with Limited Private Types in Ada: Nesting is for the Soaring Eagles". *ACM Ada Letters XI,*5 (July/Aug. 1991),79-90.
Baker, Henry. "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. "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.
Beaudoing, B., and Queinnec, C. "Mark-DURING-Sweep: A Real-Time Garbage Collector". Submitted to *PARLE'91*.
Boehm, Hans-J., and Demers, Alan. "Garbage Collection in an Uncooperative Environment". *Soft. Pract. & Exper. 18*,9 (Sept. 1988),807-820.
Brent, R. P. "Efficient Implementation of the First-Fit Strategy for Dynamic Storage Allocation". *ACM TOPLAS 11*,3 (July 1989),388-403.
Chase, David. *Garbage Collection and Other Optimizations*. Ph.D. Thesis, Rice Univ., Aug. 1987.
Chase, David. "Safety considerations for storage allocation optimizations". *Proc. ACM PLDI*, June 1988.
Hederman, Lucy. *Compile Time Garbage Collection*. MS Thesis, Rice U. Comp. Sci. Dept., Sept. 1988.
Hickey, T., and Cohen, J. "Performance Analysis of On-the-Fly Garbage Collection". *CACM 27*,11 (Nov. 1984),1143-1154.
Knuth, Donald E. *The Art of Computer Programming Vol. I: Fundamental Algorithms, 2nd Ed*. Addison-Wesley, Reading, MA, 1973,634p.
Kung, H.T., and Song, S.W. "A Parallel Garbage Collection Algorithm and its Correctness Proof". Tech. Report, Computer Science Dept., Carnegie-Mellon Univ., May 1977,20p.
Lieberman, H., and Hewitt, C. "A Real-Time Garbage Collector Based on the Lifetimes of Objects". *CACM 26*,6 (June 1983),419-429.
Moon, David. "Garbage collection in a large Lisp system". *Proc. ACM Symp. on Lisp and Funct. Prog.*, 1984,235-246.
Moss, J.E.B. "Managing Stack Frames in Smalltalk". *Sigplan '87 Symp. on Interpreters and Interpretive Techniques*, in *Sigplan Not. 22*,7 (July 1987),229-240.
Nilsen, K. "Garbage Collection of Strings and Linked Data Structures in Real Time". *SW Prac. & Exper. 18*,7 (July 1988),613-640.
Queinnec, Christian, *et al*. "Mark DURING Sweep, rather than Mark THEN Sweep". *Proc. PARLE'89*.
Robson, J.M. "Bounds for Some Functions Concerning Dynamic Storage Allocation". *JACM 21*,3 (July 1974),491-499.
Robson, J.M. "Storage Allocation is NP-Hard". *Info. Proc. Let. 11*,3 (1980),119-125.
White, Jon L. "Three Issues in Object-Oriented Garbage Collection". *Proc. ECOOP/OOPSLA'90 Workshop on Garbage Collection*, 1990.
Wilson, Paul R. "Some Issues and Strategies in Heap Management and Memory Hierarchies". *ACM Sigplan Not. 26*,3 (March 1991),45-52.
Yuasa, T. "Real-Time Garbage Collection on General-Purpose Machines". *J. Sys. Soft. 11* (1990),181-198.
Zorn, Ben. *Comparative performance evaluation of garbage collection algorithms*. Ph.D. Thesis, UC Berkeley EECS Dept., 1989.

