Stop-and-copy and One-bit Reference Counting*
Technical Report 360

David S. Wise
Indiana University
Bloomington, Indiana 47405 USA
Fax: +1 (812) 855-4829
Email: `dswise@cs.indiana.edu`

(revised) March 1993

Abstract

A stop-and-copy garbage collector updates one-bit reference counting with essentially no extra space and minimal memory cycles beyond the conventional collection algorithm. Any object that is uniquely referenced during a collection becomes a candidate for cheap recovery before the next one, or faster recopying then if it remains uniquely referenced. Since most objects stay uniquely referenced, subsequent collections run faster even if none are recycled between garbage collections. This algorithm extends to generation scavenging, it admits uncounted references from roots, and it corrects conservatively stuck counters, that result from earlier uncertainty whether references were unique.

CR categories and Subject Descriptors:
D.4.2 [**Storage Management**]: Allocation/Deallocation strategies;
E.2 [**Data Storage Representations**]: Linked representations.
General Term: Algorithms.
Additional Key Words and Phrases: multiple reference bit, MRB.

*Research reported herein was sponsored, in part, by the National Science Foundation under Grant Number CCR 90-02797.

1 One-bit Reference Counts

The most commonly used garbage collector is the stop and copy algorithm due to Minsky, Finichel, and Yokelson [15, 8]. Sometimes it is a generation scavenging [22, 26] variant that more frequently recovers shorter-lived structures with less effort. Reference counting [12] is an orthogonal strategy for storage management [11] that again is receiving attention in the context of parallel heap management because its transactions are all local [21, 30, 32].

Reference counting is best used along with a garbage collector as a hybrid manager [28, 20, 13, 29]. Although counting is too weak to handle certain circular structures [20, 16], garbage collection can back it up. On the other hand, successful reference counting can postpone need for garbage collection, or accelerate that collection [3]. Recovery of zero-count objects is worthwhile only when the per-node cost of reference-counting recovery undercuts that for garbage collecting [1]. The cost, however, must include the impact of synchronization in a real-time or multiprocessor environment. One way to improve the performance of both is to use hardware [32] to maintain the counts without any extra cycles.

Clark and Green [10] present experimental results showing that less than 3% of LISP's nodes have reference counts above one. Therefore, it is usual to use a very small field for such counts, with its maximal value designated STICKY, a limit which neither increments nor decrements can change. The only way to recover an object with a STICKY count is via garbage collection; a full collection, traversing all live links, can determine that a STICKY count really should be zero. Moreover, that same traversal can also recompute all counts from zero (unmarked)—perhaps reducing a STICKY count to a lower value without yet recovering that space [29]. In this way, a small-but-stuck counter can be reset, so that reference counting could recover that space before the next collection.

The smallest reference-count field is only a single bit, whose value can only represent "uniquely referenced" or "shared." The terms UNIQUE and STICKY will be associated with these two alternatives, which are indicated as ⓤ and ⓢ in the figures.

The first proposal for one-bit reference counting [31] only addresses the need to save space for the counter; it uses the MARK bit, already in every node, as a reference count between garbage collections. The second formulation [24] is far superior. It has since been rediscovered [9] and developed [18, 23, 19] in logic programming, where it is called a *multiple reference bit* or *MRB.*

Stoye, Clarke, and Norman [24] designated one bit within every pointer (rather than one at every node); thus, a binary node has two such bits. Similar to run-time tags that identify type, it affords information about each pointer: "This reference is unique." The remarkable feature about a tag *in* the pointer is that it anticipates the usual memory reference needed to access a reference count from the address of the object itself. For example, if a count-tag is STICKY then no memory fetch is necessary to determine that all reference-count transactions can be skipped.

UNIQUE referencing is important to *copy avoidance,* a term sometimes used to elide a node's allocation, duplication, and recovery into a single in-place side-effect. An applicative program might specify that a "model" node is copied except for a single field, but if the compiler or the run-time environment ascertains that the model node is uniquely referenced and that the current environment—the source of the reference—is about to be abandoned (as in tail recursion), then it can side-effect that field, instead [6]. In effect, the model node is pre-allocated, and much of its initialization—reflexive copying—is annihilated; the only difference results from the side-effect. So many unshared, intermediate structures (like LISP's *bignums*) can be overwritten in place—saving not only the time to collect them, but also that to reallocate and to rebuild them. Thus, copy avoidance is a special case of storage management and is implicit in the discussion that follows.

An important feature of UNIQUE reference counts appears in a multiprocessing environment. Privileges obtained in critical code at the head of a uniquely referenced chain are still valid after traversing unique links, even via non-critical code. If a process holds privilege to the only path to an object, then the privilege is valid on that object. More than one bit in reference counts is superfluous to such control.

2 Speed

Much of this paper deals with speed of transactions supporting storage management. In all cases the emphasis is on counting memory cycles, rather than processor cycles. Dominant RISC architectures make a large distinction between the two; extra processor cycles are essentially free because they are fast and usually overlapped with memory transactions. On the other hand, random access to memory is expensive because it almost surely misses the cache and requires a full memory cycle, causing a processor-wait in the case of a read. Somewhere between is the cost of serial access because many machines use block transfers between cache and memory; under such a protocol, several adjacent words are transferred after strobing a single address, and neighboring addresses can appear in cache after the required transfer has been completed. The behavior is reminiscent of page migration under virtual memory.

To account for various cache/memory protocols, a memory cycle is here associated with an address strobe. Thus, both a four-byte and an eight-byte transfer counts as a single memory cycle; although such block transfers accelerate serial access, they can actually slow the random-heap access that is our concern here. Furthermore, random reading is arguably more expensive than random writing because the latter can be delayed, perhaps elided, using cache. Nonetheless, we count either as an (eventual) memory cycle.

Under Stoye's scheme, an object is initially allocated as uniquely referenced; its birth address is tagged UNIQUE. His *only* additional memory cycle occurs upon creating a second reference to that object. Whenever a pointer is stored, an extra processor cycle checks the reference count within it. If the tag indicates UNIQUE then a second reference is being created. In that case the tag is reset STICKY, and the tagged pointer is stored *both* in the immediate destination *and* in the original field whence it was fetched. Thus, an extra random-access write is necessary.

The delicate problem about his scheme is determining the latter address; it may still be available in some processor register, because stores to memory closely follow fetches. However, our experience with similar coding problems demonstrates that the reference may be passed through several registers (or the stack) before landing as a counted reference. Such threads through the code can sometimes be unwound and split, but they are, in general, quite tortuous: more difficult to engineer correctly than are garbage collectors [32]. Although low-level systems coding like this is touchy, thank goodness it needs to be coded only rarely even though it is used frequently.

All of the above paraphrases Stoye's contribution. The following is new; it shows how an intervening garbage collection can accurately restore a STICKY count back to UNIQUE. This feature considerably relaxes the rigor on the code, which is now freer to raise a UNIQUE tag to STICKY, defending against uncertain sharing. As appropriate, the collector can later restore it to UNIQUE, compensating somewhat for earlier conservatism.

3 Collection and Count Tags

The previous section reviews how one-bit reference counts can be sustained by the mutator [14] with few memory cycles—one more for every STICKY count as it rises from UNIQUE. This section shows that the collector [14] can recompute all tags accurately for nearly the same cost: one less memory cycle for every UNIQUEly referenced object entering collection, and two extra memory cycles for each STICKY count. Nodes with only one reference require no more cycles than a conventional stop-and-copy collection, and probably less. Counts that are STICKY, which the mutator had rendered unique without being able to tag as UNIQUE, will be redecorated as UNIQUE for free; those objects might be recovered after the present garbage collection but before the next one, thereby postponing it or, if they survive until then, they will make it run faster.

A new requirement, but not a disruptive one, is that every addressable object must be large enough to contain two pointers. That is, LISP's *cons* box is the smallest heap object; the rare "boxed address" must be represented as a cons box, tolerating 50% internal fragmentation. This constraint is necessary to provide room for two "forwarding" addresses per node, instead of just one. The forwarding pointers will be called FORWARD and SOURCE.

Euphemistically, the magnitude of FORWARD is treated as a MARK bit. If it points into the range of TO space, we say MARK is "set"; if it has any other value—including a pointer into FROM space—then it is "clear." This convention is consistent with the role of marking in all garbage collectors.

The stop-and-copy collector [8, 2] is a "two finger" algorithm, whose effect is to move all nodes out of the FROM semispace, compacting them into the TO semispace. Initially, all "root" nodes are copied into one end of the TO space, without changing any content; rooting is discussed in detail in the next section. Thereafter, one "finger," called NEXT, always points to the first unused address in TO space. The other, called SCAN, starts at the beginning of TO space and updates every pointer in sequence until it catches up with NEXT. For a complete description and figure, see Appel [2]. NEXT and SCAN are denoted Ⓝ and Ⓢ in the figures; this description focuses on SCAN.

Figure 1 illustrates the effects of scanning the first and then later STICKY references to a binary node at address A. Blank entries there indicate "don't care." Figure 1.1 is the situation before scanning the first reference, Figure 1.2 shows UNIQUE tags between the two, and Figure 1.3 shows updating all

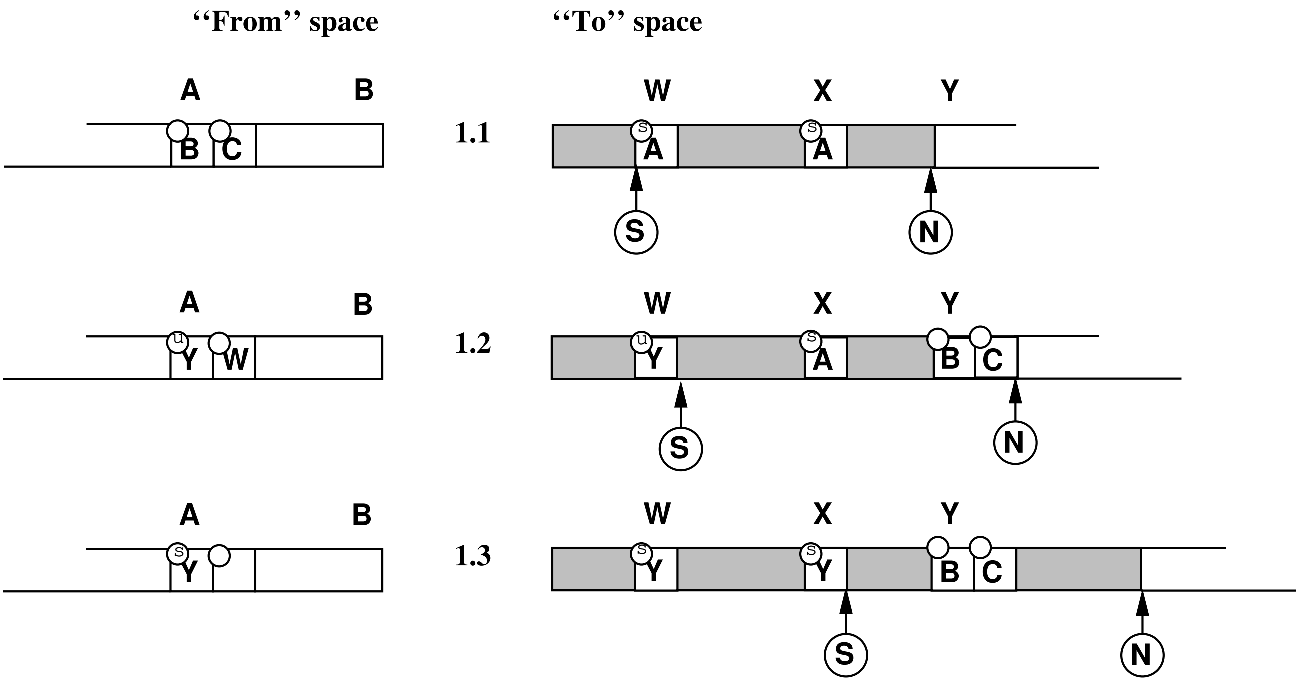

Figure 1: Scanning two references to ConsNode A.

references to STICKY counts. A description of changes to the stop-and-copy algorithm follows.

When SCAN encounters a UNIQUE pointer to FROM space, it copies the content at NEXT, but does *not* install any forwarding information. (Since the object is uniquely referenced, there aren't any other references to forward.) Thus, every UNIQUE reference (even if just installed during a previous collection) accelerates conventional collection [3].

SCAN also encounters STICKY pointers to locations in FROM space, some of whose contents have been copied—but some not yet copied—into TO space. The difference is determined by MARK on that node. When SCAN finds a pointer into FROM space, it performs a long-read (*e.g.* 64-bit read) of FORWARD (MARK) and SOURCE at that address. If MARK is clear, then the contents of that node—perhaps a lengthy vector—are transferred to NEXT, and NEXT is incremented by its length. Then, in a single long-write, that node in FROM space is overwritten with a FORWARD pointer (tagged UNIQUE) to the NEXT location whither the node was copied (a set MARK bit) and the SOURCE address—SCAN—where this initially unique reference was found. Finally, the address of the node, now relocated in TO space and still tagged UNIQUE, is stored back at SCAN. None of this yet introduces a new memory cycle into stop-and-copy collection.

If, after the long-read from the contents of SCAN, MARK is found set but the FORWARDing address is tagged STICKY, then FORWARD is overwritten at SCAN, forwarding the reference as always. While this test introduces a processor cycle to test the tag, still no new memory cycles are needed.

If, however, MARK is set but FORWARD is tagged UNIQUE, then a second reference to that node has been discovered. In this case, the tag is changed to STICKY, the FORWARDing address itself, is overwritten (to change its tag), overwritten at SCAN (to forward the second reference), and overwritten at the SOURCE of the hitherto unique reference. The first and last of these three writes are the new cost of this algorithm. However, they only occur once for each STICKY pointer.

When implemented over virtual memory, the only TO-space pages that need be memory-resident are indicated by Ⓢ and Ⓝ. Then SOURCE addresses should not be overwritten immediately. In order to avoid extra page faults they should instead be accumulated on temporary pages, sorted there after SCANning completes, and then overwritten to STICKY in sequential order.

The above description is readily adapted to generation scavenging. Any reference to an older generation, however, should be forced STICKY unless both generations are being collected.

In summary, the following invariant governs this version of stop-and-copy. If no references to a node in FROM space have been SCANned, then the node retains its pre-collection content and remains unMARKed. If exactly one reference has been SCANned, then it is MARKed, its contents have migrated into TO space, its already SCANned reference—tagged as UNIQUE—contains that address in TO space, as does its FORWARD field (unless that reference was tagged UNIQUE when collection began.) Its SOURCE field identifies the location of that unique, already forwarded reference from TO space. If two or more references have been SCANned, then it is MARKed, its contents have migrated, and its FORWARD field and all SCANned references are tagged STICKY.

4 Rooting

Now we return to the beginning of garbage collection: how to "root" the stop-and-copy collector. *Roots* are those references from the running system that are known to be live. Thus, those roots are the first to be copied from FROM space to TO space. Some ephemeral roots may not be counted at all in the reference counting scheme. Of those, some may never be in use where garbage collection is possible. This section shows how to handle those that are in use across collection.

Quite appropriately certain shared references may go uncounted. This relaxation allows, for instance, a unique reference to be relocated "atomically" without its count rising from UNIQUE to STICKY. A similar idea is used in some usual patterns of coding [6], in simple list manipulations [25], in tail-recursion protocols [7, 1, 4], and in linear languages [17, 27, 5]. However, run-time management like this can be more comprehensive than any of these.

Often uncounted references reside on a sequentially-allocated recursion stack, because it is so fast to pop it without first decrementing references counted therefrom.[1] Such stack-resident references may, in fact, be the best roots for collection. The following protocol shows how to initiate a collection under Stoye's scheme without counting the root's reference. See Figure 2.

Uncounted references are "scanned" differently from heap-resident ref-

[1] In the hardware reference-counter [32], popping is just as fast because a decrement is postponed until the stack cell is overwritten [28] later—contemporary with an increment—during a subsequent push.

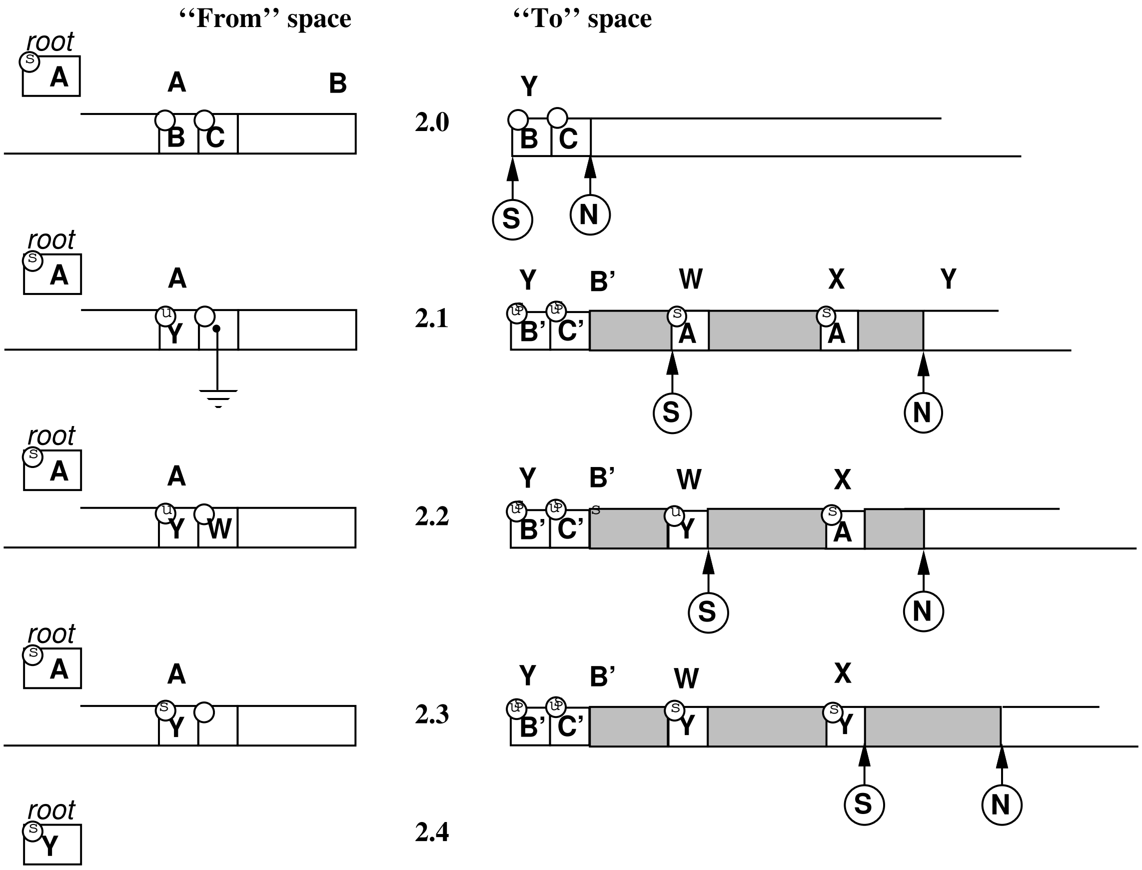

Figure 2: Initiating collection from an uncounted root.

erences and, therefore, they can occur only as roots. Each uncounted reference causes the usual long-read from FROM space; if the node is already MARKed, nothing further happens. Otherwise, the node is copied at SCAN into TO space and the following forwarding information is overwritten in FROM space: a UNIQUE-tagged FORWARDing address to SCAN (thus, a set MARK bit), and a NIL SOURCE field. (If the root is to be counted, then SOURCE is directed to any dummy address that can be clobbered.)

Now the scanner needs to be modified to deal with NIL sources. The change occurs in the previous section only in the case where MARK is set but FORWARD is tagged UNIQUE. In that case, SOURCE must first be tested for NIL; if it is non-NIL then the previous algorithm proceeds, slowed only by the processor cycle for the test. Otherwise, the value of SCAN is written as the SOURCE field in place of NIL, establishing the first counted reference.

After the copying scan completes, every root must be visited a second time to update it from the forwarding addresses (and count tag) left in FROM space. It is presumed that there is a small number of roots, lest this second traversal become extravagant; it can coincide with reloading registers prior to resuming the mutator. If desired, it is easy to verify then that each node referenced from an uncounted root is also referenced by a counted pointer: that all NIL SOURCEs have been overwritten.

Figure 2 shows what happens when A in Figure 1 is rooted by an uncounted reference. Figure 2.0 shows the result of rooting but before scanning, and Figure 2.1 illustrates scanning of the uncounted root. Figures 2.2 and 2.3 are analogous to 1.2 and 1.3, and Figure 2.4 indicates the second traversal of the root, recovering the final forwarded address. The tags on B' and C' here are unimportant: either ⓤ or ⓢ.

The remainder of the previous section is unchanged. The additional cost to the garbage collector is one memory cycle for each uncounted root (to overwrite NIL), plus one for each root (to perform the second root traversal after scanning.) However, this section is not necessary on roots that are counted and reside in addressable memory; in that case, SOURCE fields can be directed to roots as in Figure 1.

5 Speed, again

Estimating the relative timings for collection is a useful exercise here. Let us sketch an analysis on a heap of M binary nodes (*e.g.* 8-bytes big: either one 64-bit atom, or two 4-byte pointers or immediate values.) Let u be the proportion ($0 \leq u < 1$) of the heap that is actually in use. Considering immediate values (like NIL) and both halves of an atom as "dead pointers", let p be the independent proportion of pointers that are actually "alive," pointing to other nodes in the heap. With very few roots compared to nodes, $1/2 \leq p \leq 1$; if the few structures are binary trees then $p \approx 1/2$.

At issue here are the singly referenced nodes. Let q be the proportion of live nodes/atoms that are uniquely referenced. Of those, let r be the proportion that is actually tagged UNIQUE at the beginning of garbage collection. Consider the memory cycles (read or write) necessary to collect a semispace, recopying uM nodes to recover $(1-u)M$.

In its naive, two-semispace version [8] a collection requires three memory cycles to copy and forward (read, write, write) each of uM binary nodes to TO space. Scanning requires a read cycle, an expected $2p$ read cycles to get the forwarding addresses of live pointers, plus an expected $1-(1-p)^2 = 2p-p^2$ write cycle to update the node. The total cost-per-node for recovery is thus,

$$(4+4p-p^2)\frac{uM}{(1-u)M}.$$

Post-collection allocation, serial from contiguous space, requires no memory cycles.

However, if the algorithm described here is used, then uniqueness of reference nodes must be considered. Copy-and-forward now expects only $3-qr$ cycles, because forwarding of UNIQUEly referenced nodes is unnecessary. However, scanning requires an additional $2(1-q)$ cycles to store STICKY tags once over the FORWARDing address and at the SOURCE of the first reference; a live node has a second reference with probability $1-q$. The total cost-per-node for recovery is thus,

$$[4+4p-p^2-(q(r+2)-2)]\frac{u}{1-u}.$$

(Section 4 rooting is ignored.)

The modifications proposed here will actually accelerate garbage collection whenever $q(r+2) > 2$, which is likely. Define $\epsilon = q(r+2)-2$ and observe that $\epsilon > 0$ if $r > 6\%$ and $q > 97\%$, as Clark and Green [10] observed, or even when $r > 22\%$ and $q > 90\%$. Appel [1] observed $u = 1/3$ without generation scavenging. If we then approximate $4p - p^2$ as 2 ($p \approx 59\%$) and $\epsilon = 0$ conservatively, then memory-cycles-per-collected-node becomes 3. Under generation scavenging, q and r approach one, so does ϵ, and the result approaches 2.5; but u also becomes very small, further reducing this approximation.

Henry Baker suggests a further analysis to consider the impact of caching or, more archaically, paging. If there are c nodes in a cache line, then sequential traversal of the TO semispace causes a memory cycle for only $1/c$ of every read/write of a node at NEXT or SCAN. Cache hits may elide more memory access, but the effect of random hits is ignored here. The naive, two-semispace collection requires $2+1/c$ cycles to copy and forward, plus at most $1/c$ to read, $2p$ to fetch forwarding addresses, and $1/c$ cycles to write[2] the node back at SCAN. The total cost-per-node falls to

$$[2+2p+3/c]\frac{u}{1-u}.$$

Similarly, The cost-per-node for the algorithm described here falls to

$$[2+2p+3/c-\epsilon]\frac{u}{1-u}.$$

Caching accelerates memory cycles other than those counted in ϵ, because those deal with only non-sequential access. If $c=4$, again $p \approx 59\%$, $\epsilon = 0$, and $u = 1/3$ the cost already drops to 1.96, from 3.

The effect of caching on these algorithms becomes apparent if we compare the ratios of their costs: of each algorithm with caching to that without caching. Using the sample arguments, such a ratio for the original stop-and-copy algorithm is $(6-\epsilon)/6$, while that for the new algorithm is $(3.92-\epsilon)/3.92$. Thus, caching amplifies the relative improvement, ϵ, from the new collection algorithm. The relative improvement in the ϵ-term of this ratio is 2.08/3.92 here, already above one-half; this improvement more generally is

$$\frac{4+4p-p^2}{2+2p+3/c}-1=\frac{2+2p-(3/c+p^2)}{2+2p+(3/c)}.$$

So this algorithm yields even more speed with larger caches.

6 What to do with free nodes?

All the discussion so far has focused on maintaining accurate one-bit reference counts, and particularly by restoring them during stop-and-copy collection. With cheap and valuable information located so conveniently within

[2]More accurately, the likelihood of a write cycle is only $c\alpha = 1-(1-p)^{2c}$ per cache line. However, $p \geq 2^{-1}$ and so $(1-2^{-2c})/c \leq \alpha \leq 1/c$, making this per-node upper bound quite close.

pointers, what can software afford to do with them? Not much. Having so identified unique references at the point they are destroyed, we discover that the popular solution, an available-space list, is remarkably expensive.

An inexpensive remedy is to dedicate a register or a cache-resident vector to holding a few available addresses. A poor-man's available-space list, this can be effected with no memory cycles. When a node is allocated from such a register its stale contents should be inspected for additional unique references [28]. Depending on hardware, this requires another memory cycle (for the binary node illustrated), or if read-modify-write is available it can be effected by the existing write that initially fills the new node. The idea here is to enable reuse of released nodes on-the-spot.

The impact of Stoye's algorithm is minor, less than two cycles-per-node-recovered. However, the cost for his run-time reallocation becomes significant when an available-space list is used.

Building and decomposing an available-space list costs three or more memory cycles per node, unfavorable compared to the estimates above. It costs $n+1$ memory cycles to return a UNIQUEly referenced chain of n nodes to available space (n reads and a write), but often n is only one. Moreover, a new node must be read again before it is written (in absence of read-modify-write) [28], so the cost is above three memory-cycles per node-recycled, before that node can be filled.

Finally, variable sizing of nodes creates problems for any run-time recycling. One can use different registers or different available-space lists for different ranges of size, but maintaining several available-space lists [20, §2.5] requires yet more memory cycles.

7 Conclusion

Experiments with real systems are necessary to determine the success of these algorithms in different applications. For instance, a parallel, real-time program that uses constant-sized objects should find this technique quite useful.

Practical comparisons with generation scavenging and linear logic are both open questions. It is entirely possible that the bulk of the uniquely referenced nodes, that Clark and Green observed and upon which generation scavenging preys, correspond to the same nodes that Stoye's algorithm can recover,[3] especially if assisted by the garbage-collector support described

[3] "The results of applying this technique have been spectacular—on average, about

here. It conjectured that Stoye's one-bit counter is more powerful than compile-time linear-logic [27], but both can be hobbled by obfuscated code. The former may prove more appropriate in general packages (interpreted code) and the latter in general-purpose programming (compiled code).

Appel [1, p. 206] claims fractional cycle-per-node efficiency for generation-scavenging garbage-collection recovery. His arguments are valid with a heap much larger than active data structure, and when generation scavenging is effective; his application, a strict (eager) language on a large, dedicated uniprocessor, meets these criteria. Moreover, his use of continuation-passing style, which generates tons of garbage, is successful, in part, because his collector thrives on that sort of trash. That is, he enjoys a brilliant symbiosis between his mutator and his collector. Much of his garbage would never be generated under another style, that neither could attain his frequency or efficacy of collection, nor would suffer their absence. Other contexts (*e.g.* caches) have different constraints on, and different requirements of, heap management.

Some applications are space bound—if only by address space—and locality has again become desirable under cached memory, just as it once was under virtual memory. If unique references dominate (as they seem to), then this strategy may be even more efficient on a confined heap than generation scavenging is. When garbage collection is more frequent or less tolerable, as in real-time or parallel applications, then reference counting is the better strategy—and decidedly better with the counting off-processor in memory hardware [30, 32].

The idea here is to modify each garbage collection in a cost-efficient effort to postpone and to accelerate the next one. This modification to a stop-and-copy collector makes it run faster even if no nodes are recycled between collections through reference counting. Once again, the best storage manager is a symbiosis between reference counting and garbage collection.

Acknowledgement: Carl Bruggeman pointed out that the collector's later reinstallation of UNIQUE tags ameliorates a mutator's earlier dilemma whether some references remain unique. Henry Baker observed that a UNIQUEly referenced node needs no forwarding pointer, accelerating collection by minimizing "dirty" cache lines and pages.

---

seventy percent of wasted cells are immediately reclaimed [24]." "We have found 40 to 90 percent of garbage data is incrementally reclaimed by the MRB [Stoye's] scheme in benchmark [committed-choice logic] programs [18]."

References

[1] A. Appel. *Compiling with Continuations*, Cambridge Univ. Press (1992), 205-206.

[2] A. Appel. Garbage collection. In P. Lee (ed.), *Topics in Advanced Language Implementation*, Cambridge MA, M.I.T. Press (1991) 89-100.

[3] H.G. Baker. Cache-conscious copying collectors. OOPSLA '91 GC Workshop. (October 1991). ©Nimble Computer Corp, 16231 Meadow Ridge Way, Encino, CA 91436, USA.

[4] H.G. Baker. CONS should not CONS its arguments. *SIGPLAN Notices* **27**, 3 (March 1992), 24–34.

[5] H.G. Baker. Lively linear lisp—'Look Ma, no garbage!' *SIGPLAN Notices* **27**, 8 (August 1992), 89–98.

[6] J. Barth. Shifting garbage collection overhead to compile time. *Comm. ACM* **20**, 7 (July 1977), 513–518.

[7] A. Bloss & P. Hudak. Variations on strictness analysis. *Conf. Rec. 1986 ACM Symp. on Lisp and Functional Programming*, 132–142.

[8] C.J. Cheney. A nonrecursive list compacting algorithm. *Comm. ACM* **13**, 11 (November 1970), 677–678.

[9] T. Chikayama & Y. Kimura. Multiple reference management in flat GHC. In J.–L. Lassez (ed.), *Logic Programming, Proc. 4th Intl. Conf.* **1**. Cambridge, MA, M.I.T. Press (1987), 276–293.

[10] D.W. Clark & C.C. Green. A note on shared list structure in LISP. *Inf. Proc. Lett.* **7**, 6 (October 1978), 312–315.

[11] J. Cohen. Garbage collection of linked data structures. *Comput Surveys* **13**, 3 (September 1981), 341–367.

[12] G.E. Collins A method for overlapping and erasure of lists. *Comm. ACM* **3**, 12 (December 1960), 655–657.

[13] L.P. Deutsch & D.G. Bobrow. An efficient, incremental, automatic garbage collector. *Comm. ACM* **19**, 9 (September 1976), 522–526.

[14] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Sholten, & E.F.M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Comm. ACM* **21**, 11 (November 1976), 966–975.

[15] R.R. Fenichel & J.C. Yochelson. A LISP garbage collector for virtual-memory computer systems. *Comm. ACM* **12**, 11 (November 1969), 611–612.

[16] D.P. Friedman & D.S. Wise. Reference counting can manage the circular environments of mutual recursion. *Inf. Proc. Lett.* **8**, 1 (January 1979), 41–45.

[17] J.–Y. Girard. Linear logic. *Theoret. Comput. Sci.* **50** (1987), 1–102.

[18] Y. Inamura, N. Ichiyoshi, K. Rokusawa, & K. Nakajima. Optimization techniques using the MRB and their evaluation on the Multi–PSI/V2. In E.L. Lusk & R.A. Overbeek, *Logic Programming, Proc. of North American Conf. 1989* **2** Cambridge, MA, M.I.T. Press (1989), 907–921.

[19] Y. Kimura, T. Chikayama, T. Shigoni, & A. Goto. Incremental garbage collection scheme in KL1 and its architectural support of PIM. In J.G. Delgado-Frias & W.R. Moore (eds.), *VLSI for Artificial Intelligence and Neural Networks*, New York, Plenum (1991), 33–45.

[20] D.E. Knuth *The Art of Computer Programming* **I,** *Fundamental Algorithms* (2nd ed.), Reading MA, Addison-Wesley (1975).

[21] B. Lang, C. Queinnec, & J. Piquer. Garbage collecting the world. *Conf. Rec. 19th ACM Symp. on Principles of Programming Languages* (1992), 39–50.

[22] H. Lieberman & C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Comm. ACM* **26**, 6 (June 1983), 419–429.

[23] K. Nishida, Y. Kimura, A. Matsumoto, & A. Goto. Evaluation of MRB garbage collection on parallel logic programming architectures. In D.H.D. Warren & P. Szeredi (eds.), *Logic Programming, Proc. 7th Intl. Conf.* , Cambridge, MA, M.I.T. Press (1990),

[24] W.R. Stoye, T.J.W. Clarke, & A.C. Norman. Some practical methods for rapid combinator reduction. *Conf. Rec. 1984 ACM Symp. on Lisp and Functional Programming*, 159–166.

[25] N. Suzuki. Analysis of pointer 'rotation.' *Comm. ACM* **25**, 5 (May 1982), 330–335.

[26] D. Ungar. Generation scavenging: a non-disruptive high-performance storage-reclamation algorithm. *Proc. ACM SIGPLAN/SIGSOFT Software Engineering Symp. on Practical Software Development Environments, SIGPLAN Notices* **19**, 5 & *Software Engineering Notes* **9**, 3 (May 1984), 157–167.

[27] P. Wadler. Is there a use for linear logic? *Proc. Symp. on Partial Evaluation and Semantics-Based Program Manipulation, SIGPLAN Notices* **26**, 9 (September 1991), 255-273.

[28] J. Weizenbaum. Symmetric list processor. *Comm. ACM* **6**, 9 (September 1963), 524–554.

[29] D.S. Wise. Morris's garbage compaction algorithm restores reference counts. *ACM Trans. Progr. Lang. Sys.* **1**, 1 (July 1979), 115–120.

[30] D.S. Wise. Design for a multiprocessing heap with on-board reference-counting. in J.–P. Jouannaud (ed.), *Functional Programming and Computer Architecture, LNCS* **201**, Berlin, Springer (1985), 289–304.

[31] D.S. Wise & D.P. Friedman. The one-bit reference count. *BIT* **17**, 3 (September 1977), 351–359.

[32] D.S. Wise, C. Hess, W. Hunt, & E. Ost. Uniprocessor performance of reference-counting hardware heap. Computer Science Dept., Indiana Univ., Tech. Rept. (1992).