Communications of the ACM **21**, 4 (April 1978), 280-294.

List Processing in Real Time on a Serial Computer

Henry G. Baker, Jr.
Massachusetts Institute of Technology

A real-time list processing system is one in which the time required by the elementary list operations (e.g. CONS, CAR, COR, RPLACA, RPLACD, EQ, and ATOM in LISP) is bounded by a (small) constant. Classical implementations of list processing systems lack this property because allocating a list cell from the heap may cause a garbage collection, which process requires time proportional to the heap size to finish. A real-time list processing system is presented which continuously reclaims garbage, including directed cycles, while linearizing and compacting the accessible cells into contiguous locations to avoid fragmenting the free storage pool. The program is small and requires no time-sharing interrupts, making it suitable for microcode. Finally, the system requires the same average time, and not more than twice the space, of a classical implementation, and those space requirements can be reduced to approximately classical proportions by compact list representation. Arrays of different sizes, a program stack, and hash linking are simple extensions to our system, and reference counting is found to be inferior for many applications.

Key Words and Phrases: real-time, compacting, garbage collection, list processing, virtual memory, file or database management, storage management, storage allocation, LISP, CDR-coding, reference counting.

CR Categories: 3.50, 3.60, 3.73, 3.80, 4.13, 4.22, 4.32, 4.33, 4.35, 4.49

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0522.
Author's address: Computer Science Department, University of Rochester, Rochester, NY 14627.

1. Introduction and Previous Work

List processing systems such as LISP [25] have slowly gained popularity over the years in spite of some rather severe handicaps. First, they usually interpreted their programs instead of compiling them, thus increasing their running time by several orders of magnitude. Second, the storage structures used in such systems were inefficient in the use of storage; for example, compiling a program sometimes halved the amount of storage it occupied. Third, processing had to be halted periodically to reclaim storage by a long process known as garbage collection, which laboriously traced and marked every accessible cell so that those inaccessible cells could be recycled.

That such inefficiencies were tolerated for so long is a tribute to the elegance and productivity gained by programming in these languages. These languages freed the programmer from a primary concern: *storage management*. The programmer had only to call CONS (or its equivalent) to obtain a pointer to a fresh storage block; even better, the programmer had only to relinquish all copies of the pointer and the storage block would automatically be reclaimed by the tireless garbage collector. The programmer no longer had to worry about prematurely freeing a block of storage which was still in use by another part of the system.

The first problem was solved with the advent of good compilers [27,32] and new languages such as SIMULA especially designed for efficient compilation [1,5,14]. The second was also solved to some extent by those same compilers because the user programs could be removed from the list storage area and freed from its inefficient constraints on representation.[1] Other techniques such as compact list representation ("CDR-coding") [12,19] have been proposed which also offer partial solutions to this problem.

This paper presents a solution to the third problem of classical list processing techniques and removes that roadblock to their more general use. Using the method given here, a computer could have list processing primitives built in as machine instructions and the programmer would still be assured that each instruction would finish in a reasonable amount of time. For example, the interrupt handler for a keyboard could store its characters on the same kinds of lists—and in the same storage area—as the lists of the main program. Since there would be no long wait for a garbage collection, response time could be guaranteed to be small. Even an operating system could use these primitives to manipulate its burgeoning databases. Business database designers no longer need shy away from pointer-based systems for fear that their systems will be impacted by a week-long garbage collection! As memory is becoming cheaper,[2] even microcomputers could be built having these primitives, so that the prospect of controlling one's kitchen stove with LISP is not so far-fetched.

A *real-time* list processing system has the property that the time required by each of the elementary operations is bounded by a constant independent of the number of cells in use. This property does not guarantee that the constant will be small enough for a particular application on a particular computer, and hence has been called "pseudo-real-time" by some. However, since we are presenting the system

[1] In many cases, a rarely used program is compiled not to save time in its execution, but to save garbage-collected storage space.

[2] Work is progressing on 10^6 bit chips.

independent of a particular computer and application, it is the most that can be said. In all but the most demanding applications, the proper choice of hardware can reduce the constants to acceptable values.

Except where explicitly stated, we will assume the classical Von Neumann serial computer architecture with real memory in this paper. This model consists of a memory, i.e. a one-dimensional array of words, each of which is large enough to hold (at least) the representation of a nonnegative integer which is an index into that array; and a central processing unit, or CPU, which has a small fixed number of registers the size of a word. The CPU can perform most operations on a word in a fixed, bounded amount of time. The only operations we require are load, store, add, subtract, test if zero, and perhaps some bit-testing. It is hard to find a computer today without these operations.

As simple as these requirements are, they exclude virtual memory computers. These machines are interesting because they take advantage of the locality of reference effect, i.e. the nonzero serial correlation of accesses to memory, to reduce the amount of fast memory in a system without greatly increasing the average access time. However, the time required to load a particular word from virtual memory into a CPU register is not bounded because the primary memory may have to fetch it from a lower level memory. Since we are more interested in tight upper bounds, rather than average performance, this class of machines is excluded.

Since the primary list processing language in use today is LISP, and since most of the literature uses the LISP paradigm when discussing these problems, we will continue this tradition and center our discussion around it. Due to its small cells, which consist of 2 pointers apiece, LISP is also a kind of worst case for garbage collection overhead.

There are two fundamental kinds of data in LISP: *list cells* and *atoms*. List cells are ordered pairs consisting of a *car* and a *cdr*, while atoms are indecomposable. ATOM(x) is a predicate which is true if and only if x is an atom (i.e. if and only if x is not a list cell); EQ(x,y) is a predicate which is true if and only if x and y are the same object; CAR(x) and CDR(x) return the car and cdr components of the list cell x, respectively; CONS(x,y) returns a new list cell (not EQ to any other accessible list cell) whose car is initially x and whose cdr is initially y; RPLACA(x,y) and RPLACD(x,y) store y into the car and cdr of x, respectively. We assume here that these seven primitives are the only ones which can access or change the representation of a list cell.

There have been several attempts to tackle the problem of real time list processing. Knuth [22,p.422] credits Minsky as the first to consider the problem, and sketches a multiprogramming solution in which the garbage collector shares time with the main list processing program. Steele's paper [30] was the first in a flurry of papers about *multiprocessing* garbage collection which included contributions by [16,17] and [23,24]. [28] independently detailed the Minsky-Knuth-Steele method, and both [28] and [33] analyzed the time and storage required to make it work.

The Minsky-Knuth-Steele-Muller-Wadler (MKSMW) method for real-time garbage collection has two processes running in parallel. The list processor process is called the *mutator* while the garbage collector is called the *collector* (these terms are due to [16]). The mutator executes the user's program while the collector performs garbage collection over and over again. The collector has three phases: *mark*, *sweep*, and *relocate*. During the mark phase, all accessible storage is marked as such, and any inaccessible storage is picked up during the sweep phase. The relocate phase relocates accessible cells in such a way as to minimize the address space required. Since the mutator continues running while the mark and relocate phases proceed, the free list must be long enough to keep the mutator from starvation. During the sweep phase, cells must be added to the free list *faster* than they can be taken off, on the average, lest the net gain in cells from that garbage collection cycle be negative.

The behavior of this algorithm under equilibrium conditions (which is when a cell is let go for every cell CONSED, and when the rates of cell use by the mutator, and of marking, sweeping, and relocating by the collector, are all constant) was studied in [28] and [33]. If we let m be the ratio of the rate of CONSING to that of marking, s be the ratio of the rate of CONSING to that of sweeping, and r be the ratio of the rate of CONSING to that of relocating, then we can derive estimates of the size of storage needed to support an accessible population of N cells under equilibrium conditions.[3] Using these assumptions, we derive:

$$\text{Maximum MKSMW Storage Required} \le N\frac{m+(m+1)(r+1)}{1-s(r+1)} + \text{size of collector stack}$$

We note that r=0 if there is no relocation (i.e. it happens instantaneously), and we have the simpler expression:

$$\text{Maximum MKSMW Storage Required} \le N\frac{1+2m}{1-s} + \text{size of collector stack}$$

The collector stack seems to require depth N to handle the worst case lists that can arise, but each position on the stack need only hold one pointer. Since a LISP cell is two pointers, the collector stack space requirement is $0.5N$. Thus, we arrive at the inequality:

$$\text{Maximum MKSMW Storage Required} \le N\frac{1.5+2m-.5s}{1-s}$$

These estimates become bounds for nonequilibrium situations so long as the ratios of the rate of CONSING to the rates of marking, sweeping, and relocating are constant; i.e. we relativize the rates of marking, sweeping, and relocating with respect to a CONS counter rather than a clock.

The Dijkstra-Lamport (DL) method [16,17,23,24] also has the mutator and collector running in parallel, but the collector uses no stack. It marks by scanning all of storage for a mark bit it can propagate to the marked cell's offspring. This simple method of garbage collection was considered because their concern was *proving* that the collector actually collected only and all garbage. Due to its inefficiency, we will not consider the storage requirements of this method.

Both the MKSMW and the DL methods have the drawback that they are parallel algorithms and as a result are incredibly hard to analyze and prove correct. By contrast, the method we present is serial, making analyses and proofs easy.

2. The Method

Our method is based on the Minsky garbage collection algorithm [26], used by Fenichel and Yochelson in an early Multics LISP [18], elegantly refined in [11], and applied by Arnborg to SIMULA [1]. This method divides the list space into two *semispaces*. During the execution of the user program, all list cells are located in one of the semispaces. When garbage collection is invoked, all accessible cells are traced, and instead of simply being marked, they are moved to

[3] Of course s<1, or else the storage required is infinite.

the other semispace. A *forwarding address* is left at the old location, and whenever an edge is traced which points to a cell containing a forwarding address, the edge is updated to reflect the move. The end of tracing occurs when all accessible cells have been moved into the "to" semispace (*tospace*) and all edges have been updated. Since the tospace now contains all accessible cells and the "from" semispace (*fromspace*) contains only garbage, the collection is done and the computation can proceed with CONS now allocating cells in the former fromspace.

This method is simple and elegant because 1) it requires only one pass instead of three to both collect and compact, and 2) it requires no collector stack. The stack is avoided through the use of two pointers, *B* and *S*. *B* points to the first free word (the *bottom*) of the free area, which is always in the tospace. *B* is incremented by COPY, which transfers old cells from the fromspace to the bottom of the free area, and by CONS, which allocates new cells. *S scans* the cells in tospace which have been moved, and updates them by moving the cells they point to. *S* is initialized to point to the beginning of tospace at every flip of the semispaces and is incremented when the cell it points to has been updated. At all times, then, the cells between *S* and *B* have been moved, but their cars and cdrs have not been updated. Thus, when *S*=*B* all accessible cells have been moved into tospace and their outgoing pointers have been updated. This method of pointer updating is equivalent to using a queue instead of a stack for marking, and therefore traces a spanning tree of the accessible cells in breadth-first order.

Besides solving the compaction problem for classical LISP, the Minsky-Fenichel-Yochelson-Cheney-Arnborg (MFYCA) method allows simple extensions to handle nonuniformly sized arrays and CDR-coding because free storage is kept in one large block. Allocation is therefore trivial; one simply adds *n* to the "free space pointer" to allocate a block of size *n*.

Copying garbage collectors have been dismissed by many as requiring too much storage for practical use (because they appear to use twice as much as classical LISP), but we shall see that this judgement was, perhaps, premature.

We present the MFYCA algorithm here in pseudo-Algol-BCPL notation. The notation "$\alpha[\beta]$" means the contents of the word whose address is the value of α plus the value of β, i.e. the contents of $\alpha + \beta$. If it appears on the left hand side of ":=", those contents are to be changed. Thus, *p*[*i*] refers to the *i*th component of the vector pointed to by *p*. The function s*ize(p)* returns the size of the array pointed to by *p*. The notation "α & β" is similar to the notation "α; β" in that α and β are executed in order; however, "α & β" returns the value of α rather than the value of β. Thus, ";" and "&" are the duals of one another: "α_1; α_2; …; α_n" returns the last value (that of α_n) whereas "α_1& α_2& …& α_n" returns the first value (that of α_1).

Our conventions are these: the user program has a bank of *NR* registers *R*[1], …, *R*[*NR*]. *The user program may not "squirrel away" pointers outside of the bank R during a call to CONS* because such pointers would become obsolete if garbage collection were to occur. (We will show later how to deal with a user program stack in such a way that the real-time properties of our system are not violated.) Pointers either are *atoms* or refer to *cons cells* in fromspace or tospace. A cons cell *c* is represented by a 2-vector of pointers: *car*(*c*)=*c*[0], *cdr*(*c*)=*c*[1]. *FLIP*, *FROMSPACE* and *TOSPACE* are implementation-dependent routines. FLIP interchanges the roles of fromspace and tospace by causing CONS and COPY to allocate in the other semispace and the predicates FROMSPACE and TOSPACE to exchange roles. FLIP also has the responsibility of determining when the new tospace is too small to hold everything from the fromspace plus the newly consed cells. Before flipping, it checks if *size*(fromspace) is less than (1+*m*)[*size*(tospace)-(*T*-*B*)], where *T* is the top of tospace, and if fromspace (the new tospace) is too small, either it must be extended, or the system may later stop with a "memory overflow" indication.

```
% The Minsky-Fenichel-Yochelson-Cheney-Arnborg [26,18,11,1] Garbage Collector.
pointer B;                              % Bottom; points to bottom of free area.
pointer S;                              % Scan; points to first untraced cell.
pointer T;                              % Top; points to top of tospace.
                                        % Assertions: S≤B≤T and T-B is even.

pointer procedure CONS(x,y) ≡           % Allocate the list cell (x.y)
  begin
    if B = T                            % If there is no more free space,
      then                              %   collect all the garbage.
        begin                           % This block is the "garbage collector".
          flip();                       % Interchange semispaces.
          for i=1 to NR                 % Update all user registers.
            do R[i]:=move(R[i]);
          x:=move(x); y:=move(y);       % Update our arguments.
          while S<B                     % Trace all accessible cells.
            do begin
                 S[0]:=move(S[0]);      % Update the car and cdr.
                 S[1]:=move(S[1]);
                 S:=S+2                 % Point to next untraced cell.
               end
        end;
    if B≥T then error;                  % Memory is full.
    B[0]:=x; B[1]:=y;                   % Create new cell at bottom of free area.
    B & (B:=B+2)                        % Return the current value of B
  end;                                  %   after stepping it to next cell.
pointer procedure CAR(x) ≡ x[0];        % A cell consists of 2 words:
```

```
pointer procedure CDR(x) ≡ x[1];          %   car is 1st; cdr is 2nd.
procedure RPLACA(x,y) ≡ x[0]:=y;          % car(x) := y
procedure RPLACD(x,y) ≡ x[1]:=y;          % cdr(x) := y
boolean procedure EQ(x,y) ≡ x=y;          % Are x,y the same object?
boolean procedure ATOM(x) ≡               % Is x an atom?
  not tospace(x);
pointer procedure MOVE(p) ≡               % Move p if not yet moved; return new address.
  if not fromspace(p)                     % We only need to move old ones.
    then p                                % This happens a lot.
    else begin
           if not tospace(p[0])           % We must move p.
             then p[0]:=copy(p);          % Copy it into the bottom of free area.
           p[0]                           % Leave and return forwarding address.
         end;
pointer procedure COPY(p) ≡               % Create a copy of a cell.
  begin                                   % Allocate space at bottom of free area.
    if B≥T then error;                    % Memory full?
    B[0]:=p[0]; B[1]:=p[1];               % Each cell requires 2 words.
    B & (B:=B+2)                          % Return the current value of B
  end;                                    %   after moving it to next cell.
% TOSPACE, FROMSPACE test whether a pointer is in that semispace.
```

In order to convert MFYCA into a real-time algorithm, we force the mark ratio m to be constant by changing CONS so that it does k iterations of the garbage collection loop before performing each allocation. But this means that both semispaces contain accessible cells at almost all times. In order to simplify the algorithm and the proof, *we trick the user program into believing that garbage collection ran and finished at the time of the last flip*; i.e. we assert that, as before, the user program sees addresses only in tospace.

Some slight effort must be made to keep up this appearance. When the semispaces are interchanged, all the user program registers must be updated immediately to point to tospace. This gives the collector a head start on the mutator. Since the only operations that might violate our assertion are CAR and CDR, we make sure that CAR and CDR cause forwarding addresses to be followed, and cells to be moved, when necessary. This ensures that the mutator cannot pass the collector. It turns out that preserving our assertion is much simpler than preserving the corresponding assertions of DL [16,17,23,24]. In particular, RPLACA and RPLACD cannot cause any trouble at all!

There is another problem caused by interleaving garbage collection with normal list processing: the new cells that CONS creates will be interleaved with those moved, thereby diluting the moved cells which must be traced by CONS. Of course, new cells have their cars and cdrs already in tospace and therefore do not need to be traced. We avoid this waste of trace effort through the use of the pointer T, which points to the *top* of the free area, and by allocating all *new* cells there.

```
% Serial Real-Time System (SRT).
integer k;                                % Global trace ratio parameter:
                                          %   the number of cells to trace per cons.
pointer T;                                % Top; Points to top of free area.
pointer procedure CONS(x,y) ≡             % Do some collection, then allocate (x.y).
  begin
    if B=T                                % Check if free area is empty.
      then begin                          % Switch semispaces.  Memory is full
             if S<B then error;           %   if tracing is not finished.
             flip();                      % Flip semispaces.
             for i=1 to NR
               do R[i]:=move(R[i]);       % Update user registers
             x:=move(x); y:=move(y);      %  and our arguments.
           end;
    for i=1 to k while S<B                % Do k iterations of gc.
      do begin
           S[0]:=move(S[0]);              % Update car and cdr.
           S[1]:=move(S[1]);
           S:=S+2                         % Go on to next untraced cell.
         end;
    if B=T then error;
    T := T-2;                             % Actually create the cell.
    T[0] := x; T[1]:=y;                   % Move in car and cdr.
    T                                     % Return address of new cell.
  end;
```

```
pointer procedure CAR(x) ≡ x[0]:=move(x[0]);        % Move, update and return x[0].
pointer procedure CDR(x) ≡ x[1]:=move(x[1]);        % Move, update and return x[1].
% Procedures not redefined here are as before.
```

The time required by all of the elementary list operations in this algorithm, with the exception of CONS, can easily be seen to be bounded by a constant because they are straight-line programs composed from primitives which are bounded by constants. CONS is also bounded by a constant because the number of mutator registers is a (small) fixed number (e.g. 16), and the parameter k is fixed. In principle, given the number of registers and the parameter k, the two loops in CONS could be expanded into straight-line code; hence the time it requires is also bounded by a constant.

The proof that the incremental collector eventually moves all accessible cells to tospace is an easy induction. Upon system initialization there are no accessible cells, hence none in tospace, and so we have a correct basis. Suppose that at some point in the computation we have just switched semispaces so that tospace is empty. Suppose further that there are N accessible cells in fromspace which must be moved to tospace. Now, every cell which was accessible at the time of flipping eventually gets moved when it is traced, unless lost through RPLACA and RPLACD, and as a result appears between S and B. Furthermore, a cell is moved only once, because when it is moved it leaves behind a forwarding address which prevents it from being moved again. When the pointer S reaches a cell, its edges are traced—i.e. the cells they point to are moved, if necessary. Finally, only cells which have been moved appear between S and B. Therefore, the number of those accessible, unmoved cells in fromspace decreases monotonically, eventually resulting in no accessible, unmoved cells in fromspace. At this point, the collector is done and can interchange the two semispaces.

It should be easy to see why the other list operations cannot adversely affect the progress of the collector. A CAR or CDR can move a cell before the collector has traced it. But since moving it increases B but not S, it will be traced later. RPLACA and RPLACD can affect connectivity, but since all of their arguments are already in tospace, they have already been moved and may or may not have been traced. Consider RPLACA(p,q). Suppose that p has been traced and q has not. But since q has been moved but not traced, it must be between S and B and will not be missed. Suppose, on the other hand, that q has been traced and p has not. Then when p is traced, the old CAR of p will not be traced. But this is all right, because it may no longer be accessible. If it still is the target of an edge from some accessible cell, then it either already has, or will be, traced through that edge. Finally, if either both p and q have been traced or both have not been, there is obviously no problem.

This algorithm can also be proved correct by the methods of DL [16,17,23,24], because this particular sequence of interleaving collection with mutation is only one of the legal execution sequences of the DL algorithm on a serial machine. Therefore, if the DL algorithm is correct, then so is this one. The correspondence is this: *white* nodes are those which reside in fromspace, i.e. those which have not yet been moved; *grey* nodes are those which have been moved but have not yet been traced, i.e. those between S and B; and *black* nodes are those which have been moved and traced, and those which have been allocated directly in tospace (cells below S or above T). Then the assertions are:

A) each node will only darken monotonically;
B) no edge will ever point from a black node to a white one; and
C) the user program sees only grey or black nodes.

We can now see why the burden is on CAR and CDR rather than RPLACA and RPLACD—the latter will not violate B so long as the former does not violate C. Using these assertions, we see that the mutator and the mark phase of the collector are essentially doing the same thing: tracing accessible cells. The difference is that the collector goes about it systematically whereas the mutator wanders. Thus, only the collector knows for sure when all the cells in fromspace have been traced so that the two semispaces can be interchanged. Assertion C also allows CAR and CDR to update a cell in which a pointer to fromspace is found, thus reducing pointer-chasing for cells which are accessed more than once.

We must now analyze the storage required by this algorithm. Suppose that at some flip of the semispaces there are N accessible nodes. Then the collector will not have to move or trace any more than N cells. If it traces (makes black) exactly k cells per CONS, then when the collector has finished, the new semispace will contain $\leq N+N/k=N(1+m)$ cells, letting $m=1/k$. If only N of these are accessible, as in equilibrium conditions, then the next cycle of the collector will copy those N cells back to the first semispace, while performing Nm CONSES. Hence, we have the inequality:

Maximum SRT Storage Required $\leq N(2+2m) = N(2+2/k)$

Therefore, for a program which has a maximum cell requirement of N cells operating on a fixed-size real memory of $2M$ cells, the parameter k must be greater than $N/(M-N)$ to guarantee that tracing is finished before every flip.

If we compare the bound for our algorithm with the bound for MKSMW, using the unlikely assumption that sweeping and relocation take no time ($s=r=0$), we find that they are quite similar in storage requirements.

Maximum MKSMW Storage Required $< N(1.5+2m)$

Maximum SRT Storage Required $\leq N(2+2m)$

If $m=1$ (which corresponds to one collector iteration per CONS), the two algorithms differ by only 1 part in 8, which is insignificant given the gross assumptions we have made about MKSMW's sweeping and relocation speeds. It is not likely that the storage requirements of a MKSMW-type algorithm can be significantly improved because it cannot take advantage of techniques such as stack threading or CDR-coding. Stack threading cannot be done, because accessible cells have both their car and cdr in use.[4] CDR-coding using MKSMW is very awkward because CONS must *search* for a free cell of the proper size and location before allocating a cell, since the free space is fragmented. On the other hand, our algorithm can be easily modified to use CDR-coding and thereby reduce storage requirements to approximately $N(1+m)$.

[4] The Deutsch-Schorr-Waite collector [22,p.417-418] "threads" the stack but temporarily reverses the list structure, thus locking out the mutator for the duration.

3. The Parameter m (= $1/k$)

If k is a positive integer, then the parameter m (=$1/k$) will lie in the interval $0<m\leq 1$. Therefore, the factor of $1+m$ in our bounds must lie between 1 and 2. This means that the storage requirements for our method can be adjusted by varying k, but they will not vary by more than a factor of 2 as long as k is integral. Now, the time to execute CONS is proportional to $k+c$, for some suitable constant c. Therefore, one can trade off storage for CONS speed, but only within this limited range. Furthermore, as k rises above 4 the storage savings become insignificant; e.g. doubling k to 8 yields a storage savings of only 10%, yet almost doubles CONS time. Of course, if storage is limited and response time need not be fast, larger k's might be acceptable.

If the method is used for the management of a large database residing on secondary storage, k could be made a positive rational number less than 1, on the average. For example, to achieve an average k=1/3 (m=3), one could have CONS perform an iteration of the collector only every third time it was called. The result of this would double the storage required (m+1=4), but would reduce the average CONS time by almost 2/3. Of course, the worst case time performance for CONS would still be the same as though k were 1.

This improvement is significant because each iteration of the collector traces all the pointers of one record. This requires retrieving that record, updating all of its pointers by moving records if necessary, and then rewriting the record. If there are t pointers to be updated, then t+1 records must be read and written. This sounds like a lot of work, but this much work is done only when a record is created; if there are no record creations, then with the exception of the first access of a record via a pointer stored in another record, the accessing and updating functions will be as fast as on any other file management scheme. Therefore, since secondary storage is usually cheap but slow, choosing k<1 in a file management system allows us to trade off storage space against average record creation time.

With a little more effort, k can even be made *variable* in our method, thus allowing a program to dynamically optimize its space-time tradeoff. For example, in a database management system a program might set k=0 during an initial load of the database because it knows that, even though there are many records being created, none are being let go, and therefore the continual copying of the collector will achieve no compaction. The function READ in LISP might want to exercise the same prerogative and for the same reason. Of course, any reduction of k should not take effect until the next flip to avoid overflowing storage before then.

4. A User Program Stack

If the user program utilizes its own stack as well as a bank of registers, the stack may (in theory) grow to an unbounded size and therefore cannot be wholly updated when the semispaces are flipped and still preserve a constant bound on the time for CONS. This problem may be trivially solved by simulating the stack in the heap (i.e. PUSH(x) ≡ CONS(x,*stack*) and POP() ≡ CDR(*stack*)); this simulation will satisfy the bounded-time constraints of classical stack manipulation. However, this simulation has the unfortunate property that accessing items on the stack requires time proportional to their distance from the top.

In order to maintain constant access time to elements deep in the stack, we keep stack-like allocation and deallocation strategies but perform the tracing of the stack in an incremental manner. We first fix the stack accessing routines so that the user program never sees pointers in fromspace. This change requires that the MOVE routine must be applied to any pointers which are picked up from the user stack. We must then change CONS to save the user stack pointer when the semispaces are flipped so that it knows which stack locations must be traced. Finally, the user stack POP routine must keep this saved pointer current to avoid tracing locations which are no longer on the user stack [28].

The only remaining question is how many stack locations are to be traced at every CONS. To guarantee that stack tracing will be finished before the next flip, we must choose the stack tracing ratio k' (the number of stack locations traced per CONS) so that the ratio k'/k is the same as the ratio of stack locations in use to cons cells in use. We recompute k' at each flip, because the "in use" statistics are available then. Due to this computation, a constant bound on the time for CONS exists only if the ratio of stack size to heap size is bounded, and it is proportional to that ratio.

```
% Serial Real-Time System with User Stack.
% The user stack resides in the array "ustk" and grows upward from "ustk[0]". The global variable SP is the user stack pointer
% and points to the current top of the user stack. The global variable SS scans the user stack and points to the highest stack
% level which has not yet been traced by the collector.
integer SP init(0);                         % User stack pointer.
integer SS init(0);                         % User stack scanner.
procedure USER_PUSH(x) ≡                    % Push x onto user stack.
  begin                                     % Note: x will not be in fromspace.
    SP:=SP+1;
    ustk[SP]:=x
  end;
pointer procedure USER_POP() ≡              % Pop top value from user stack.
  move(ustk[SP]) &                          % Move value if necessary.
  begin
    SP:=SP-1;                               %   then update stack pointer.
    SS:=min(SS,SP)                          % Keep stack scanner current.
  end;
pointer procedure USER_GET(n) ≡             % Get nth element from top of stack.
  ustk[SP-n]:=move(ustk[SP-n]);             % Move and update if necessary.
```

```
pointer procedure CONS(x,y) ≡                 % Collect some, then allocate (x.y).
  begin
    if B=T                                    % Check if free area is empty.
      then begin                              % Interchange semispaces.
             if SS>0 or S<B                   % Check for memory overflow.
               then error;
             N:=flip();                       % Set N to number of cells in use.
             SS:=SP;                          % Start stack scan at top of stack.
             k':=ceil(k*SS/N);                % Calculate stack trace effort.
             for i=1 to NR
               do R[i]:=move(R[i]);           % Update user registers
             x:=move(x); y:=move(y);          %   and our arguments.
           end;
    for i=1 to k' while SS>0                  % Move k' user stack elements and
      do begin                                %   update scan pointer.
           ustk[SS]:=move(ustk[SS]);
           SS:=SS-1
         end;
    for i=1 to k while S<B                    % Do k iterations of gc.
      do begin
           S[0]:=move(S[0]);                  % Trace and update car, cdr.
           S[1]:=move(S[1]);
           S:=S+2
         end;
    if B=T then error;
    T:=T-2;                                   % Actually create the cell.
    T[0]:=x; T[1]:=y;                         % Install car and cdr.
    T                                         % Return address of new cell.
  end;
```

The preceding code exhibits these changes.

The complexity involved in this conversion is essentially that necessary to make the serial real-time method work for several different *spaces* [27]. In such a system, each space is a contiguous area in the address space disjoint from the other spaces, and has its own representation conventions and allocation (and deallocation) strategies. The system of this section thus has two spaces, the heap and the user stack, which must be managed by cooperating routines.

5. CDR-Coding (Compact List Representation)

In this section, we discuss the interaction of our algorithm with a partial solution to the second big problem with list structures: their inefficient use of storage. Whereas a list of 5 elements in a language like Fortran or APL would require only a 5 element array, such a list in LISP requires 5 cells having two pointers apiece. So-called "CDR-coding" [12,19] can reduce the storage cost of LISP lists by as much as 50%. The idea is simple: memory is divided up into equal-sized chunks called *Q*'s. Each *Q* is big enough to hold 2 bits plus a pointer *p* to another *Q*. The 2 bits are decoded via the table:

00—NORMAL;	CAR of this node is *p*; CDR is *in* the following *Q*.
01—NIL;	CAR of this rode is *p*; CDR is NIL.
10—NEXT;	CAR of this node is *p*; CDR *is* the following *Q*.
11—EXTENDED;	The cell extension located at *p* holds the car and cdr for this node.[5]

CDR-coding can reduce by 50% the storage requirements of a group of cells for which CDR is a 1-1 function whose range excludes non-nil atoms. This is a non-trivial saving, as all "dotless" s-expressions read in by the LISP reader have these properties. In fact, it has been found [12] that, after linearization, 98% of the non-NIL cdrs in several large LISP programs referred to the following cell. These savings are due to the fact that CDR-coding takes advantage of the implicit linear ordering of addresses in address space.

What implications does this coding scheme have for the elementary list operations of LISP? Most operations must dispatch on the CDR code to compute their results, and RPLACD needs special handling. Consider RPLACD(*p*,*q*). If *p* has a CDR code of NIL or NEXT, then it must be changed to EXTENDED, and the result of CONS(CAR(*p*),*q*) placed in *p*.[6]

The number of memory references in the elementary operations has been minimized by making the following policies [20]: 1) every EXTENDED cell has a NORMAL extension; 2) the user program will never see a pointer to the extension of an EXTENDED cell; and 3) when COPY copies an EXTENDED cell, it reconstitutes it without an extension.

CONS, CAR, CDR, RPLACA, and RPLACD must be changed to preserve these assertions, but EQ and ATOM require no changes from their non-CDR-coded versions. Since an EXTENDED cell cannot point to another EXTENDED cell, the forwarding of EXTENDED pointers need not be iterated. These policies seem to minimize memory references because each cell has a constant (between flips) canonical address, thereby avoiding normalization [30] by every primitive list operation.

CDR-coding requires a compacting, linearizing garbage collector if it is to keep allocation simple (because it uses two different cell sizes) and take full advantage of the sequential coding efficiency. The MFYCA algorithm

[5] These conventions are slightly different from [19].

[6] We note in this context that if RPLACD is commonly used to destructively reverse a list—e.g. by LISP's "NREVERSE"—the system could also have a "PREVIOUS" CDR-code so that RPLACD need not call CONS so often.

presented above compacts, but does not linearize cdrs due to its breadth-first trace order. However, the trace order of an MFYCA collector can easily be modified at the cost of an additional pointer, *PB*. *PB* keeps track of the previous value of *B* (i.e. *PB* points to the last cell copied), so that tracing the cdr of the cell at *PB* will copy its successor into the next consecutive location (*B*), thus copying whole lists into successive contiguous locations.

The meaning of the scan pointer *S* is then changed slightly so that it points to the next *word* which must be updated rather than the next *cell*. Finally, the trace routine is modified so that tracing the cdr of *PB* has priority over tracing the edge at *S* and the condition on the trace loop is modified to amortize both the copying effort (measured by movements of *B*) and the tracing effort (measured by movements of *S*) over all the CONSES. These modifications do not result in a depth-first trace order, but they do result in cdr-chains being traced to the end, with few interruptions. Thus an MFYCA collector can minimize the amount of memory needed by CDR-coded lists.

```
% Serial Real-Time System with CDR-Coding
pointer S;                                  % Next cell whose car needs tracing.
pointer PB;                                 % Pointer to previous value of B.
pointer L,H;                                % Low and high limits of tospace.
                                            % Assertion: L≤S≤PB≤B≤T≤H.

pointer procedure CONS(x,y) ≡               % Create a new cell in tospace with
  begin                                     %   car of x and cdr of y.
    if T-B<2                                % Flip when free area is exhausted.
      then begin                            % This part is the same as usual.
             if S<B then error;             % Interchance semispaces.
             flip();
             for i=1 to NR
               do R[i]:=move(R[i]);         % Update user registers.
             x:=move(x); y:=move(y)         % Update our arguments.
           end;
    while (S+B)/2-L < k*(H-T) and S<B       % Trace and copy a measured amount.
      do if PB<B                            % Extend current list, if possible.
           then PB:=(B&CDR(PB));            % CDR will trace this edge for us.
           else begin
                  S[0]:=move(S[0]);         % Update this edge.
                  S:=S+1                    % Step S over this word.
                end;
    if B=T then error;                      % Check for memory overflow.
    T:=T-1;                                 % Create new cell at top of free area.
    if y=nil
      then code(T):="NIL"                   % If y is special case,
      else if y=T+1                         %   then create a short cell
             then code(T):="NEXT"           %   with appropriate cdr-code,
             else begin                     % Otherwise, create a normal cell.
                    if B=T then error;      % Need more space for the cdr.
                    T:=T-1;
                    code(T):="NORMAL";      % Set in "NORMAL" cdr-code.
                    T[1]:=y                 % Set in the cdr.
                  end;
    T[0]:=x;                                % Set the car in the new cell.
    T                                       % Return the new cell.
  end;
pointer procedure CAR(x) ≡                  % CAR must move cell it uncovered.
  brplaca(x,move(bcar(x)));                 % Update this edge.
procedure RPLACA(x,y) ≡                     % x[0]:=y.  May require subtlety.
  brplaca(x,y);
pointer procedure BCAR(x) ≡                 % Basic car; dispatch on CDR-code.
  if code(x)="EXTENDED"                     % Type "EXTENDED" means
    then (x[0])[0]                          %   indirect car.
    else x[0];                              % All other types have normal cars.
pointer procedure BRPLACA(p,q) ≡            % Basic rplaca; dispatch on CDR-code.
  if code(p)="EXTENDED"                     % If extended cell, clobber indirectly.
    then (p[0])[0]:=q
    else p[0]:=q;                           % All others have normal car.
pointer procedure CDR(x) ≡                  % CDR moves uncovered cell, but updates
  brplacd(x,move(bcdr(x)));                 %   only if still possible after move.
```

```
procedure RPLACD(x,y) ≡                         % x[1]:=y.  May require brute force.
  begin
    if code(x)="NIL" or code(x)="NEXT"          % Test for screw cases.
      then                                      % EXTENDED case will fall through.
        begin pointer p;                        % Extend the cell x.
          p:=CONS(CAR(x),"DUMMY");              % Construct guaranteed NORMAL cell.
          x:=move(x); y:=move(y);               % Update arguments in case CONS flipped.
          x[0]:=p;                              % Leave forwarding address in old cell.
          code(x):="EXTENDED"                   % The old cell has now been extended.
        end;
    brplacd(x,y)                                % Finally replace the cdr.
  end;
pointer procedure BCDR(x) ≡                     % Basic cdr; dispatch on CDR-code.
  if code(x)="NORMAL" then x[1]                 % NORMAL cells have a second word.
  else if code(x)="NIL" then nil                % Interpret NIL CDR-code.
  else if code(x)="NEXT" then x+1               % Interpret NEXT CDR-code.
  else (x[0])[1];                               % EXTENDED cells point to NORMAL cells.
pointer procedure BRPLACD(p,q) ≡                % Handle easy cases of RPLACD.
  if code(p)="EXTENDED"                         % We have extended cell;
    then (p[0])[1]:=q                           %   clobber the NORMAL indirect.
  else if code(p)="NORMAL"                      % The easiest case of all.
         then p[1]:=q
  else q;                                       % In all cases, return q as value.
integer procedure SIZE(p) ≡                     % Find the size of p from its CDR-code.
  if code(p)="NORMAL"                           % "NIL", "NEXT" and "EXTENDED" all have
    then 2 else 1;                              %   size(p)=1.
pointer procedure COPY(p) ≡                     % Copy the cell p; append to current
  begin                                         %   train if possible.
    if PB=B-2 and bcdr(PB)=p                    % See if we can hop this NEXT train.
    then begin
           code(PB):="NEXT";                    % Convert NORMAL cell to NEXT cell.
           B:=B-1                               % Reuse extra space now available.
         end;
    if bcdr(p)=nil                              % Create a NIL cell, if appropriate.
      then code(B):="NIL"
      else code(B):="NORMAL";                   % Otherwise, all cells are NORMAL.
    B[0]:=bcar(p);                              % Copy over car;
    brplacd(B,bcdr(p));                         %   and cdr too, if necessary.
    PB:=B;                                      % PB is end of current NEXT train.
    B:=B+size(B);                               % Step B over newly copied cell,
    if B>T then error;                          %   check for memory overflow,
    PB                                          %   and return pointer to new copy.
  end;
% Procedures not redefined here are as before.
```

The size of the tospace needed for CDR-coding is $(1+m)$ times the amount of space actually used in fromspace. With a coding efficiency improvement of e over the classical storage of LISP cells, and under equilibrium conditions, we have the inequality:

Maximum SRTC Storage Required $\leq Ne(2+2m)$

Since we have claimed that $e\approx.5$, we get the following estimate:

SRTC Storage Required $\approx N(1+m)$ (!)

But this latter expression is less than the bound computed for MKSMW. Thus, CDR-coding has given us back the factor of 2 that the copying garbage collector took away.

The real-time properties of our algorithm have not been affected in the least by CDR-coding; in fact, good microcode might be able to process CDR-coded lists faster than normal lists since fewer references to main memory are needed.

CDR-coding is not the final answer to the coding efficiency problems of list storage, because far more compact codes can be devised to store LISP's s-expressions. For example, both the car and cdr of a cell could be coded by relative offsets rather than full pointers [12]. However, a more compact code would represent some cells in so few bits that the pointer we need for a forwarding address would not fit, rendering our scheme unworkable. Part of the problem is inherent in LISP's small cell size; small arrays can perform much better.

6. Vectors and Arrays

Arrays can be included quite easily into our framework of incremental garbage collection by simply enclosing certain parts of the collector program in loops which iterate through all the pointers in the array, not just the first and second. The convergence of the method with regard to storage space can also be proved and bounds derived. However, the method can no longer claim to be *real-time* because neither the time taken by the array allocation function (ARRAY-CONS) nor the time taken by the array element accessing function is bounded by a constant. This unbounded behavior has two sources: copying an array and tracing all its pointers both require time proportional to the length of the array. Therefore, if these operations are included in a computer as noninterruptable primitive instructions, hard interrupt response time bounds for that computer will not exist. However, an arbitrary bound (say 10), placed on the size of all arrays by either the system or the programmer, allows such bounds to be derived.

A scheme which overcomes some of these problems has been devised [31]. In it, each vector is given a special link word which holds either a forwarding pointer (for vectors in fromspace which have been partially moved), a backward link (for incomplete vectors in tospace), or NIL (for complete vectors). MOVE no longer copies the whole array, but only allocates space and installs the forward and backward links. Any reference to an element of a moved but incompletely updated vector will follow the backward link to the fromspace and access the corresponding element there. When the scan pointer in the tospace encounters such a vector, its elements are incrementally updated by applying MOVE to the corresponding elements of its old self; after the new one is complete, its link is set to NIL. Element accesses to incomplete vectors compare the scan pointer to the element address; access is made to the old (new) vector if the scan pointer is less (greater or equal). Tracing and updating exactly $k \bullet n$ vector elements (not necessarily all from the same vector) upon every allocation of a vector of length n guarantees convergence.

Steele's scheme has the following properties: the time for referencing an element of any cell or vector is bounded by a constant while the time to allocate a new object of size n is bounded by c_1kn+c_2, for some constants c_1 and c_2. Hence, a sequence of list and vector operations can be given tight time bounds.

7. Hash Tables and Hash Links

Some recent artificial intelligence programs written in LISP have found it convenient to associate *property lists* with list cells as well as symbolic atoms. Since few cells actually have property lists, it is a waste of storage to allocate to every cell a pointer which points to the cell's property list. Therefore, it has been suggested [9] that one bit be set aside in every cell to indicate whether the cell has a property list. If so, the property list can be found by looking in a hash table, using the *address* of the list cell as the key.

Such a table requires special handling in systems having a relocating garbage collector. Our copying scheme gives each semispace its own hash table, and when a cell is copied over into tospace, its property list pointer is entered in the "to" table under the cell's new address. Then when the copied cell is encountered by the "scan" pointer, its property list pointer is updated along with its normal components. A "CDR-coding" system with two "scan" pointers should also keep a third for tracing property list pointers to prevent property lists from destroying chains of "next"-type cells.

8. Reference Counting

In this section we consider whether reference counting can be used as a method of storage reclamation to process lists in real time; i.e. we try to answer the question "is reference counting worth the effort in a real-time system, and if so, under what conditions?"

A classical reference count system [13,34] keeps for each cell a count of the number of pointers which point (refer) to that cell; i.e. its in-degree. This *reference count* (*refcount*) is continually updated as pointers to the cell are created and destroyed, and when it drops to zero, the cell is reclaimed. When reclaimed, the refcounts of any daughter cells it points to are decremented, and are also reclaimed if zero, in a recursive manner.

Reference counting appears to be unsuitable for real-time applications because a potentially unbounded amount of work must be done when a cell is let go. However, if a *free stack* is used to keep track of freed objects instead of a *free list* [34], the newly freed cell is simply pushed onto the free stack. When a cell is needed, it is popped off the stack, the refcounts of its daughters are decremented, and if zero, the daughters are pushed back onto the stack. Then the cell which was popped is returned. In this way, only a bounded amount of work needs to be done on each allocation.

We now consider the storage requirements of a reference counting (RC) system. In addition to the memory for N cells, we also need room for N refcounts and a stack. Since the refcounts can go as high as N, they require approximately the same space as a pointer. So we have:

Maximum RC Space Required
$$\leq 1.5N + \text{the size of the "free stack"}$$

The worst case stack depth is N. However, whenever a cell is on the stack, its refcount is zero, so we can thread the stack through the unused refcounts! So we now have:

Maximum RC Space Required $\leq 1.5\ N$

Reference count systems have the drawback that directed cycles of pointers cannot be reclaimed. It has been suggested [15,22] that refcounts be used as the "primary" method of reclamation, using garbage collection (GC) as a fallback method when that fails. Since RC will not have to reclaim everything and since the average refcount is often very small, it has also been suggested that a truncated refcount (a bounded counter which sticks at its highest value if it overflows) be used to save space.

We say that garbage in a combination RC and GC system is *ref-degradable* if and only if it can be reclaimed by refcounts alone. Cells whose truncated refcounts are struck are therefore *non-ref-degradable*.

What is the effect of a dual system in terms of performance? Whatever the RC system is able to recycle puts off flipping that much longer. By the time a flip happens in such a two level system, there is no ref-degradable garbage left in tospace. Therefore, the *turnover* of the semispaces is slowed.

How much memory does the dual system require? If truncated refcounts are used, the free stack cannot be threaded through a cell's refcount because it is not big enough to hold a pointer. Therefore, using this method and assuming only a few bits worth of truncated refcount per cell, we have:

Maximum SRT + RC Space Required
$$\leq N(2+2m) + \text{RC free stack} \leq N(2.5+2m)$$

So it appears that we have lost something by adding refcounts (even tiny ones), because we still need room for the free stack.

Let us now examine more closely the average timing of CONS under a pure RC versus a pure SRT system. The average time for CONS under the RC system is the same as the maximum time since there is no freedom in the algorithm. The time for CONS in SRT is $c_1k + c_2$, where c_1 and c_2 are constants. Now c_2 is simply the time to allocate space from a contiguous block of free storage. Certainly incrementing a pointer is much less complex than popping a cell from a stack, following its pointers, decrementing their refcounts, and if zero, pushing them onto the stack. Therefore, we can choose k small enough[7] so that the average time to perform CONS with our SRT method is smaller than the average time to perform CONS in an RC system.[8] This analysis does not even count the additional time needed to keep the refcounts updated. Of course, the storage required for our "pure" SRT system may be many times the storage of the RC system, but SRT will have a smaller average CONS time.

Since this seems counterintuitive, or at least reactionary (given the current penchant for recycling), we give a rationale for why it is so. Reference counting traces the garbage cells, while normal garbage collection traces the accessible cells. Once the number of garbage cells exceeds the number of accessible cells in a region of storage, it is faster to copy the accessible cells out of the region and recycle it whole. When $m>1$, reference counting cannot compete timewise with garbage collection because RC must trace a cell for every cell allocated while GC traces on the average only a fraction ($1/m$) of a cell for every cell allocated.

On the other hand, if we wish to minimize storage by making $m<1$, a dual scheme with truncated refcounts should reduce the average CONS time over that in the pure scheme. However, CDR-coded lists and other variable sized objects cannot be easily managed with reference counting because the object at the top of the free stack is not necessarily the right size for the current allocation. Thus, CDR-coding can reduce the storage requirement of a "pure" scheme below that of a "dual" system with the same m. But even on a system with objects of uniform size, we are skeptical whether the increased average efficiency of CONS in the "dual" system will offset the increase in k needed to keep the storage requirements the same as the "pure" system. We conclude that, at least on a real memory computer, *reference counting probably is not a good storage management technique unless one a) has uniformly sized objects; b) uses full counts; and c) guarantees no cycles*.

This is not to say that reference counts are not useful. If the LISP language were extended with a function to return the current refcount of an object, and suitably clean semantics were associated with this function, then one might be able to make use of this information *within the user program* to speed up certain algorithms, such as structure tracing or backtracking, *á la* Bobrow and Wegbreit [8]. This author is not aware of any language which makes this information available; if it were available, good programmers would certainly find a use for it.

9. The Costs of Real-Time List Processing

The amount of storage and time used by a real-time list processing system can be compared with that used by a classical list processing system using garbage collection on tasks not requiring bounded response times. The storage required by a classical noncompacting garbage collector is $N(1+\mu)$, if the system uses the Deutsch-Schorr-Waite (DSW) [22,p.417-418] marking algorithm, and $N(1.5+\mu)$ if it uses a normal stack, for some positive μ. If CDR-coding is used, copying must be done; the storage requirement is then $Ne(2+2\mu)$, where e is the efficiency of the coding. Since e is near .5 [12], the requirement is about $N(1+\mu)$, so that CDR-coding requires approximately the same space as DSW. Comparing these expressions with those derived earlier for our real-time algorithms, we find that *processing LISP lists in real-time requires no more space than a non-real-time system using DSW*. If larger non-uniform-sized objects like arrays must be managed, real-time capability requires no more space than the MFYCA system, since a copying collector is already assumed.

The *average* time requirement for CONS in our real-time system is virtually identical to that in a classical MFYCA system using the same cell representation and the same amount of storage. This is because 1) a classical system can do μN CONSES after doing a garbage collection which marks N nodes—thus giving an average cons/mark ratio of μ and allowing us to identify μ with m—and 2) garbage collection in our real-time system is almost identical to that in the MFYCA system, except that it is done incrementally during calls to CONS. In other words, the user program pays for the cost of a cell's reclamation at the time the cell is created by tracing some other cell.

CAR and CDR are a bit slower, because they must test whether the value to be returned is in fromspace. However, as noted above, any cell movement done inside CAR or CDR should not be charged to CAR or CDR because it is work which the collector would otherwise have to do and therefore has already been accounted for in our analysis of CONS. Therefore, CAR and CDR are only slower by the time required for the semispace test.[9]

Since RPLACA, RPLACD, EQ, and ATOM are unchanged from their classical versions, their timings are also unchanged.

The overhead calculated for our serial system can be compared to that in the parallel system of [33]. According to these calculations, a parallel garbage collector requires significantly more total time than a nonparallel collector. But this contradiction disappears when it is realized that the author's parallel collector continues tracing even in the absence of any cell creation activity. Since our system keys collector activity to cell creation, the collector effort is about the same as on a non-real-time system.

10. Applications

1) A Fixed Size, Real Memory Computer.

This application covers the classical 7090 LISP [25] as well as a LISP for a microcomputer. We conceive of even 16-bit microcomputers utilizing this algorithm for real-time process control or simulation tasks. Each of the list processing primitives is intended to run with interrupts inhibited, so that all interrupt processing can make use of list storage for its buffers and other needs. Multiple processes may also use these primitives so long as CONS, CAR, and CDR are used by one process at a time; i.e. they are protected by one system-wide lock. Of course, the system must be aware of the registers of every process.

For these real memory applications, we want to put as much of the available storage under the management of the algorithm as possible. Thus, both atoms (here we mean the whole LISP atom-complex, not just the print-name) and list nodes are stored in the semispaces. CDR-coding is usually a good idea to save memory, but unless the bit testing is done

---

[7] Section 3 deals with nonintegral k's.

[8] We can discount the additional time occasionally required by CAR and CDR in our method because any relocation and pointer updating done by them is work that we have charged to CONS, and does not have to be recounted.

[9] In Greenblatt's LISP machine [19], the virtual memory map performs the semispace test as an integral part of address translation.

in microcode, it may be faster to use normal cells and increase the parameter k to keep the storage size small.

The average CONS time is reduced by putting off flipping until all of the free space in tospace is exhausted, i.e. $B=T$. Thus, after all moving and tracing is done, i.e. $S=B$, allocation is trivial until $B=T$. As a result, the average CONS time in our real-time system is approximately the same as that in a classical system. Of course, with a memory size of $2M$, the maximum number of cells that can be safely managed is still $Mk/(k+1)$.

2) A Virtual Memory Computer.

The current epitome of this application is Multics LISP with an address space of 2^{36} ($\approx 10^{11}$) 36-bit words, room for billions of list cells. The problem here is not in reclaiming cells that are let go, but keeping accessible cells compact so that they occupy as few pages of real memory as possible. The MFYCA algorithm does this admirably and ours does almost as well.

Our scheme is still real-time on a virtual memory computer, but the bounds on the elementary list operations now have the order of magnitude of secondary storage operations.

There are some problems, however. Unlike MFYCA, wherein both semispaces were used only during garbage collection, our method requires that they both be active (i.e. partially in real memory) at all times. This may increase the average working set size. A careful analysis needs to be made of our algorithm in order to estimate the additional cost of incremental garbage collection. Brief consideration tells us that the active address space varies from a minimum of $N(1+m)$ just before a flip to $N(2+2m)$ just after. Since at a flip the user program registers are updated in numerical order, relatively constant pointers should be placed in the lower numbered registers to keep the trace order of constant list structure similar between flips. If the average size of an object is much larger than the size of a pointer, the working set may also be reduced by storing the forwarding addresses in a separate table instead of in the old objects in fromspace [7].

In a virtual memory environment, the active address space will automatically expand and contract in response to changes in the number of accessible cells if 1) FLIP re-adjusts the size of fromspace to $(1+m)$[cells in tospace] just before interchanging the semispaces; and 2) flipping occurs when tracing finishes rather than when B meets T. This policy, plus a smaller k than a real memory computer would use, should give both a fast CONS and a tolerable working set size. The parameter k can also be dynamically adjusted to optimize either running time (including paging) or cost according to some pricing policy by following an analysis similar to that of Hoare and others [2,10,21].

3) A Database Management System.

We conceive of a huge database having millions of records, which may contain pointers to other records, being managed by our algorithm. Examples of such databases are a bill of materials database for the Apollo Project, or a complete semantic dictionary and thesaurus of English for a language understanding program. Performing a classical garbage collection on such a databank would be out of the question, since it might require days or weeks to complete, given current disk technology.

Some of these large database systems currently depend on reference counts for storage reclamation, and so do not allow directed cycles of pointers. Since our method performs general garbage collection, this restriction could be dropped. Moreover, given enough space, our algorithm can take even *less time* than a reference count system. When compared with a classical garbage collection system, our method would not save any *total* time in processing transactions against such a database, but it would avoid the catastrophic consequences of a garbage collection during a period of heavy demand.

This case is very much like case 1, the real memory computer, because we assume that the database is orders of magnitude too big to fit into primary memory and thus that there is little hope for a speedup from the locality of reference effect. "Read memory" and "store memory" instructions here apply to secondary storage; the constant bounds for the elementary operations are now on the order of milliseconds rather than microseconds. Therefore, almost everything that we say about real memory implementations also applies to large database implementations, except that space is cheaper and time is more dear.

4) A Totally New Computer Architecture.

We conceive of an architecture in which a CPU is connected to a *list memory* instead of a random access memory. Machines of this architecture are similar to "linking automata" [22,p.462-463] and "storage modification machines" [29]. At the interface between the CPU and the memory sits a bank of *pointer registers*, which point at particular cells in the list memory. Instead of a bus which communicates both addresses and values, with read and write commands, the memory would have only a data bus and commands like CAR, CDR, CONS, RPLACA, RPLACD, EQ, and ATOM, whose arguments and returned values would be in the pointer registers. The CPU would not have access to the bit strings stored in the pointer registers, except those which pointed to atoms (objects outside both fromspace and tospace). This restriction is necessary to keep the CPU from depending upon memory addresses which might be changed by the management algorithm without the CPU's knowledge.

An advantage of such a system over random access memory is the elimination of the huge address bus that is normally needed between the CPU and the memory, since addresses are not dealt with directly by the CPU. As the number of bits on a chip increases, the number of address lines and supporting logic becomes a critical factor.

Our method of garbage collection can also be used with a random access *write-once* memory by appending an extra word to each cell which holds the forwarding address when that cell is eventually moved. Using such a system, the cells in tospace cannot be updated until they are moved to the new tospace after the *next* flip. In other words, *three* semispaces need to be active at all times. In addition to these changes, RPLACA and RPLACD must actually perform a CONS, just like RPLACD occasionally does in our CDR-coding system. Perhaps the write-once property can eliminate the need for transaction journals and backup tapes.

11. Conclusions and Future Work

We have exhibited a method for doing list processing on a serial computer in a real-time environment where the time required by all of the elementary list operations must be bounded by a constant which is independent of the number of list cells in use. This algorithm was made possible through: 1) a new proof of correctness of parallel garbage collection based on the assertion that the user program sees only marked cells; 2) the realization that the collection effort must be proportional to new cell creation; and 3) the belief that the complex interaction required by these policies makes parallel collection unwieldy. We have also exhibited extensions of this algorithm to handle a user program stack, "CDR-coding," vectors of contiguous words, and hash linking. Therefore, we consider our system to be an attractive alternative to reference counting for real-time storage management and have shown that, given enough storage, our method will outperform a reference count system, without requiring the topological restrictions of that system.

Our real-time scheme is strikingly similar to the incremental garbage collector proposed independently by Barbacci for a microcoded LISP machine [3]. However, his non-real-time proposal differs in the key points listed above. Our system will itself appear in microcoded form in Greenblatt's LISP machine [19].

There is still some freedom in our algorithm which has not been explored. The *order* in which the cells are traced is not important for the algorithm's correctness or real-time properties. The average properties of the algorithm when run on a virtual memory machine need to be extensively investigated.

The space required by our algorithm may be excessive for some applications. Perhaps a synthesis of the *area* concept [6,7] with our method could reduce the memory requirements of a list processing system while preserving the bounded-time properties of the elementary operations.

A garbage collection algorithm can be viewed as a means for converting a Von Neumann-style random access memory (with "side-effects" [25]) into a list memory (without "side-effects"). Perhaps a list memory can be implemented directly in hardware which uses considerably less energy by taking advantage of the lack of side-effects in list operations [4].

Acknowledgments. I wish to thank the people at M.I.T.'s Artificial Intelligence Laboratory and Laboratory for Computer Science (formerly Project MAC) for their time discussing these ideas, and especially Peter Bishop, John DeTreville, Richard Greenblatt, Carl Hewitt, Al Mok, Guy Steele, and the referees for their copious comments and helpful suggestions after reading early versions of this paper. I also wish to thank John McCarthy for ignoring David Hilbert's advice about "leaving elegance to the tailors" when he created the LISP language.

Received September 1976; revised May 1977

References

1. Arnborg, S. Storage administration in a virtual memory SIMULA system. *BIT 12* (1972), 125-141.
2. Arnborg, S. Optimal memory management in a system with garbage collection. *BIT 14* (1974), 375-381.
3. Barbacci, M. A LISP Processor for C.ai. Memo CMU-CS-71-103, Computer Sci. Dept., Carnegie-Mellon. Pittsburgh, Pa., 1971.
4. Bennett, C.H. Logical reversibility of computation. *IBM J. Res. Develop. 17* (1973), 525.
5. Birtwistle, G.M., DahI, O.-J., Myhrhaug, B, and Nygaard, K. *Simula Begin*. Auerbach, Philadelphia, Pa., 1973.
6. Bishop, P.B. Garbage collection in a very large address space. Working Paper 111, M.I.T. A.I. Lab., M.I.T., Sept. 1975.
7. Bishop, P.B. Computer systems with a very large address space and garbage collection. Ph.D. Th., TR-178, MIT Lab. for Computer Sci., Cambridge, Mass., May 1977. Forthcoming.
8. Bobrow, D.G. and Wegbreit, B. A model and stack implementation of multiple environments. *Comm. ACM 16*, 10 (Oct. 1973), 591-603.
9. Bobrow, D.G. A note on hash linking. *Comm. ACM 18*, 7 (July 1975), 413-415.
10. Campbell, J.A. Optimal use of storage in a simple model of garbage collection. *Inform. Processing Letters 3*, 2 (Nov. 1974), 37-38.
11. Cheney, C.J. A nonrecursive list compacting algorithm. *Comm. ACM 13*, 11 (Nov. 1970), 677-678.
12. Clark, D.W., and Green, C.C. An empirical study of list structure in LISP. *Comm. ACM 20*, 2 (Feb. 1977), 78-87.
13. Collins, G.E. A method for overlapping and erasure of lists. *Comm. ACM 3*,12 (Dec. 1960), 655-657.
14. Dahl, 0.-J., and Nygaard, K. SIMULA—an ALGOL-based simulation language. *Comm. ACM 9*, 9 (Sept. 1966), 671-678.
15. Deutsch, L.P., and Bobrow, D.G. An efficient, incremental, automatic garbage collector. *Comm. ACM 19*, 9 (Sept. 1976), 522-526.
16. Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C. S., Steffens, E.F.M. On-the-fly garbage collection: An exercise in cooperation. E.W.Dijkstra note EWD496, June 1975.
17. Dijkstra, E.W. After many a sobering experience. E.W. Dijkstra note EWD500.
18. Fenichel, R.R., and Yochelson, J.C. A LISP garbage-collector for virtual-memory computer systems. *Comm. ACM 12*, 11 (Nov. 1969), 611-612.
19. Greenblatt, R. LISP Machine Progress Report memo 444. A.I. Lab., M.I.T., Cambridge, Mass., Aug. 1977.
20. Greenblatt, R. Private communication, Feb. 1977.
21. Hoare, C.A.R. Optimization of store size for garbage collection. *Inform. Processing Letters 2* (1974), 165-166.
22. Knuth, D.E. *The Art of Computer Programming, Vol. I: Fundamental Algorithms*. Addison-Wesley, Reading, Mass., 1968.
23. Lamport, L. Garbage collection with multiple processes: An exercise in parallelism. CA-7602-2511, Mass. Computer Associates, Wakefield, Mass., Feb. 1976.
24. Lamport, L. On-the-fly garbage collection: Once more with rigor. CA-7508-1611, Mass. Computer Associates, Wakefield, Mass., Aug. 1975.
25. McCarthy, J., *et al. LISP 1.5 Programmer's Manual*. M.I.T. Press, Cambridge, Mass., 1965.
26. Minsky, M.L. A LISP garbage collector algorithm using serial secondary storage. Memo 58, M.I.T. A.I. Lab., M.I.T., Cambridge, Mass., Oct. 1963.
27. Moon, D.A. *MACLISP Reference Manual*. Project MAC, M.I.T., Cambridge, Mass., December 1975.
28. Muller, K.G. On the feasibility of concurrent garbage collection. Ph.D. Th., Tech. Hogeschool Delft, The Netherlands, March 1976 (in English).
29. Schonhage, A. Real-time simulation of multidimensional Turing machines by storage modification machines. TM-37, Project MAC, M.I.T., Cambridge, Mass., Dec. 1973.
30. Steele, G.L. Jr. Multiprocessing compactifying garbage collection. *Comm. ACM 18*, 9 (Sept. 1975), 495-508.
31. Steele, G.L. Jr. Private communication, March 1977.
32. Teitelman, W., *et al. INTERLISP Reference Manual*. Xerox Palo Alto Res. Ctr., Palo Alto, Calif, 1974.
33. Wadler, P.L. Analysis of an algorithm for real-time garbage collection. *Comm. ACM 19*, 9 (Sept. 1976), 491-500.
34. Weizenbaum, J. Symmetric list processor. *Comm. ACM 6*, 9 (Sept. 1963), 524-544.