A New List Compaction Method

Kai Li and Paul Hudak

Research Report YALEU/DCS/RR-362
February 1985

This research was supported in part by NSF Grant MCS-8302018 and DCR-8106181.

Table of Contents

A New List Compaction Method

Kai Li and Paul Hudak

Yale University
Department of Computer Science

Abstract

List compaction, or so called "cdr-coding," can greatly reduce the storage needs of list-processing languages. However, existing methods do not perform well when several lists are being constructed simultaneously from the same heap, since the non-contiguous nature of the cells being allocated eliminates the opportunity for compaction. This situation arises not only in true parallel systems sharing a common memory, and sequential systems supporting multiple processes, but also quite often in purely sequential systems, where it is not uncommon to build several different lists simultaneously within a single loop.

In this paper a new list compaction method is presented that performs well during both sequential and "parallel" list generation. The method is essentially a generalization of cdr-coding; lists are represented as linked vectors, and an encoding scheme is used that is as simple or simpler than all known encodings. In addition, a simple memory organization is proposed that supports the underlying representation and allows very rapid access to random elements in the list. Support of destructive operations is discussed, as well as primitive instructions specially tailored to the proposed memory structure. Performance figures in a simulated environment suggest that the strategy consistently performs better than conventional cdr-coding, with essentially the same complexity.

1. Introduction

Programming languages relying heavily on the list data structure have become very popular in recent years. For example, various dialects of Lisp [McCarthy 62], including Scheme [Rees 82, Sussman 75], Common Lisp [Steele 83], and others [Foderaro 82, Moon 78]; functional languages such as ALFL [Hudak 84], SASL [Turner 76], and ML [Milner 84]; and logic programming languages such as Prolog [Clocksin 81], are playing an ever-increasing role in artificial intelligence, systems programming, and many other fields. Furthermore, data-flow languages [Mcgraw 82], variations of functional and logic programming languages [Hudak 85, Shapiro 84], and new dialects of Lisp [Hastead 84, Gabriel 84] have entered into the realm of parallel processing. All of these languages depend a great deal on the list data structure, and thus supporting lists efficiently has become a major concern. The overall problem is rather complex, and involves issues such as the amount of storage consumed, how it is structured, ways to reclaim "garbage" nodes, locality of reference, and degree of page faulting.

In this paper we concentrate on how lists are represented, and the effect that has on storage requirements and other system performance parameters. Several list compaction methods that can be collectively called "cdr-coding" have been proposed in the past [Henson 69, Clark 76, Bobrow 75, Bobrow 79], and together with empirical data have been used in designing special-purpose hardware to support lisp implementations [Bobrow 79, Roads 83, Weinreb 81]. We summarize these approaches in Section 2. However, deficiencies in these schemes have led us to consider an alternative list compaction strategy that can be viewed as a generalization of cdr-coding, in which lists are represented as linked vectors. In Section 3 we point out the situations where the conventional strategies perform poorly, and in Section 4 we present our new strategy, showing why it does much better. In Section 5 we discuss further improvements and implementation issues, and in Section 6 we present performance results for some sequential and parallel program simulations.

2. Cdr-Coding

A *cons cell* is traditionally represented by two contiguous memory locations, one for the *car* and one for the *cdr*. Each of these contains either an atomic datum (such as an integer or boolean value), or a pointer to some other cons cell. Since the memory locations are usually full machine words, this allows the car and cdr to address any other location in memory.

It turns out that statistical studies have shown that in most cases the cdr field is a pointer to some other cons cell [Clark 76]. The main idea behind *cdr-coding* is to take advantage of this fact by arranging for that other cons cell to directly follow the car of the first, and to encode that information in a few extra bits contained in the car of the first cell, rather than having an explicit pointer to it. If all lists could be represented in this way, it is easy to see that such a strategy could save almost half of the overall storage requirement.

The known methods of cdr-coding can be classified into three kinds, according to the coding technique. One kind is to let the car field of a cons cell have full addressing capability and to use two extra bits to indicate different cdr types. This style of coding is used in the MIT lisp machine [Bawden 77], and is shown in Table 1. Code 00 indicates that the cdr is NIL; code 01 indicates that the next cell is the car of the cdr, and can be viewed as an "implicit pointer" to the cdr; code 10 indicates that the *contents* of the next cell is the cdr; and code 11 means that the current cell contains an indirection pointer to the "real" pair – this happens as a result of destructive operations. The coding technique used in the Symbolics 3600 [Roads 83] is similar, but does not have the indirection code, since it has a transparent addressing strategy built into its memory addressing mechanism.

The second kind of cdr-coding is to let the car field of a cons cell have full addressing capability, but to encode the cdr field to allow referencing cells by offsetting from the current location. This is a generalization of the first scheme, in that the first scheme only allows encoding the "next" location as the cdr. It is the method used in the Xerox lisp machine [Deutsch 73], where 8 bits are used for the cdr code, as shown in Table 2.

coding-bits	meaning	storage/cell
00	the CDR is NIL	1 word
01	the CDR begins in next word	1 word
10	the CDR is contained in next word	2 words
11	current cell contains indirect pointer	2 or 3 words

Table 1: MIT lisp machine cdr-coding

cdr-coding bits	cdr	storage/cell
0	NIL	1 word
1 to 127	offset of cdr	1 word
128	extended	2 or 3 words
129 to 255	offset of indirect cdr	2 words

Table 2: Xerox lisp machine cdr-coding

Although this method requires 8 bits for the cdr code, in practice it supports as large an addressing space as the MIT lisp machine. This is because the MIT machine actually uses 5 or more bits in each word as its data type tag, whereas the Xerox machine uses page numbers to identify data types, by only storing data of the same type on each page. Thus both machines support a 24-bit virtual address within a 32-bit word. An advantage of the MIT machine is that its immediate value range is as large as the car field of a cons cell minus the space for its type-tag field. The Xerox machine, however, needs to use a reserved address, called an *unboxed value*, to represent an immediate value [Moore 76]. The range of such a value is small, and the need arises for efficient ways to implement "boxing" and "unboxing."

The third coding strategy is to use page offsets for *both* the car and cdr fields [Clark 76, Bobrow 79]. It has been shown that this method can save more storage than either of the other two in total number of bits [Bobrow 79]. However, it introduces the complexity of requiring an "escape" mechanism for both the car and cdr fields (to reference cells that are too far away to encode as an offset), because neither of them has full addressing capability. Another disadvantage is that many list copying and garbage collection algorithms [Baker 78, Cheney 70, Clark 76, Deutsch 76, Fisher-D 75] cannot be used with this method because it cannot store forwarding addresses conveniently.

3. Parallel Consing

We use the phrase "parallel consing" to refer to any situation where two lists are being constructed simultaneously from the same heap. It is easy to see that whenever such a situation arises, the interleaving of heap allocations makes the basic cdr-coding technique perform poorly. Such behavior can take place in both sequential and parallel systems, as the following discussion demonstrates.

Consider a parallel environment in which there are an arbitrary number of processes sharing a single heap, and in which any process can perform a CONS operation on any list at any time. Since CONS is a frequently used operation [Clark 76], one can expect such a parallel system to have

many interleaved CONS operations within a given interval of time. (We assume that appropriate mutual exclusion mechanisms exist to handle simultaneous CONSes.) As an example of a parallel program meeting this description, consider the following T program (T is a dialect of Scheme [Rees 82]), extended with a "parallel let" construct that evaluates the defined names in parallel:

```
(define (par-merge-sort l)
    (cond ( (null? (cdr l))
            l )
          ( else
            (par-let ( (left  (merge-sort (left-half  l)))
                       (right (merge-sort (right-half l))) )
              (merge left right) ) ) ) )
```

In this example we assume that there are two processors available, so that each will sort half of the list by some sequential procedure called MERGE-SORT, and then one of the processors will merge the results. For long lists, this program will have a speedup factor of almost 2. If the sequential sort is purely functional, then each processor will make about $(\frac{N}{2}) \, log \, (\frac{N}{2})$ cell allocations, where N is the length of the list. If we assume that both processors run at approximately the same speed, then most of the allocations will be interleaved.

To see how parallel consing can occur in a sequential environment, consider the following piece of sequential T code:

```
(loop (while (some-condition))
      (do (set L1 (cons e1 L1))
          ...
          (set L2 (cons e2 L2))
          ...
          (set LN (cons e3 LN))
          ...
                    ))
```

The "loop" here is a widely-used macro [Charniak 80] whose meaning should be clear. It could be replaced by its tail-recursive equivalent, where the updated lists are passed explicity as arguments in the recursive call. As an example of a case where this situation occurs, consider the subfunction of quicksort that splits the list into two, one containing all elements less than a certain value, the other containing all other elements:

```
(define (split l head)
    (let ( (low  ())
           (high ()) )
      (loop (while (not (null? l)))
            (do (let ( (x (car l)) )
```

```
        (if (< x head)
            (set low  (cons x  low))
            (set high (cons x high)) ) )
      (set l (cdr l)) )
    (return (list low high)) ) ) )
```

In both cases the CONS operations inside the loop are likely to become interleaved, and thus the cons cells of each list will not be contiguous.

The non-contiguous memory allocation of a list that results from parallel consing hurts some list compaction methods severely. Indeed, for the first kind of cdr-coding identified earlier, there will be no compaction at all! The other two strategies will have compaction, but at the expense of requiring more bits and thus reducing the effective virtual address size. (We are particularly interested in strategies that will provide as many bits of addressability as possible on today's plethora of 32-bit machines – 24 bits is simply not enough for many applications.) Of course, the non-contiguous memory allocations can be "linearlized" by a special pass over the heap (and also may happen as a result of garbage collection), but this introduces considerably more work [Clark 76].

An interesting question in its own right is how often parallel consing of this sort occurs in practice. It depends a great deal on the algorithm, of course, and to some extent on the programmer's coding style. As an exersize in estimating how common this is, we scanned a rather large program (the Bulldog Compiler [Ellis 84b] built at Yale for very-long-word instruction architectures [Ellis 84a, Fisher-J 84] consisting of about 50,000 lines of code). We found that several critical "inner loops" used parallel consing, enough that a noticeable performance difference would occur depending on the cdr-coding technique used. Since sequential environments seem to have a non-trivial amount of parallel consings, we can expect that parallel environments will have even more.

4. A New List Data Structure and Compaction Strategy

To summarize our goals, we want a representation for lists that:

- Allows significant compaction of long lists.
- Does not degrade during parallel consing.
- Allows the full range of list operations, including destructive ones, without loss (and hopefully a gain) in efficiency.

Our approach is to represent lists as linked vectors. This idea was first proposed by Hansen [Hansen 69], but his strategy does not provide a mechanism to perform destructive operations on the resulting lists. This is a "feature" that all Lisp systems allow, and something that most Lisp programmers are loath to live without. Our strategy is a variant of Hansen's that allows destructive operations.

4.1. List representation using linked vectors

The car or cdr of a list (actually a pair) may be either an atomic value (such as NIL, an integer, or a boolean), or another list (represented as a pointer). As discussed earlier, statistics show that the cdr is usually another list, so we wish to optimize the storage requirements in that situation. Our representation is described below.

We distinguish between two kinds of cells: *content cells* and *indirection cells.* A content cell represents the car of a list, whose cdr is interpreted by looking at the next contiguous cell. If the next cell is an indirection cell, the cdr *is contained* in that cell; if it is a content cell, the cdr is a list whose car *is* that cell. There is no need to distinguish between pointer types with this strategy, because a pointer in a content cell means that the car of that list is itself a list, whereas a pointer in an indirection cell is actually an indirect reference to the real list. An efficient encoding scheme for this strategy is described in Section 5.1.

The list itself is stored as n linked vectors $v^1, v^2, \ldots, v^n$. A vector v^i has a number of cells $v^i_1, v^i_2, \ldots, v^i_k$. The first $k-1$ cells are all content cells, each containing either the special value "#" to indicate that it is unused, or a value representing an element of the list. The last cell v^i_k is an indirection cell, and always contains a value representing the cdr of the list that begins at v^i_{k-1}. This value could be NIL for a proper list, some other atomic value for an improper list, or a pointer to another list. The distinction between content cells and the indirection cell is significant: for $j = 1, \ldots, k-2$, the cdr of a list starting at content cell v_j is generally the list starting at v_{j+1}, whereas the cdr of v_{k-1} is *contained in* v_k. (Destructive operations change these relationships somewhat, as will be described shortly.) Asking what is the cdr of v_k is the same as asking what is the cdr of the object it points to – if v_k contains an atomic value, taking its cdr should result in error.

The above strategy provides a general way to construct a list. If we use vectors with a fixed-length of two, the structure degenerates to the standard cons cell. v^i in this case is the ith cons cell of the list. The "left" cell of v^i (i.e., v^i_1) contains the ith car of the list, and the "right" cell v^i_2 contains the ith cdr. In general the lengths of the vectors can vary. For example, the list (A B C (D E) F G) constructed using variable-length vectors is shown in Figure 1. In this and subsequent figures, a box with no horizontal lines is a content cell, and a box with one horizontal line nears its top is an indirection cell.

When referring to a list l, we consider l to be a pointer to its first element, which will usually be a content cell v_j in some vector v. For convenience we will assume that cells differ in address by one, so that $l+1$ refers to the next cell v_{j+1} in the vector v. Finally, the contents of the cell pointed to by l is denoted contents(l).

4.2. List compaction using linked vectors

The basic idea behind the allocation scheme is to allocate vectors one at a time, always filling in the empty slots before allocating the next vector. More precisely, a vector is allocated as a result

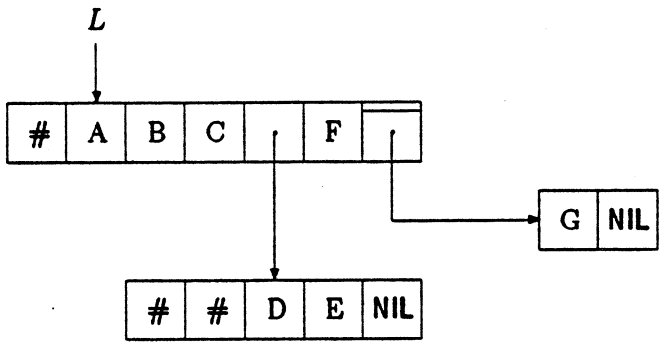


Figure 1: The representation of list L = (A B C (D E) F G)

of a CONS operation whenever a new list is being created or an element is being added to an old vector that does not have an unused cell. If there *is* an unused cell, the new element is simply placed there. The following definition of CONS describes this behavior more formally:

```
(define (CONS x y)
    IF   <y is a list pointer and y-1 is unused>
    THEN <put x into cell y-1;
          return y-1>
    ELSE <allocate a new vector u of length k;
          put x in cell u_{k-1};
          put y in cell u_k (i.e., the indirection cell);
          return a pointer to u_{k-1}> )
```

Note that no check is made to see if y is the first cell in the vector before checking to see if $y - 1$ is unused. This is because if y *is* the first cell, then $y - 1$ will *never* be unused.

During parallel consing, this version of CONS will preserve compaction within a vector. The degree of compaction depends of course upon the length of the vector and the total number of unused cells in the program. The goal is naturally to make vectors as large as possible while keeping the number of unused cells to a minimum. Unfortunately, these two goals are in direct conflict with one another, since larger vectors tend to create more unused cells. In Section 4.3 we present a variable-length vector strategy that eliminates this problem to some degree.

Given the above definition of CONS, the definitions of CAR and CDR are straightforward:

```
(define (CAR x)
    IF   <x is not a list> THEN ERROR
    ELSE IF   <x is an indirection cell>
         THEN (CAR contents(x))
         ELSE contents(x) )
```

```
(define (CDR x)
    IF   <x is not a list> THEN ERROR
    ELSE IF   <x is an indirection cell>
         THEN (CDR contents(x))
         ELSE IF   <x+1 is an indirection cell>
              THEN contents(x+1)
              ELSE x+1 )
```

All of the known cdr-coding techniques require special treatment of destructive operations such as RPLACA and RPLACD (which in this paper we will call SET-CAR and SET-CDR, respectively), and our scheme is no different. The algorithm for SET-CAR is:

```
(define (SET-CAR x y)
    IF   <x is not a list> THEN ERROR
    ELSE IF   <x is an indirection cell>
         THEN (SET-CAR contents(x) y)
         ELSE <change contents(x) to y;
               return y> )
```

The destructive operation SET-CDR changes the *data dependencies* between elements in a list, and is thus a bit more complex. We need to consider two cases, *explicit cdr* and *implicit cdr*. A list has an *explicit* cdr only when the next cell in the vector is an indirection cell; this always happens for the last content cell v_{k-1}, and also happens as a result of SET-CDR operations, as will be described. In all other cases an *implicit* cdr exists, because the first element in the cdr is stored in the next cell in the vector, so there is no explicit pointer to it. Doing a SET-CDR in the former case is easy if the indirection cell is also the last cell in the vector: we just change the contents of the indirection cell to the new cdr (and is exactly what would be done in a conventional lisp implementation). This works because it can never happen that some other cell is pointing to the last cell in a vector, so there is no harm in modifying it.

However, if the indirection cell is not the last cell, or if the cdr is implicit, we need to copy the car of the original list L into the last content cell u_{k-1} of a new vector u, change the car of L to a pointer to u_{k-1}, and set the indirection cell of u to the new value we are setting L's cdr to. For example, (SET-CDR L '(X Y)) has the effect shown in Figure 2, where L is the same list as in Figure 1. Note however, that if the list '(X Y) has room in its first vector to contain the car of L, we could avoid creating a new vector; this case is shown in Figure 3. It turns out that whether or not a new vector is required is conveniently captured within the definition of CONS, leading us to the following definition for SET-CDR:

```
(define (SET-CDR x y)
    IF   <x is not a list> THEN ERROR
```

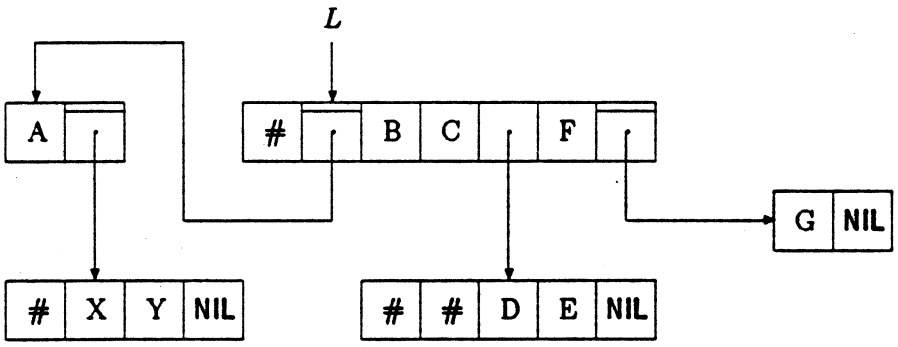


Figure 2: The consequence of (SET-CDR *L* '(X Y)) with a vector allocation

```
ELSE IF   <x is an indirection cell>
     THEN (SET-CDR contents(x) y)
     ELSE <IF   <x+1 is an indirection cell and last cell in the vector>
          THEN <put y in x+1>
          ELSE <temp := (CONS contents(x) y);
                put temp in x;
                make x an indirection cell>;
          RETURN y> )
```

Note that the statement "`temp := (CONS contents(x) y)`" will create a new vector if need be, but will otherwise place the car of x into an unused cell of y.

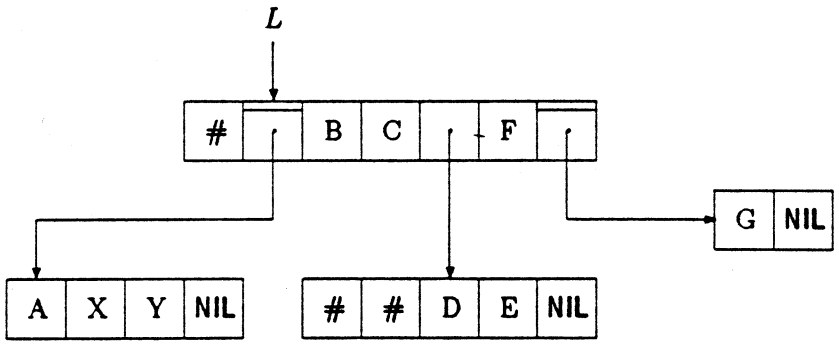


Figure 3: The consequence of (SET-CDR *L* '(X Y)) without vector allocation

4.3. Further opportunities for compaction

It has also been observed that NIL is the most common element in a list [Clark 76], so it might be worthwhile to optimize such occurrences as is done in conventional cdr-coding. We call

this *CDR-NIL compaction*, and it can be implemented by extending our coding strategy to include a code for NIL. Figure 4 shows how the list in Figure 1 might be represented with CDR-NIL compaction, where a box with two horizontal lines near its top is a content cell whose cdr is NIL. Ways to represent this information explicitly are discussed in Section 5.1.

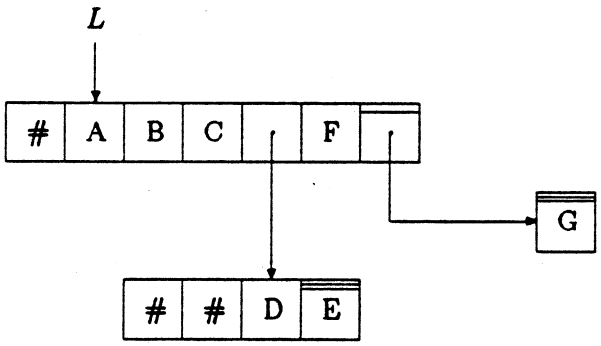


Figure 4: The list L = (A B C (D E) F G) represented using CDR-NIL compaction

Another useful idea in a system using variable-length vectors is to give control of the vector size to the programmer. This might be done by providing a function SET-CONS-VECTOR-SIZE that takes an integer argument, or perhaps by letting the CONS function take an optional argument indicating the length of the allocated vector (if any). In the latter case the programmer could then write (CONS x y) to use the default vector length, or (CONS x y k) to specify the length to be k.

Such dynamic control of the vector length can be very useful in optimizing the storage needs of a program. In particular, if a client knows, either statically or dynamically, the exact length of a list being constructed, much better compaction can be obtained. An example of using CONS in this way is in the definition of SET-CDR given earlier, where one might assume that destructive operations do not occur very often, so that if a new vector is allocated it should be of length two. This can be accomplished by changing the statement "`temp := (CONS contents(x) y)`" to "`temp := (CONS contents(x) y 2)`."

It should be noted that there is a significant advantage to using the variable-length method in that if two vectors of a list are contiguous, they can become one. This situation happens often when one long list is being allocated during which there are no intervening allocations for other lists. We can modify the CONS algorithm to take advantage of this situation as follows:

```
(define (CONS x y)
    IF   <y is a list pointer and y-1 is unused>
    THEN <put x into cell y-1;
          return y-1>
    ELSE IF   <y is a pointer to a cell v1,
```

```
        and v is the last vector allocated from the heap>
THEN <extend v by k cells, so that v_1 is now v_{k+1};
        put x in v_k;
        return a pointer to v_k>
ELSE <allocate a new vector u of length k;
        put x in cell u_{k-1};
        put y in cell u_k (i.e., the indirection cell);
        return a pointer to u_{k-1}> )
```

The reader should compare this to the previous definition of CONS. It is interesting to note that when $k = 1$, this compaction method is functionaly equivalent to the MIT lisp machine cdr-coding method.

5. Implementation Issues

5.1. Encoding strategies

In order to implement the algorithms described in the last section, we need to know whether or not a cell:

- is an indirection cell or a content cell.
- is the last cell of a vector.
- contains the unused value #.
- contains NIL.

As is true of most representational issues such as this, there are many ways this information could be stored. We discuss in this section several suitable coding alternatives. The total requirements of a particular implementation will dictate the final strategy used.

In most Lisp systems, values have a type-tag indicating their type. We could assume that NIL and the special unused value # are suitably represented within this typing scheme, either by introducing an extra type or by designating special values. One could also have different types to distinguish indirect pointers from normal ones, but this will sometimes require coercing pointers from one kind to the other. A better strategy might be to allocate a special bit for each cell to indicate whether it is an indirection cell or a content cell.

Several possibilites also exist for indicating whether or not a cell is the last cell in a vector. If we use a fixed-length $k = 2^i$ for all vectors, we do not have to explicitly remember which cell is the first or last, as long as we allocate vectors on even k-word boundaries (often called *k-word allignment*). This is because the address of the first cell will always have zeros for its i low-order bits, whereas the address of the last cell will always have ones. However if we use variable-length vectors, we really need an extra bit to encode the fact that a cell is the last.

It should be noted that the only routine that needs to know whether a cell is the last or not is SET-CDR. This need could be eliminated altogether at the expense of having SET-CDR always perform a CONS instead of performing the optimization of setting the last cell. This is not a great loss, since it is easy to argue that such an optimization rarely happens, especially since SET-CDR is a rare operation itself and one that most programmers try to avoid.

If CDR-NIL compaction is used, a particularly concise encoding of the cells is possible that takes advantage of the fact that a cell cannot simultaneously be "several things at once." It also provides an encoding for an unused cell. The encoding is shown in Table 3.

coding-bits	meaning	storage/cell
00	the CDR is NIL	1 word
01	the CDR begins in next cell	1 word
10	the current cell is an indirection cell	2 words
11	the current cell is unused	N/A

Table 3: Concise encoding for linked-vectors

Note that this encoding is as compact as, and very similar to, that used by the MIT lisp machine shown in Table 1. An advantage of having a separate code for an unused cell is that if an "auxiliary memory" is used as described in Section 5.3, entire areas of the heap may be cleared with little effort.

5.2. Garbage collection

Garbage collection algorithms based on copying between two semi-spaces (such as Baker's algorithm [Baker 78], in which cdr-coding has been considered) can be used with our list representation without modification. This is because the interface to the list representation is the same in our scheme as with conventional cdr-coding, and the car field can be used to store a forwarding address. Standard mark-sweep algorithms can be used also, but the fractured free-list that results will be difficult to allocate new vectors from, warranting some sort of compaction phase as well. The details of such algorithms are beyond the scope of this paper.

If a reference counting strategy is used for garbage collection, the performance of SET-CDR can be improved considerably. Specifically, when (SET-CDR x y) is performed, if x is not an indirection cell and the reference count of x+1 is 1, then we can just change the contents of x+1 to y. This holds regardless of whether x+1 is a content cell or an indirection cell. The strategy also eliminates the need to know which cell in a vector is its last, since that cell will never have a reference count greater than one. Since measurements of Lisp programs show that about 97% of list cells have just one reference to them [Clark 76], this optimization will eliminate most of the unnecessary vector allocations and resultant indirect pointers.

5.3. Auxiliary memories and list traversing functions

Most lisp dialects support a variety of functions that need to traverse lists, such as APPEND, REVERSE, LENGTH, NTH, NTHCDR, LAST, etc. Some of these functions provide the user

with a notion of "random access" to elements in a list, but a reference to the ith element typically requires at least i memory references. The slowness of these operations is one reason why many lisp dialects also provide contiguous data types such as vectors and arrays, whose elements can be accessed in constant time. In this section we present a memory architecture based on linked vectors that can speed up list traversing operations substantially, and has other advantages as well.

In a system using vectors of fixed-length k, one could skip over the first $\frac{i}{k}$ vectors when searching for the ith element in a list L, giving an average speedup of k. However, this strategy does not work in a variable-length vector system unless we store the length of each vector, nor in a system allowing destructive operations such as SET-CDR since they introduce indirection cells that require a scan of the whole vector to determine their presence.

In this section we describe a simple memory architecture that solves this problem by providing hardware support for the "scanning" process. For generality, suppose we have a system using variable-length vectors, CDR-NIL compaction, and the SET-CDR optimization that requires knowing which cell is the last in each vector. The main idea is to extract the indirection-cell bit, the CDR-NIL bit, and the last-cell bit from each word and put them into separate *auxiliary memories*, as shown in Figure 5 (alternatively, one could use the more concise encoding described in the last section). The main memory containing the contents of each cell is called the *content memory*. Each bit in an auxiliary memory is associated with one word in the content memory. Thus if the content memory contains $N = 2^n$ words, and $W = 2^m$ is the number of bits per word in the auxiliary memory, then there are $\frac{N}{W} = 2^{n-m}$ words in each auxiliary memory.

We assume that when the contents of a cell is fetched, the corresponding word from each auxiliary memory is loaded into the registers $R_{indirect}$, R_{nil}, and R_{last}. Similarly, if a word is written in content memory, we assume that the corresponding bits are set appropriately in the auxliliary memories. In all cases these memory accesses can be done in parallel, so we assume that no extra time is incurred. We also allow for the auxiliary memories to be read or written independently of the content memory, by providing suitable machine instructions to do so.

With such a memory architecture, we can design special instructions to aid list processing using linked vectors. In particular, to speed up list traversing operations, we would like to find the next indirection cell or the next cell whose cdr is NIL. Let us define a word as *interesting* iff it satisfies these properties. The instruction we want is then:

```
FINDI S, B, R
```

where S is a register containing the memory location from which the search is to begin, B contains an upper bound on the number of words to "effectively" search through, and R is a result register. Suppose i is the address of the next interesting word, and let r be the minimum of $contents(B)$ and $i - contents(S)$. Then after instruction execution, the following things happen:

- R is loaded with r.

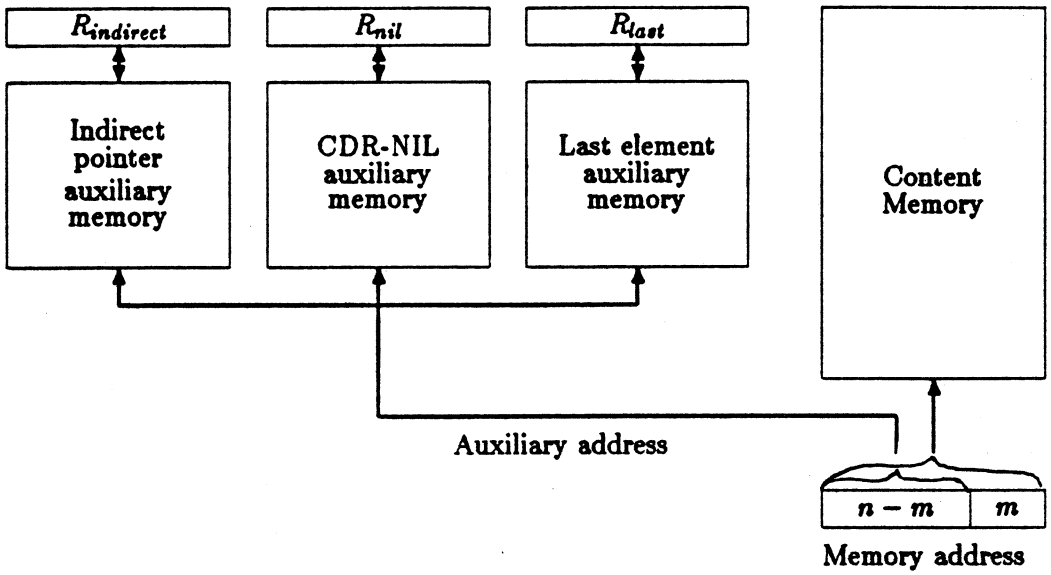


Figure 5: Auxiliary memories and content memory

- B is decremented by r.
- S is incremented by r.

It is easy to see how this instruction might be implemented given our memory design: The auxiliary memories are scanned sequentially until a non-zero word is found in either $R_{indirect}$ or R_{nil}. The first such bit that corresponds to a word in content memory greater than S is the desired one. The scan stops prematurely if more than contents(B) words are effectively scanned in content memory. We imagine that such an instruction could be implemented in micro-code very efficiently. Note that the number of memory references needed to find the nth element in a contiguous list is at most $\left(\frac{n}{W}\right)+1$.

We would also like conditional branch instructions that branch based on the execution of FINDI. For example:

```
BI    <effective-address>    ; branch if interesting word found
BNIL  <effective-address>    ; branch if interesting word is NIL
BBE   <effective-address>    ; branch if bound exceeded (contents(B)=0)
   ...
```

The instructions mentioned above can be used to implement operations requiring list traversals, such as APPEND, REVERSE, LENGTH, NTH, NTHCDR, and LAST. The algorithms are fairly straightforward, and are all variations of NTH, described below. NTH takes two arguments, a list L and index n. We assume that they are in registers R1 and R2, respectively, and that R0 is available to store a result. The code for NTH is then:

```
NTH:    FINDI   R1,R2,R0        ; find next interesting word
        BBE     EXIT            ; found nth element
```

```
        BNIL    RTNIL           ; index is out of range
        MOVE    (R1),R1         ; follow indirect pointer
        JMP     NTH             ; loop
RTNIL:  MOVE    NIL,R0          ; put NIL into R0
        RTS                     ; return
EXIT:   CAR     R1,R0           ; put nth car into R0
        RTS                     ; return
```

Note that the "inner loop" here consists of 5 instructions, the same number likely to be needed by an implementation on a conventional architecture. Of course, one loop here corresponds to an arbitrary number of loops on a conventional machine, and therein lies the increase in performance. More precisely, whereas a conventional implementation might loop n times (corresponding to n elements), the algorithm above only loops once for each indirect pointer found in the first n elements of the list, and each of these loops involves one execution of FINDI whose micro-coded implementation scans through the auxiliary memory as described earlier. Generally speaking, the number of memory references saved when scanning a list of length n is:

$$n - aux - ind$$

where aux is the number of auxiliary memory references made by FINDI, and ind is the number of indirect pointers in the list. It is easy to show that this quantity is always $>= 0$, because $ind < n$ and an auxiliary memory reference occurs only when the distance between the given address and the next interesting word is greater than 1, in which case at least one reference in content memory is saved. The upper bound of the speedup factor depends on W; the larger it is, the greater the gain in speed, with a modest tradeoff in hardware cost.

5.4. Other uses for auxiliary memories

It is worth noting that the auxiliary memory concept can be used for other purposes as well. In this section we point out two such applications.

Efficient garbage collection is an important issue in Lisp implementations. For example, the Symbolis 3600 lisp machine [Moon 84, Roads 83] uses hardware to overcome the inefficiency of scanning pointers. In addition to word-tags, each physical page of memory has a page-tag bit that indicates when a pointer to temporary space has been written into some location on that page. When the garbage collector wants to reclaim the temporary space, it only needs to scan the pages whose page-tag bits were set. This speeds up selecting the pages to scan, but does not help in the scan of the page itself – the garbage collector must do this sequentially. However a "GC auxiliary memory" could be used to speed up the sequential scan. When a pointer to temporary space is written into a word, its associated bit in the GC auxiliary memory is set (and is of course cleared if that pointer is subsequently removed). During garbage collection the next word containing a pointer to temporary space can be found in a way similar to that of FINDI described earlier.

Another use of auxiliary memories is in the support of parallel access to many elements in a list. Since many of the elements will be contiguous, appropriate hardware to support vector operations could be used just as in conventional vectorization strategies in high-performace machines. The auxiliary memory would play the role of controlling the extent of the individual vector operations and controlling the looping mechanism to traverse the entire list. Used in this way, one can imagine parallel versions of the standard mapping functions such as MAP, MAPCAR, and MAPCDR (to be used, of course, only when the parallel computations are simple enough and do not interfere through side-effects).

5.5. Impact on Destructive Operations

In this section we discuss the advantage of representing lists as linked vectors when performing destructive operations. In Section 4.2 we described how SET-CDR worked; here we concentrate on two other frequently used functions that produce side-effects: APPEND! (destructive version of APPEND) and REVERSE! (destructive version of REVERSE).

Implementing APPEND! by SET-CDR is fairly easy and very efficient, with or without CDR-NIL compaction. Without using CDR-NIL compaction, the end of a list L is characterized by the value NIL in the last cell of a vector; to accomplish (APPEND! L M) one simply replaces this value with a pointer to M. When using CDR-NIL compaction, the cell containing the last element of the list is specially marked, and if the list M has no unused cells in front of it, one cannot avoid allocating a new vector. Whether or not a new vector is required is easily determined using CONS, in a way almost identical to that of SET-CDR described in Section 4.2.

In all previous list compaction strategies, if one were to implement REVERSE! in the obvious way using SET-CDR, the result would be a list of almost all normal (i.e., two-element) cons cells - all compaction is lost. Doing this using our method is even worse when $k > 2$, because each SET-CDR operation will create a new vector containing $k - 2$ unused cells! Of course, no matter which compaction strategy is used, the correct way to solve this problem is to make REVERSE! a primitive, as described below.

In a system not using CDR-NIL compaction, the algorithm is rather simple:

(define (REVERSE! L)

- Let L consist of n vectors v^1 through v^n, each vector v^i having (variable) length k_i.
- For each vector v^i having form:

  $$v^i = < v^i_1,\ v^i_2,\ \ldots\ ,\ v^i_{k_i-1},\ v^i_{k_i} >,$$

  reverse its content elements destructively yielding:

  $$v^i = < v^i_{k_i-1},\ \ldots\ ,\ v^i_2,\ v^i_1,\ v^i_{k_i} > .$$

- Reverse all pointer cells destructively such that:

  $$v^1 = < v^1_{k_1-1},\ \ldots\ ,\ v^1_2,\ v^1_1,\ NIL >$$

and for $i = 2, \ldots, n$:

$$v^i = < v^i_{k_i-1}, \ \ldots \ , \ v^i_2, \ v^i_1, \ \texttt{pointer-to-}v^{i-1} >,$$

- Return a pointer to v^n.)

Note that this algorithm can be done in one pass over the list.

In a system using CDR-NIL compaction, the algorithm becomes slightly more complex. To avoid allocating extra storage, every vector except the first needs to shift one of its elements into its left neighboring vector, since there is no slot in the last vector to store a pointer to its new cdr, yet there is an extra slot in the first vector because one indirect cell was freed up by the CDR-NIL encoding. The resulting algorithm is as follows:

(define (REVERSE! L)

- Let L consist of n vectors v^1 through v^n, each vector v^i having (variable) length k_i.
- Reverse the elements in v^1 and shift in v^2_1, such that:

  $$v^1 = < v^2_1, \ v^1_{k_1-1}, \ \ldots \ , \ v^1_2, \ v^1_1 >$$

  and set the CDR-NIL bit of the last cell.
- For $i = 2, \ldots, n-1$, reverse v^i such that:

  $$v^i = < v^{i+1}_1, \ v^i_{k_i-1}, \ \ldots \ , \ v^i_2, \ \texttt{pointer-to-}v^{i-1} > .$$

- Reverse v^n such that:

  $$v^n = < v^n_{k_n}, \ \ldots \ , \ v^n_2, \ v^n_1, \ \texttt{pointer-to-}v^{n-1} > .$$

  and make the last cell an indirection cell.
- Return a pointer to V_n.)

This algorithm can also be implemented in a single pass.

6. Simulation

Most of the algorithms and data structures described in this paper have been simulated on real programs by making modifications to a compiler for T [Rees 82]. The compiler was changed to produce new closed code for the primitive operations CAR, CDR, CONS, SET-CAR, and SET-CDR (and also involved re-compiling all list routines used by the compiler). The new versions of these primitives allocate lists as linked vectors (with CDR-NIL compaction) in a simulated heap that is represented as a very large vector in T. Thus a list in the simulated environment is a word whose type-tag is the same as a cons cell in the normal environment, but whose value is an index into the vector representing the heap. After programs were ran, statistical information was gathered by scanning through the entire vector representing the heap. Since the simulation was in essence done

"on top of" the original environment, it's correctness was preserved even during garbage collection. Overall this was a non-trivial effort, but was worthwhile since it allowed us to run any T program without modification in the new environment.

One of our primary goals was to simulate parallel execution to maximize the effect of "parallel consing." We did this by first running the parallel program segments sequentially, while keeping track of every CONS operation that was invoked. We could then simulate parallel allocations from the heap by either randomly or regularly interleaving the resulting "cons traces." In the examples below the basic model is one of two parallel processors sharing the same heap memory.

In the first example we simulate the simultaneous copying of two independent lists – one would expect an extensive amount of parallel consing in such a case. Table 4 shows the results when copying a list of 1001 elements on each processor, simulated by perfectly interleaving the cons traces. A more descriptive curve is given in the Appendix.

Vector length	Memory used	Unused elements	Indirect pointers
1	4002	0	2000
2	4000	0	1998
3	3000	0	998
4	2672	4	666
5	2500	0	498
6	2400	0	389
7	2338	4	332
8	2288	2	284
9	2250	0	248
10	2240	13	222
11	2200	0	198
12	2184	2	180

Table 4: Memory usage of copying two lists in parallel

Several interesting (although quite predictable) things are worth noting. First, with vector length one we are essentially simulating the MIT cdr-coding scheme – note that two cells get allocated for each list element, as expected. With vector length two, essentially no improvement is realized, since with CDR-NIL compaction the two schemes behave almost identically. Finally, note that as the vector length increases, substantial improvements are obtained, which asymptotically should approach a factor of two. The purpose of this first example is to show the worst case of MIT cdr-coding and the best case of ours – we present a more realistic example below.

Consider the version of parallel merge sort (PAR-MERGE-SORT) given in Section 3, where we assume that the sequential procedure MERGE-SORT is written without side-effects. The reason for selecting this program and writing it in this particular way is that we want to create many lists of different lengths, in order to avoid both the worst case of MIT cdr-coding and the best case of our strategy. A recursive merge sort is ideal for this, since near the root of the recursion it creates

long lists, but towards the fringes very small lists are generated. We would expect our strategy to do poorly at the fringes since many unused cells would be created, but do well near the root where long lists are made. It seems like a good "average" test case.

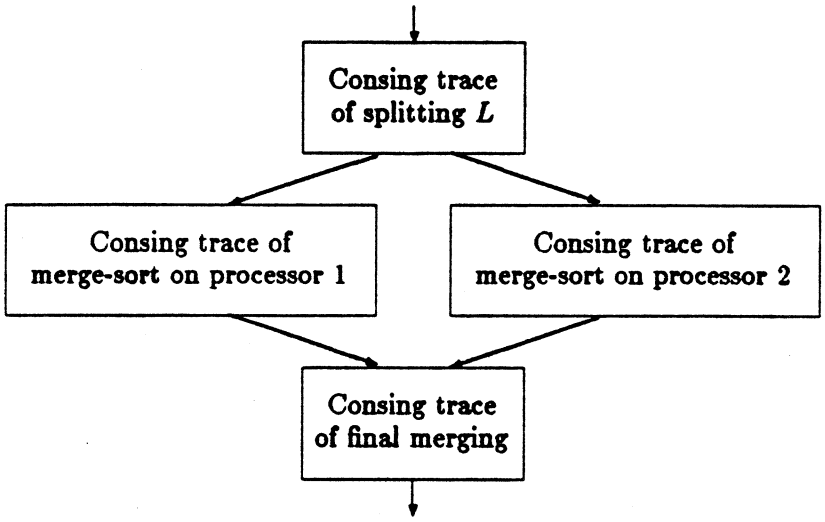


Figure 6: Consing trace graph of a parallel merge sort

The resulting cons traces for PAR-MERGE-SORT were combined as shown in Figure 6. The trace for the inital split and final merge (which are both very short) remain sequential, whereas the parallel executions of MERGE-SORT were simulated by interleaving the cons traces randomly. We ran the PAR-MERGE-SORT program on a large existing database file. The results are shown in Table 5.

Vector length	Memory used	Unused elements	Indirect pointers
1	6713	0	2792
2	6697	54	2721
3	5817	318	1598
4	5516	373	1122
5	5725	855	949
6	5934	1182	831
7	6076	1413	732
8	6208	1629	658
9	6651	2095	625
10	6970	2446	603
11	7348	2842	585
12	7788	3297	576

Table 5: Memory usage of a parallel merge sort

A normal cons-cell implementation using no compaction at all uses 8042 words of memory for this same program. Vector length 1 (corresponding to the MIT scheme) only requires 6713 words, an

immediate improvement of 16%. As the vector length increases beyond that, the improvement also increases, reaching a maximum of 31% at vector length 4 – almost twice as much improvement as the MIT scheme. Beyond vector length 4 the improvement begins to decrease, but even at vector length 12 it is an improvement over no compaction at all, and up to vector length 10 there is still an improvement over the MIT strategy. A more descriptive version of this data is given in the Appendix.

7. Conclusion

In comparing our compaction scheme with previous ones, we can first of all conclude that it performs as well and usually better than the MIT cdr-coding strategy, with no greater complexity in encoding. It is more difficult to compare to the Xerox lisp machine strategy since the latter is in some sense only a "pseudo-compaction" scheme in that it uses a relatively large code that acts as sort of a "local address" for nearby elements. Indeed, such a trick is not incompatible with our scheme, whose encoding could easily be extended to include an offset field. In terms of maximum compaction with minimum overhead in coding bits, our method seems superior to either of the other two.

We have also suggested a physical memory organization to support our encoding scheme. This "auxiliary memory" structure is quite simple, yet provides invaluable support for certain very common operations, such as rapid access to random elements in the list, and garbage collection tasks. When coupled with micro-coded primitives for implementing common operations, this structure seems to provide a realistic design choice for a fast lisp machine. Indeed, it might obviate the need for vectors as a separate data-type.

Another way of viewing our list compaction strategy is to think of it as allocating many small "mini-heaps" that collectively make up larger lists. As a list grows, its new elements are first allocated from these mini-heaps, which are in turn allocated from a single main heap within which conventional garbage collection is performed. A possibility for future research is the generalization of this idea to distributed processing implementations.

Acknowledgement

We wish to thank John Ellis, Norman Adams, and Jonathan Rees for their invaluable help in the simulation work. Without help from the latter two, the simulations using T would have been impossible. Also thanks to Christopher Riesbeck and James Spohrer for providing AI programs with which to run experiments. Finally, we wish to thank Professor Alan Perlis for his continual help and inspiration.

This research was supported in part by NSF Grant MCS-8302018 and DCR-8106181.

References

[Baker 78] Baker, Henry G Jr., "List Processing in Real Time on a Serial Computer," CACM Vol. 21, No. 4, April 1978, pp. 280-294

[Bawden 77] Bawden, A. , R. Greenblatt, J. Holloway, T. Knight, D. Moon and D. Weinreb, "Lisp Machine Progress Report," Memo 444, MIT AI Lab., Aug. 1977.

[Bobrow 75] Bobrow, Daniel G., "A Note on Hash Linking," CACM Vol. 18, No. 7, July 1975, pp. 413-415.

[Bobrow 79] Bobrow, Daniel G. and Douglas W. Clark, "Compact Encodings of List Structure," ACM Trans. on Programming Languages and Systems, Vol. 1, No. 2, Oct. 1979, pp. 266-286.

[Cheney 70] Cheney, C. J. , "A Nonrecursive List Compacting Algorithm," CACM Vol. 13, No. 11, Nov. 1970, pp. 677-678.

[Clark 76] Clark, Douglas W., "List Structure: Measurements, Algorithms, and Encodings," Ph. D. thesis, Carnegie-Mellon University, August 1976.

[Clocksin 81] Clocksin, W. F. and Mellish, C. S. , "Programming in Prolog," Springer-Verlag, 1981.

[Deutsch 73] Deutsch, L. Peter, "A Lisp Machine with Very Compact Programs," Proceeding of 3rd IJACI, Stanford, 1973, pp. 697-703.

[Deutsch 76] Deutsch, L. Peter and Daniel G. Bobrow, "An Efficient, Incremental, Automatic Garbage Collector," CACM Vol. 19, No. 9, Sept. 1976, pp. 522-526.

[Ellis 84a] Ellis, John R. , "Bulldog: A Compiler for VLWI Architectures", Ph. D. Thesis, Yale University, expected Dec. 1984.

[Ellis 84b] Ellis, John R. , Private communication, 1984

[Fisher-D 75] Fisher, David A. , "Copying Cyclic List Structures in Linear Time Using Bounded Workspace," CACM Vol. 18, No. 5, May 1975, pp. 251-252.

[Fisher-J 84] Fisher, Joseph A. , John R. Ellis, John C. Rutternberg, and Alexandru Nicolau, "Parallel Processing: A Smart Compiler and a Dumb Machine," Proceedings of The SIGPLAN '84 Symposium on Compiler Construction, ACM SIGPLAN, Vol. 19, No. 6, June 1984, pp. 37-47.

[Foderaro 82] Foderaro, J. K. and K. L. , Sklower, *The Franz LISP Manual*, Department of Electrical Engineering and Computer Science, Univeristy of California, Berkeley, April 1982.

[Gabriel 84] Gabriel, Richard P. and John McCarthy, "Queue-based Multi-processing Lisp," 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, Aug. 1984, pp. 25-44.

[Halstead 84] Halstead, Robert H. Jr. , "Implementation of Multilisp: Lisp on a Multiprocessor," 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, Aug. 1984, pp. 9-17.

[Hansen 69] Hansen, Wilfred J. , "Compact List Representation: Definition, Garbage Collection, and System Implementation," CACM Vol. 12, No. 9, Sept. 1969, pp. 499-507.

[Hudak 84] Hudak, P. , "ALFL Reference Manual and Programmers Guide," Yale Department of Computer Science Technical Report YALEU/DCS/TR-322, Oct. 1984.

[Hudak 85] Hudak, P. and Smith, L. , "Explicit Mapping of Functional Programs to Multiprocessor Systems" submitted to 1985 Symposium on Functional Programming and Computer Architecture, 1985.

[McCarthy 62] McCarthy John, et al, *LISP 1.5 programmer's manual*, MIT Press, Cambridge, Mass. , 1962.

[Mcgraw 82] Mcgraw, J. R. , "The VAL language: description and analysis," ACM Trans. on Prog. Lang. and Sys., 4(1), Jan. 1982, pp 44-82.

[Milner 84] Milner, R. , "A Proposal for Standard ML," in Proc. of ACM Sym. on LISP and Functional Programming, August 1984, pp. 184-197.

[Moon 78] Moon, David, *MacLisp Reference Manual*, Version 0, Technial Report, MIT Laboratory for Computer Science, 1978

[Moon 84] Moon, David, "Garbage Collection in a Large Lisp System," In Proceedings of the 1984 ACM symposium on Lisp and Functional Programming. August 1984.

[Moore 76] Moore, J. Strother, "The Interlisp Virtual Machine Specification," Xerox PARC, CSL-76-5, 1976.

[Rees 82] Rees, Jonathan A. and Norman I. Adams IV. "T: A Dialect of Lisp or, Lambda: The Ultimate Software Tool," In Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming. August 1982.

[Roads 83] Roads, Curtis B., *3600 Technical Summary*, Symbolics, Cambridge, Mass. , Feb. 1983

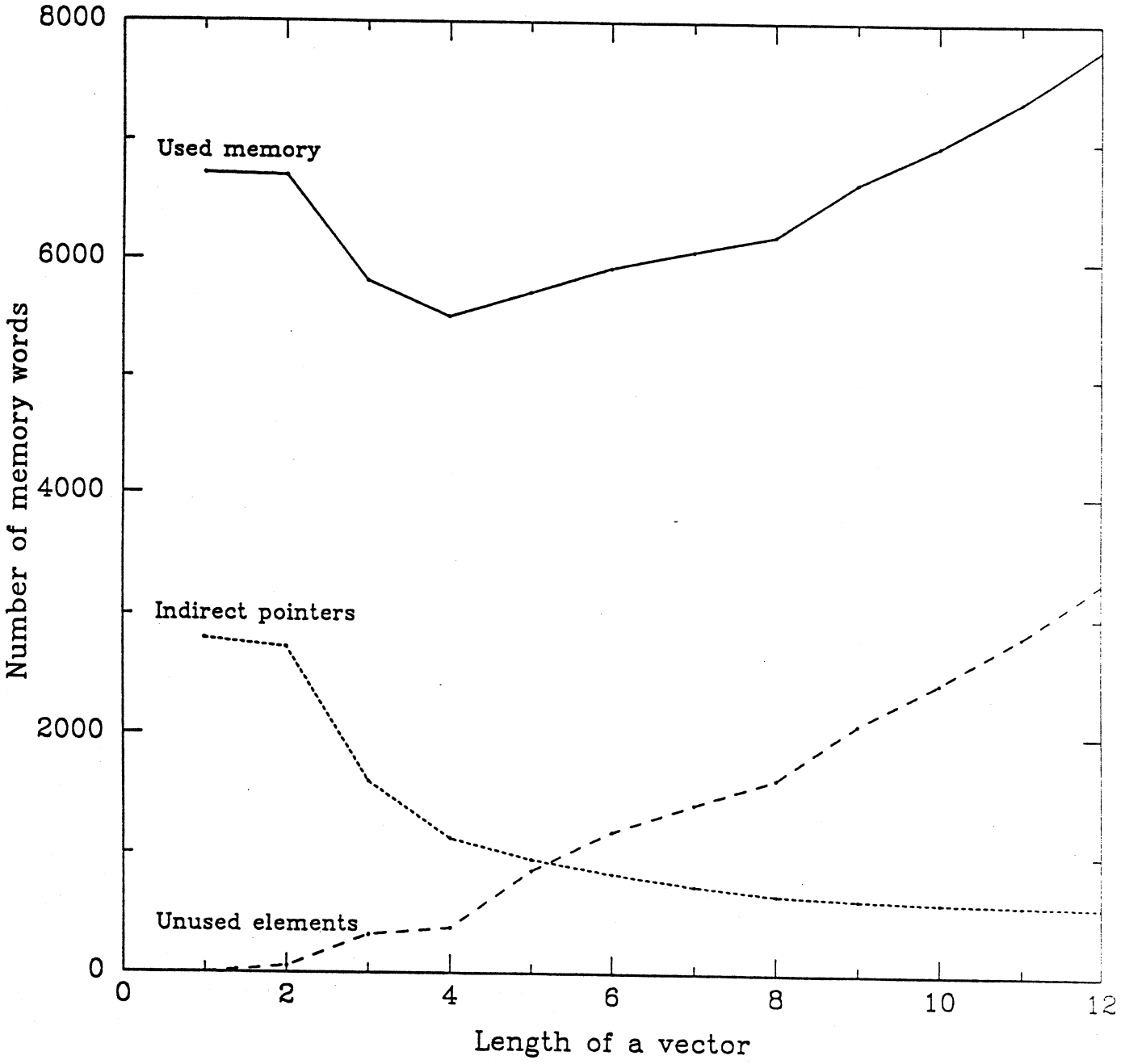
[Steele 83] Steele, G. L. , *Common Lisp Reference Manual*, Carnegie-Mellon Univerisity, Computer Science Department, 1983

[Steele 80] Steele, G. L. and Sussman, G. J. , "Design of a Lisp-Based Microprocessor," CACM Vol. 23, No. 11, Nov. 1980. pp. 628-645

[Sussman 75] Sussman, Gerald Jay and Guy Lewis Steele, "Scheme: An Interpreter for Extended Lambda Calculus," MIT AI Lab Memo 349, Dec. 1975.

[Turner 76] Turner, D. A. , "SASL Language Manual," University of St. Andrews Technical Report, 1976.

[Weinreb 81] Weinreb, Daniel and David Moon, "Lisp Machine Manual," MIT AI lab, July 1981



Vector length vs. memory usage of a parallel merge sort