A New List Compaction Method

KAI LI AND PAUL HUDAK

Department of Computer Science, Yale University, New Haven, CT 06520, U.S.A.

SUMMARY

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, in which lists are represented explicitly as linked vectors rather than implicitly as compacted memory. In addition, an encoding scheme is used that is as simple or simpler than all known encodings, and destructive operations are supported with no greater overhead than existing schemes. Performance figures in a simulated environment suggest that the strategy consistently performs better than conventional cdr-coding, with essentially the same complexity.

KEY WORDS List structure List compaction Memory management LISP programming Shared-memory parallel computing

INTRODUCTION

Programming languages relying heavily on the list data structure have become very popular in recent years. For example, various dialects of Lisp, including Scheme, Common Lisp, and others; functional languages such as ALFL, SASL, and ML; and logic programming languages such as Prolog, are playing an ever-increasing role in artificial intelligence, systems programming and many other fields. In addition, dataflow languages, variations of functional and logic programming languages, and new dialects of Lisp 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, and together with empirical data have been used in designing special-purpose hardware to support Lisp implementations. These approaches are summarized in the next section. However, deficiencies in these schemes have led us to consider an alternative list

0038-0644/86/020145-19$05.90
© 1986 by John Wiley & Sons, Ltd.

Received 7 January 1985
Revised 10 June 1985
compaction strategy that can be viewed as a generalization of cdr-coding, in which lists are represented as linked vectors. After presenting the basics of our new method, we discuss further improvements and implementation issues, and present performance results for some sequential and parallel program simulations.

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. Although these empirical data are now almost ten years old, there is good reason to believe that programmers still tend to build lists by placing an element in the car of a cell, with the cdr pointing to the rest of the list, i.e. another cons cell. The main idea behind cdr-coding is to take advantage of this situation by arranging for that second cons cell to directly follow the car of the first, and to encode that information in a few extra bits contained in the first cell; thus the first cell does not need a cdr field at all. 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 for lists.

There are three prominent methods of cdr-coding, which we refer to as the MIT, Xerox and hybrid methods. In the MIT method (used in the MIT Lisp machine\textsuperscript{21,22}), the car field of a cons cell has full addressing capability and two extra bits are used to indicate different cdr types, as shown in Table I. 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 is needed to accommodate destructive operations. The coding technique used in the Symbolics 3600\textsuperscript{23} is similar, but does not have the indirection code, since it has a transparent addressing strategy built into its memory addressing mechanism.

<table>
<thead>
<tr>
<th>cdr-coding bits</th>
<th>meaning</th>
<th>storage/cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>cdr is NIL</td>
<td>1 word</td>
</tr>
<tr>
<td>01</td>
<td>cdr begins in next word</td>
<td>1 word</td>
</tr>
<tr>
<td>10</td>
<td>cdr is contained in next word</td>
<td>2 words</td>
</tr>
<tr>
<td>11</td>
<td>current cell contains indirect pointer</td>
<td>2 or 3 words</td>
</tr>
</tbody>
</table>

In the Xerox method of cdr-coding, the car field of a cons cell also has full addressing capability, but the cdr is constrained to a smaller address, which is interpreted as an offset from the current location. In this scheme the code 0 refers to NIL, and the code 128 means that the current cell contains an indirect pointer to the 'real' pair (these correspond to the MIT cdr-codes 00 and 11, respectively). Additionally, a value from 1 to 127 denotes an offset from the current cell to a cell that contains the cdr, and a value
from 129 to 255 denotes an offset to a cell that contains a pointer to the cdr. This can be viewed as a generalization of the MIT method, since the MIT strategy only allows encoding the 'next' location (i.e. an offset of one) as the cdr.

<table>
<thead>
<tr>
<th>cdr-coding bits</th>
<th>meaning</th>
<th>storage/cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>cdr is NIL</td>
<td>1 word</td>
</tr>
<tr>
<td>1 to 127</td>
<td>offset of cdr</td>
<td>1 word</td>
</tr>
<tr>
<td>128</td>
<td>current cell contains indirect pointer</td>
<td>2 or 3 words</td>
</tr>
<tr>
<td>129 to 255</td>
<td>offset of indirect cdr</td>
<td>2 words</td>
</tr>
</tbody>
</table>

The Xerox method as implemented on the Xerox Lisp machine uses 8 bits for the cdr code, as shown in Table II. Despite the large number of cdr-coding bits, in practice this method 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 a 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. The range of such a value is small, and the need arises for efficient ways to implement 'boxing' and 'unboxing'.

The third cdr-coding strategy, the hybrid method, is to use limited page offsets for both the car and cdr fields. Under proper circumstances this method can save more storage then either of the other two. On the other hand, it requires the additional complexity of an 'escape' mechanism for both car and cdr (to reference cells that are too far away to encode as an offset), and list copying and garbage collection algorithms that rely on forwarding pointers become difficult to use since the method has no convenient way to store them.

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 behaviour 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. (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 program written in T (a dialect of Scheme), extended with a
`parallel let' construct that evaluates the defined names in parallel:

```scheme
(define (par-merge-sort l)
  (cond ((null? (cdr l)) 1)
        (else
         (par-let ((left (sort (left-half l)))
                    (right (sort (right-half l))))
                   (merge left right)))
)
```

In this example, we assume that there are two processors available. Each will sort half of the list by some sequential procedure called SORT, and then one of the processors will merge the results. For long lists, this program will have a speed-up factor of almost 2. If the sequential sort is purely functional, then each processor can be expected to make at least \(2^{\log_2 N}\) cell allocations, where \(N\) is the length of the list. If we assume that the two processors run at approximately the same speed, then most of the allocations will be interleaved.

One way to partially solve this problem is to give each processor its own heap, allocated in large blocks from some common memory resource. This requires the additional complexity of maintaining separate heaps, but more seriously it does not solve the problem of several processors consing elements to the same list. This would happen, for example, in a parallel system where processors are dynamically assigned tasks as the need arises. It is easy to argue that in such a situation each list should have its own heap, not each processor. That is essentially what our method accomplishes, as explained below.

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

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

The loop here is a widely-used macro\(^{26}\) whose meaning should be clear. (Alternatively, the loop could be expressed as a recursion, passing the updated lists explicitly as arguments in the recursive call.) As an example of a case where this situation occurs in practice, consider the subfunction of quicksort that splits a list into two, one containing
all elements less than a certain value (the 'pivot'), the other containing all other elements:

\[
\text{define (split l pivot)} \\
\text{(let ( (low []) )} \\
\text{(high []) )} \\
\text{(loop (while (not (null? l))))} \\
\text{(do (let ( (x (car l))) )} \\
\text{(if (< x pivot) } \\
\text{(set low (cons x low))} \\
\text{(set high (cons x high)) } ) \\
\text{(set l (cdr l)))} \\
\text{(result (list low high)) ) )
\]

Assuming relative random distribution of the element values, 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 that results for lists created during parallel consing generally degrades the performance of list compaction methods significantly. Indeed, with perfect interleaving the MIT method produces no compaction at all! Some compaction will be realized with the other two strategies, but at the expense of a larger number of bits for encoding, 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 lists can be 'linearized' by a special pass over the heap (which also may happen as a result of garbage collection), but this introduced considerably more work.

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 exercise in estimating how common this is, we scanned a rather large program (the Bulldog compiler for very-long-word instruction architectures,\textsuperscript{27–29} consisting of over 30,000 lines of Elisp code). We found many occurrences of parallel consing, in particular loops having the following pattern:

\[
\text{(loop (for x in l)} \\
\text{(do (if (some-property x) } \\
\text{(then (push 11 x) } \\
\text{... ) )} \\
\text{(else (push 12 x) } \\
\text{... ) ) ) )}
\]
where (PUSH 1 ×) is essentially equivalent to (SET 1(CONS × 1)). We show in the simulation section that significant performance variations take place with this kind of loop, depending on the cdr-coding technique used. If sequential programs such as this tend to have non-trivial amounts of parallel consing, we can expect that parallel programs will have even more.

A NEW LIST DATA STRUCTURE AND COMPACTI ON STRATEGY

To summarize our goals, we want a representation for lists that
(a) allows significant comparison of long lists.
(b) experiences minimal degradation during parallel consing
(c) allows the full range of list operations, including destructive ones, without loss
   (and it is hoped, a gain) in efficiency
(d) requires minimal encoding space, providing maximal addressability.

Our solution is to represent lists as linked vectors. This idea first appeared in Reference 16, but the strategy proposed there 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 method is a variant of that of Reference 16, which supports destructive operations.

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.

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 below.

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 $v^i_j$ is generally the list starting at $v^i_{j+1}$, whereas the cdr of $v^i_{k-1}$ is contained in $v^i_k$. (Destructive operations change these relationships somewhat, as described below.) Asking what is the cdr of $v^i_k$ is the same as asking what is the cdr of the object it points to — if $v^i_k$ 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. In this case $v^i$ is the $i$th cons cell of the list, where the ‘left’ cell of $v^i$ (i.e. $v^i_1$) contains the $i$th car of the list,
and the 'right' cell \((v_j)\) contains the \(i\)th cdr. In general the lengths of the vectors can vary. For example, the list \((A \ B \ C \ (D \ E) \ F \ G)\) may be constructed using variable-length vectors as 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.

![Figure 1. The representation of list \(L = (A \ B \ C \ (D \ E) \ F \ G)\)](image)

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 \(\text{contents}(l)\).

**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 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 behaviour more formally:

```scheme
(define (CONS x y)
  (if (\text{\textbf{\textit{y}} is a list pointer and \(y-1\) is unused})
      (\text{\textbf{\textit{put \(x\) into cell \(y-1:\) return \(y-1\)}}}
      (\text{\textbf{\textit{allocate a new vector \(u\) of length \(k\):}}}
                  \text{\textbf{\textit{\(u\) in cell \(u_{k-1:\) put \(y\) in cell \(u_k\) (i.e., the indirection cell):}}}
                                 \text{\textbf{\textit{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 it is easy to maintain the invariant that 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. We describe below 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:

\[
\begin{align*}
\text{(define (CAR x))} \\
\quad \text{IF } <x \text{ is not a list} \text{ THEN ERROR} \\
\quad \text{ELSE IF } <x \text{ is an indirection cell} > \\
\quad \quad \text{THEN (CAR contents(x))} \\
\quad \quad \text{ELSE contents(x) )}
\end{align*}
\]

\[
\begin{align*}
\text{(define (CDR x))} \\
\quad \text{IF } <x \text{ is not a list} \text{ THEN ERROR} \\
\quad \text{ELSE IF } <x \text{ is an indirection cell} > \\
\quad \quad \text{THEN (CDR contents(x))} \\
\quad \quad \text{ELSE IF } <x+1 \text{ is an indirection cell} > \\
\quad \quad \quad \text{THEN contents(x+1)} \\
\quad \quad \quad \text{ELSE x+1 )}
\end{align*}
\]

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. (SET-CAR x y) replaces the car of x with y, and is defined by

\[
\begin{align*}
\text{(define (SET-CAR x y))} \\
\quad \text{IF } <x \text{ is not a list} \text{ THEN ERROR} \\
\quad \text{ELSE IF } <x \text{ is an indirection cell} > \\
\quad \quad \text{THEN (SET-CAR contents(x) y)} \\
\quad \quad \text{ELSE <change contents(x) to y; return y > )}
\end{align*}
\]

The destructive operation SET-CDR changes the data dependencies between elements in a normal list, and is thus a bit more complex. We need to consider two cases, which we call 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 \(u_{k-1}\), 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 \(U\) (i.e. \(u_k\)) to the new value
we are setting L's cdr to. For example, (SET-CDR L '(XY)) 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:

\[
\text{(define (SET-CDR x y)}
\begin{align*}
&\quad\text{IF } \langle x \text{ is not a list}\rangle \text{ THEN ERROR} \\
&\quad\text{ELSE IF } \langle x \text{ is an indirection cell}\rangle \\
&\quad\quad\text{THEN (SET-CDR contents(x) y)} \\
&\quad\text{ELSE IF } \langle x+1 \text{ is an indirection cell and last cell in the vector}\rangle \\
&\quad\quad\text{THEN put y in x+1} \\
&\quad\quad\text{ELSE temp := (CONS contents(x) y)}; \\
&\quad\quad\quad\text{put temp in x}; \\
&\quad\quad\quad\text{make x an indirection cell}; \\
&\quad\text{return y} \rangle
\end{align*}
\]

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.

**Further opportunities for compaction**

Empirical studies have also demonstrated that NIL is the most common element in a list,$^{17,18}$ so it might be worth while to optimize such occurrences as is done in
conventional cdr-coding. We call such an optimization *cdr-nil compaction*, which can be implemented by extending our encoding strategy to include a representation 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 represents a content cell whose cdr is NIL. Ways to represent this information explicitly are discussed in the next section.

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 globally by providing a procedure `SET-CONS-VECTOR-SIZE` that takes an integer argument, or locally by letting the CONS function take an optional argument indicating the length of any newly allocated vector. 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.

**Figure 4.** The list $L = (A B C (D E) F G)$ represented using cdr-nil compaction

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 a significant optimization is available when using the variable-length method, in that if two vectors of a list are contiguous, they can be merged into one. This situation happens often; in particular 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 Uv, and v is the last vector allocated from the heap>
  THEN <extend v by k cells, so that v is now Uv+1; put x in Uv; return a pointer to Uv>
  ELSE <allocate a new vector U of max(k,2); put x in cell Uv-1; put y in cell Uv (i.e., the indirection cell); return a pointer to Uv-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 functionally equivalent to the MIT cdr-coding method.

IMPLEMEN\-TATION ISSUES

Encoding strategies

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

(i) is an indirection cell or a content cell
(ii) is the last cell of a vector
(iii) contains the unused value \# 
(iv) 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 overall requirements of a particular implementation will dictate the final strategy used.

In most LISP systems, values have a type-tag indicating their type. One 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 possibilities also exist for indicating whether or not a cell is the last cell in a vector. If a fixed-length \( k = 2 \) is used for all vectors, it is not necessary to explicitly mark which cell is the first or last, as long as we allocate vectors on even \( k \)-word boundaries (often called \( k \)-word alignment). 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 variable-length vectors are used, an extra bit is needed 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 III.

Note that this encoding is as compact as, and very similar to, the MIT encoding shown in Table I.
Table III. Concise encoding for linked-vectors

<table>
<thead>
<tr>
<th>coding-bits</th>
<th>meaning</th>
<th>storage-cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>the cdr is NIL</td>
<td>1 word</td>
</tr>
<tr>
<td>01</td>
<td>the cdr begins in next cell</td>
<td>1 word</td>
</tr>
<tr>
<td>10</td>
<td>the current cell is an indirection cell</td>
<td>2 words</td>
</tr>
<tr>
<td>11</td>
<td>the current cell is unused</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Garbage collection

Garbage collection algorithms based on copying between two semi-spaces (such as Baker's algorithm, for 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 percent of list cells have just one reference to them, this optimization will eliminate most of the unnecessary vector allocations and resultant indirect pointers.

Implementing other destructive list operations

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

Implementing APPEND! by SET-CDR is fairly easy and very efficient, with or without cdr-nil compaction. Without 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 earlier.

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 would be 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.

Using our method without cdr-nil compaction, the algorithm is rather simple:

(define (REVERSE! L)
  1. Let L consist of n vectors v\(^1\) to v\(^n\), each vector v\(^i\) having (variable) length k\(i\).
  2. For each vector v\(^i\) having form:
     v\(^i\) = (v\(_{i1}\), v\(_{i2}\), ..., v\(_{ik}\_1\), v\(_{ik}\))
     reverse its content elements destructively, yielding
     v\(^i\) = (v\(_{ik}\_1\), ..., v\(_{i2}\), v\(_{i1}\), v\(_{ik}\))
  3. Reverse the pointers linking the vectors, so that
     v\(^i\) = (v\(_{ik}\_1\), ..., v\(_{i2}\), v\(_{i1}\), NIL)
     and for i = 2, ..., n
     v\(^i\) = (v\(_{i1}\), ..., v\(_{i2}\), v\(_{i1}\), pointer to v\(^i-1\))
  4. 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 neighbouring 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 indirection cell was freed up by the cdr-nil encoding. The resulting algorithm is as follows:

(define (REVERSE! L)
  1. Let L consist of n vectors v\(^1\) to v\(^n\), each vector v\(^i\) having (variable) length k\(i\).
  2. Reverse the elements in v\(^1\) and shift in v\(^2\), such that:
     v\(^1\) = (v\(_{11}\), v\(_{12}\), v\(_{21}\))
     and set the cdr-nil bit of the last cell.
  3. For i = 2, ..., n - 1, reverse each vector v\(^i\) and shift one element from v\(^i+1\), such that
     v\(^i\) = (v\(_{i1}\), v\(_{i2}\), ..., v\(_{ik}\_1\), pointer to v\(^i-1\))
  4. Reverse v\(^n\) such that
     v\(^n\) = (v\(_{n1}\), ..., v\(_{nk}\), v\(_{n1}\), pointer to v\(^n-1\))
     and make the last cell an indirection cell.
  5. Return a pointer to v\(^n\).

This algorithm can also be implemented in a single pass.

SIMULATION RESULTS

Most of the algorithms and data structures described in this paper have been simulated on real programs written in T. This was accomplished by modifying a compiler for T\(^3\) so that it produced new closed code for the primitive operations CAR, CDR, CONS, SET-CAR and SET-CDR (and also involved recompiling 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 run, 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, its correctness was preserved even during garbage collection. Overall this was a non-trivial effort, but was worth while 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

Table IV. Memory usage when copying two lists in parallel

<table>
<thead>
<tr>
<th>Vector length</th>
<th>Memory used</th>
<th>Unused elements</th>
<th>Indirect pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4002</td>
<td>0</td>
<td>2000</td>
</tr>
<tr>
<td>2</td>
<td>4000</td>
<td>0</td>
<td>1998</td>
</tr>
<tr>
<td>3</td>
<td>3000</td>
<td>0</td>
<td>998</td>
</tr>
<tr>
<td>4</td>
<td>2672</td>
<td>4</td>
<td>666</td>
</tr>
<tr>
<td>5</td>
<td>2500</td>
<td>0</td>
<td>498</td>
</tr>
<tr>
<td>6</td>
<td>2400</td>
<td>0</td>
<td>389</td>
</tr>
<tr>
<td>7</td>
<td>2338</td>
<td>4</td>
<td>332</td>
</tr>
<tr>
<td>8</td>
<td>2288</td>
<td>2</td>
<td>284</td>
</tr>
<tr>
<td>9</td>
<td>2250</td>
<td>0</td>
<td>248</td>
</tr>
<tr>
<td>10</td>
<td>2240</td>
<td>13</td>
<td>222</td>
</tr>
<tr>
<td>11</td>
<td>2200</td>
<td>0</td>
<td>198</td>
</tr>
<tr>
<td>12</td>
<td>2184</td>
<td>2</td>
<td>180</td>
</tr>
</tbody>
</table>
A NEW LIST COMPACTION METHOD

'these 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 IV and Figure 5 show the results when copying a list of 1001 elements on each processor, simulated by perfectly interleaving the cons traces.

Several interesting (although quite predictable) things are worth noting. First, with vector length one we are essentially simulating the MIT cdr-coding method—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 should asymptotically 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 earlier, where we assume that the sequential procedure 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.

The resulting cons traces for PAR-MERGE-SORT were combined as shown in Figure 6. The trace for the initial split and final merge (which are both very short) remain sequential, whereas the parallel executions of sort were simulated by interleaving the

![Figure 6. Consing trace graph of a parallel merge sort](image-url)
Table V. Memory usage of a parallel merge sort

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

Figure 7. Graph of Table V

cons traces randomly. We ran the PAR-MERGE-SORT program on a large existing database file. The results are shown in Table V and Figure 7.

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 method) only requires 6713 words, an immediate improvement of 16 per cent. As the vector length increases beyond that, the improvement also increases, reaching a maximum of
Table VI. Memory usage of a parallel consing loop pattern

<table>
<thead>
<tr>
<th>Vector length</th>
<th>Memory used</th>
<th>Unused elements</th>
<th>Indirect pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3252</td>
<td>0</td>
<td>1252</td>
</tr>
<tr>
<td>2</td>
<td>3251</td>
<td>1</td>
<td>1251</td>
</tr>
<tr>
<td>3</td>
<td>3003</td>
<td>1</td>
<td>1001</td>
</tr>
<tr>
<td>4</td>
<td>2668</td>
<td>3</td>
<td>667</td>
</tr>
<tr>
<td>5</td>
<td>2505</td>
<td>5</td>
<td>501</td>
</tr>
<tr>
<td>6</td>
<td>2406</td>
<td>5</td>
<td>401</td>
</tr>
<tr>
<td>7</td>
<td>2338</td>
<td>5</td>
<td>334</td>
</tr>
<tr>
<td>8</td>
<td>2288</td>
<td>2</td>
<td>286</td>
</tr>
<tr>
<td>9</td>
<td>2259</td>
<td>8</td>
<td>251</td>
</tr>
<tr>
<td>10</td>
<td>2240</td>
<td>17</td>
<td>224</td>
</tr>
<tr>
<td>11</td>
<td>2211</td>
<td>10</td>
<td>201</td>
</tr>
<tr>
<td>12</td>
<td>2016</td>
<td>2</td>
<td>182</td>
</tr>
</tbody>
</table>

31 per cent 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 method.
A final solution example is based on the abstraction given earlier of the loop pattern found in the Bulldog compiler.\textsuperscript{27, 29} Table VI and Figure 8 show the results of running this program fragment on a list of 2001 elements whose 'properties' were chosen at random. Not surprisingly, the results are similar to the first simulation example. Vector length 1 (functionally equivalent to MIT cdr-coding) saves about 19 per cent of storage. As the vector length increases, the savings approach 50 per cent.

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 method, with no greater implementation complexity. It is more difficult to compare to the Xerox or hybrid methods, since they are in some sense only 'pseudo-compaction' schemes in that they use a relatively large cdr-code that acts as sort of a 'local address' for nearby elements. Indeed, such a technique is not incompatible with our scheme, whose encoding could easily be extended to include an offset field. An open empirical problem is determining for our scheme the right default vector length to ensure maximum average compaction.

An alternative 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.

ACKNOWLEDGEMENTS

We wish to thank John Ellis, Norm Adams and Jonathan Rees for their invaluable help in the simulation work. Without the latter two, the simulations using T would not have been possible. Also thanks to Chris Riesbeck and Jim 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 Grants MCS-8302018 and DCR-8106181.

REFERENCES

A NEW LIST COMPACTION METHOD