Comparison of Compacting Algorithms for Garbage Collection

JACQUES COHEN and ALEXANDRU NICOLAU
Brandeis University

The relative efficiencies of four compactors of varisized cells are estimated by constructing their time-formulas. These are symbolic formulas expressing execution times as functions of the time to perform common, elementary operations such as assignment, addition, subscripting, and loop overhead. By binding the variables to numeric values corresponding to a specific machine one can estimate program execution times without resorting to empirical tests. The first of the compactors (Lisp 2) requires additional storage for pointer readjustment. The second (based on the work of Haddon and Waite) attempts to reduce these storage requirements at the expense of processing time. The last two (Morris' and Jonkers') are recently proposed compactors that require minimal additional storage and that update pointers by first threading them into linear lists. The paper provides unified descriptions of the algorithms and presents curves expressing the relative efficiencies of the compactors when run on a specific machine (PDP-10). It is straightforward to modify the given formulas to estimate compactors' efficiencies when run on other computers.

Categories and Subject Descriptors: D.2.8 [**Software Engineering**]: Metrics; D.4.2 [**Operating Systems**]: Storage Management; E.1 [**Data**]: Data Structures; E.2 [**Data**]: Data Storage Representations

General Terms: Algorithms, Measurement

Additional Key Words and Phrases: Garbage collection, compaction, varisized cells, storage management, pointer readjustment, time-formulas

1. INTRODUCTION

Garbage collection is a term that denotes the process of reclaiming unused storage. Methods for garbage collection usually comprise two separate phases:

(1) identifying the storage that may be reclaimed;
(2) incorporating this reclaimable storage into the memory area that can be made available to the user.

Phase (1) is usually performed by keeping a list of immediately accessible cells and following the links contained in them to trace and mark every accessible cell. This method of identification is usually called *marking*. Phase (2) can be subdi-

This work was supported by the National Science Foundation under grant MCS 79-05522.
Authors' addresses: J. Cohen, Computer Science Program, Ford Hall, Brandeis University, Waltham, MA 02254; A. Nicolau, Computer Science Department, Yale University, New Haven, CT 06520.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1983 ACM 0164-0925/83/0700-0532 $00.75

vided into two classes:

(2a) incorporation into a free list in which available cells are linked by pointers;
(2b) compaction of all used cells into one end of the memory, the other end containing contiguous words that are made available to the allocator.

Knuth [8] and Cohen [2] describe several algorithms for performing these two phases of garbage collection.

In this paper we present detailed analyses of four algorithms that perform Phase (2b) of garbage collection. The first algorithm, referred to as Lisp 2 [8, pp. 602–603], is a classical one. The second is an improved version of another classical compactor also known as the "rolling table" compactor [5, 6, 13]. The last two, by Morris [11, 12] and Jonkers [7], have been proposed only recently. Our analyses consist of constructing the time-formulas [1, 3] for these compacting algorithms. Time-formulas are symbolic formulas that express execution times as functions of variables representing the time needed to perform common, elementary operations (e.g., addition, assignment, subscripting, loop overhead). By binding the variables to numeric values corresponding to a specific machine, one can estimate program execution times without resorting to empirical tests.

All four of the selected algorithms can compact varisized cells, that is, cells of different sizes. The following storage requirements should be kept in mind when comparing the four compactors:

Lisp 2	One extra field per cell is required to store information for pointer readjustment. The content of this field is an address.
Table Compactor	In principle, no additional storage is required since the information for readjusting pointers can be stored within garbage cells. This information consists of a table of records, each being able to hold two addresses. An additional table may be used to speed up pointer readjustment.
Morris	Assuming that cells may contain pointers and data, two bits per field are needed to identify pointers, swapped pointers, data, and garbage cells. If each field has its own mark bit, only an additional bit is needed.
Jonkers	One bit per cell is required to recognize if the space occupied by data has been previously occupied by a pointer. This presupposes that the space needed to store the data of a cell is also capable of holding a pointer.

Although the four chosen algorithms are (essentially) linear, only a microanalysis, as presented here, can reveal their relative efficiencies on specific machines.

In the following two sections we present unified descriptions of the algorithms and show how to derive their time-formulas. In Section 4 we compare the performance of the algorithms by considering a specific computer (PDP-10). However, the data provided by the time-formulas enable the reader to estimate the performance on various other machines.

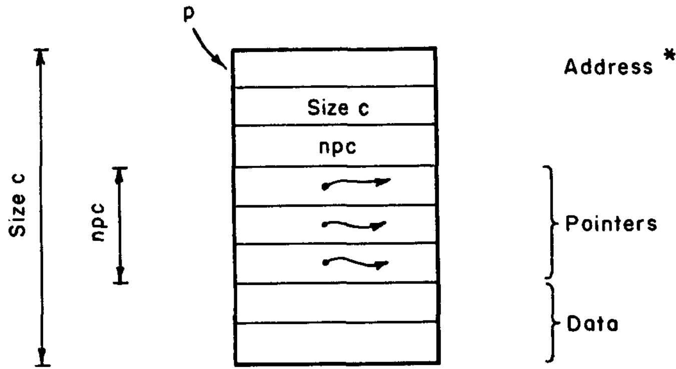

Fig. 1. Assumed layout of a cell.

2. DESCRIPTION OF THE ALGORITHMS

A cell is a number (≥1) of contiguous computer words that can be made available to the user. For description purposes let us consider that the varisized cells have the configuration depicted in Figure 1. The first field of the cell is large enough to store an address and is only required in the Lisp 2 compactor. The second and third fields contain (1) the size of the cell and (2) the number, NPC, of fields containing pointers. It is assumed that these pointers immediately follow the field storing NPC and that they reference the first element of a cell. The data of the cell are placed following the pointers.

It would be straightforward to modify our descriptions and analyses of the compactors if the information about the size of a cell and its pointers were stored in a different manner.

Each algorithm compacts an area of the memory M between two addresses, START and LAST (START < LAST). This area contains a number (≥1) of cells, which may be either active or garbage cells. An active cell is one that can be accessed by a program through a set of initial pointers. In contrast, garbage (also called inactive) cells cannot be accessed through that set. All four compactors are of the *sliding* type; that is, they move the active cells toward one end of the memory.

The first phase of collection marks each active cell by setting a special bit on. (Initially, mark bits are off.) The mark bits of the garbage cells remain off. Upon completing its task, the compactor turns off the mark bits of the active cells to prepare for a subsequent collection.

The marking (or tagging) fields of a cell vary according to the compactor. In Lisp 2 the first word of a cell (see Figure 1) contains a nonnil pointer if the cell is active and a nil pointer otherwise.

Before we proceed to the presentation of the compactors, it is helpful to describe the common data used in evaluating them. Table I presents the parameters used in time-formulas to indicate the number of cells (classified according

Table I. Loop Data Applicable to the Four Compactors

NC	Total number of cells between START and LAST (NC = NMC + NGC)
NMC	Number of marked (i.e., active) cells between START and LAST
NGC	Number of garbage cells between START and LAST
NGB	Number of garbage blocks; a garbage block is a sequence of adjacent garbage cells
NMB	Number of marked blocks
NPC	Number of pointers per cell (NPC = NNP + NAPC)
NNP	Number of nil pointer fields per cell
NAP	Number of active pointers (i.e. nonnil ones) including the initial pointers
NAPC	Number of nonnil pointers per cell
NIP	Number of initial pointers
NIAP	Number of initial nonnil pointers
NFP	Number of forward pointers plus the number of initial pointers
NBP	Number of backward pointers
NSP	Number of pointers referencing their own cell
NIPF	Number of initial pointers outside the area to be compacted
SIZEC	Number of words in a cell

Note. Average values of these quantities are actually used in the time-formula computations.

Table II. Time-Variables and Their Bindings

Time-variable	PDP-10 bindings (μs)
ADDITION	0.61
ASSIGN	1.21
AND	0.63
COND	1.30
OR	0.63
FOROH	2.02
IFOH	1.30
MULT	2.69
NEGATION	0.20
REPEATOH	0.93
SUB1	2.27
SUBTRACT	0.77
WHILEOH	0.92

to their types), the number of pointers, and related information. Time formulas are expressed as a function of these parameters and of the time-variables indicating the time to perform the basic operations common to all the compactors. A list of these operations and their bindings to a specific computer are presented in Table II. SUB1, for example, is the time needed to subscript a one-dimensional array; COND denotes the time to test a conditional expression (such as $a > b$ or $d \neq a$). The time-variables ending in OH denote the time overhead needed to execute the corresponding operation.

We have chosen to provide detailed program versions of only the first and last compactors. A careful examination of the time-formula for the first compactor will enable the reader to reconstruct the time-formulas for the others.[1] The last

[1] The detailed programs for all four compactors and their time-formulas are available from the authors.

compactor is also presented in detail since it is one of the most likely to be used in cases where memory is scarce.

2.1 Lisp 2

The algorithm as described by Knuth [8, pp. 602–603] uses three linear passes of the memory space. The first pass has two objectives:

(1) To combine all adjacent garbage cells into one garbage cell; although this step is not strictly necessary, it may speed up the subsequent passes.
(2) To compute the new address of each active cell; this new address is the sum of the sizes of the preceding active cells, and it is stored in the first field of each active cell as it is encountered (see Figure 1).

The second pass simply updates the pointer fields of each active cell so that they point to the new address where the cell will be relocated. Finally, the third pass relocates the active cells toward the lowest address part of the memory. This pass also resets to nil the link field of each active cell. As mentioned earlier, this field is used in the marking phase of each collection.

Figure 2 is a program that implements the Lisp 2 algorithm. Our purpose in presenting this straightforward algorithm is to illustrate the generation of the time-formula. The program is presented in a PASCAL-like language.[2] It contains comments indicating the number of times each of its branches and loops are executed. This information, which is complemented by Table I, is needed to derive the time-formula for the algorithm.

The snapshots shown in Figure 3 illustrate the action of each pass. To improve clarity, only the pointers to one of the cells, C, are shown in the figure. The sole initial pointer (root) also references C, and it lies in a word outside the area being compacted. There are, of course, pointers emanating from C and linking all the marked cells shown in the shapshots.

2.2 Table Compactors

The table in this class of compactors [5, 6, 13] is used for storing data needed for pointer readjustment. These compactors usually operate in three phases:

1. Computation of readjustment data and its storage within the inactive cells ("holes"). This scan also does the compacting and involves the following steps:
   a. determine the next hole, its initial address a_i, and its size S_i;
   b. compute the ith pair (a_i, Sum + S_i); Sum is initialized to zero, and it is updated by the assignment Sum := Sum + S_i;
   c. add the above pair to a (break) table BT that is stored within the holes and that may need to be "rolled" to make room for the cells being compacted. This rolling alters the order in which pairs appear in the table;[3]
   d. compact; that is, move the cell adjacent to the hole toward the address a_i.

[2] This has been done to simplify the presentation. The actual programs were tested using a PASCAL compiler.

[3] It can be shown [6] that there is always room in the holes for storing the BT.

2. Sorting of the pairs in the BT table according to increasing values of a. This is needed to speedup Phase 3 below.
3. Pointer readjustment. Each element in the compacted area is scanned, and each pointer p is adjusted as follows:

   a. search through the BT table and determine the adjacent pairs (a, S) and (a', S') such that $a \leq p < a'$;
   b. the readjusted value of p is $p - S$.

Phase 1 is linear; one of its time-consuming operations is the copying and rolling of the table as explained in Phase 1c.

Let n be the number of blocks of active cells. The worst case complexity of Phase 2 occurs when active cells alternate with inactive cells. In that case the BT table contains n entries, and its sorting has complexity $n \log n$.

Phase 3 also has complexity $n \log n$ since a binary search is needed to readjust each pointer by consulting the already sorted BT table.

Although the complexity of the last two phases is theoretically $n \log n$, the results presented by Fitch and Norman [5] and confirmed by our analysis show that in most practical situations the compactor behaves linearly.

Our version of the table compactor is that suggested by Fitch and Norman [5] and is briefly described as follows:

A table H (of size $h = 2^p$) containing pointers to BT is constructed just after Phase 2 and stored in the reclaimed space together with BT. This table is used as a hash table providing fast access to BT. When a pointer is redjusted in Phase 3, the p most significant bits are used to look up the start and end addresses of a region in BT containing the values of a and a' needed for readjustment (see Phase 3a). Fitch and Norman suggest that h be roughly twice the size of BT when possible.

Microanalysis of the table compactor enabled us to estimate the importance of the sorting required in Phase 2. An initial analysis was made assuming that sorting would take γ NMB log NMB time units, in which γ is a constant depending on the sorting algorithm and on the computer characteristics and NMB is the number of marked blocks, that is, the number of entries in BT. We then determined experimentally that value of γ for Quicksort running on the PDP-10, the numbers to be sorted being randomly generated. To our surprise, the results indicated that the time-formula for the table compactor was not only strongly nonlinear but also yielded compacting times above those of the other compactors. (Our results indicated that sorting required about 30 to 40 percent of the compactor's time.) This was contrary to the almost linear results obtained by Fitch and Norman [5]. Our findings led to a more careful and realistic analysis, which in turn suggested the proper algorithm for sorting BT.

Note that, every time the BT table is rolled, the order of its contents is changed, and additional effort is needed to sort BT in Phase 2. Therefore, the appropriate strategy in Phase 1 is to avoid rolling the BT table unless this is strictly necessary.

We now estimate the size of the portion of the BT that must be sorted, that is, the index S such that sorting is only needed for elements BT[1] through BT[S].

```
procedure COMPACT_LISP2;
(* compacts used cells towards lowest address *)
begin

(* first pass: combine adjacent garbage cells and
   calculate new address of used cells *)
   P:=START;
   NEW_ADDRESS:=START;
   while P <> NIL do
      begin
         (* the loop header is executed (NGB+NMC+1) times, and
            the body (NGB+NMC) times *)
         if MARKED(P) then
            begin
               (* this branch is executed (NMC) times *)
               LINK(P):=NEW_ADDRESS;
               NEW_ADDRESS:=NEW_ADDRESS+SIZE(P);
            end;
         else
             while not MARKED(NEXT(P)) do
                 (* the loop header is executed (NGC) times, and
                    the body (NGC-NGB) times *)
                 COMBINE(P)
         P:= NEXT(P)
      end;   (* end while loop *)

   UPDATE_ROOTS;   (* updates initial pointers *)

(* second pass: update all pointer fields in each used cell *)
   P:=START;
   repeat
      (* this loop is executed (NMC+NGB) times *)
      if MARKED(P)
         then UPDATE_CHILDREN(P);
      P:=NEXT(P)
   until P = NIL;   (* end repeat loop *)

(* third pass: relocate marked cells *)
   P:=START;
   repeat
      (* this loop is executed (NMC+NGB) times; a temporary location
         is needed to save the information contained in the old size
         field of the cell being moved *)
      T:=NEXT(P);
      if MARKED(P) then
         begin
            (* this branch is executed (NMC) times *)
            NEW_ADDRESS:=LINK(P);
            (* set link field to unmarked for future use *)
            UNMARK(P);
            MOVE(P,NEW_ADDRESS)
         end;
      P:=T
   until P = NIL;   (* end repeat loop *)
end; (* procedure COMPACT_LISP2 *)

procedure UPDATE_ROOTS;
begin
   (* this loop is executed (NIP) times *)
   for (all initial pointers i) do
      if (M[i] is not nil) then UPDATE(i)
end;
```

Fig. 2. PASCAL-like program implementing the Lisp 2 compactor.

As the scan of Phase 1 proceeds, the total number of freed words steadily increases, from zero to the value NGB $*$ SIZEC (see Table I). The expected number of free words after processing b blocks of marked cells is at least $b *$ SIZEC.

```
procedure UPDATE_CHILDREN(P):
begin
   (* this loop is executed (NPC) times *)
   for (all pointers i in the cell P) do
      if (M[i] is not nil) then UPDATE(i)
end;

procedure MOVE(P,NEW_ADDRESS):
begin
   (* this loop is executed (SIZEC) times *)
   for i:=0 to M[P+1]-1 do
      M[NEW_ADDRESS+i]:=M[P+i]
end;

Variables Used in Lisp2 Compaction
==================================

P           -  address of the first element of a cell

START       -  address of the first location in the area being compacted

LAST        -  address of the last location in the area being compacted

NEW_ADDRESS -  address to which a cell will be relocated after compaction


Fields Used in the Lisp 2 Compactor
===================================

LINK(P)              -  contains the address of the future location of a cell,
                        i.e. after compaction.  When this field is nil, the
                        cell is considered to be unmarked.

MARKED(P)            -  contains the value TRUE if the cell is marked, otherwise
                        contains the value FALSE


SIZE(P)              -  yields the size (number of words) of the cell pointed to
                        by P


Auxiliary Procedures Used in the Lisp 2 Compactor
=================================================

COMBINE(P)           -  combines two adjacent garbage cells by changing the
                        contents of the field SIZE(P) to SIZE(P) + SIZE(NEXT(P))

MOVE(P)              -  moves the cell pointed to by P to its new address

NEXT(P)              -  yields the address of the cell following the cell pointed
                        to by P.  NEXT(P) returns the value NIL if the computed
                        address exceeds the value of LAST.

UPDATE(A)            -  update pointer field A, so that it now points
                        to the new location of the cell

UPDATE_CHILDREN(P)   -  updates each of the NPC pointers in the cells pointed
                        to by P

UPDATE_ROOTS         -  updates each of the initial (root) pointers

UNMARK(P)            -  unmarks the cell P
```

Figure 2 continued.

Let T be the size of the largest cell; then, when $T < b * \text{SIZEC} - b$, the freed space should be large enough to accommodate both the break table and the next marked cell, so we would not have to roll the table. Thus, an upper bound for the average value of S is $T/(\text{SIZEC} - 1)$. If T is relatively small, only a short initial

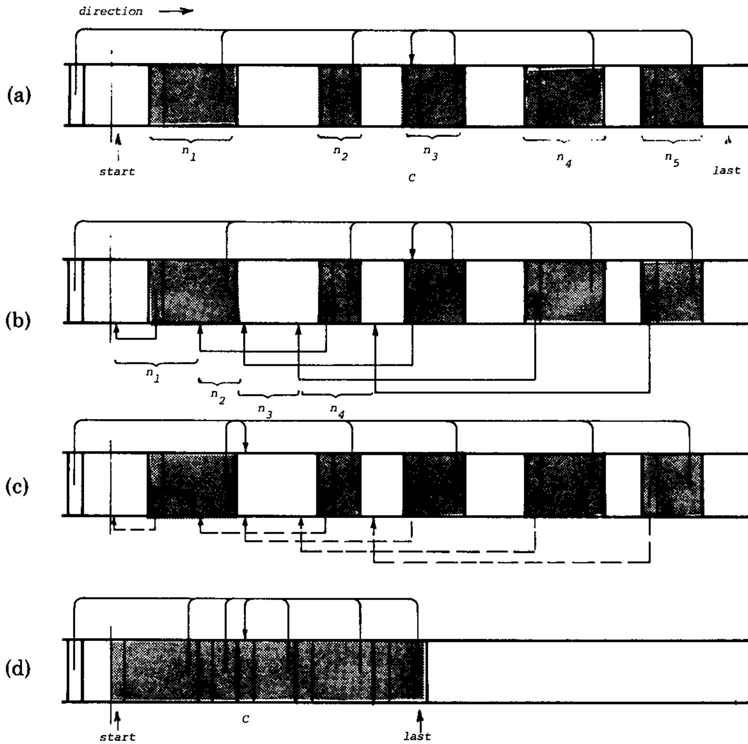

Fig. 3. Snapshots used in describing the Lisp 2 compactor: (a) initially; (b) after pass 1; (c) after pass 2; (d) after pass 3.

segment of the final break table will need to be sorted. This suggests that a few scans of the break table be made to determine the size of this initial segment. A suitable sorting strategy then proceeds as follows:

1 Scan the BT from higher to lower addresses, finding the first entry such that

$$a_{j-1} > a_j.$$

2 Find the largest value V between a_1 and a_{j-1}.
3 Scan the BT from $j - 1$ onward and find the largest index S such that

$$a_{S-1} \leq V < a_S.$$

4 Sort the values between a_1 and a_S.

The time-formula for Step (2) above becomes γ S log S where S is at most $T/(\text{SIZEC} - 1)$. This result suggests that sorting can become time consuming when there are large active arrays in memory.

The expense of sorting caused by excessive rolling can be avoided if each BT entry contains an extra field. This field stores the final position of the entry in the table, and it is assigned a value in Phase 1 when the entry is created. The table can now be rolled with impunity, since an obvious linear algorithm can be used to restore the table to the order indicated by the extra field.

2.3 Morris

Morris' algorithm is based on the following property: assume that the contents of locations A, B, C, ... point to location T. Then no information is lost if this tree structure, with root T, is transformed into a linear list by stringing together locations $T, \ldots, C, B, A$ and placing the contents of T in A. Once the new position of T, say T', is known, it is simple to reconstruct the original tree by making A, B, and C point to T'.

This property has been used by Fisher in a special type of collector [4]. Morris pioneered the development of general compactors based on that property. (We see below that Jonker's algorithm uses it as well.)

Morris' algorithm requires two tag bits for each word in a cell.[4] These bits indicate different configurations of pointers and data according to the following convention:

0: inactive (garbage);
1: pointer;
2: swapped pointer;
3: nonpointer (data).

Morris' algorithm may require three passes through the memory space. The first pass only readjusts forward-pointing references and references pointing to their own cell. The second pass readjusts backward-pointing references. Finally, a third pass moves the active cells toward the highest address in the memory space. The first pass is in the forward direction. The other two passes are in the backward direction and may be combined into one pass.

One advantage of Morris' compactor is that it does not require that pointers reference the beginning of a cell. It would be inefficient in terms of storage to modify Lisp 2 to handle these general pointers. (Jonkers remarks that his compactor may handle general pointers by introducing tag bits.)

The heart of Morris' algorithm is a procedure that is called within the first two passes, that is, in the forward and in the backward directions. Its parameters are the current field being examined and the direction of the pass. Basically, this procedure *threads* pointers to a given cell as they are encountered. Assume cells A, B, and C point to location T whose contents are X. Then threading A, B, and C consists of linking T, C, B, and A by pointers and placing X in A. Note that threading involves swapping information, and this is indicated by an appropriate setting of the tag bits.

The operation of updating consists of unthreading, that is, transforming the threaded list into its original tree form. In our example unthreading consists of making A, B, and C point to the new address T', where the original contents of T are now stored.

The first pass of Morris' algorithm consists of threading and updating forward pointers (as well as pointers which reference their own cell). The second pass, in the backward direction, also calls this procedure to thread and update backward pointers. Finally, the third pass moves the cells toward the highest addresses of the memory space.

[4] In its original presentation, Morris used cells containing only pointers. In that case only one tag bit is required.

The reader interested in the actual implementation of Morris' algorithm should examine Figure 4, which presents the details of the thread and update operations.

The snapshot shown in Figure 4b is taken when cell C is reached during the first pass. This is followed by an updating of the pointers originally pointing to C (see Figure 4c). The updating of pointers referencing their own cell is shown in Figure 4d.

Figure 4e shows a snapshot during the second pass when cell C is reached. Figure 4f is a snapshot taken after updating the backward pointers to C. Figure 4g shows the memory space after the higher address cells have been moved.

A source of inefficiency in Morris' algorithm is that the backward pass has to scan every word of a cell until it accesses the header information needed to locate the cell's pointers (see Figure 1). Recall that a complete scan within cell boundaries is unnecessary in the forward pass since unmarked cells are easily skipped and access to marked cell's pointers is immediate.

In his final remarks Morris suggests that this inefficiency may be surmounted by storing in inactive cells (during the forward pass) a pointer to the header of the preceding active cell. Some speedup is possible by using this approach to skip garbage cells in backward scans, and this was taken into account in constructing the time-formulas.

2.4 Jonkers

Jonkers improves Morris' algorithm by eliminating the need for tag bits and by using two passes, both in the forward direction. Note that Morris needed the tag bits to differentiate between swapped and unswapped pointers in both forward and backward directions. The need for tag bits is eliminated by assuming that a word of the cell (originally containing data) is large enough to store an address. In our description this word is the first word of the cell.

Let C be a typical cell. The first pass threads the forward pointers to cell C. When C is reached, the forward pointers are updated. As the first pass continues, pointers which refer back to cell C are threaded. Pointers which reference their own cell are treated as backward pointers.

The second pass compacts the previously updated cells until cell C is reached. From then on it updates the backward reference pointers and moves their cells to the compaction area. The detailed program is presented in Figure 5.

Figure 6b is a snapshot taken during the first pass after cell C is reached. It is assumed that the contents of the first word of C is X. At this stage all cells of lower addresses containing pointers to C have been threaded. The snapshot in Figure 6c is taken after the threaded lists are updated. The configuration after processing of a self-referencing cell is depicted in Figure 6d. A snapshot analogous to that in Figure 6b appears in Figure 6e and is taken after the threading of cells whose addresses are higher than C's.

During the second pass all cells of addresses lower than C's are compacted (see Figure 6f). Figure 6g (similar to Figure 6c) shows the memory configuration after updating of the lists threaded as shown in Figure 6e.

3. TIME-FORMULAS

The information contained in Table I and in the program's comments enable us to generate time-formulas for the compactors described in the previous sections.

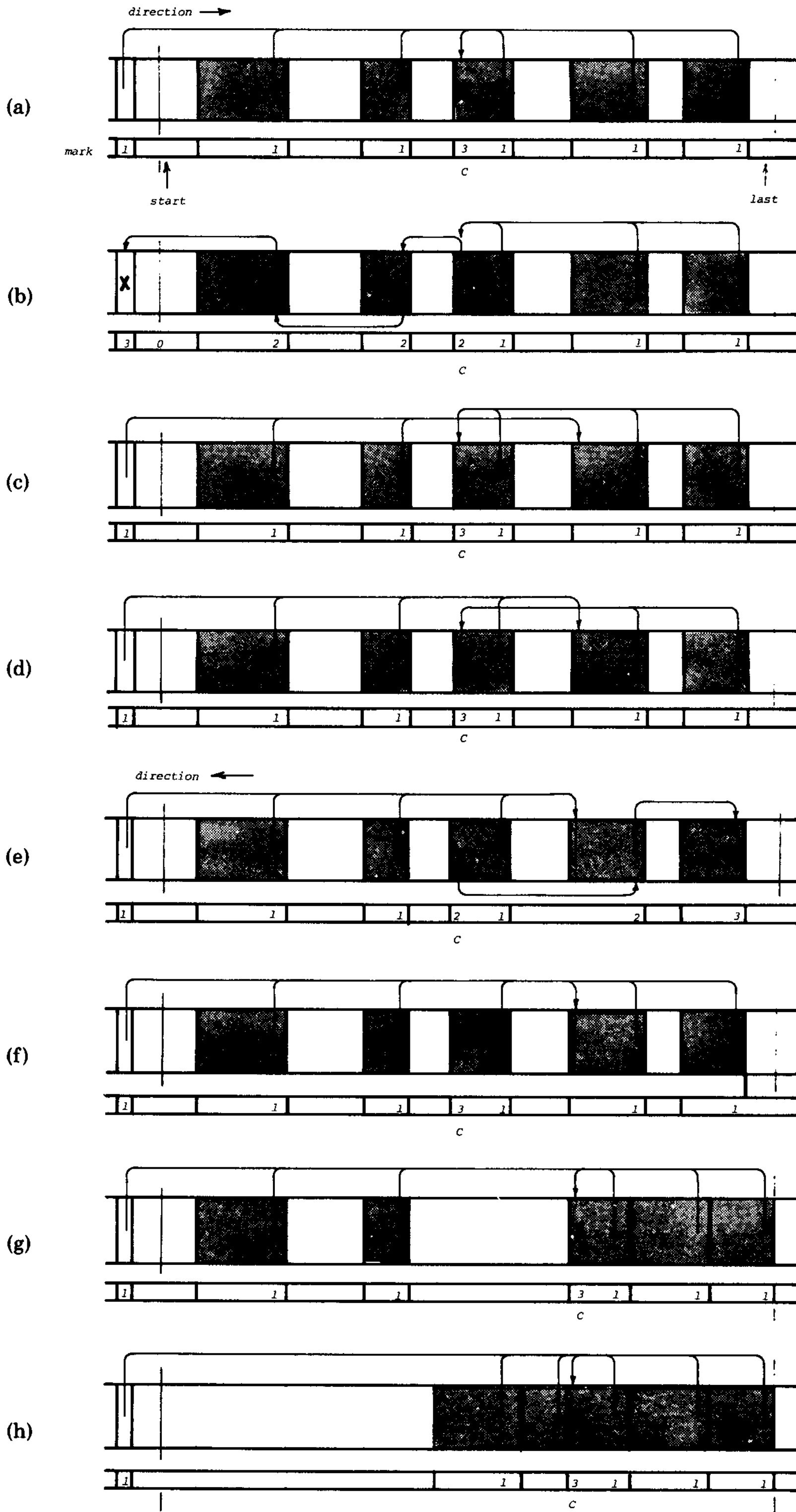

Fig. 4. Snapshots used in describing Morris' compactor.

```
procedure COMPACT_JONKERS;
(* compacts used cells towards lowest address *)
begin

(* first pass: link all pointers, update forward pointers *)
   THREAD_ROOTS;
   P:=START;
   NEW_ADDRESS:=START;
   repeat
      (* this loop is executed (NC) times *)
      if MARKED(P) then
         begin
            (* this branch is executed (NMC) times *)
            UPDATE_J(P,NEW_ADDRESS);
            THREAD_CHILDREN(P);
            NEW_ADDRESS:=NEW_ADDRESS+SIZE(P)
         end;
      P:=NEXT(P)
   until P = NIL;   (* end repeat loop *)

(* second pass: update backwards pointers and relocate
   used cells towards lowest address *)
   P:=START;
   NEW_ADDRESS:=START;
   repeat
      (* this loop is executed (NC) times *)
      if MARKED(P) then
         begin
            (* this branch is executed (NMC) times *)
            UPDATE_J(P,NEW_ADDRESS);
            MOVE(P,NEW_ADDRESS);
            NEW_ADDRESS:=NEW_ADDRESS+SIZE(P)
         end;
      P:=NEXT(P)
   until P = NIL   (* end repeat loop *)
end;   (* procedure COMPACT_JONKERS *)


procedure THREAD(P);
var T: word;
begin
   (* let T be the address contained in P. Then THREAD makes the location T
      point to P and stores the old contents of T in P; Fig. 6 illustrates
      graphically the action of consecutive calls of this procedure. *)
   if (M[P] <> nil) then
      begin
         (* this branch is executed (NAP) times *)
         T:=M[P];
         M[P]:=M[T];
         M[T]:=P
      end
end;   (* procedure THREAD *)


procedure UPDATE_J(P,NEW_ADDRESS);
var S,T: word;
begin
   (* this procedure replaces a linked list pointed to by P, by a tree
      structure in which all elements of the list point to the new address
      of P. (see illustration in Fig. 6 ) *)
   T:=M[P];
   while  POINTER(T) do
      begin
         (* this loop is executed (NAP) times *)
         S:=M[T];
         M[T]:=NEW_ADDRESS;
         T:=S
      end;
   M[P]:=T
end;   (* procedure UPDATE *)

NOTE: The predicate "pointer(P)" is easily implemented if the conventions of
      Fig.1 are used. Otherwise, a special bit may be needed for this purpose.
```

Fig. 5. Program implementing Jonkers' compactor.

```
Variables Used in Jonker's Compaction
=====================================

P,START,LAST,NEW_ADDRESS
                    - as in the Lisp 2 Compactor


Fields Used in Jonker's Compactor
=================================

MARKED(P),SIZE(P) - as in the Lisp 2 Compactor


Procedures Used in Jonker's Compactor
=====================================

MOVE(P,NEW_ADDRESS)    - relocates the cell pointed by P to its  new address.
                         This procedure also unmarks a cell.

NEXT(P)                - as in the Lisp 2 Compactor

THREAD_CHILDREN(P)     - for (all pointer fields i contained in cell P) do
                         THREAD(i)

THREAD_ROOTS           - for (all initial pointers i) do THREAD(i)
```

Figure 5 continued.

In this section we only present the time-formula for the Lisp 2 compactor. The reader should have no difficulty reconstructing the time-formulas for the other three compactors.

Our time-formulas have been generated semiautomatically by a revised version of the program described in [3]. This version allows the user to specify the flows in selected branches of a program being analyzed. Kirchhoff's law is then applied using the algorithm proposed by Knuth and Stevenson [9] to determine the flows in the various program branches as functions of the flows specified by the user.

It should be mentioned that the task of generating time-formulas is akin to that of compiling. Therefore, if a given implementation optimizes certain parts of an algorithm, that optimization should be taken into consideration in constructing its time-formula. We have endeavored to produce the "most reasonable" optimized version of the compactors. Although we expect the reader to trust our results, he or she should keep in mind that the techniques proposed herein will enable him or her to reconstruct time-formulas for a particular compactor taking advantage of particular machine features. With the information in the program (Figure 2) and in Table I we can easily derive the time-formula for the Lisp 2 compactor (see Figure 7).

Once a time-formula is generated, it can easily be processed by a symbolic formula manipulator, for example, Macsyma [10]. This processing consists of simplifying a formula and expressing it in terms of certain parameters. By binding the time-variables to specific numeric values applicable to a given machine, we are able to do a microanalysis of the compactors. The results of this analysis are presented in the next section.

4. RESULTS

When plotting curves describing the time efficiencies of the algorithms, it is important to reduce the number of variables in the time-formulas.

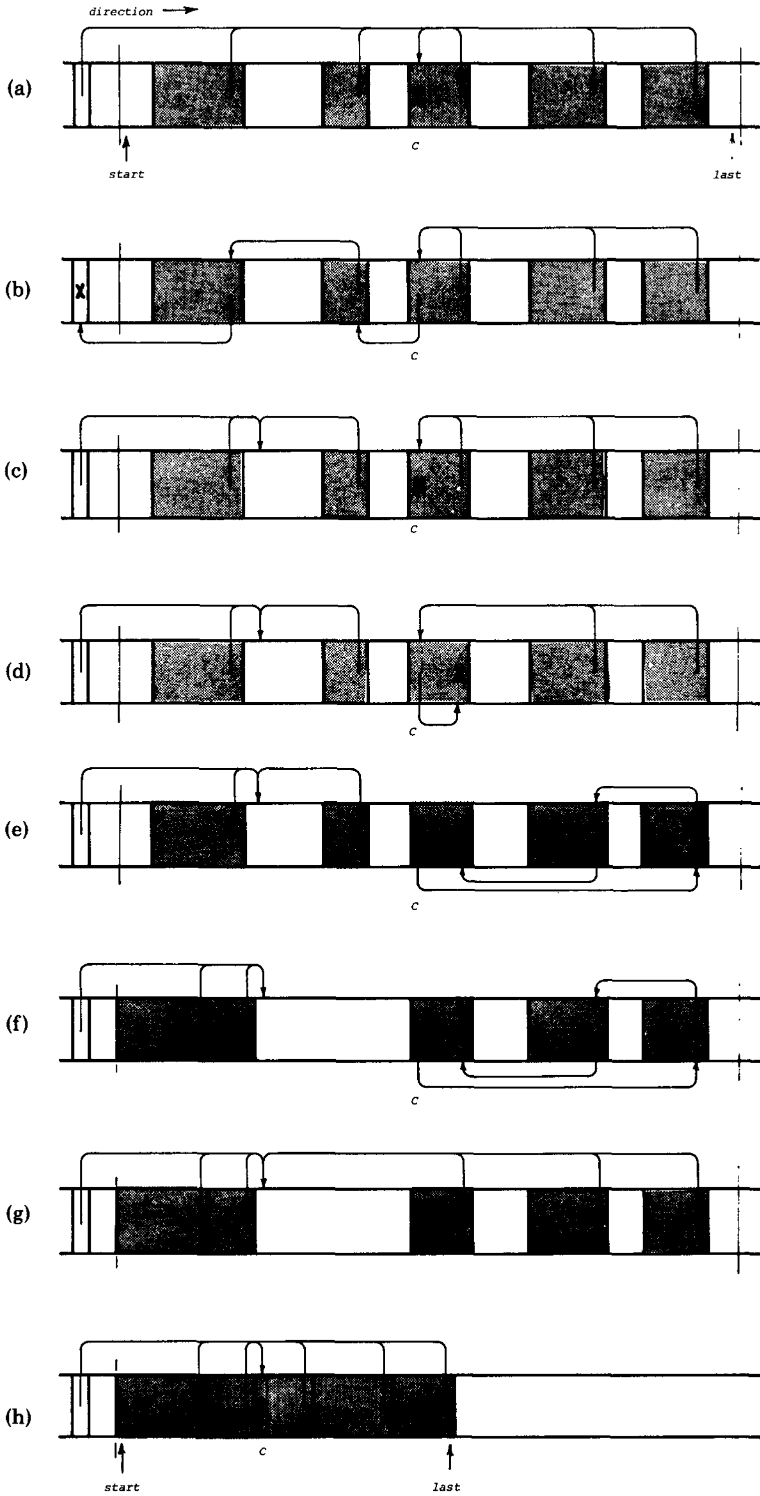

Fig. 6. Snapshots used in describing Jonker's compactor.

```
TLISP2:=
2*ASSIGN+
(NGB+NMC+1)*(WHILEOH+COND)+
        (NGB+NMC)*(IFOH+MARKED)+
                (NMC)*(2*ASSIGN+LINK+ADDITION+SIZE)+
                (NGC)*(WHILEOH+NEGATION+MARKED+NEXT)+
                        (NGC-NGB)*(COMBINE)+
        (NGB+NMC)*(ASSIGN+NEXT)+
UPDATEROOTS+
ASSIGN+
(NMC+NGB)*(REPEATOH+COND+
        (IFOH+MARKED))+
            NMC*(UPDATECHILDREN)+
         (NMC+NGB)*(ASSIGN+NEXT)+
ASSIGN+
(NMC+NGB)*(REPEATOH+COND+
        (ASSIGN+NEXT)+
        (IFOH+MARKED))+
            NMC*(ASSIGN+LINK+UNMARK+MOVE)+
         (NMC+NGB)*(ASSIGN);

NEXT:=2*ADDITION+SUB1;
COMBINE:=ASSIGN+6*ADDITION+4*SUB1;
MARKED:=SUB1+COND;
LINK:=SUB1;
SIZE:=SUB1+ADDITION;
UPDATEROOTS:=NIP*(IFOH+COND+SUB1+FOROH)+NIAP*(3*SUB1+ASSIGN);
UPDATECHILDREN:=SUB1+4*ADDITION+
      NPC*(IFOH+COND+SUB1+FOROH)+NAPC*(3*SUB1+ASSIGN);
UNMARK:=SUB1+ASSIGN;
MOVE:=ADDITION+SUBTRACT+SUB1+SIZEC*(FOROH+2*SUB1+2*ADDITION+ASSIGN);
```

Fig. 7. Time-formula for the Lisp 2 compactor.

First, we assume that there is only one active initial pointer to the area being compacted. Two ratios are critical in estimating the efficiencies of the compactors:

α: (number of marked cells)/(total number of cells) (according to Table I, α = NMC/NC);

β: (number of nonnil pointers in marked cells)/(total number of pointers in marked cells) (i.e., $\beta = (\text{NAP} - 1)/(\text{NPC} * \text{NMC})$).

We have also assumed that the cells have an average size of five words and have an average of two pointers. Therefore, there are $2 * \beta$ active pointers per cell.

In what follows we show how to express NGB, the number of garbage blocks, as a function of α, the quantity indicating the percentage of marked cells. Let $\alpha' = (1 - \alpha)$, that is, the ratio of garbage cells (NGC) to the total number of cells (NC). The average size of garbage blocks A is determined by

$$A = \alpha(1 + 2\alpha' + 3\alpha'^2 + \cdots) = \frac{1}{\alpha}.$$

Therefore, NGB $= \alpha(1 - \alpha)$NC.

Given a time-formula such as that of Figure 7, it would be trivial to obtain simplified formulas that hold for a different set of assumptions. The curves presented here were obtained by binding the values of the time-variables to those of a PDP-10–PASCAL configuration (see Table II).

It is interesting to note that, although the compactors behave almost linearly in α, the actual time-formulas using NGB have a (small) nonlinear component.

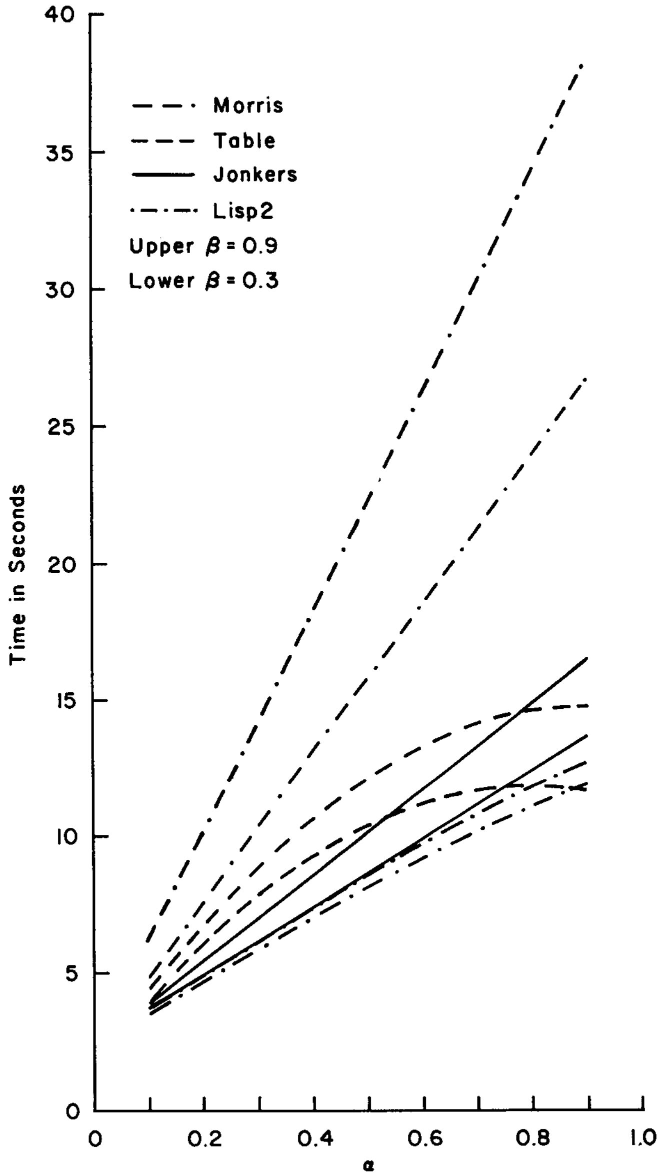

Fig. 8. Time comparisons for the four compactors.

For example, the use of Macsyma in processing the time-formula for Lisp 2 (Figure 7) yielded

$$\text{execution time} = -1.45\alpha^2 = 12.97\alpha = 2.21 \text{ seconds.}$$

Figure 8 shows the results obtained for the four compactors given the assumptions made at the beginning of this section and with $\beta = 0.9$ (i.e., with a large percentage of active pointers) and $\beta = 0.3$. The total number of cells was assumed to be 100,000, but similar results were obtained by considering other memory sizes.

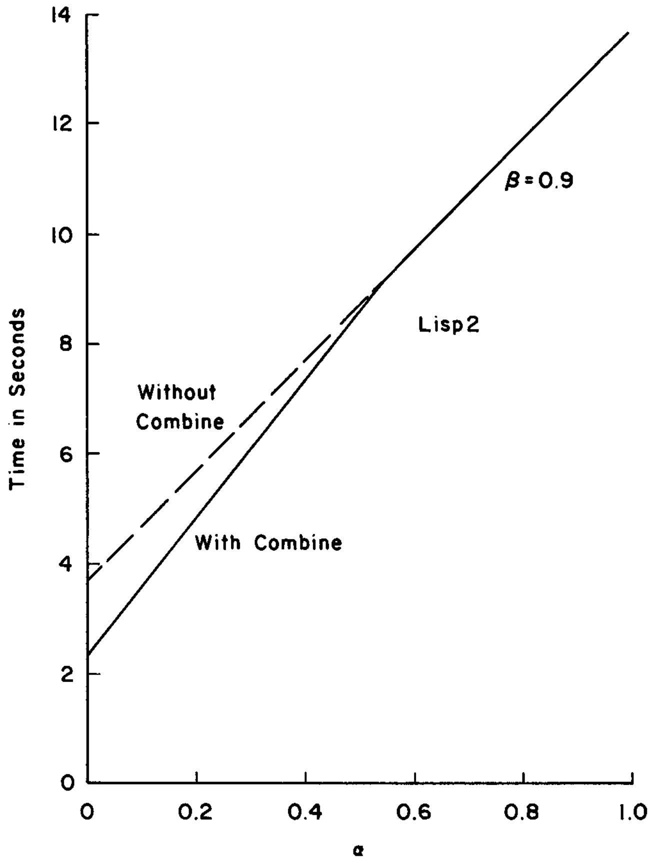

Fig. 9. Effect of combining the garbage cells in the Lisp 2 compactor.

Benchmarks obtained by actually running the compactors yielded results within 5 percent of those shown in Figure 8. The discrepancies are due to several factors, among them the fact that the times presented in Table II are average times.

The curves show that Lisp 2 is the most time efficient of the compactors. However, it should be kept in mind that the Lisp 2 compactor requires an additional word per cell. For the assumed cell distribution, Morris' compactor is the least efficient even when assuming that an extra (small) field was available at the end of each cell to store the cell size, therefore permitting a speedup of the backward scan. Figure 8 also indicates that the Lisp 2 compactor exhibits the smallest overhead when the number of pointers increases.

The effects of introducing changes in two of the compactors are depicted in Figures 9 and 10. Figure 9 shows how the efficiency of the Lisp 2 compactor decreases when garbage cells are not combined during the first scan (see Section 2.1). Figure 10 shows the nonlinear behavior of the table compactor when the entire break table needs to be sorted. As mentioned earlier, this may occur if there are large discrepancies in the sizes of cells in the memory. The more linear

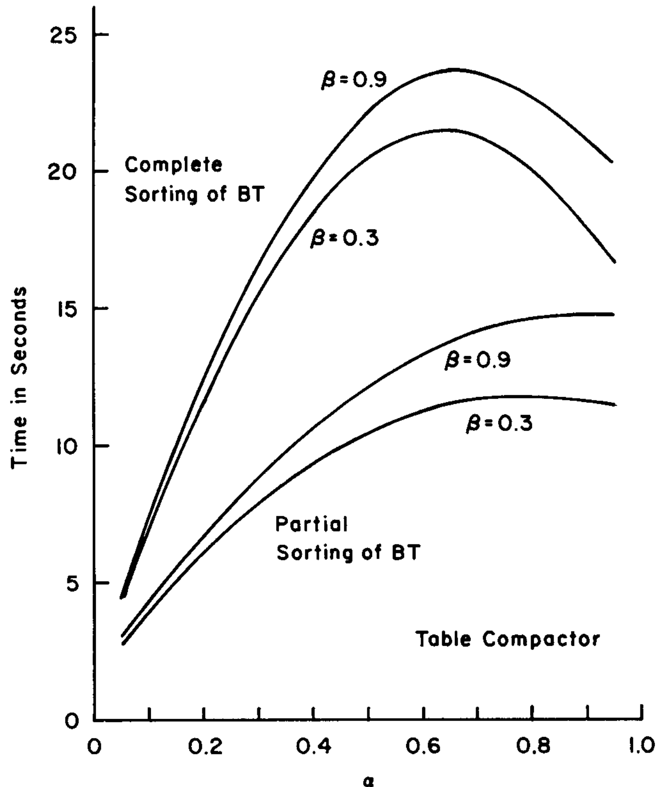

Fig. 10. Effect of increased sorting in the table compactor.

curves in Figure 10 (which are identical to those of Figure 8 for the table compactor) were obtained by assuming that the largest cell was ten times larger than the average cell.

Figures 11 and 12 show the effect of varying some of the parameters of the time-formulas. The curves shown in Figure 11 indicate how the time efficiency of the compactors varies with cell size. The results are based on the assumption that the total memory size remains constant. Also, the number of pointers per cell was considered to remain constant and equal to five. Results indicate that compactor efficiency increases as cell size increases. This is expectable since there are fewer pointers to be readjusted. The relative efficiencies of the compactors, however, remain essentially the same.

The effects of changes in the value of the time-variable SUB1 are shown in Figure 12. (Recall that SUB1 stands for the time to access an item in the memory.) The results indicate that the relative efficiencies of the compactors remain approximately the same as the value of SUB1 is doubled or halved.

An added advantage of the availability of time-formulas for the compactors is that time-space trade-offs may be studied. For example, the choice between two compactors (say, Jonkers and Lisp 2) could be made by evaluating the product $S * P$ (i.e., space * time) for each compactor and selecting the compactor yielding the smallest product. Note, however, that the determination of S is machine

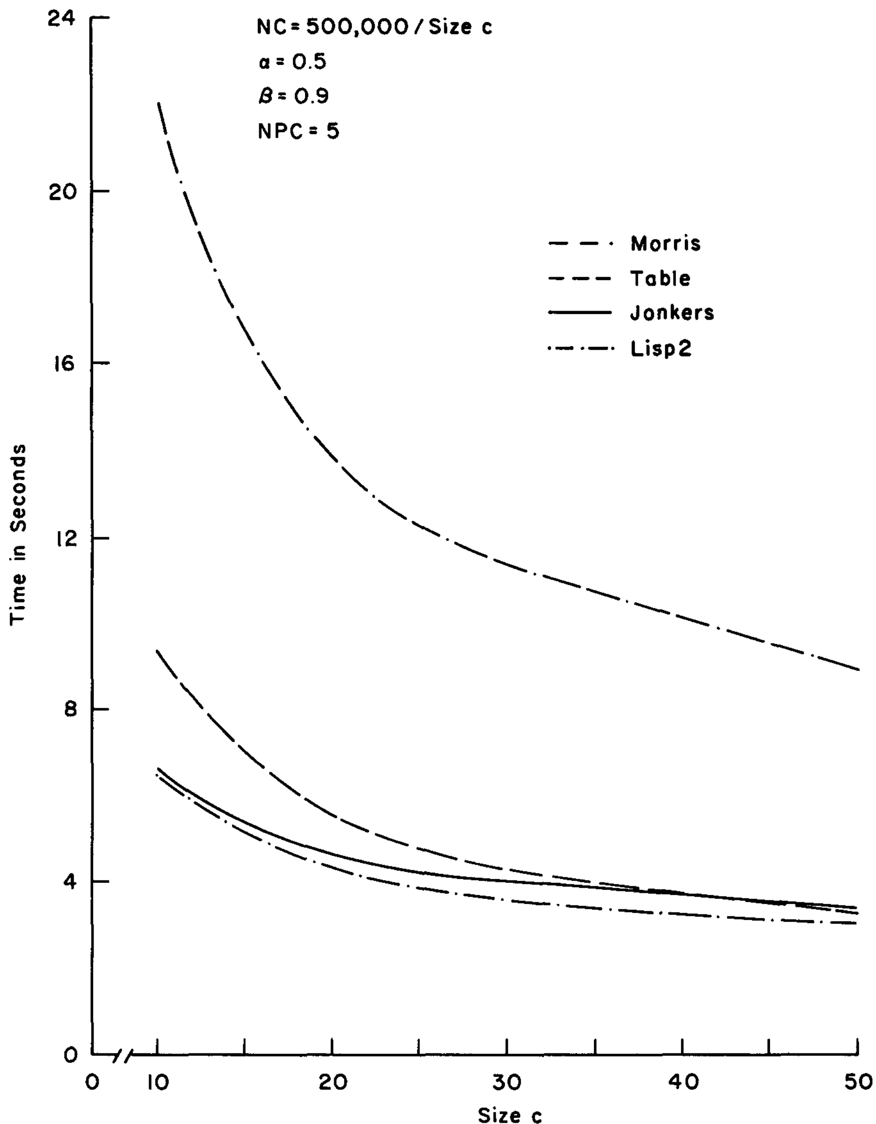

Fig. 11. Effect of cell size.

dependent since word boundaries have to be taken into account. The following estimate, suggested by one of the referees, allows us to approximate the space-time impact of the extra pointer required for the Lisp 2 algorithm. When the size of memory considerably exceeds the steady-state requirements of the application program, the primary effect of larger cells will be more frequent garbage collection. Assume that the application requires cells at a rate of w per time unit. Then with the Lisp 2 collector the requirement will be Θw per time unit where

$$\Theta = \frac{\text{Lisp 2 cell size}}{\text{nominal cell size}}.$$

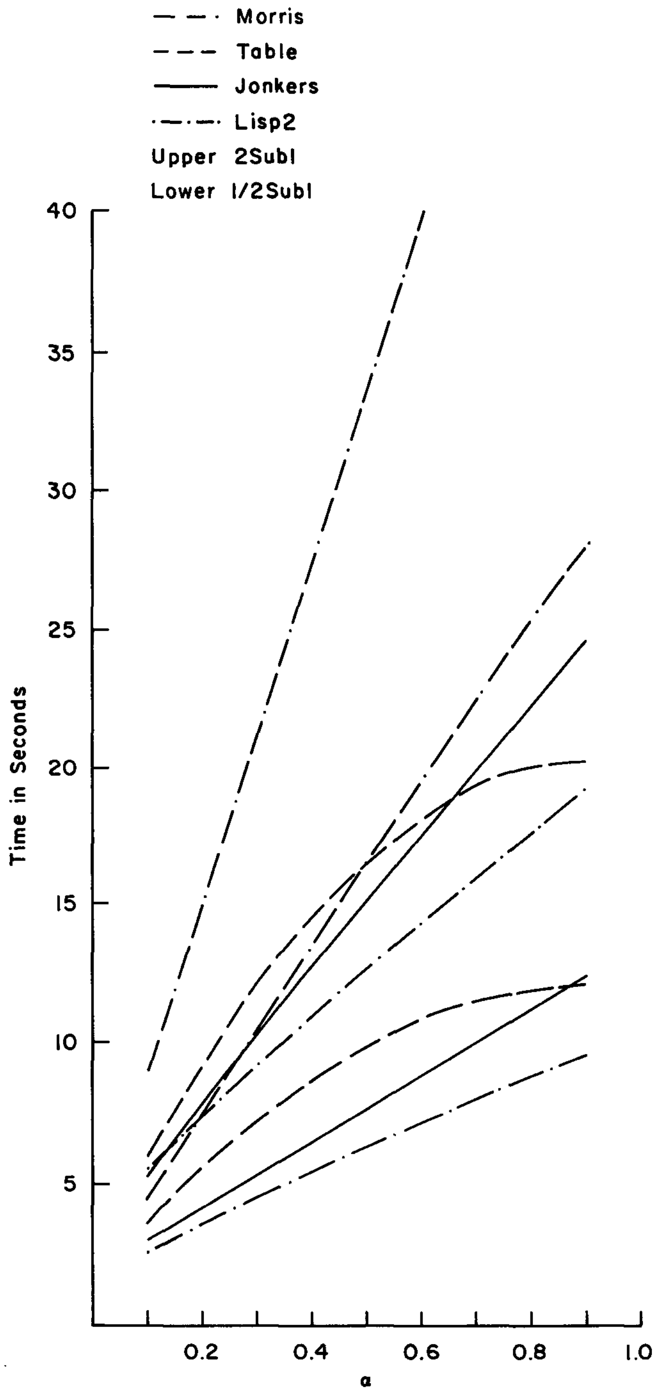

Fig. 12. Effect of doubling and halving the value of SUB1.

With a fixed memory size, Lisp 2 will have to be called Θ times as often as a nominal size collector. Thus, its cost should be multiplied by Θ. This has the effect of moving the Lisp 2 curves up slightly. In Figure 8 the space-penalized Lisp 2 curves would lie between the curves for Jonkers (and slightly above for small α).

Great caution should be exercised in generalizing the presented results to cases involving assumptions far removed from the ones made herein. It is not improbable that the relative efficiencies of the compactors will change in special cases.

One should also keep in mind that, although Jonkers' algorithm exhibits the best performance with minimal additional space, the algorithm requires that there must exist enough space in a cell to store a pointer. The selection of an algorithm over another may therefore depend on factors other than time efficiency.

ACKNOWLEDGMENTS

The authors wish to thank Carolyn Boettner, Peter Pih, and Sandra Belloni for the help provided during the gestation phase of this work. Susan Denker's participation in revising the time-formulas and obtaining further results is gratefully appreciated. Timothy Hickey made a thorough study of the table compactors and assisted in the processing of time-formulas using Macsyma. Finally, the careful and constructive remarks made by one of the referees enabled the authors to produce a better version of the paper.

REFERENCES

1. COHEN, J. Computer-assisted microanalysis of programs. *Commun. ACM 25*, 10 (Oct. 1982), 724–733.
2. COHEN, J. Garbage collection of linked data structures, *Comput. Surv.* (ACM) *13*, 3 (Sept. 1981), 341–367.
3. COHEN, J., AND ZUCKERMAN, C. Two languages for estimating program efficiency. *Commun. ACM 17*, 6 (June 1974), 301–308.
4. FISHER, D.A. Bounded workspace garbage collection in an address order preserving list processing environment. *Inf. Process. Lett. 3*, 1 (July 1974), 29–32.
5. FITCH, J.P., AND NORMAN, A.C. A note on compacting garbage collection. *Comput. J. 21*, 1 (Feb. 1978), 31–34.
6. HADDON, B.K., AND WAITE, W.M. A compaction procedure for variable length storage elements. *Comput. J. 10* (Aug. 1967), 162–165.
7. JONKERS, H.B.M. A fast garbage compaction algorithm. *Inf. Process. Lett. 9*, 1 (July 1979), 26–30.
8. KNUTH, D.E. *The Art of Computer Programming*, vol. 1: *Fundamental Algorithms*. Addison-Wesley, Reading, Mass., 1973.
9. KNUTH, D.E., AND STEVENSON, F.R. Optimal measurement points for program frequency counts. *BIT 13* (1973), 313–322.
10. MATH LAB GROUP. Macsyma reference manual. Laboratory for Computer Science, Massachusetts Inst. of Technology, Cambridge, Mass., Dec. 1977.
11. MORRIS, F.L. On a comparison of garbage collection techniques. *Commun. ACM 22*, 10 (Oct. 1979), 571.
12. MORRIS, F.L. A time- and space-efficient garbage compaction algorithm. *Commun. ACM 21*, 8 (Aug. 1978), 662–665.
13. WEGBREIT, B. A generalized compactifying garbage collector. *Comput. J. 15*, 3 (Aug. 1972), 204–208.

Received July 1980; revised August and November 1982; accepted December 1982