A COMPACTIFYING GARBAGE COLLECTOR FOR ECL'S NON-HOMOGENEOUS HEAP

(DRAFT)

2-74

William R. Conrad

Center for Research in Computing Technology

Harvard University
Cambridge, Massachusetts 02138

A COMPACTIFYING GARBAGE COLLECTOR FOR ECL'S NON-HOMOGENEOUS HEAP

(DRAFT)

2-74

William R. Conrad

Harvard University
Center for Research in Computing Technology
Cambridge, Massachusetts

This work was supported in part by the Advanced Research Projects Agency under contract F19628-71-C-0173.

Summary

This paper gives the algorithms for the implementation of a compactifying garbage collector for ECL's non-homogeneous heap. It also presents a storage allocation technique made possible by the presence of a compactifying garbage collector which can greatly reduce the number of garbage collections.

Introduction

ECL is an extensible language whose source language resembles Algol but which evaluates statements in much the same way Lisp does. It features a powerful data definition facility with regard to both the structure and the behavior of data types. It has a working compiler, and arbitrary mixtures of compiled and interpreted routines may be run together. As might be expected with such a language, storage management is relatively complex; the heap has eight or so distinct regions of dynamically varying size in not necessarily contiguous locations. The state of the heap is defined by use of a QMAP, a directory of each page (unit of allocation equalling 512 words) specifying the space to which the page belongs, and by use of separate free-lists for each space. The algorithms given below illustrate how such a heap can be compactified so as to preserve QMAPping and implement separate free-lists for each space.

After presenting the techniques for compactifying a non-homogeneous heap, ideas are presented concerning the proper use of a compactifying garbage collector. In particular, it will be shown that, for many programs, the optimal use of a compactifying garbage collector is to use it very sparingly.

1.0 Basic Algorithm for Compactification

The basic algorithm used is taken from Wegbreit[1], and is a variant of a compactification algorithm proposed by Hadden and Waite [2] in 1967. The method presented here consists of using the gaps created by free areas in a heap to compute and store the relocation information for the active segment immediately above each free area. That information is then used to relocate pointers to all of the active segments. When all pointers have been relocated, all active segments will be slid downward towards the origin of the heap. As an example, envision a homogenous heap with the following regions: (1) an active area of 10 words, (2) a free area of 25 words, (3) an active area of 35 words, (4) a free area of 15 words, and finally (5) an active area of 40 words. Observe that a pointer into region 1 would have a zero relocation factor; that a pointer into region 3 would have a relocation factor of -25 since that whole region would be moved down 25 words to fill up region 2; that a pointer into region 5 would have a relocation factor of -40 since that region would be moved down 25 words as a result of squeezing out region 2 and 15 words as a result of squeezing out region 4.

Now formalize these observations by introducing an ascending free-list such that the free-list entries are of the form <size, link>, and let each entry be located in the first word of the free area which it describes. Let R[I] be the relocation factor associated with the Ith free-list entry and set

$$R[0] = -S[0]$$
$$R[I+1] = R[I] - S[I+1]$$

where the S's are the sizes of each free region. Note that the sizes of each free region may be recovered from the relocation factors when they are needed after relocation to provide information to slide the active fragments down so as to squeeze out the free regions. This fact suggests that the relocation factors may be stored in the size field of each free-list entry. Wegbreit shows that there is always a place to store the required relocation factors, and he gives an algorithm for obtaining the correct relocation factor for any given pointer. The algorithm is: search the free-list until the given pointer is bracketed between two free-list entries and then use the size field (now containing a relocation factor) of the lower bracketing free-list entry to relocate the given pointer. For further details of the basic algorithm (and for a discussion of the more or less conventional trace and mark phases of garbage collection), it is suggested that the paper by Wegbreit be consulted.

1.1 Extended Algorithm for a Non-homogeneous Heap

The extension of the basic algorithm to a non-homogeneous heap is straightforward. The relocation terms R[I] are prepared exactly as in the basic algorithm using each space's free-list. In addition, a page correction factor and an origin correction factor are computed to take into account the fact that a non-homogeneous heap is to be compactified. The page correction factor is minus the sum of the sizes of all pages QMAPped differently from the given page from the first page of the entire heap to the page in question; and, if added to the relocation terms given by the basic algorithm, it would relocate and compactify each separate space to the first page of the heap (to an address of zero). The use of this page correction factor is conceptually equivalent to treating intervening pages which are QMAPped differently as being free space which is to be squeezed out during the sliding of active fragments downward. The origin correction factor is then merely the planned origin (the address of the planned first page) of the space in question after compactification is done.

In practice, an extended relocation factor which is the sum of the basic relocation factor and the appropriate correction factors is used. Symbolically, if I is the ordinal number of a free-list entry for the Kth space on the Jth page, then

$$Re[I,J,K] = R[I,K] + Pc[K,J] + O[K]$$

where Re is the extended relocation factor, R the basic relocation factor, Pc is the page correction factor, and O the planned origin of the space in question. As in the basic algorithm, the extended relocation factors are stored in the size fields of free-list entries. Additionally, the fact that a page without a free-list entry may require a relocation factor is handled by the introduction of a "directory" whose entries are identical to free-list entries. The size of the directory is such that there is always at least one entry corresponding to the first word of each page. The directory is an essential feature of the extended algorithm since it allows the specification of a page dependent relocation factor even when there is no free-list entry on that page. By the use of a directory, both a defect in the extended algorithm is remedied, and the relocation process is greatly speeded up since the free-list for a given space is entered at a point "near" to the pointer to be bracketed.

1.2 Relocation of Pointers

Relocation Environment

Having given the general extension of the algorithm to a QMAPped heap, we will proceed to a description of the specific data structure and algorithm as applied to the ECL system. At the time compactifying garbage collection is to take place, an ordinary garbage collection will have been just finished, and there will be an updated free-list for each space. In addition, the entire heap will be covered by a double (two bits for each word) bitmap* specifying for each word in the heap whether it is (1) free, (2) active with no pointers, (3) with a pointer in the right half, or (4) with a pointer in both halves. After the generation of a directory and the transformation of the free-list size fields into extended relocation factors, this bitmap will be swept and the contents of each word appropriately relocated.

Computation of the Plan

The head of the free-list for the Kth space will be found in SMHEAD[K] and each free-list entry will consist of a <size,link> pair as in the basic algorithm. For the purposes of the algorithms to be given below, the terminator of a free-list will be a link pointing to an address which is numerically greater than any possible heap address. The first step in compactification is to compute the plan which is the origin and number of pages needed for each space after compactification by consideration of the free counts for each space and the number of pages currently in use. This information will be stored in the table SMFREE for each space as the planned number of free words after compactification; and, in the table SMPLAN as minus the number of planned pages (18 bits) and the planned first page (18 bits). Two additional tables are used in the routines

- - - - - - - - - - - - - -

* The only concession made to compactification in the organization of data objects in the heap is that pointers not cross half word boundaries and that the a word contain ing a single pointer contains it in the right half of the word. Because of the fact that two pointers may reside in a single machine word, it was determined that a second relocation trace using a single bitmap would not work since it would not be possible to specify which half of the word had already been relocated. Once it was decided that a double bitmap would be needed and that all pointers could be properly marked for relocation during the first (and only trace), the savings in time (the cost of a second trace) and the simplification of trace logic (a pointer may be marked for relocation many times but it can be relocated only once) appear to have more than made up for the extra space used.

to be given below. The QMAP is a table, one word for each page of heap, which contains the following fields: FREE (9 bits), LINK (9 bits), and TYPE (18 bits). SMCODE is a table, one word for each space, such that SMCODE[K].LH gives the first page of the Kth space and SMCODE[K].RH gives the mode of the Kth space.

Computation of the Directory

At this point we allocate a directory the size of which is a multiple of a power of two and the number of pages currently in the heap. Since the size of a page (512 words) is a power of two, directory entries will not cross page boundaries. Futhermore, there will be at least one directory entry for the first word of each page. The basic format of a directory entry is a relocation factor (18 bits) to be used if the next free-list entry is above the given pointer to be relocated and a free-list pointer (18 bits). The parameters relevant to the directory are DIRLOC, DIRSIZ, and DIRNUM (number of words spanned by a directory entry) which will be set up when the directory is allocated. Below is the inner routine to set up the directory and compute the extended relocation factors for a given space.

```
NEXTGROUP <-
  EXPR(NPG:INT SHARED,GBGN:INT SHARED,GEND:INT SHARED; BOOL)
      BEGIN
         NPG = 0 => TRUE;
         GBGN <- PAGESZ * NPG;
         TILL QMAP[NPG].TYPE # CMODE DO INCREMENT(NPG,1);
         GEND <- PAGESZ * NPG;
         NPG <- QMAP[NPG - 1].LINK;
         FALSE;
      END;
```

```
MAKEDIR <-
   EXPR(SPNDX:INT; NONE)
      BEGIN
         FL <- SMHEAD[SPNDX];
         NEXTPG <- SMCODE[SPNDX].LH;
         CMODE <- SMCODE[SPNDX].RH;
         RF <- SMPLAN[SPNDX].PLANNEDPG * PAGESZ;
         TILL NEXTGROUP(NEXTPG, GSTART, GFIN)
            DO BEGIN
                  INCREMENT(RF, - GSTART);
                  FOR GSTART FROM GSTART BY DIRNUM
                     TILL GSTART GT GFIN
                     DO BEGIN
                           TILL FL GT GSTART
                              DO BEGIN
                                    INCREMENT(RF, - FL.LH);
                                    FL.LH <- RF;
                                    FL <- FL.RH;
                                 END;
                           GSTART LT GFIN ->
                           DIRLOC[GSTART / DIRSIZ] <-
                              HALFWORD(RF, FL);
                        END;
                  INCREMENT(RF, GFIN);
               END;
      END;
```

Actual Relocation of Pointers

After having created the directory and replaced the size field in all the free-lists with extended relocation factors, the double bitmap is scanned, and each pointer, as indicated by its bitmap entry, is relocated. The routine given below performs the actual relocation of the pointer P. Note that if the directory entry's free-list link address is already greater than the pointer to be relocated, the directory entry's relocation factor will be applied.

```
RELOC <-
    EXPR(P:INT; INT)
       BEGIN
        FL1 <- DIRLOC[P / DIRSIZ];
        TILL (FL2 <- FL1.LINK) GT P
           DO   FL1 <- FL2;
        P + FL1.SIZE;
       END;
```

1.3 Sliding of Active Fragments

After all pointers, as specified by the bitmap, are relocated, all active fragments of each space will be slid downward (towards a lower core address) so that with the possible exception of a partially full last page all pages of a given space will be completely full although not necessarily contiguous. During the sliding operation the QMAP will have the format: unused (9 bits), link to next page (9 bits), and mode (18 bits). The QMAP will be used in conjunction with the free-list for each space to do the sliding. After the sliding is done, for each space the link to next page field in the QMAP will be terminated at the last page which actually contains some active fragment, and all pages on the the chain above marked as free (a zero QMAP entry). Below are given the routines which do the sliding for each space as given by its space index. The variable ASIFADR is the address to which the next fragment would be moved if each fragment were to be moved to its final address during sliding. For an address X which is to be the first address of the fragment to be relocated by an extended relocation factor Re[I,J,K] the following algebraic relation holds:

$$X + Re[I,J,K] = ASIFADR$$

which can be solved for X. Below are the routines which do the sliding for a given space. (Note: BLT(FROM,TO,UNTIL) is a routine simulating the PDP-10 block transfer instruction.)

```
FINDFRAG <-
   EXPR(; BOOL)
      BEGIN
 AGAIN: (FRAGBGN # FL OR
            BEGIN
               FRAGBGN <- ASIFADR - FL.LH;
               FL <- FL.RH;
               FRAGBGN = FGROUPEND;
            END)
           AND NEXTGROUP(FRAGPG,FRAGBGN,FGROUPEND) => TRUE;
         FRAGBGN = FL -> GOTO AGAIN;
         FRAGEND <- MIN(FL,FGROUPEND);
         FALSE;
      END;
```

```
SLIDE <-
   EXPR(SPNDX:INT; NONE)
      BEGIN
        (FL <- SMHEAD[SPNDX]) = NIL => NOTHING;
        CMODE <- SMCODE[SPNDX].RH;
        FRAGPG <- SMCODE[SPNDX].LH;
        FGROUPEND <- (FRAGPG + 1) * PAGESZ;
        ASIFADR <- SMPLAN[SPNDX].PLANNEDPG * PAGESZ;
        TILL FGROUPEND GE FL OR
           NEXTGROUP(FRAGPG,FRAGBGN,FGROUPEND)
        DO INCREMENT(ASIFADR, MIN(FL,FGROUPEND) - FRAGBGN);
        HOLEPG <- FRAGPG; HOLEND <- FGROUPEND;
        FRAGBGN <- FRAGEND <- HOLEBGN <- FL;
        TILL FRAGBGN = FRAGEND AND FINDFRAG() DO
          BEGIN
            LENGTH <- MIN(HOLEND-HOLEBGN,FRAGEND-FRAGBGN);
            BLT(FRAGBGN,HOLEBGN,HOLEBGN+LENGTH-1);
            INCREMENT(ASIFADR, LENGTH);
            INCREMENT(FRAGBGN,LENGTH);
            INCREMENT(HOLEBGN,LENGTH) = HOLEND
              -> NEXTGROUP(HOLEPG,HOLEBGN,HOLEND);
          END;
      END;
```

1.4 Shuffling of Pages to Final Location

By consulting the plan and the current QMAP, the page to which a given page is to be moved and the page which is to replace it will be determined. This information will be stored in the QMAP entry for each page. Thus, on entering the shuffle routine, the QMAP will have the following format: page number to move the page to (9 bits), which if zero implies the page is empty, page number from which to fill the page (9 bits), and current mode of the page (18 bits). The number MAXPAGE is the highest page which will be active after compactification is complete, and it plays a special role in the shuffling algorithm. The shuffling algorithm makes use of a page sized buffer (obtained from the bitmap which is no longer needed) and the fact that every time a page is moved to its final destination, a page is freed into which another page may be moved. However, if the page just vacated is above MAXPAGE, there will be no page to move into it as a final destination, and, hence, that page may have to be used as a temporary buffer. To avoid this situation the first step in shuffling is to make sure that all free pages below MAXPAGE are filled with their final contents. Once this is done, it is certain that when a page is freed by moving it to the buffer or its final destination there will be another page also below MAXPAGE to be moved into it. Below are given the routines to do the shuffling.

```
SHUFFLE <-
   EXPR(; NONE)
      BEGIN
         FOR I FROM FIRSTPAGE  TO MAXPAGE
            DO  QMAP[I].TO = 0 ->
                BEGIN
                   J <- QMAP[I].FROM = I => NOTHING;
                   PAGEMOVE(J, I); I <- FIRSTPAGE - 1;
                END;
         FOR I FROM FIRSTPAGE TO MAXPAGE
            DO QMAP[I].FROM # I ->
               BEGIN
                  J <- I; QMAP[J].TO # 0 -> BUFFIL(J);
            SLOOP:K <- QMAP[J].FROM = BUFPAGE
                        => BUFEMPTY(J);
                  PAGEMOVE(K, J); J <- K;
                  GOTO SLOOP;
               END;
      END;


PAGEMOVE <-
   EXPR(F:INT, T:INT; NONE)
      BEGIN
         BLT(F*PAGESZ, T*PAGESZ, (T+1)*PAGESZ-1);
         QMAP[F].TO <- QMAP[T].TO <- 0;
         QMAP[T].FROM <- T;
         QMAP[T].TYPE <- QMAP[F].TYPE;
      END;

BUFEMPTY <-
   EXPR(T:INT; NONE)
      BEGIN
         BLT(BUFLOC, T*PAGESZ, (T+1)*PAGESZ-1);
         QMAP[T].TO <- 0; QMAP[T].FROM <- T;
         QMAP[T].TYPE <- BUFMODE; BUFPAGE <- 0;
      END;

BUFFIL <-
   EXPR(F:INT; NONE)
      BEGIN
         BLT(F*PAGESZ, BUFLOC, BUFLOC+PAGESZ-1);
         BUFMODE <- QMAP[F].TYPE;
         BUFPAGE <- F; QMAP[F].TO <- 0;
      END;
```

1.5 Treatment of Special Problems

There are typically several pages at the low end of the heap which are allocated at system initialization and which are pointed to from the ECL high segment. Since pointers in the high segment cannot be changed, these pages will not be compactified or relocated although they will be subject to regular garbage collection. In addition to these pages, there are IO buffer pages* which are extremely difficult to move once setup because of the PDP-10 operating system. This constraint has been handled by the introduction of a parameter, RELPAG, into the making of the plan. RELPAG is the first page which may be relocated and is the highest system or IO buffer page plus one. During the making of the plan, RELPAG is used to partition the freelist, free counts, and first page tables into two groups. The group which refers only to pages greater than or equal to RELPAG is the group which the above algoritms see. After the compactification down to RELPAG is done, a fixup routine combines the two groups again, and, additionally, properly adjusts the CMAP, free-lists, and free counts to reflect the changes which compactification has caused.

CEXPRs (compiled expressions) are ECL data structures which, with one exception, are properly relocated by the normal tracing and relocation algorithms. The exception is a sequence of integers which are actually machine language code which must be relocated relative to the CEXPR's new origin. This problem is solved by giving the node CEXPR a special garbage collection trace function which consults the CEXPR's own internal bitmap to determine which words contain relocatable addresses. These words are then appropriately marked in the double bitmap for later relocation.

The problem of user requested breaks occurring at an "unclean point" with respect to garbage collection has been handled by use of a "breakcell" which is executed in the interpreter's main loop and in certain time consumming system routines at "clean points". A "clean point" is one at which there are no pointers in the accumulators and the stacks have only proper entries on them. When a break is requested, the contents of the "breakcell" are changed from a no-op to a break calling instruction. In addition, all routines save any live accumulators before calling for an error break or a garbage collection. The garbage collector never needs to consider either the tracing or relocation of the contents of accumulators.

---

*It is possible for a user to specify that a compactification occur on the opening of an IO port so that all IO buffer pages are placed in contiguous low core locations.

1.6 Performance as Implemented in ECL

It has not been possible to run the experiments necessary to give an accurate formula for garbage collection times based on the number of pointers to be traced and relocated. However, some facts have been informally noted which will give an idea of the performance of the algorithm just presented. For a heap of 80 pages with a high proportion of pointers, total time for a garbage collection is about 6 seconds, of which the compactification phase takes about 3 to 4 seconds. Since compactification is an optional procedure which takes place after a regular garbage collection is complete, it should be noted that the figures given above indicate that it takes about the same time to trace and mark a given heap environment as it does to relocate and compactify the same environment. The size of the entire garbage collector is about 1800 words, of which about 900 are devoted to the compactifying part. A bitmap equal to 1/18 of the current heap size and a directory of 256 words are additionally used.

2.0 Storage Allocation Strategy without a CGC

Storage allocation without a compactifying garbage collector tends to be a very conservative procedure since there is no way of undoing the damage which may be done by the liberal allocation of free pages. This damage is commonly referred to as "fragmentation" and it occurs in two varieties: (1) the free-list for a space may contain the requested number of words but they are not in a contiguous block or (2) only a few words on many different pages of a space may be active thereby preventing the reallocation of some of the pages to another space where the demand is greater. The fear of fragmentation has, in the past, led to the practice of doing a garbage collection every time the free-list for a given space was exhausted in the hope that enough free space would be obtained from already allocated pages that no new pages would need to be allocated. While this technique reduces fragmentation and tends to keep core size down, actual experience over a two year period has shown it to be terribly wasteful of run time. In extreme cases, the total garbage collection time can be two or three times the program execution time.

2.1 Storage Allocation with a CGC

It is important to understand that a compactifying garbage collector's primary virtue is that it allows storage to be allocated in any manner whatsoever without the fear of being caught in a fragmentation bind. Once this is realized, it is possible to consider any philosophy of storage allocation between the extremes of doing frequent garbage collections and not doing any garbage collections at all until forced to do so by total exhaustion of core. The way is clear if it is only desired to minimize either total run time or core size, but the choice of a strategy to minimize the integral of total run time (including garbage collection time) and core size is not so simple. If a program could know the number of discarded but not freed words at any given time, it could calculate the cost/gain ratio of doing a garbage collection at that time. However, such knowledge is not obtainable (except as an approximation based on statistics of previous garbage collections) without doing a garbage collection. The difficulty of determining optimal times to do garbage collections led to the concept of using compactifying garbage collections strictly as a corrective procedure triggered by the growth of core size since the last garbage collection. This method, which will be discussed below, could be considered as allowing a user to set the marginal cost (in terms of an increase in core size) that he is willing to pay to reduce the number of garbage collections and thus the total run time.

2.2 The FREEBIE Count Method

The strategy used in ECL is to consider the number of pages allocated since the last garbage collection and to garbage collect every "n" pages given out, where "n" is a user settable parameter. It should be pointed out that this method is a logical extension of the idea of desirable storage minima ("minfs") used by Bobrow[3] in BBN Lisp (and in ECL until recently). It is, so to speak, a virtual desirable storage minimum in that rather than allocating so many free pages by space each garbage collection, permission is granted to the storage allocator to give out on demand up to "n" pages before doing a garbage collection. It has certain advantages over a table of desirable storage minma by space. First, it is virtual; a page is not allocated until requested. Second, it does not require a detailed knowledge of the specific storage usage of a program by each space. Rather, all that is required is the approximate total number of pages a program uses. The exact distribution is immaterial. This method has been nicknamed the FREEBIE count method because it allows a program to obtain a certain number of pages of core free of the cost of a garbage collection.

2.3 Experimental Results

Below are presented the results of varying the FREEBIE count for two different and rather massive programs. The first of these tests consists of compiling an accounting program written in ECL using a compiled version of the compiler. The second of the tests consists of running an interpreted version of the parser table generator on the full ECL grammar. In the tests, every garbage collection was a compactifying garbage collection, and the space freed by the garbage collection was removed from the top of the program core image. Before commenting on the results, one additional fact should be mentioned. In both of the tests, approximately ten of the garbage collections were caused by the openings of IO ports* rather than by the demands of an expanding program. Despite these gratuitous garbage collections, the pattern of the results is clear. As the FREEBIE count is reduced, total runtime increases, and, after a certain point, both the average size and the product of size times runtime stops decreasing. It becomes clear that whether one is trying to minimize size or time or the product of both, a point is reached, and reached fairly quickly, after which additional garbage collections serve no useful purpose. In the mathematical model presented below, these results will be made plausible.

To save space and allow easy compairison between the experimental and predicted results based on the mathematical model, the first line of each run will give the actual experimental results and the second line the predicted results using the average of constants computed for each run according to the model. The predicted results are given only to indicate that it is possible to program the model given below and come up with reasonable results. It is not suggested that the crude model there presented is sufficient for more than a rough estimation of the behavior of a given program.

- - - - - - - - - - - - - - - - - - -

*It was decided that the effects of extra garbage collections would be easier to handle in the model than the inability to compactify which would be caused by IO buffers high in the heap. The actual model which produced these results contained an adjustment for the garbage collections produced by the opening of IO ports of the form (N-NO)/N*F for F in the model given below. NO is the number of forced GCs.

FREEBIE	#GCs	GC Time	Size	Time
Case 1 Compilation of Accounting Program				
32 (actual)	13	64	43	280
(predicted)	13	71	43	290
16	15	76	42	300
	16	87	42	310
8	22	120	41	350
	22	120	42	340
4	38	220	42	440
	35	190	42	410
2	65	370	42	590
	61	330	42	550
1	115	670	43	890
	113	620	42	840
Case 2 Generation of Parse Table for ECL Grammar				
32	11	12	22	400
	11	14	18	400
16	12	14	19	410
	12	15	18	410
8	14	17	18	420
	14	18	18	410
4	18	24	16	410
	18	23	18	410
2	27	38	16	420
	27	35	18	420
1	43	65	16	450
	44	57	18	450

Time in seconds, FREEBIE count and size in K=1024 words.

Time in the right hand column is total runtime. All numbers are given to only two significant figures explaining the variations of the quantity: Time minus Time.

2.4 A Mathematical Model

The model below was developed in an attempt to explain why relatively few garbage collections are usually sufficient to obtain nearly optimal time-size performance and to show the counterproductiveness of more frequent garbage collections. In addition, it provides a theoretical foundation for the FREEBIE count method of storage allocation which the author developed for use in the ECL system.

Method of Analysis

A given process can be logically separated into two processes, the task process and the garbage collection process, and the time and size of each process computed separately. If there were no garbage collections, then the time and average size of the task process would represent the total cost of completing the task. Of course, if there were no garbage collections, the average size of the task process would probably be much bigger than otherwise. To further the analysis, we will characterize the behavior of the task process by two functions of time, both non-decreasing, called the acquisition function and the discard function. If garbage collection were continuously going on, then the product of time and core size for the task process would be the integral of the acquisition function minus the discard function over the task function runtime. However, to obtain this minimal result would result in an infinite runtime for the garbage collection process. Therefore, we consider a finite number of garbage collections and replace the integral of the discard function with a sum. At the same time, we take into account the cost of the garbage collection process by considering the number of garbage collections, the size during which each takes place, and the time for each. It is desired, of course, to determine under what conditions the combined cost of both the task process and the garbage collection process is minimized.

Definitions

F	The FREEBIE count.
N	The number of garbage collections.
S	The average size of both processes.
St	The average size of task process.
Sg	The average size of GC process.
T	The total runtime of both processes.
Tt	The runtime of the task process.
Tg	The runtime of the GC process.
Tgi	The assumed constant time for an individual GC.
A(t)	The total number of words a task process acquires in time t. The only restrictions placed on the function are that it be non-decreasing, bounded, and with a finite set of discontinuities in the interval [0,Tt]. These conditions also apply to D below.
D(t)	The total number of words a task process discards in time t. A GC must take place for these discarded words to be put on a free-list, or, in the case of the model, removed from the size of the program's core image.

Equations of the Model

$$N = A(Tt) / F$$

The definition of the FREEBIE count and the acquisition function is such that a garbage collection is to occur every time F pages are acquired. This equation is basically a statment of the decision procedure used in the FREEBIE count method of storage allocation.

$$S*T = Tt*St + Tg*Sg$$

The combined cost is taken as the sum of the time-size products for each process. This is another way of saying that for well behaved functions, the integral of the sum of two functions is equal to the sum of the integrals of the two functions. As will be seen below, both of the right hand terms could have been defined as integrals or equivalent sums.

$$Tg*Sg = \sum_{1}^{N} (A(T_i) - D(T_{i-1}))*Tg_i$$

The size of the GC process at each GC is the size of the task process when the GC is initiated minus the discard function at the time of the last GC. For simplicity, the time of each GC has been assumed to be a constant and the size overhead of the bitmap and directory has been ignored. These simplifications do not in any way contradict the argument that as the number of garbage collections increase, so does the product of the GC task runtime and size.

$$Tt*St = \int_{0}^{Tt} A(t)\ dt - \sum_{1}^{N} D(T_i)*(T_i - T_{i-1})$$

The sum represents the integral of the step function produced when the discard function is taken at N points. The whole right hand side thus represents the integral of A(t)-E(t), where E is the step function related to D. Since all the functions in this model are assumed to be well behaved in the interval [0,Tt], E(t) approaches D(t) arbitrarily close as N and, hence, the number of steps becomes larger. We therefore state that the product of task runtime and average task size approaches a minimum, the integral of A(t)-D(t), as the number of garbage collections increases. Restating this fact in stronger terms, we have the essential reason why only a small number of garbage collections can produce nearly optimal results: The minimum cost of a task process is approached as the number of garbage collections is increased in exactly the same manner as the value of an integral is approached by an approximating sum as the number of approximating steps is increased.

Discussion

It is a well known fact of numerical analysis that the error in approximating an integral over the interval [a,b] by the rectangular rule (The approximating sum of the model is either rectangular or close to it.) is of the form:

$$((b-a)*h/2)*f'(e),\ a<e<b$$

where $h=(b-a)/n$ is the step size and n the number of steps. We note that the error is proportional to the step size and the value of the first derivative of the function to be integrated at some point in the interval of integration. We now present three observations which will strongly suggest that a relatively small number of compactifying garbage collections will generally produce the best overall results. First, doubling the number of garbage collections (which approximately doubles the cost of the garbage collection task) only halves the error in calculating the integral of the discard function. In general, halving the error in approximating the integral of the discard function will in no way come even close to halving the cost of the task process. Second, if the discard function is relatively smooth, then a small number of steps suffices to approximate its integral with a very small error. Third, if the error is large, then the number of garbage collections necessary to reduce it beyond a certain point becomes very great. For example, if the initial error is 100% with 10 garbage collections, it will take a 1000 garbage collections to reduce it to 1%.

Fiting the Model to the Experiments

In this section, we will give an outline of how the model was used to obtain the predicted results given in section 2.3. The acquisition function was taken to be a linear function of the form C1*t. C1 was determined by the following equation:

$$C1=(N-NO)*F/Tt$$

The discard function was taken to be a square root function of the form C2*SQRT(t) and C2 determined by the following equation:

$$C2 = \frac{\int_0^{Tt} A(t)+C3\ dt - S*T + \sum_1^N (A(T_i)+C3)*Tg_i}{\sum_1^N \sqrt{T_{i-1}}*Tg_i + \sqrt{T_i}}$$

where C3 is the constant size of the ECL interpreter and the equation was obtained by using the definition of average size times runtime and factoring out the constant C2 from the sums involving the discard function. The average garbage collection time was calculated by simply dividing the total GC time for each run by the number of garbage collections for that run. For each case, the average value of the constants calculated from the six runs was then used to calculate the predicted results. It should be emphasized that for each case, only three constants were used to calculate the results of all six runs thus ruling out the possibility that the closeness of the predicted results with the experimental results was purely the result of "curve fitting."

2.5 Conclusions

The primary virtue of a compactifying garbage collector is that it allows a program great freedom in how it allocates storage.

In particular, this freedom allows the use of a liberal allocation policy with a corrective feedback mechanism such as the FREEBIE count method.

Based on the experimental and mathematical results presented above, it appears that a relatively small number of compactifying garbage collections is sufficient to obtain "optimal" results. This is because the reduction in task process cost becomes marginal after a certain point while the cost of each garbage collection remains essentially fixed.

Ironically, the best compactifying garbage collector is one that is hardly ever used. It is the author's belief that, in the future, the important areas of investigation will not be in obtaining marginal increases in speed for a particular garbage collection algorithm, but in the investigation of techniques to minimize the total number of garbage collections necessary to run a particular program efficiently.

REFERENCES

[1]Wegbreit, B.
A GENERALIZED COMPACTIFYING GARBAGE COLLECTOR.
The Computer Journal(Vol 15:204-208) 1972.

[2]Hadden,B.K. and Waite,W.M.
A COMPACTION PROCEDURE FOR VARIABLE-LENGTH STORAGE ELEMENTS.
The Computer Journal(vol 10:162,165) 1967.

[3]Bobrow, D. G., et al.
BBN-LISP TENEX REFERENCE MANUAL.
Bolt, Beranek, and Newman, Inc.
Cambridge, Massachusetts. 1971.