ALLOCATION AND COMPACTION - A MATHEMATICAL MODEL FOR MEMORY MANAGEMENT

Dennis W. Ting
Computer Science Department
The Pennsylvania State University
University Park, PA. 16802

ABSTRACT

A mathematical model for the processes of allocation and compaction is developed. The behavior of the most commonly used compaction algorithm is studied under this model. The relationship between time and space and trade-offs between "conservative" and "non-conservative" compaction algorithms are also investigated.

INTRODUCTION

One important problem in the area of what is generally known as "systems programming" is the problem of dynamic memory allocation: given a fixed size memory, how do we allocate parts of it to satisfy program demands. This problem is of critical importance in the design of operating systems and some language processors because:

- 1) Main memory is one of the most expensive components in a computer system. Good performance of the entire system hinges to a very large extent on the efficient utilization of main memory.
- 2) In some memory configurations, many programs will be forced to abort for lack of a single free area of main memory sufficiently large. The configurations are not under the control of any single program. Hence a program may or may not complete successfully depending on what other programs do. This introduces some amount of non-determinism which is undesirable in a good computer system. The attitude of "if it doesn't work this time, submit it again and maybe it will work the next time" is certainly intolerable.

Furthermore, a designer should be equally concerned with the speed of an allocation algorithm considering how often it is called upon to perform

its function. Thus while a memory allocation strategy should conserve as much memory as possible, this should not be done at a great expense of speed.

Thus the dynamic memory allocation problem is central to operating systems design. A complementary problem is the compaction problem: given that memory is already badly fragmented, what steps can be taken to compact all the busy areas and consolidate some (or all) of the free areas in memory?

These two problems are complementary in that when we speak of dynamic memory management, we really mean both the allocation and, when necessary, compaction as well. We try to be as clever with allocation as possible to reduce memory fragmentation, but when overflow (i.e., when memory is so fragmented that although there are enough free words in total for some request, none of the individual contiguous free areas is large enough to fit the request) does occur, we also want to do compaction in such a way that as few words as possible would have to be relocated.

A few very well known allocation algorithms are the best-fit, the first-fit, and the buddy system. (We exclude the boundary-tag system since the system itself does not define an allocation algorithm). All three algorithms use some sort of linked lists which chain all of the free areas in memory together. The first-fit algorithm searches the linked-list sequentially and allocates the first free area which fits the current request. The best-fit algorithm finds the free area which "best" fits the current request in the sense that the size of the area used is closest to, and yet not smaller than, the size of the request. The buddy system breaks the memory up into blocks of sizes which are powers of 2. The

smallest such block which can fit the request is allocated. There are good general discussions of these allocation algorithms in Knuth [Kn].

On the subject of compaction, however, very little is known. In most (if not all) of the applications of compaction, only one compaction algorithm is employed - namely the "conventional" compaction algorithm which "slides" all busy blocks to the low end of memory. On the subject of this particular compaction algorithm, most papers dealt with the detailed implementation techniques of this algorithm (e.g., Lang and Wegbreit [LW]). It is the purpose of this paper to investigate behaviors of compaction algorithms. Measure their performances, and study the trade-off between time and space.

ALLOCATION

The process of assigning an area of free storage to a request is called (storage) allocation. By allocating storage judiciously, overflow does not take place as often. In this paper, we view an allocation policy as a mapping from one memory state (the one prior to satisfying the request) to another (the one after an area has been assigned). We further define the notion of a conservative allocation as one which does not fragment memory unnecessarily, and which will always use whatever free space is available, i.e., does not fail unnecessarily. Of the three allocation algorithms mentioned before, both the first-fit and best-fit algorithms are conservative, as they always succeed whenever there is in memory a free area which fits the request. The buddy system, however, is not conservative because when a request for 2^{k+1} words is presented to the allocator, and there is a free area of size 2^{k+1} which consists of two subareas of size 2^k each, but they are not "buddies", the allocator will not succeed in allocating this area. It has been shown by Robson [Ro], however, that none of the three allocation algorithms mentioned above is the optimal allocation strategy in that even in the very restricted case where requests are made only for 1 or 2 words, there is an allocation strategy which uses less memory to satisfy all request sequences without possibility of overflow than any of the three methods mentioned above. In fact, Robson showed that there is a request sequence which would cause any allocation to fail if the store used is

only even one word fewer than that used in the optimal algorithm proposed by him. So aside from efficiency considerations, the allocation algorithms used do affect the amount of memory needed to prevent overflow. Any cost used must reflect the overhead time spent doing compaction. Depending on the implementation of the accompanying allocation algorithm, the cost may or may not reflect the time spent on "marking" the free space in memory. In any case, compaction involves relocating busy blocks in memory. The relocation in turn involves, on the one hand, the actual moving of busy blocks, and on the other hand, the modification of pointers to the relocated blocks. If the cost of the actual moving is dominant, we would conceivably want to minimize the number of memory words moved (with respect to the number of words requested). If the cost of modifying pointers dominates, however, we would then want to minimize the number of blocks relocated. Practically, it is conceivable to limit the total number of busy memory words in order to prevent (or at least cut down on the number of) overflows. It is out of the question, however, to limit the number of jobs (or segments) in memory as a large number of very small jobs would under-commit memory. It is for this reason that we are unable to arrive at a cost function which would reflect to a closer extent the cost of modifying pointer in the process of compaction. For the rest of this paper we therefore define the cost of compaction to be the ratio of number of words moved to the number of words requested, (i.e., the size of the request which prompted the compaction). Fortunately, minimizing the number of words relocated would at the same time minimize the number of busy blocks relocated to some extent.

COMPACTION

The operation of consolidating some or all of the free areas into fewer but larger free areas is called compaction. Compaction involves the relocation of some of the busy areas. In this paper we view compaction also as a mapping from one memory state (before compaction) to another (after compaction). The net effect of a compaction is that some larger free area is created as a result. In order to evaluate a compaction algorithm, it is necessary to define a cost associated with each application

of compaction. In this paper, we also analyze compaction algorithms in light of the cost function we just defined. Any compaction algorithm we propose in this paper will be in consideration of the cost. There is another criterion by which we must judge a compaction algorithm, namely the "instantaneous cost". Over a relatively lengthy period of time (i.e., over a lot of requests), there may have been only a few overflows, so the overall ratio of words moved to the number of words requested may be low. However, there may have been quite a few words relocated when the few overflows did occur. Thus without some knowledge of the ratio: number of words moved to the number of words requested immediately preceding the compaction, which we call the "instantaneous cost", we never know how much work is involved in doing a given compaction. While devising compaction algorithms with low costs, we should keep in mind given any request for some number (say x) of words, and overflow occurs, the algorithm should not relocate too many busy words relative to x . A low instantaneous cost tells how much compaction has to be done at any time before the compaction takes place. Fortunately, these two measures (cost and instantaneous cost) are compatible in that a low instantaneous cost automatically implies a low overall cost (although the converse is not true).

NOTATION AND DEFINITIONS

We view the entire memory as a vector $V = (v_i)$ of M components, each of which takes on non-negative integer values, where M is the memory size. If $v_i = v_{i+1} = \dots = v_k = p > 0$, and $v_{i-1} \neq p$, $v_{k+1} \neq p$, we say that the words i through k are occupied by (allocated to) the same request. We denote by $v_j = 0$ that the j^{th} word is free. The choice of $p \neq 0$ in any instance is not important, as long as it serves to distinguish one request from another. We will use α to denote an arbitrary allocation algorithm and g to denote an arbitrary compaction algorithm. We use β to denote some pair (α, g) , which we will call a memory management scheme. We also denote by an integer $q > 0$, a request for q words of memory, and by an integer $q < 0$, the release of those words of memory i through k such that $v_i = v_{i+1} = \dots = v_k = -q$. A request-release string Q is simply some (valid) concatenation of requests and releases. Given β , V , and q , we let

$r(\beta, V, q)$ denote the instantaneous cost of compacting memory with algorithm β , memory state V , and request q . Given β and Q , i.e., the algorithm and the entire request-release string, we let $K(\beta, Q)$ (or alternatively $K(\alpha, g, Q)$) denote the "overall" cost, i.e., the ratio of the number of words relocated throughout the entire sequence, of β working on Q . If Q is of infinite length, we use the limit as time approaches infinity instead. Finally we let $K^*(\beta)$, or alternatively $K^*(\alpha, g)$ be the maximum

$$K^*(\beta) = \max_Q \{K(\beta, Q)\}.$$

Definition 1: A memory state V is an n -dimensional vector which specifies the status (free or busy) of each memory word, if the memory size is n . A zero entry in the i^{th} component means that the i^{th} word is free. If $v_i = v_{i+1} = \dots = v_k = p \neq 0$ then words $i, i+1, \dots, k$ were all allocated to the same request.

Definition 2: An allocation algorithm α is a mapping $\alpha: V \times N \rightarrow V \cup \{\omega\}$ such that, if $q > 0$, $\alpha(V, q) = V' \neq \omega$ for some V' with $v_i = \dots = v_{i+q-1} = 0$ but $v_i' = \dots = v_{i+q-1}' = p \neq v_j$ for all j , and $p \neq 0$. For all other j such that $j < i$ or $j > i+q-1$, $v_j' = v_j$. If, on the other hand, for all i such that there exists a j , $i \leq j \leq i+q-1$, $v_j \neq 0$, then $\alpha(V, q) = \omega$. If $q < 0$ then $\alpha(V, q) = V'$ for some V' with $v_i' = 0$ if $v_i = -q$, and $v_i' = v_i$ if $v_i \neq -q$.

Definition 3: A compaction algorithm g is a mapping $g: V \times N \rightarrow V$. $g(V, q) = V'$ for some V' if and only if $\exists i$ such that $v_i = v_{i+1} = \dots = v_k \neq 0$, where $k \geq i$, $\exists c$ such that $v_{i+c}' = v_{i+1+c}' = \dots = v_{k+c}' = v_i'$, and furthermore, $\exists j, \ell$ such that $v_j' = \dots = v_\ell' = 0$ but not both $v_j = 0$ and $v_\ell = 0$.

Definition 4: A compaction algorithm g is a "complete" compaction if $g(V, q) = V'$ and $v_i' = 0$, $v_{i+1}' \neq 0$, for some i , then $\forall j > i$, $v_j \neq 0$.

In other words, a compaction g is a complete compaction if all the free areas are consolidated into one.

Definition 5: Given allocation and compaction algorithms α and g , we define the "composition" β to be as follows:

$$\beta(V,q) = \alpha(V,q) \text{ if } \alpha(V,q) \neq \omega, \text{ and}$$

$$\beta(V,q) = \alpha(g(V,q),q) \text{ if } \alpha(V,q) = \omega.$$

Definition 7: We extend the definitions of α , g , and β from mappings of $V \times N \rightarrow V$ to $V \times N^+ \rightarrow V$ by

$$\hat{\alpha}(V,q) = \alpha(V,q), \text{ and } \hat{\alpha}(V,Qq) = \alpha(\hat{\alpha}(V,Q),q).$$

Where there is no ambiguity, we use α to denote $\hat{\alpha}$. β is extended similarly.

Since we measure the performance of a compaction algorithm in connection with the associated allocation algorithm, we shall parameterize the problem by making some assumptions about the allocation algorithm, namely we assume that

- (i) the total number of busy words in memory at any time does not exceed m , and
- (ii) the size of each request does not exceed n .

Given an allocation α , a compaction g , and a specific request sequence Q , a certain number of words are relocated each time when g is called into action. We define the notion of "instantaneous cost" which relates the amount of compaction to be done at any overflow relative to the size of the request causing the overflow here:

Definition 8: We say a memory state V' is "reachable" from another memory state V by β if there exists a valid request sequence Q such that $\beta(V,Q) = V'$.

Definition 9: The ratio: $\frac{\text{number of words relocated}}{\text{size of req. causing ovfl.}}$ at any particular compaction, denoted $r(\beta,V,q)$, is called the "instantaneous cost" of β acting on request q (when the compaction in question took place), at a state V reachable from $\bar{0}$, the state where all of memory is free.

Definition 10: The instantaneous cost of β acting on Q is defined to be $r^*(\beta,Q) = \max_{q \in Q} \{r(\beta,q)\}$ if Q is finite, and $r^*(\beta,Q) = \limsup_{t \rightarrow \infty} \{r(\beta,q)\}$ if Q is infinite.

Definition 11: The uniform instantaneous cost of β is defined to be $r'(\beta) = \max_Q \{r^*(\beta,Q)\}$.

Thus the instantaneous cost $r^*(\beta,Q)$ tells us how much compaction may be involved for any word requested in the sequence Q , whereas the uniform instantaneous cost $r'(\beta)$ gives an index to how well the pair (α,g) performs with respect to

limiting the words relocated for any one word requested. Besides the uniform instantaneous cost, we need some overall measure of the performance of β . This overall measure should reflect the percentage over-head time involved in compaction. Thus we define a cost function $K()$ associated with each β and Q to be:

Definition 12: $K(\beta,Q) = \lim_{t \rightarrow \infty} \frac{\text{number of words relocated}}{\text{number of words requested}}$.

While the cost function $K()$ measures the cost of doing allocation and compaction for Q using β , we would like to have a measure of efficiency on β uniformly, i.e., over all Q . A natural choice for worst case analyses would of course be the worst possible Q . So we define the uniform cost $K^*(\beta)$ to be:

Definition 13: $K^*(\beta) = \limsup_{t \rightarrow \infty} \frac{\text{number of words relocated}}{Q \text{ number of words requested}}$.

Finally, we want to single out those compaction strategies which preserve "locality" in some sense of the word, since, as experiments [CV] show that preserving locality is one of the most important considerations for efficient execution of programs. Thus we define

Definition 14: A compaction algorithm g is locality-preserving if, for some state V , request q , $g(V,q) = V'$ such that for all i, j with $v_i = v'_\ell \neq 0$, $v_j = v'_k \neq 0$, $v_i \neq v_j$. $i < j$ implies $\ell < k$.

In other words, a locality-preserving compaction algorithm is one which "slides" busy blocks rather than "picking up" a busy block and places it somewhere else beyond its neighboring busy blocks. The conventional compaction algorithm is therefore a complete, locality-preserving compaction strategy.

THE EFFICIENCY OF THE CONVENTIONAL COMPACTION

We show a theorem here which states that the conventional compaction algorithm is grossly inefficient (in the worst case sense).

Theorem 1: $K^*(\alpha,g^*) \geq \frac{m-2}{3}$ for all conservative α .

Proof: (We shall use g^* to denote the conventional compaction). To establish the upper bound, it is sufficient to demonstrate a request sequence Q

which has a cost $K(\alpha, g^*, Q)$ of at least $\frac{m-2}{3}$ for all conservative α . Recall that an allocation is conservative if it does not fragment memory unnecessarily. Now consider the following request-release sequence:

- 1) Make m requests for one word each.
- 2) Release the two words on the two edges.
- 3) Make a request of 2 words.
- 4) Release the two-word block requested in 3.
- 5) Make a request of one word.
- 6) Release the one word block on an edge not adjacent to a "gap".
- 7) Repeat steps 3 through 6 indefinitely.

It is obvious that after step 2 there are two free words which are not adjacent to each other. Furthermore, there are $m-2$ busy words between them. The two-word block requested in step 3 will thus cause an overflow and then the $m-2$ busy words must therefore be moved. The one word block requested in step 5 will fill one of the two empty spaces vacated by the release in step 4. After step 6, memory will look exactly the same as it did after step 2. The repetition will therefore force the cost to be $(m-2)/3$. Illustration in Figure 1 will help make this analysis clearer. $\square$

```

after 1  |X X X X X ..... X X X X X|
after 2  |_ X X X X ..... X X X X _|
after 3  |X X X X X ..... X X X Y Y|
after 4  |X X X X X ..... X X X _ _|
after 5  |X X X X X ..... X X X X _|
after 6  |_ X X X X ..... X X X X _|

```

Figure 1

Corollary: $r'(\alpha, g^*) = (m-2)/2$.

Proof: We merely observe that a block of 1 cannot cause an overflow, so it takes a request of size at least 2 to cause overflow, and thus the total number of busy words in memory cannot exceed $m-2$. $\square$

MOTIVATION

If we inspect the proof of Theorem 1 closely, we would discover that the way to introduce a high cost is for a request sequence to somehow place a single large block in the "middle" of memory. A small request would then force the relocation of the large block. If the allocator can keep large blocks

on the edges, such relocation involving a lot of words would not be possible. Since it is impossible to define what a "large" block is, a viable solution would be therefore to keep busy blocks in such a way as to have the smallest blocks in the middle.

Algorithm A: When a request for s words is presented, search memory from both ends and find the first memory word to the "inside" of the last busy block whose size is at least s . Allocate the s words starting at the word chosen above going inwards to the request. If any of the s words so allocated is busy, i.e., occupied by other (necessarily smaller than s) busy blocks, relocate them by making requests for their sizes recursively.

Proposition 2: Algorithm A eventually stops with the request properly allocated.

Proof: When the request arrives, either there are s free words to the inside of the last busy block whose size is at least s or not. If there are, the request is satisfied with the s free words without further action. If there are not, just enough busy blocks are removed to make room. Since these removed blocks are each smaller than s , eventually all requests (generated by the removals) have to be satisfied. $\square$

We now consider the cost of this algorithm. Given sufficiently large n , suppose all sizes from 1 to n are allowed and assume that sufficiently many busy blocks of sizes 1 through $n-1$ occupy both ends of memory with the only free area in the middle. Suppose now that the next request is for a block of n words. Since for any $i > 2$, $i-1 \nmid i$, the request for n words will result in the removal of 2 blocks of size $n-1$ each. The 2 blocks of size $n-1$ will in turn cause the removal of 3 blocks of size $n-2$ which will finally cause the removal of blocks of size 2. Hence for the block of size n , a total of $2(n-1)+3(n-2)+\dots+(n-1)(2)$ words would have to be relocated. But $2(n-1)+3(n-2)+\dots+(n-1)(2)$

$$\begin{aligned}
 &= (2n+3n+\dots+(n-1)(n)) - (2 \times 1 + 3 \times 2 + \dots + (n-1) \times (n-2)) \\
 &= \sum_{i=2}^{n-1} n \cdot i - \sum_{i=2}^{n-1} i(i-1) \\
 &= (n+1) \sum_{i=2}^{n-1} i - \sum_{i=2}^{n-1} i^2 \\
 &= (n+1) \left(\frac{n(n-1)}{2} - 1 \right) - \frac{1}{3} n(n-\frac{1}{2})(n-1) - 1
 \end{aligned}$$

$$\begin{aligned}
&= \frac{1}{2} \left(\frac{n^3}{3} - n \right) - (n+1) - \frac{n^3}{3} + \frac{n^2}{2} - \frac{n}{2} - 1 \\
&= \frac{n^3}{6} + \frac{n^2}{2} - \frac{5n}{3} - 2.
\end{aligned}$$

So a total proportional to n^3 words are to be relocated. Although this compares favorably with any algorithm which has to move a total proportional to m , rather than n , words, it is not as desirable as something proportional to, say, n . We investigate this algorithm further with some restrictions in hope to reduce the cost somewhat.

Algorithm B: For a request of size s , where $c^{i-1} < s \leq c^i$, for some constant c , a request of size c^i is generated instead, and the algorithm proceeds as defined in Algorithm A.

With this restriction (or rounding up), we see that we never have to remove more than c^i words for a request of size c^i . We summarize the effect of this modification in the following theorem:

Theorem 3: $r(\alpha, g, q) \leq q \log_c q \quad \forall q$ where (α, g) is the algorithm described in Algorithm B.

Proof: At the overflow due to a request for q words, where $q = c^k$, at most q words are removed at the current recursion. Since all blocks are of size c^j for some j , it takes at most k recursions to reach blocks of size 1. So at most qk words have to be moved. But $qk = q \log_c q$, and the theorem is proved. $\square$

Corollary: $K^*(\beta) \leq \log_c n$

Proof: This follows directly from the theorem and the definition of $K()$. $\square$

In any case, the performance of the algorithm is now bounded by a function of q , the current request size, instead of m , the maximum number of words needed, as in the case of the conventional compaction. In the restrictive case, the instantaneous cost is bounded from above by $\log_c q$, which is quite an improvement over the non-restrictive case for which the instantaneous cost is of order q^2 . The intuitive idea behind Algorithm B is that the work of compacting memory is "spread around" at almost every request so there will be no unexpected large-scale compaction due to small requests. It is well worth pointing out that the price of requiring each request size to be a power of some constant c

is not altogether so unusual. After all, the buddy system requires rounding up to the next power of 2.

A comment on the choice of c may be appropriate here. It is evident that as c increases, $\log_c n$ decreases. But with n fixed, the number of choices of permissible request sizes decreases exponentially as c increases. Another way of looking at it is that the amount of "internal fragmentation" increases as c becomes large.

Note that this algorithm employs a non-conservative allocation algorithm (because it calls the compactor even when there may be a free area large enough to hold the current request) as well as a non-locality-preserving compaction algorithm. (If the compactor preserves locality, much more than $q \log_c q$ words may have to be relocated.) Observe also that the function $\log_c n$ increases as n increases, while the bound $m-n$ applied for any compaction algorithm. There must be a point beyond which the $\log_c n$ bound is no longer applicable. The question is then this: Does this point actually coincide with $m-n$ for some value of n ? We notice, however, that we can "approach" the middle of memory from both edges. In other words, we can "choose the better side" in which to do our compaction. In this case, the obvious bound would be $\frac{1}{2}(m-n)$ as opposed to $m-n$, for a savings of a factor of two.

We observe that at the expense of generality, and possibly some running time, we can indeed achieve the goal of being able to predict the cost of compaction in advance of the compaction process. Furthermore, we notice that Algorithm B is not a locality preserving compaction algorithm. So the trade-off is not between time and space, but rather a choice between whether a "large" compaction or lots of "small" compactations is more desirable. We now investigate the trade-off between time and space, i.e., we try to minimize compaction efforts at the expense of more memory (with respect to m).

When the management algorithm is given a request for s words, and all free areas in memory are of sizes less than s , it is conceivable that we may get away with relocating just a few busy words to make enough room for the incoming request. Furthermore, it seems natural that the place to do the compaction is where busy blocks are the most

sparse. We can thus give a preliminary algorithm using these ideas:

Algorithm C: Let s be the current request size, km be the memory size where $k > 1$, α be any conservative allocation algorithm, and V be the current memory state. If $\alpha(V, s) = \omega$ then divide memory into $m(k-1)/s$ regions of size $ks/(k-1)$ each. (So the total size is km). We consolidate all the free areas in the region with the least number of busy words and leave all the other regions intact. After this step we then call on α to do the allocation.

We must now show that this allocation-compaction algorithm succeeds in allocating an area to the current request.

Lemma 4: There is at least one region (as described in Algorithm C) with at least s free words.

Proof: If the current request size is s , consistent with our assumption that there are at most m busy words in memory at any time, there can only be less than or equal to $m-s$ busy words in memory at the time of the current request. Thus the overall density of busy words in memory is at most $(m-s)/km$. Since all of the $m(k-1)/s$ regions are of equal size, namely $ks/(k-1)$, there must be at least one such region with density at most $(m-s)/km$. So there are no more than

$$\frac{m-s}{km} \cdot \frac{ks}{k-1} = \frac{m-s}{m} \cdot \frac{s}{k-1} < \frac{m}{m} \cdot \frac{s}{k-1} = \frac{s}{k-1}$$

busy words in this particular region. Hence there are at least $\frac{ks}{k-1} - \frac{s}{k-1} = s \cdot \left(\frac{k-1}{k-1} \right) = s$ free words in the region. Since our algorithm consolidates all free areas (which total at least s words) into one, after the compaction there is a free area of size least s in this region. Since the allocation algorithm α is conservative, it will allocate (part of) this free area to the request, and thus completing its job successfully. $\square$

Corollary: There is a region (as described in Algorithm C) with at most $s/(k-1)$ busy words.

Proof: Since there is at least one region with at least s free words, and the size of the region is $sk/(k-1)$, so there are at most $\frac{sk}{k-1} - s = \frac{s}{k-1}$ busy words in the region. $\square$

Since all the relocation activities take place in

this region, no more than $s/(k-1)$ busy words are relocated. So we have

Theorem 5: $K(\alpha, g) \leq r'(\alpha, g) \leq \frac{1}{k-1}$ where α is any conservative allocation algorithm, g is the compaction algorithm described in Algorithm C, and km is the memory size, where $k > 1$.

Proof: The theorem follows directly from Lemma 4 and the definitions of $r^*(\cdot)$ and $K(\cdot)$. $\square$

Note that if we choose $k=2$, then for any request of s , we need only to relocate no more than s words. In addition to the advantage of being a locality-preserving compaction strategy, the algorithm has the definite edge on costs. It not only is independent of m , but is a very nice linear function of the current request size only.

To summarize the paper, we gave a mathematical model to the process of memory management and we studied how various desirable properties of memory management algorithms can be achieved. Note that the author is not of the opinion that Algorithms B and C should be implemented in every memory manager. Rather than proposing Algorithms B and C as useful algorithms per se, we merely used them to study the relationships among various parameters in the allocation--compaction process.

REFERENCES

- [Kn] Knuth, D. E., "The Art of Computer Programming", V1, Addison-Wesley, Reading, Mass., 1968.
- [LW] Lang, B. and Wegbreit, B., "Fast Compactification", Report No. 25-72, Center for Research in Computing Technology, Harvard University, Nov. 1971.
- [Ro] Robson, J. M., "An Estimate of the Store Size Necessary for Dynamic Storage Allocation", JACM 3 (1971), pp. 416-423.
- [CV] Coffman, E. G., Jr. and Varian, L. C., "Further Experimental Data on the Behavior of Programs in a Paging Environment", CACM 11, 7 (July 1968), pp. 471-474.