# A Note on Storage Fragmentation and Program Segmentation

B. RANDELL

*IBM Thomas J. Watson Research Center*

*Yorktown Heights, New York*

**The main purpose of this paper is the presentation of some of the results of a series of simulation experiments investigating the phenomenon of storage fragmentation. Two different types of storage fragmentation are distinguished: (1) external fragmentation, namely the loss in storage utilization caused by the inability to make use of all available storage after it has been fragmented into a large number of separate blocks; and (2) internal fragmentation, the loss of utilization caused by rounding up a request for storage, rather than allocating only the exact number of words required. The most striking result is the apparently general rule that rounding up requests for storage, to reduce the number of different sizes of blocks coexisting in storage, causes more loss of storage by increased internal fragmentation than is saved by decreased external fragmentation. Described also are a method of segment allocation and an accompanying technique for segment addressing which take advantage of the above result. Evidence is presented of possible advantages of the method over conventional paging techniques.**

**KEY WORDS AND PHRASES: storage allocation, storage fragmentation, paging, segmentation, addressing**
**CR CATEGORIES: 6.2,6.20**

## Introduction

The performance of many current computer systems is critically dependent on the level of utilization of execution storage that can be maintained. The present paper views effective storage utilization as involving (i) accommodating as many requested sets of information (data arrays, subroutines, etc.) concurrently in storage as possible, and (ii) making as much use of these sets of information as possible.

In recent studies [1, 6] of storage utilization achieved by using paging techniques, the above two factors have not been distinguished. In fact, it has been assumed that all information has been allocated names in a large linear name space [10] and that the original structuring of information into data arrays, etc., is no longer explicit. Therefore the storage management system is neither presented with explicit requests to fetch a particular set of information, nor can it interpret an attempted access to an individual item of information as such a request. The above studies have been based on simulations of the behavior in a paging environment of programs and data whose allocation in a linear name space has been performed by already existing compilers. The results of such simulations of course depend not only on the paging system, but also on the method of compilation. In fact it has become apparent that the way in which a linear name space is allocated can be crucially significant to system performance (see for example Comeau [2]).

The distinction between effects caused by the compiler and effects due to the storage management system is more obvious in systems which permit use of a segmented name space [10]. The reason of course is that an explicit decision is made by the compiler as to which set of items of information (or segment) a particular item should belong. This is true even though the decision is often a simpler one than that involved in allocating names within a linear name space, because it merely reflects the way in which a programmer structured his program and data.

The present paper is concerned with only the first of the two factors described above as being involved in storage utilization. The effects caused by the way a programmer structures his program and data and the way in which a compiler uses this information are not considered. Instead it is assumed that sets of information have been identified as segments, and that the problem is to accommodate as many of these segments as possible in storage. Questions as to the use made of segments once allocated in core storage are explicitly ignored. Some of the results obtained from a recent set of simulation experiments (programmed using APL/360 system [5]), in which various storage allocation techniques were modeled are described.

## Storage Fragmentation

The original intent of the simulation experiments was to investigate the various factors which might affect storage fragmentation. Here storage fragmentation (also called fracturing or checker-boarding) refers to the fragmenting of available storage into a large number of blocks separated by occupied (unavailable) areas. This phenomenon is statistical in nature and occurs when blocks of various sizes are being allocated and then de-allocated after varying amounts of time. Fragmentation will affect system performance when each of the available blocks is too small to serve a waiting request for storage, although the total extent of available storage is sufficient. When this happens, one has the choice of tolerating the decreased storage utilization or of expending system resources in order to reorganize the contents of storage so as to coalesce the separate blocks of available storage. The appropriate choice of action depends on the extent to which storage is a critical system resource.

The simulation model was quite abstract. Storage was represented by a list of block sizes, each accompanied by an indication of whether the block was occupied, and if so, when it would be released. Adjacent entries in the list represented physically adjacent blocks of storage. Requests for storage were characterized by amount of storage required and length of time required, both of these values being chosen from statistical distributions. No attempt was made to model the use of these blocks of storage, and it was assumed that there were always requests waiting for storage (the queue of requests was regarded as being ordered-there was no "cutting in" of smaller requests when the first-in line would not fit). The model was thus intended to investigate the effect of various parameters on storage utilization as measured simply by the amount of storage occupied by allocated requests.

One of these parameters indicated which placement algorithm (i.e. a strategy for deciding where a request for storage should be allocated) was to be used in a particular experiment. Three separate placement algorithms were included in the model:

(i) MIN–an algorithm which maintained a list of available blocks, ordered in increasing size. Each request would be allocated from the smallest available block that was of sufficient size, with no attempt being made to move blocks of occupied storage in order to collect together separate fragments of available storage.

(ii) RANDOM–the choice would be made at random from the entire set of available blocks that were large enough. Again there would be no attempt to move occupied blocks.

(iii) RELOC–each time an occupied block became available, the blocks to its right would be moved so as to keep the available space always in one contiguous block at the end of storage. Each request would be allocated using words from the start of this available block.

It will be seen that the RELOC algorithm achieves the highest possible degree of storage utilization. Any loss of utilization that does result is due to a request which cannot possibly fit into storage with those currently there, even though these requests do not exactly completely fill the store. (Assuming requests are for varying lengths of time, full utilization could only be achieved if each request was for the same fixed fraction of total storage.)

The difference in utilization shown by an experiment using either MIN or RANDOM and by an otherwise identical experiment using RELOC is due to the fragmentation that these nonrelocating placement algorithms cause. For reasons that become clear below, we choose to call this difference in utilization external fragmentation. The particular algorithms MIN and RANDOM were chosen as being probably near the two extremes of the spectrum of likely nonrelocating placement algorithms, as regards both complexity and level of performance. The RELOC algorithm was chosen, not as a particularly practical algorithm, but for its value in indicating the maximum possible gains that could be achieved by using relocation.

The group of experiments which are of interest here involved an investigation of the effects of rounding up requests for storage to the nearest multiple of a given quantum of allocation $Q$, in order to assess the effect of reducing the number of different sizes of blocks coexisting in storage. (Figure 1 represents a store which has been so allocated.) Quite predictably, as $Q$ was increased in size the amount of external fragmentation decreased. On the other hand, a new cause of loss of storage utilization was introduced-namely that wasted within each block by the process of rounding up the request. The second type of loss of utilization we term internal fragmentation. (It should be noted that the extra space used within each block is regarded as being wasted because it is assumed that the exact segments required are known. Thus there is no chance of accidental benefit in bringing more information into core storage than was requested, in case it might be used in the near future.)

Figures 2 and 3 are representative samples of the type of results obtained. Figure 2 was obtained from an experiment using an exponential distribution of storage request sizes, with a mean of 1024 words, the memory size being 32768 words. (In all the experiments described here, the time required in core storage by a given request was chosen at random from the integers 1, 2, . . . , 5.) Figure 3 shows the results of a similar experiment but using the "SDC" distribution of storage request sizes. (This is a distribution which generates requests so as to approximate to the experimental observations of program sizes on the SDC Time Sharing System, given by Totschek [11]). Similar results were obtained, using both MIN and RANDOM, with various means and distributions of storage request sizes.

All the results show the unexpected (to the author at least) fact that as $Q$ increases the loss of utilization due to increased internal fragmentation distinctly outweighs the gain due to decreased external fragmentation. This is a very interesting result, though of course analytical confirmation, as well as determination of the region of validity of the result, is sorely lacking. (In fact very little analytical work has been done in this area at all, although Knuth [7] gives an analysis of the internal fragmentation caused by rounding up requests to the next power of two.) The result is of particular interest in relation to the various current strategies of implementing program segmentation, and in fact prompted consideration of a form of segment allocation, which we term "partitioned segmenting," described below.

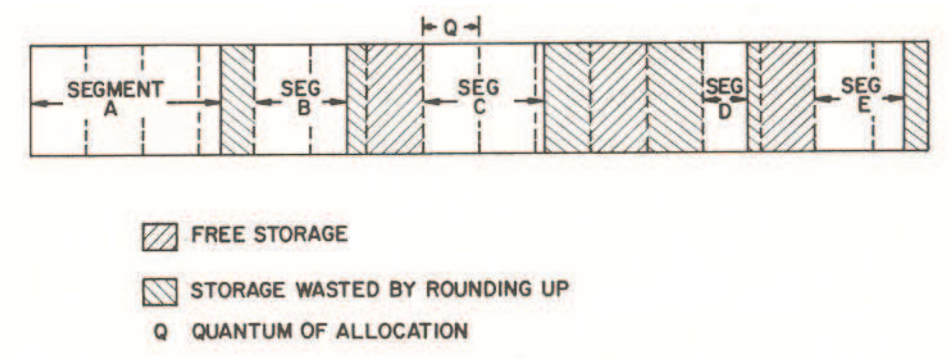

FIG. 1. A fragmented store

## Segment Allocation

There are two well-known methods of storage allocation in systems which provide a segmented name space.

The method used for example in the B5000 [13], which we term simple segmenting, allocates each segment in a single block of storage, of a size appropriate to the extent of the segment. The contiguity of words within a segment that is apparent to a programmer is provided simply by an underlying physical contiguity of words. Therefore a strict limit on the maximum size of segments is enforced (at most the actual memory size, though typically some very small fraction of the memory size).

On the other hand, paged segmenting techniques, such as used in the S360/Model 67 [3], provide segments which are an integral number of pages and regard physical storage as being logically equi-partitioned into a set of page frames. There is then no necessity to allocate the entirety of a segment to storage at anyone time. The segment can in fact be much larger than the size of physical storage. The apparent contiguity of items within a segment is provided using an address mapping mechanism.

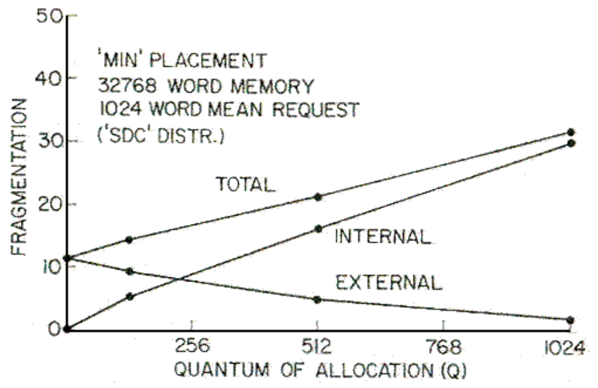

FIG. 2. The effects of rounding up storage requests (exponential distribution)

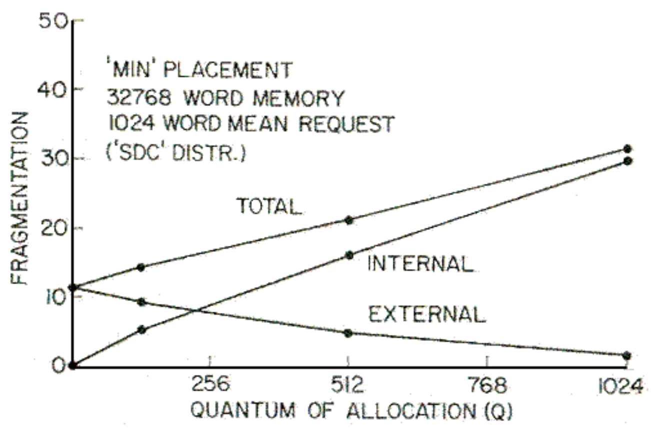

FIG. 3. The effects of rounding up storage requests ('SDC' distribution)

The results typified by Figure 2 are of relevance to a simple segmenting scheme and provide evidence of the desirability of allocating exactly the required number of words to each segment. However the strict limit on segment size that is inherent in simple segmenting is undesirable. These observations prompted the modification of the simulation model to permit investigation of a variant of paged segmenting, called partitioned segmentation, in which it was not necessary to round up each segment to a multiple of the page size.

## Partitioned Segmenting

As before, $Q$ is the quantum of allocation, i.e. the accuracy with which segment requests are honored. Thus the size $N$ of a segment will be rounded up to the nearest multiple of $Q$ to form $N_Q$, which will be the total number of words allocated to the segment. It is assumed that $Q$ is a power of two.

We define page size $P$ as the largest size of block of physically contiguous words of storage that will be provided in honoring a request for fetchinv a segment. $P$ will be a power of two and will be greater than or equal to $Q$. A segment of $N$ words will be allocated as $[N/P]$ blocks of $P$ words, plus a block of $N_{PQ}$ words, where $N_{PQ} = N_Q \bmod P$.

The storage allocation mechanism will therefore have to respond to requests for blocks of contiguous storage of various different sizes. The number of different sizes will in fact be $s = P/Q$.

Thus the situation as regards storage allocation is no different from the case with simple segmentation where the maximum permitted extent of a segment is $P$. Since there is no insistence that each block of $P$ words be placed on a location in storage whose first address is a multiple of $P$, any of the standard placement algorithms can be used. Similarly, the problems of allocating backing storage and arranging for information transfer between backing storage and working storage are exactly those encountered in existing simple segmenting schemes.

A series of experiments were performed to investigate the performance of the partitioned segmenting technique. The results are shown in Figures 4–6. (It should be noted that these figures show plots of storage utilization directly rather than as one of the causes of loss of utilization, as in Figures 2 and 3). For example Figure 4 shows the effects of varying the quantum of allocation $Q$ from 32 to 1024, with a page size $P$ of 1024, in a 32768 word memory. An exponential distribution of segment sizes, with a mean of 1024, was used. It will be seen that all three placement algorithms have the same performance when the quantum of allocation $Q$ is set equal to the page size $P$

(when paged segmenting is being simulated). As *Q* is decreased, all the placement algorithms achieve significantly increased utilization, with RELOC naturally performing better than MIN, which in turn is better than RANDOM. (For example, storage utilization increased from 63 to 92 percent with MIN.)

Figure 5 shows the results obtained using the MIN algorithm with two other distributions of segment size, namely a uniform distribution and the SDC distribution referred to earlier. It will be seen that there is very little difference between the results from the various distributions, particularly for small values of *Q*. There is however slightly more variation when ordinary paging is occurring, with storage utilization varying between 63 and 72 percent, according to the particular distribution used.

Finally, Figure 6 shows that the amount of increase of storage utilization achieved by the partitioned segmenting scheme over a paged segmenting scheme depends critically on the mean segment size. Three sets of experiments are shown (one repeated from Figure 2) all using the MIN algorithm, an exponential distribution of segment sizes, and a 32768 word memory. With *Q* set equal to *P*, so as to simulate paging, storage utilization varies from 55 percent when the mean request is half the page size to 89 percent when it is four times the page size. In all three cases partitioned segmenting shows an improvement, which increases steadily as *Q* is decreased. Obviously, however, the scope for improvement is diminished when the mean request size is large enough, relative to the page size, for paging to be fairly efficient.

In actual practice, because of the overhead involved in storing and processing page tables, few designers have been prepared to choose a single page size of much less than 1024 words. One's view of what is a likely mean segment size depends on one's conceptions of how programs and data are structured and how compilers should use this structuring. Designers of paged segmenting schemes naturally tend to expect segments to be large relative to the page size. On the other hand, many segments in the B5000 system and its successors have been quite small, with the mean size probably being of the order of 500 words. In fact McKeeman [9] refers to an analysis of compiled scientific programs in which the mean segment size was 60 words. However, he stressed that this was not a well-defined mean, and that segment sizes varied considerably. It is this latter point which is probably the most important. Compilers and programming conventions are likely to have a considerable effect on the mean segment size but less likely to remove the problem of the great variation in sizes. (An example of the dilemma that designers face in choosing a page size is illustrated by the decision in MULTICS to have two kinds of pages, of 64 and 1024 words respectively [4].)

## Segment Addressing

The results described above provide an obvious motivation for investigating means of storage addressing in a partitioned segmenting system.

The standard method of storage addressing in a simple segmenting system such as the B5000 involves one level of mapping using a program reference table (PRT). Each entry in the PRT indicates the address of the first word of a segment, the size of the segment, and whether the segment is currently in execution storage. If desired, recently used PRT entries can be kept in a small associative memory, in order to reduce the average access time. This is done for example in the B8500 [8].

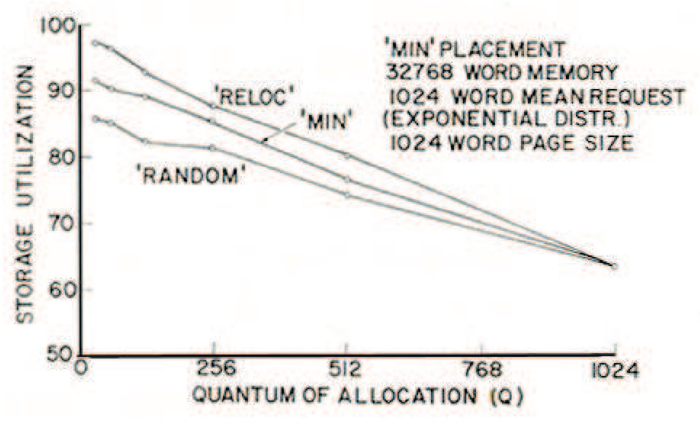

FIG. 4. Partitioned segmenting – a comparison of the three placement algorithms

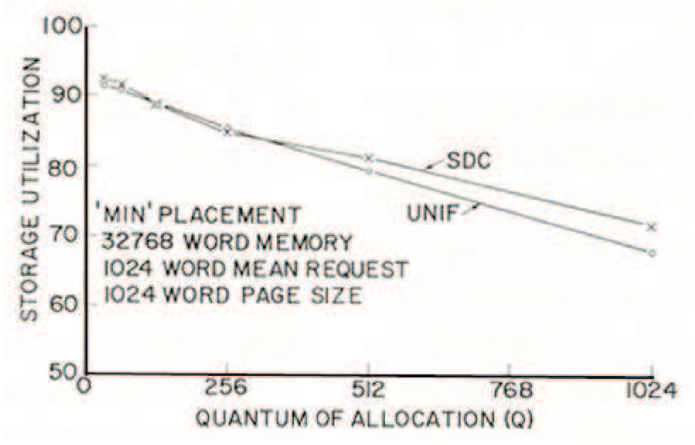

FIG. 5. Partitioned segmenting –'SDC' and uniform request distributions

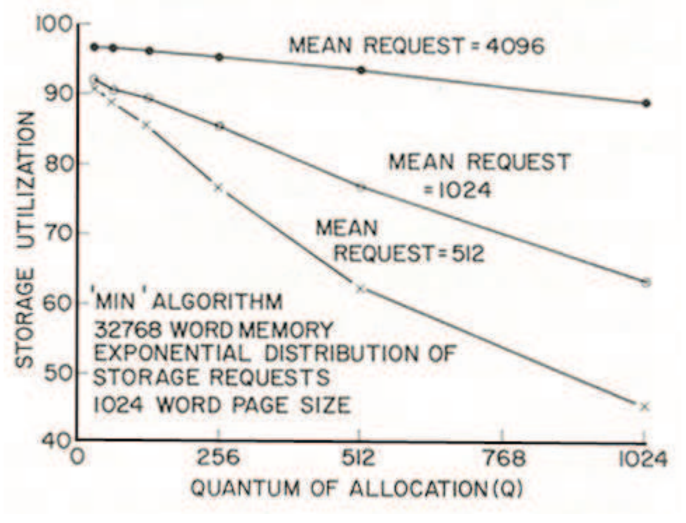

FIG. 6. Partitioned segmenting – the effect of varying the mean request size

The method of storage addressing used in paged segmenting systems such as MULTICS and the S360jModei 67 involves two levels of mapping. The first mapping corresponds to the mapping used in a simple segmenting system but produces a page table address. The second mapping obtains a word address from the page table. Again small associative memories are used to reduce the average access time.

The proposed mechanism for storage addressing in a partitioned segmenting system is a development of the above two addressing methods. Two levels of indirection are used in the case that a segment is larger than the page size *P*. Otherwise only a single level is used, as in a simple segmenting system.

The format of PRT entries is basically as it is in a simple segmenting system. For convenience, an extra single-bit field is used to indicate whether the segment is larger than the page size and hence has been allocated as a set of separate blocks or has been allocated as a single block of up to *P* words. (This information is also given implicitly by the field which indicates the segment size.)

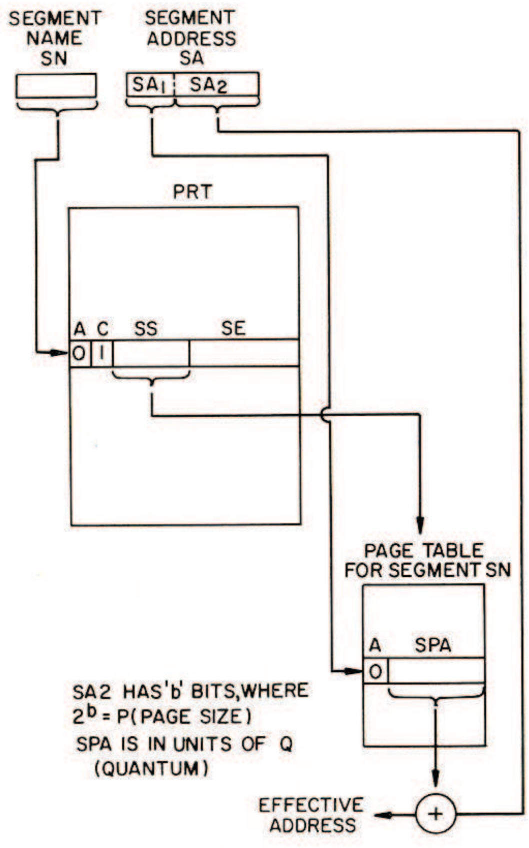
SEGMENT NAME SN
SEGMENT ADDRESS SA
SA1
SA2
PRT
A
C
SS
SE
0
1
PAGE TABLE FOR SEGMENT SN
A
SPA
0
SA2 HAS 'b' BITS, WHERE 2^b = P (PAGE SIZE)
SPA IS IN UNITS OF Q (QUANTUM)
EFFECTIVE ADDRESS
+

FIG. 7. The addressing mechanism (segment larger than the page size)

Figure 7 shows the addressing technique for the case of a segment whose PRT element indicates, by a nonzero C field, that the segment has been allocated as a set of separate blocks. (For comparison, the addressing technique for the case of a segment which is smaller than the page size is shown in Figure 8.) The sequence of actions is as follows:

(i) The field SN, segment name, is used to access the appropriate element of the PRT.

(ii) The address SA of the item in segment SN is checked against the extent SE of the segment, given in the PRT element.

(iii) The field SS in the PRT element is added to the field SA1 of SA (SA is considered as comprising two parts SA1 and SA2, the number of bits in SA2 being *b*, where the page size $P = 2^b$).

(iv) The result is used to access the appropriate entry of the page table corresponding to segment SN.

(v) The contents of this page table entry, regarded as being in units of *Q*, are *added* to SA2 to form the effective address.

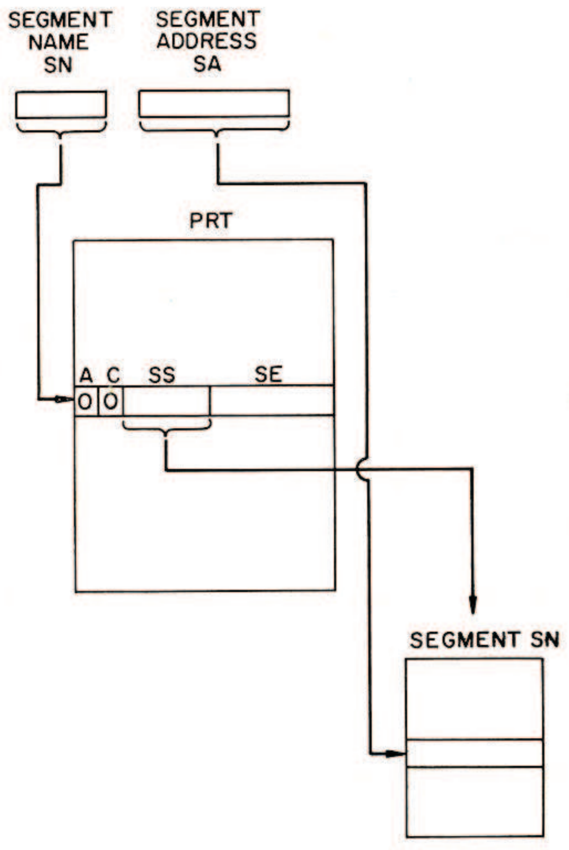

FIG. 8. The addressing mechanism (segment within page limitations)

The above sequence of actions describes the simple case when both the page table and the appropriate block of a segment are available in working storage (the *A* fields in the PRT element and page table entry indicate this fact). As in other systems such as the S360/Model 67 a small associative memory could be used to reduce the average accessing time.

The addressing technique differs from the typical two levels of indirection used in paged segmentation systems, since an addition rather than a concatenation is used to form the final effective address. In fact the addition might well be performed by special purpose hardware designed to take advantage of the fact there can be only a partial overlap in nonzero digits between the two factors. The overlap is at most m bits, where $2^m = P/Q$.

A further difference is that the last entry in the page table might point to a block of storage which is less than *P* words in extent. This will cause no problem because SA will already have been checked against SE, so an access to this last block of words is bound to be within limits.

## Alternative Mapping Mechanisms

It turns out that the mapping mechanism provided on the CDC 3300 [12] has several similarities to the one described above. Storage is divided into page frames (of 2048 words), which are further subdivided into quarters. Blocks of one quarter, one half, three quarters, or a complete page in size can be allocated. It so happens that only a very limited number of segments of somewhat limited size are provided. Therefore the actual mapping involves one level of mapping through a complete page table, maintained in a special fast core memory. Each page table entry indicates the page frame to which a block has been allocated and a 2-bit number representing the starting location (measured in quarter pages) of the block in the page frame. This 2-bit number is added to the appropriate 2 bits of the programmer-specified address within a segment as part of the formation of the effective address. However no carry is permitted into higher bit positions of the effective address. Thus for example a three quarter page block could be allocated within a page frame, starting in the last quarter of the frame.

Thus the basic difference between this technique and the method incorporated in the partitioned segmenting scheme is that, at the expense of a somewhat more complicated method of allocating and relocating storage, the possibility of a lengthy carry propagation during the formation of an effective address is avoided in the CDC 3300.

## Conclusions

The above study has attempted to differentiate between the effects on system performance caused by the way a program and its data are structured and the actions of the dynamic storage allocation system which is used during program execution. Some evidence, obtained from simulation experiments, has been given of the undesirability of rounding up segment sizes to an integral number of page sizes in an attempt to remove storage fragmentation. Finally, descriptions are given of segment allocation and addressing mechanisms for a system of partitioned segmenting, whose investigation was prompted by this experimental evidence.

*Acknowledgments*. The author is grateful for many useful discussions with L. A. Belady, C. J. Kuehner, M. Lehman, R. W. O'Neill, and F. W. Zurcher.

## REFERENCES

1. BELADY, L. A. A study of replacement algorithms for a virtual-storage computer. *IBM Systems J.* 5, 2 (1966), 78-101.
2. COMEAU, L. W. A study of the effect of user program optimization in a paging system. ACM Symposium on Operating System Principles, Oct. 1-4, 1967, Gatlinburg, Tenn.
3. COMFORT, W. T. A computing system design for user service. Proc. AFIPS 1965 Fall Joint Comput. Conf. Vol. 27, Pt. 1, Spartan Books, New York, pp. 619-628,
4. CORBATO, F. J. AND VYSSOTSKY, V. A. Introduction and overview of the Multics system. Proc. AFIPS, 1965 Fall Joint Comput. Conf., Vol. 27, Pt. 1, Spartan Books, New York, pp. 185-196.
5. FALKOFF, A. D, AND IVERSON, K. E. The APL/360 terminal system. ACM Symposium on Interactive Systems for Experimental Applied Mathematics, Washington, D.C., Aug. 26-28, 1967.
6. FINE, G. H., JACKSON, C. W. AND MCISAAC, P. V, Dynamic program behavior under paging. Proc. ACM 21st Nat. Conf. 1966, Thompson Book Co., Washington, D.C., pp. 223-228.
7. KNUTH, D. E. *The Art of Computer Programming, Vol. 1.– Fundamental Algorithms*. Addison Wesley, Reading, Mass., 1968, p. 448.
8. MCCULLOUGH, J. D., SPEIERMAN, K. H., AND ZURCHER, F. W. A design for a multiple user multiprocessing system. Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27, Pt. 1. Spartan Books, New York, pp. 611-617.
9. MCKEEMAN, W. M. Language directed computer design. Proc. AFIPS 1967 Fall Joint Comput. Conf., Vol. 31, Thompson Book Co., Washington, D.C., pp.413-417.

10. Randell, B. and Kuehner, C. J, Dynamic storage allocation systems. Comm. ACM 11, 5 (May 1968), 297-306.
11. Totschek, R, A. An empirical investigation into the behavior of the SDC time-sharing system. Rep. SP 2191, System Development Corp., Santa Monica, Cal., 1965,  AD 622 003.
12. 3300 Computer System Reference Manual. Pub. No. 60157000, Control Data Corp., St, Paul, Minn., 1966.
13. The Descriptor–a definition of the B5000 information processing system. Burroughs Corp., Detroit, Mich., 1961.