Programming Techniques

R.L.Rivest,* S.L.Graham
Editors

# Dynamic Memory Allocation in Computer Simulation

Norman R. Nielsen
Stanford Research Institute

**This paper investigates the performance of 35 dynamic memory allocation algorithms when used to service simulation programs as represented by 18 test cases. Algorithm performance was measured in terms of processing time, memory usage, and external memory fragmentation. Algorithms maintaining separate free space lists for each size of memory block used tended to perform quite well compared with other algorithms. Simple algorithms operating on memory ordered lists (without any free list) performed surprisingly well. Algorithms employing power-of-two block sizes had favorable processing requirements but generally unfavorable memory usage. Algorithms employing LIFO, FIFO, or memory ordered free lists generally performed poorly compared with others.**

**Key Words and Phrases: algorithm performance, dynamic memory allocation, dynamic memory management, dynamic storage allocation, garbage collection, list processing, memory allocation, memory management, programming techniques, simulation, simulation memory management, simulation techniques, space allocation, storage allocation**

**CR Categories: 3.74, 4.49, 5.25, 8.1**

* This paper was submitted prior to the time that Rivest became editor of the department, and editorial consideration was completed under the former editor, G. K. Manacher.

Author's address. N.R. Nielsen, Information Systems Group, Stanford Research Institute, Menlo Park, CA 94025. This research was supported in part by the National Science Foundation under Grant GJ-599 and by Stanford Research Institute.

## 1. Introduction

The dynamic allocation and deallocation of memory space is a characteristic of most simulation programs, particularly in connection with their timing, queueing, or other list processing activities. Thus, the efficiency of memory allocation procedures is often an important consideration in the development of a model. This is true whether one is programming in a simulation language (where the choice of algorithm is implicit in the language choice) or in a general purpose language (where the choice of algorithm can be made explicitly).

The efficiency of an algorithm can be measured in terms of the required:

- Computer time (for allocating, deallocating, and recombining space) and
- Extra memory space [for pointers as well as for internal (within block) and external (between blocks) memory fragmentation].

Both of these measures will be used in the material presented herein.

Dynamic memory allocation is not a new problem. Knuth [14] provides a good categorization of general techniques and their relative advantages and disadvantages. The various simulation languages have been applying some of these techniques. For example, the original Simscript [17] uses a binary splitting and recombination technique (buddy system), while GASP II [19] uses a LIFO-ordered free space list with constant sized blocks, and Simscript II [12] relies on the operating system to perform this function. However, none of these publications discusses or compares the advantages of the algorithms employed.

Other authors have addressed performance, but only in a limited context. For example, Comfort [4] describes a first-fit and a multiple-list technique (entries being multiples of one size) for use in variable size list processing. Bailey, Barnett, and Burleson [1] and Berztiss [2] outline similar first-fit algorithms for symbol manipulation. Both Shore [24] and Fenton [6] investigate first-fit and best-fit algorithms. However, they consider only the question of memory utilization, and they permit requests to be delayed when insufficient memory is available. Campbell [3] shows that his optimal fit method is superior to first-fit algorithms under certain conditions.

Knowlton [13] describes a binary block splitting technique for use in a list processing language. Purdom [20] looks at the statistical properties of the external space fragmentation of this technique under the assumptions of Poisson requests and exponential holding times. Hinds [7] presents a coding scheme for locating the buddy of a block of storage. Isoda, Goto, and Kimura [11] provide a modified buddy system for greater efficiency of memory usage. Hirschberg [8] provides a Fibonacci-based generalization of the buddy system to reduce the internal space fragmentation problem, although this is achieved at the expense of serious table-space or processing-time side effects.

Cranston and Thomas [5] provide a buffer recombination scheme to overcome these side effects. Shen [23] describes a weighted buddy system, again oriented toward reducing internal space fragmentation.

Iliffe and Jodeit [9] describe the algorithm used to allocate storage for the variable-sized segments of the B5000 system. Margolin, Parmelee, and Schatzoff [16] demonstrate the merits of a combination approach with block sizes tailored to the CP/67 operating system. Randell [21] shows that rounding-up storage requests to an even multiple of block size tends to increase internal fragmentation faster than it reduces external fragmentation. Ross [22] outlines the AED system in which the programmer can control the dynamic memory allocation algorithm; however, proper algorithm selection is not discussed.

Despite the volume of past work on the subject, only a few of the results contribute to an understanding of dynamic memory allocation and algorithm performance within a simulation program. Many of the authors have investigated dynamic allocation algorithms in the context of operating systems or string handling languages. In these situations memory requests vary widely in size, in contrast to simulations which generally request memory in a small number of fixed sizes corresponding to event notices, queue entries, temporary entities, and similar data structures. Also, string and list processing applications usually operate in a stack fashion, so that memory is deallocated in reverse order to allocation (LIFO). Simulation programs generally operate in a more parallel manner, leading to a more random deallocation pattern.

Other studies have incorporated various restrictions that limit their applicability to general simulation activities. For example:

- Restrictions are made as to the arrival patterns of requests and holding times of allocated memory space.
- Execution times of the algorithms are ignored.
- Requests are temporarily suspended when available memory is insufficient.

This latter action is acceptable in systems with ongoing activities that can be "slowed down" (e.g. an operating system), but this action is unacceptable for a simulation inasmuch as it would destroy the faithful representation of the modeled activity. Since the relative advantages and disadvantages of various algorithms for dynamically allocating memory in a simulation environment have not been well established, this investigation was undertaken.

## 2. Memory Demand Test Loads

Inasmuch as a "standard simulation program" has never been defined, there is no standard memory demand pattern against which algorithms can be tested. Rather, because of the variety of behaviors that simulations may exhibit, it is necessary to examine algorithm performance over a wide variety of memory demand patterns. To aid in the development of such sets of demand data, a memory request data generator was constructed. The generator was parameterized, so that a wide variety of allocation or deallocation behaviors could be studied. These parameters covered the number of sources of memory demands as well as the distributions to be used for the frequencies, lifetimes, and sizes of demands. Parameters also controlled the on-off cycling of demand sources to reflect the behavioral changes that some simulations undergo periodically. The pseudo-random number generator (and seed) used to sample from the probability distributions were taken from Lewis, Goodman, and Miller [15].

The assumption was made that at particular points in time the simulation program would implicitly or explicitly request the allocation of memory space and that at subsequent points in time it would implicitly or explicitly call for the deallocation of that space. Actual holding times for each specific block were used by the generator in developing the appropriate sequence of allocate and deallocate requests. However, the output of the generator consists only of the series of requests (and some summary data); the time of each request is not required by the algorithms and has thus been omitted. This means that an algorithm's memory utilization efficiency can only be addressed in terms of a "high water mark" or after "every *X* requests"; it cannot be addressed on a time-weighted basis. This is not, however, a serious limitation, as it is often the high water mark of memory use rather than the time weighted average that governs the memory required to run a simulation.

Another assumption of the test data generator is that memory blocks are released at particular points in time, independent of the algorithm employed. Only by assuming independence can a set of constant test data be applied to a variety of algorithms. Without using an actual simulation model, it is impossible to test garbage collection algorithms that, on running out of free space, search for space blocks to which there are no pointers, and construct a new free list. By using the latter approach, a variety of garbage collection algorithms for space compaction can still be tested (as discussed in Section 3). When the test data stream indicates the release of a space, the algorithm need only mark it "unused" and take no further action. Then, when free space is exhausted, the desired garbage collection algorithm can be applied to compact the unused area. This understates the algorithm execution required in a true garbage collection case (such as that of SIMULA 67) to determine "are there any pointers left to this space?" However, the reported effort would be on a consistent basis with the nongarbage collecting algorithms and would accurately reflect the required effort if used with languages (such as Simscript and GPSS) using explicit deallocation.

A total of 18 sets of test demand data were generated for testing the various dynamic memory allocation algorithms. These sets were selected to reflect a wide variety of conditions, both typical and extreme. Each set or test load consisted of 21 periods, permitting the first period to be used for start-up purposes and the following 20 to be used for algorithm performance evaluation. The period length was selected to approximate the average lifetime of the test's allocations. As a result, the number of separate allocations in a test load ranged between 3,000 and 30,000. The choice of 21 periods and the selected period length for each test load was based in part on the findings of Knuth [14, Chapter 2.5, Section D]. The suitability of these choices was subsequently confirmed by the experimental results.

The base test load (demand load number 1) consisted of ten different demand sources and exhibited a wide range in the number of coexisting allocations outstanding from each source. A wide variety of memory sizes was requested by these sources, and there was a wide variation in the lifetimes of items requested by either the same source or different sources. A more specific description of this test load may be found in Table I.

The other 17 test demand loads (demand loads numbers 2-18) were developed to reflect the likely requirements of other "typical" simulations or to reflect particular types of pathological demand patterns to test the limits of algorithm performance. Thus, for example, demand loads numbers 10 and 7 were designed to reflect the requirements of a large and a small queuing network type of simulation. Demand load number 3 was designed to reflect a simulation which from time to time undergoes sharp changes in the character of its memory requirements, and demand load number 18 was designed to test a sequential usage pattern where every deallocation was followed by an allocation request for a block of the same size. The characteristics of the various test demand loads are summarized in Table II and are further described in Nielsen [18].

The reliance on hypothetical rather than actual demand data for the tests described below raises a question concerning the applicability of the results to actual modeling situations. However, inasmuch as the test data contained extreme cases as well as assumed typical cases exhibiting a variety of demand patterns, and inasmuch as the seven algorithms selected in Section 5 exhibited very little performance sensitivity to these demands (see Table V), it is likely that the conclusions of the research will hold for a wide variety of real-world simulation problems even if the characteristics of those problems are not reflected in the data used in this study.

## 3. Dynamic Memory Allocation Algorithms

A total of 35 different dynamic memory allocation algorithms were selected for test. These can be divided into six categories.

Table I. Characteristics of Base Test Demand Load.

| Demand Source | Interval between demands | Lifetime of allocation | Size of allocation |
|---|---|---|---|
| 1 | normal (100,15) | exponential (3000) | 38 |
| 2 | uniform (50,100) | exponential (1500) | 15 |
| 3 | exponential (125) | exponential (4000) | 7 |
| 4 | normal (75,5) | exponential (2500) | 20 |
| 5 | uniform (25,45) | exponential (250) | 3 |
| 6 | exponential (165) | uniform (500,2500) | 14 |
| 7 | exponential (200) | normal (3000,900) | 20 |
| 8 | constant (150) | normal (900,275) | 2 |
| 9 | exponential (90) | exponential (3000) | 14 |
| 10 | normal (125,25) | normal (4500,1000) | 45 |

All demand sources were active throughout the simulated period (no on-off cycling of demand sources). Each demand was a request for a single block of space of the indicated size (no multiple requests). Each of the 21 periods in the request stream was 4,000 time units long. For the 20 periods for which algorithm performance was measured:

| | Memory in use | Allocates | Deallocates |
|---|---|---|---|
| Highest value | 5768 | 474 | 473 |
| Lowest value | 4684 | 413 | 396 |

There were a total of 8,861 allocations and 8,795 deallocations made. The parameters for the distributions are "mean" for the constant, mean for the exponential, mean and variance for the normal, and lower and upper bounds for the uniform.

The first category contains eight algorithms that operate on a single list maintained in memory order. There is no separate free list. Options tested included the search strategy, the search starting point, the linkage between blocks, and the memory compaction procedures.

The second category contains eight algorithms that operate on a free space list maintained in memory order. Options tested included the search strategy, the search starting point, the free list linkage between blocks, and the memory compaction procedure.

The third category contains six algorithms that operate on a free space list. Options tested included the search strategy, the order in which the list is maintained, the linkage between blocks, and the memory compaction procedure.

The fourth category contains five algorithms that maintain free space lists for each possible size of memory block, subject to a maximum size limit. Allocation is from the appropriate free list if it is nonempty, otherwise space is obtained from the residual queue (which initially contains all of the dynamic memory). A variety of free list linkages and compaction procedures were tested.

The fifth category contains six algorithms that are based on a repeated halving of free space into blocks of appropriate size. Memory compaction takes place only with the buddies split from the same parent. Options tested included the ordering of the free space lists, the linkage between blocks, and the memory compaction procedure.

The sixth category contains two algorithms that operate with a single free space list maintained in block size order.

Table II. Characteristics of Test Demand Loads.

| Demand load no. | No. generators | No. coexisting mean/spread | Size range | Allocation lifetimes one generator / all generators | Generator cycling | Total allocations |
|---|---|---|---|---|---|---|
| 1 | 10 | 22 / wide | wide | dissimilar / dissimilar | none | 9,277 |
| 2 | 20 | 40 / wide | wide | dissimilar / dissimilar | long,uncoord. | 13,315 |
| 3 | 20 | 40 / wide | wide | dissimilar / dissimilar | long,coord. | 13,318 |
| 4 | 20 | 40 / wide | wide | dissimilar / dissimilar | short | 13,325 |
| 5 | 10 | 22 / wide | narrow | dissimilar / dissimilar | none | 9,235 |
| 6 | 10 | 22 / wide | wide,mult. | dissimilar / dissimilar | none | 9,357 |
| 7 | 3 | 27 / wide | wide | dissimilar / dissimilar | none | 2,647 |
| 8 | 3 | 27 / wide | narrow | dissimilar / dissimilar | none | 2,620 |
| 9 | 3 | 27 / wide | wide,mult. | dissimilar / dissimilar | none | 2,638 |
| 10 | 25 | 36 / wide | wide | dissimilar / dissimilar | none | 29,517 |
| 11 | 25 | 36 / wide | narrow | dissimilar / dissimilar | none | 29,685 |
| 12 | 25 | 36 / wide | wide,mult. | dissimilar / dissimilar | none | 29,462 |
| 13 | 10 | 33 / wide | wide | similar / similar | none | 9,382 |
| 14 | 10 | 33 / wide | wide | dissimilar / similar | none | 9,266 |
| 15 | 10 | 22 / wide | wide | similar / dissimilar | none | 9,315 |
| 16 | 10 | 7 / narrow | wide | dissimilar / dissimilar | none | 9,303 |
| 17 | 10 | 50 / narrow | wide | dissimilar / dissimilar | none | 17,524 |
| 18 | 10 | 22 / wide | wide | sequential / dissimilar | none | 9,365 |

Notes: Demand load no.—a number assigned for purposes of subsequent identification.
No. generators—the number of different types or sources of allocation demands used in constructing the test load.
No. coexisting—the approximate mean number of outstanding allocated memory blocks from *each* generator and the spread in the mean value across all generators.
Size range—the range in the memory sizes requested to be allocated by the various generators. The term "mult." denotes that all requests (including overhead space) were for multiples of 5 words.
Allocation lifetimes—the range in allocation lifetimes for the requests from any one generator and the range in average lifetimes for the requests from all the generators used.
Generator cycling—the cycling (if any) of the generators between the active and inactive states.
Coord. and uncoord. indicate whether a generator's cycle shifts were coordinated or uncoordinated with the shifts of other generators.
Total allocations—the total number of allocation demands generated for the 21 reporting period test load.

Summary descriptions of each algorithm are presented in Table III. More specific descriptions may be found in [18].

Each of the 35 algorithms tested was programmed in Fortran IV and compiled under IBM's H level optimizing compiler. Each separate path was then "timed" through an algorithm using the optimized code listing and the instruction timings for the IBM 360/65 [10]. Algorithm performance was computed using this data along with the passage counts collected for each path during execution. (Although the performance figures are particularized for the IBM 360/65, the comparisons between algorithms should be fairly indicative of their relative performance on other computer systems.) The size of the program code for each algorithm is shown in Table IV. Although sizes range from 256 bytes to 1500, there is surprisingly little variation in size. Most of the simpler algorithms are clustered between 400 and 600 bytes, while the more complex algorithms are clustered between 500 and 800 bytes. Only algorithms with very complex compaction procedures exceed 800 bytes in size.

The comparison of the maximum space requested with the actual space allocated during any period provides one indication of an algorithm's space efficiency (internal fragmentation and overhead space usage).

The measurement of external fragmentation is more crude. Three trials were made with each algorithm—demand load combination tested. The amount of memory provided on each trial was adjusted so that the maximum space requested (excluding pointers and internal fragmentation) would constitute 50, 66.7 and 80 percent, respectively, of the available memory space. The level at which an algorithm "ran out of usable space" provides a rough measure of external space fragmentation.

## 4. Experiments—Stage I

Since testing 35 algorithms on 18 sets of data at three levels of available memory space would have resulted in 1890 test runs, a two-stage test procedure was used. All 35 algorithms were first run against the base demand load. Then a set of seven promising algorithms was selected and run against the other 17 test demand loads. Such a procedure presumes that the base demand load is sufficiently representative to serve as a rough screening device, and a posteriori results provide justification for this approach.

Table IV summarizes the performance of all 35 algorithms on the base demand load. The figures under each of the three major column headings reflect algorithm performance under one of the three memory sizes.

The Space use columns reflect the maximum amount of space actually allocated by an algorithm. The ">100%" notation indicates failure to find sufficient contiguous space for a requested allocation. Table IV shows that all algorithms were successful at the 50

Table III. Dynamic Memory Allocation Algorithms

*Category 1 Algorithms: Memory Order without Free Space List*

These algorithms operate on a single list (containing both available and unavailable blocks) which is maintained in memory address order. The features distinguishing each algorithm are as follows:

| Algorithm | Search strategy | Overhead space | Memory compaction |
|---|---|---|---|
| I-1 | First Fit | S + BP | Deallocation |
| I-2 | First Fit + R | S + BP | Deallocation |
| I-3 | Exact/Best + R | S + BP | Deallocation |
| I-4 | First Fit | S | Allocation |
| I-5 | First Fit + R | S | Allocation |
| I-6 | First Fit + R | S | Deallocation |
| I-7 | First Fit + R | S | Garbage Collection |
| I-8 | Exact/Best + R | S | Garbage Collection |

*Category 2 Algorithms: Memory Ordered Free List*

These algorithms operate on a single free space list which is maintained in memory address order. The features distinguishing each algorithm are as follows:

| Algorithm | Search strategy | Overhead space | Memory compaction |
|---|---|---|---|
| II-1 | First Fit | S + DP | Deallocation |
| II-2 | First Fit + R | S + DP | Deallocation |
| II-3 | Exact/Best | S + DP | Garbage Collection |
| II-4 | Exact/Best + R | S + DP | Garbage Collection |
| II-5 | First Fit | S + FP | Limbo |
| II-6 | First Fit + R | S + FP | Limbo |
| II-7 | Exact/Best | S + FP | 2 Stage |
| II-8 | Exact/Best + R | S + FP | 2 Stage |

*Category 3 Algorithms: LIFO/FIFO Ordered Free Lists*

These algorithms operate on a single free space list which is maintained in LIFO or FIFO order. The features distinguishing each algorithm are as follows:

| Algorithm | Search strategy | Overhead space | Memory compaction |
|---|---|---|---|
| III-1 | LIFO + First Fit | S + DP | Allocation |
| III-2 | FIFO + First Fit | S + DP | Allocation |
| III-3 | LIFO + First Fit | S + FP | Garbage Collection |
| III-4 | LIFO + Exact/Best | S + FP | Garbage Collection |
| III-5 | FIFO + First Fit | S + FP | Garbage Collection |
| III-6 | FIFO + Exact/Best | S + FP | Garbage Collection |

*Category 4 Algorithms: Multiple Free Lists*

These algorithms operate on a series of free space lists, one list for every possible sized block of space. All free space lists are maintained in LIFO order. The features distinguishing each algorithm are as follows:

| Algorithm | Sizes used | Overhead space | Memory compaction |
|---|---|---|---|
| IV-1 | Actual | S + FP | Garbage Collection |
| IV-2 | Round Up | S + FP | Garbage Collection |
| IV-3 | Actual | S + FP + B | Compaction, GC |
| IV-4 | Actual | S + DP + B | Compaction, Memory Order |
| IV-5 | Actual | S + DP + B | Compaction, LIFO |

*Category 5 Algorithms: Buddy Blocks (Binary Splitting)*

These algorithms operate on a series of free space lists, each list containing blocks having a size equal to an integer power of two. Memory initially consists of a set of large blocks that are repeatedly split in half as needed to form smaller blocks until blocks of appropriate size are obtained to satisfy allocation requests. Only the two blocks coming from the same parent can be combined to form a larger block. The features distinguishing each algorithm are as follows:

| Algorithm | Search strategy | Overhead space | Memory compaction |
|---|---|---|---|
| V-1 | LIFO | S + FP | Deallocation |
| V-2 | FIFO | S + FP | Deallocation |
| V-3 | LIFO | S + DP | Deallocation |
| V-4 | FIFO | S + DP | Deallocation |
| V-5 | LIFO | S + DP | Garbage Collection |
| V-6 | LIFO | S + FP | Garbage Collection |

*Category 6 Algorithms: Size Ordered Free List*

These algorithms operate on a free space list that is maintained in available block size order. Accordingly searches always begin from the front of the list. The features distinguishing such algorithm are as follows:

| Algorithm | Overhead space | Memory compaction |
|---|---|---|
| VI-1 | S + FP | Garbage Collection |
| VI-2 | S + DP + B | Deallocation |

| | |
|---|---|
| Search strategy | |
| First Fit | A search is made for the first block of space that is at least as large as required to satisfy the request. |
| Exact/Best | A search is made for the first block of space that is exactly the size needed to satisfy the request. If none is found, then the block which contains the least extra space is selected. |
| Rotate | The search begins not from the beginning of the dynamic memory area but from the point at which the last allocation was made. The first block is treated as immediately following the last block in the dynamic memory area. |
| LIFO | Last in, first out, the last block put on the free space list is the first one searched for an allocation request. |
| FIFO | First in, first out, the first block put on the free list is the first one searched for an allocation request. |
| Overhead space | |
| S | Space is needed within each block for a size field. |
| BP | Space is needed within each block for a backwards pointer to the preceding block. |
| FP | Space is needed within each block for a forward pointer on the free space list. The pointer space can be included as part of the user space when the block is not on a free space list. |
| DP | Space is needed within each block for double pointers, one forward and one backward on the free space list. The pointer spaces can be included as part of the user space when the block is not on a free space list. |
| B | Space is needed within each block for a backwards pointer to the preceding block of space in physical memory. |
| Memory compaction | |
| Allocation | Adjacent available blocks are combined at the time an allocation search is made. |
| Deallocation | Adjacent available blocks (buddies only for Algorithm V) are combined at the time a block is deallocated. |
| Garbage Collection | Blocks are deallocated to the appropriate list without any attempt at compaction. When a sufficiently large block cannot be found for a request, a garbage collection routine is called to compact free space. For algorithms employing several free lists, combined blocks are moved to the appropriate lists. |
| Limbo | Deallocated blocks are marked available but are not returned to a free list. When a sufficiently large block cannot be found on the free list for a request, a garbage collection routine is called to "find" and compact the available blocks and to reconstruct an available space list. |
| 2 Stage | Like Limbo procedure except that the first stage garbage collection procedure only collects available blocks and reconstructs a free list (no compaction). If there is still not a sufficiently large block of space, then compaction of the free list is attempted. |
| Compaction GC | Compaction attempted upon deallocation. Otherwise proceed as for garbage collection. |
| Compaction MO | Compaction is attempted upon deallocation. The residal free space list is maintained in memory order. |
| Compaction LIFO | Compaction is attempted upon deallocation. The residual free space list is maintained in LIFO order. |
| Sizes used | |
| Actual | The block size used is the requested size plus any required overhead space for pointers. |
| Round up | The block size used is the requested space plus any required overhead space for pointers plus any additional space required to make the block a multiple of five words. |

Table IV. Algorithm Performance on Base Demand List.

| | | Maximum user memory = 50% | | | Maximum user memory = 66.7% | | | Maximum user memory = 80% | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Algo-rithm | Size | Space use | Alloc. | Deall oc. | Space use | Alloc. | Deall oc. | Space use | Alloc. | Deall oc. |
| Memory Order | | | | | | | | | | |
| I-1 | 392 | 55.0% | 742 | 31 | 73.3% | 742 | 31 | 88.0% | 742 | 31 |
| I-2 | 466 | 54.5 | 67 | 42 | 73.0 | 75 | 41 | >100.0 | – | – |
| I-3 | 568 | 55.1 | 2,274 | 30 | 73.4 | 2,274 | 30 | 88.1 | 2,274 | 30 |
| I-4 | 256 | 52.5 | 1,294 | 10 | 70.0 | 1,294 | 10 | 84.0 | 1,294 | 10 |
| I-5 | 276 | 52.6 | 71 | 10 | 70.2 | 83 | 10 | >100.0 | – | – |
| I-6 | 466 | 52.3 | 59 | 1,227 | 69.8 | 65 | 1,140 | >100.0 | – | – |
| I-7 | 392 | 52.5 | 124 | 17 | 70.0 | 149 | 23 | >100.0 | – | – |
| I-8 | 490 | 52.2 | 1,466 | 10 | 69.6 | 1,466 | 10 | 83.6 | 1,469 | 10 |
| Memory Ordered Free List | | | | | | | | | | |
| II-1 | 558 | 52.9% | 196 | 120 | 70.5% | 209 | 118 | 84.4% | 215 | 117 |
| II-2 | 630 | 52.3 | 46 | 96 | 70.0 | 48 | 94 | >100.0 | – | – |
| II-3 | 592 | 52.7 | 609 | 66 | 70.3 | 609 | 66 | 84.4 | 603 | 67 |
| II-4 | 628 | 52.7 | 524 | 63 | 70.3 | 524 | 63 | 84.4 | 520 | 63 |
| II-5 | 458 | 52.6 | 97 | 22 | 70.3 | 144 | 26 | 84.4 | 182 | 33 |
| II-6 | 494 | 52.4 | 76 | 26 | 69.8 | 93 | 35 | >100.0 | – | – |
| II-7 | 626 | 52.3 | 636 | 28 | 69.7 | 473 | 35 | 83.8 | 448 | 42 |
| II-8 | 676 | 52.2 | 323 | 29 | 69.7 | 349 | 35 | 84.0 | 387 | 43 |
| LIFO/FIFO-Ordered Free List | | | | | | | | | | |
| III-1 | 384 | 53.2% | 210 | 23 | 70.9% | 210 | 23 | >100.0% | – | – |
| III-2 | 388 | 53.6 | 190 | 23 | 71.1 | 259 | 23 | 85.6 | 341 | 23 |
| III-3 | 462 | 52.7 | 456 | 23 | 70.5 | 276 | 34 | >100.0 | – | – |
| III-4 | 520 | 52.2 | 231 | 17 | 69.7 | 231 | 17 | 83.6 | 233 | 17 |
| III-5 | 502 | 52.4 | 1,401 | 20 | 70.1 | 744 | 22 | 84.4 | 506 | 26 |
| III-6 | 560 | 52.2 | 1,591 | 20 | 69.7 | 1,591 | 20 | 83.6 | 1,580 | 20 |
| Multiple Free Lists | | | | | | | | | | |
| IV-1 | 534 | 53.5% | 20 | 21 | 71.4% | 20 | 21 | 85.6% | 20 | 21 |
| IV-2 | 534 | 56.6 | 20 | 21 | 75.4 | 20 | 21 | 90.5 | 20 | 21 |
| IV-3 | 916 | 55.8 | 20 | 43 | 74.3 | 20 | 43 | 89.2 | 20 | 43 |
| IV-4 | 1500 | 56.0 | 52 | 67 | 74.6 | 53 | 68 | 89.5 | 52 | 67 |
| IV-5 | 1000 | 55.9 | 42 | 57 | 74.6 | 42 | 57 | 89.4 | 43 | 57 |
| Buddy Blocks | | | | | | | | | | |
| V-1 | 510 | 73.8% | 53 | 53 | >100.0% | – | – | >100.0% | – | – |
| V-2 | 568 | 73.8 | 47 | 56 | 98.4 | 47 | 56 | >100.0 | – | – |
| V-3 | 498 | 73.8 | 47 | 59 | >100.0 | – | – | >100.0 | – | – |
| V-4 | 512 | 73.8 | 48 | 58 | 98.4 | 48 | 58 | >100.0 | – | – |
| V-5 | 746 | 73.8 | 47 | 29 | >100.0 | – | – | >100.0 | – | – |
| V-6 | 750 | 73.8 | 54 | 22 | >100.0 | – | – | >100.0 | – | – |
| Size-Ordered Free List | | | | | | | | | | |
| VI-1 | 770 | 52.3% | 1,338 | 779 | 69.8% | 1,338 | 779 | 84.2% | 1,338 | 779 |
| VI-2 | 794 | 55.1 | 135 | 200 | 73.4 | 135 | 200 | 88.1 | 135 | 200 |

Notes. Size – the bytes of memory (on an IBM 360/65) required to hold the algorithm program.

Maximum user memory – the maximum percentage of available memory theoretically required to satisfy allocation requests at any one point in time (ignoring internal fragmentation, external fragmentation, overhead space for pointers).

Space use – the maximum percentage of available memory actually required to satisfy allocation requests at any one point in time (includes internal fragmentation, overhead space for pointers but ignores external fragmentation.

Alloc. – the average number of microseconds (on an IBM 360/65) required to allocate a block of memory. The average includes an apportionment of any garbage collection time required by an algorithm.

Dealloc. – the average number of microseconds (on an IBM 360/65) required to deallocate a block of memory. The average includes an apportionment of any garbage collection time required by an algorithm.

percent level, that only four failed at the 66.7 percent level, and that 14 failed at the 80 percent level.

The Alloc. and Dealloc. columns indicate the average microseconds (on an IBM 360/65) required to allocate and deallocate respectively a block of memory. These timings include a prorata share of any garbage collection or periodic memory compaction time used by an algorithm.

The algorithm execution performance was stable from one period to the next, permitting use of the average figures for gross comparisons. (The use of the same test demand loads for each algorithm further reduces the variance associated with algorithm performance differences.) For the seven algorithms selected for Stage II tests, the standard deviations for allocation and deallocation times on the base demand load under each of the three memory loadings were all less than 8 percent of the mean. Similar results were obtained for other algorithms having allocation or deallocation times of less than 100 microseconds (excepting those whose period-to-period performance varied between two values depending on whether the period contained a garbage collection). In the case of several algorithms, there was no variation at all in performance across the 20 measurements.

The period to period stability of reported performance also verified the adequacy of the selected start-up and reporting period lengths.

The most striking feature of Table IV is the general stability of the algorithms' execution performance across the three memory sizes. In most cases there is no tendency toward good performance when memory is plentiful and poor performance when it is not. (There is, however, a clear difference in the ability of the algorithms to use space effectively, as is indicated by the allocation failure frequencies.) Even the period by period performance figures (not shown) for the algorithms that failed the 80 percent memory trial were very similar to the corresponding figures on the 50 and 66.7 percent trials. Only just before failure did performance deteriorate, if at all.

The causes of the performance differences that did exist between trials often appeared fairly obvious. For example, execution time for Algorithm II-5 increased as available memory decreased because of increasing garbage collections. Algorithm III-5, however, exhibited the opposite behavior given the decreasing number of items to search on the free list.

The memory order algorithms (category I) exhibited a wide range of behavior. The first-fit algorithm with a rotating starting point was generally effective over the conditions tested. The first-fit technique with a constant starting point and the exact / best-fit techniques were generally slow (although they were able to perform at higher memory utilizations). Compacting space upon deallocation, without having appropriate pointers (Algorithm I-6), was also time consuming.

The memory order free list algorithms (category II) also exhibited a range of performance but without nearly as wide a variation. The advantage of the first-fit technique was again demonstrated, as was the advantage of the rotating starting point when used with first fit. The performance of the first-fit algorithm using a constant starting point only became advantageous when the free list was relatively small, as was the case with Algorithm II-5 (in which space was not returned to the free list until compacted during garbage collection). The exact / best-fit algorithms again performed poorly. The use of the free list did, however, permit a larger number of these algorithms to perform successfully at the 80 percent memory utilization level. However, the simpler algorithms of category I were surprisingly competitive in terms of execution speed.

The LIFO/FIFO algorithms (category III) generally exhibited poor performance. The LIFO algorithms with garbage collection were particularly poor, but even the best algorithms in this category were not really competitive with the better algorithms of most of the other categories. The advantage that might be offered by a LIFO list is too often lost because of intervening requests breaking up the larger blocks. This was reflected in both speed and memory utilization. Checking the free list for possible compaction also slowed execution. The FIFO algorithms did, however, tend to be able to use the available memory space more effectively than the LIFO algorithms.

The multiple free list algorithms (category IV) were good performers in terms of both execution time and memory utilization. These algorithms require slightly more memory space for their operation than many other algorithms, but this is adequately compensated for by the apparent reduction in external space fragmentation. Since no garbage collection turned out to be required to process the base demand load, the compaction done by the nongarbage collecting algorithms in this category (IV-3, IV-4, and IV-5) was superfluous and led to the higher execution times observed. Nevertheless, all of these algorithms were very competitive with the better ones of categories I and II.

The buddy or binary splitting algorithms (category V) all had favorable execution characteristics and were competitive with the better memory order algorithms. However, they all suffered from a serious internal fragmentation of space. Most of the algorithms failed at the 66.7 percent utilization level and all failed at the 80 percent level. Thus, unless memory is plentiful (or unless most space requests are slightly smaller than an integer power of two), these algorithms would not be very useful. As was the case for the category IV algorithms, no garbage collection was required to process the base demand load.

The size-ordered algorithms (category VI) were fairly slow in execution, although their memory utilization was satisfactory. The time and space use figures for these algorithms clearly illustrate the trade-off between memory space (for additional pointers) and CPU execution time.

## 5. The Experiments — Stage II

The results of the stage I experiments were reviewed to select a set of promising algorithms for further test against the other 17 test demand loads. This review resulted in the selection of seven algorithms: I-2, I-5, II-2, II-6, IV-1, V-1, and V-5.

Of the eight category I algorithms (Memory Order Without Free Space List), three (I-2, I-5, and I-7) dominated the execution performance of the others. Because I-7 is similar to I-5 (differing only in memory compaction procedure) and is somewhat poorer in execution performance, only algorithms I-2 and I-5 were selected for further testing. Although the selected algorithms were not as effective in memory space utilization as other category I algorithms, the nonselected algorithms were dominated in execution performance by algorithms in other categories having effective space utilization.

Of the eight category II algorithms (Memory Ordered Free List), three (II-2, II-5, and II-6) dominated the execution performance of the others. Because II-6 is similar to II-5 (except for use of a rotating search starting point), only II-2 and II-6 were selected for further testing. Again, as for the category I algorithms, the selected algorithms were not as effective in memory space utilization as some of the slower algorithms, but these slower algorithms were dominated by algorithms in other categories having good space utilization performance.

The execution performance of the six category III algorithms (LIFO/FIFO Ordered Free List) was disappointing, since intuitively, better results had been anticipated. Because the execution performance of these algorithms was dominated by other algorithms, none were selected for further test.

Because of the similarity of the five category IV algorithms (Multiple Free Lists) and because of the stability of their execution performance over the memory sizes tested, it was felt that any of these could suitably be used to represent the others. Algorithm IV-1 was selected as the category representative because it had the fastest execution speed and the smallest internal space usage.

Although the performance of the six category V algorithms (Buddy Blocks) was very similar, two were selected for further testing. This stems in part from a desire to make a more detailed test of the type of algorithm used in Simscript and from a suspicion that the memory compaction scheme might have a significant effect in the face of different types of demands. Algorithms V-1 and V-5 were selected as being the fastest and least space consuming algorithms of each type (compaction attempted on each deallocation and compaction performed via a periodic garbage collection).

Because of the relatively slow execution performance of the two category VI algorithms (Size-Ordered Free List), neither was selected for further testing.

Table V provides a summary of the selected algorithms' performance over all 18 test demand loads. Load by load performance is reported in [18]. The Minimum and Maximum columns refer to the minimum and maximum average microseconds required by an algorithm to allocate and deallocate one block of space on any of the tests. The Successes columns refer to the number of test demand loads (out of 18) which were processed successfully without encountering insufficient usable space.

Table VI summarizes the execution performance of the seven selected algorithms relative to the performance of the multiple free space list algorithm (IV-1), which had the most favorable execution characteristics of those tested. Algorithm performance at each memory utilization level was calculated on the basis of the median average time required over the 18 test demand loads to allocate and deallocate one block of space. A failure designation is substituted for those algorithms that were unable to process more than half of the test loads at a given memory utilization level without encountering insufficient usable space.

The multiple free space algorithm (IV-1) had a more favorable computing efficiency than the other algorithms in nearly every case. It did require slightly more memory than most of the others, but it was able to make much better use of the available space. External fragmentation resulted in failure on only three of the 18 test cases at the 80 percent memory utilization level, which was the best performance among the seven algorithms. (It is interesting to note that the three failures all involved situations in which relatively few blocks of space were allocated at any one time. Block sizes were thus large relative to memory size, causing fragmentation effects to be more pronounced.)

The multiple free space list algorithm was able to operate without garbage collection in all cases at the 50 and 66.7 percent memory levels and in all but two cases (involving cycling of demand sources) at the 80 percent level. Thus, this approach must be given high marks on both computational efficiency and memory utilization.

The second memory order algorithm without a free space list (I-5) was surprisingly effective considering its simplicity. At low memory utilizations its processing requirements were more favorable than all algorithms except the multiple free list algorithm (IV-1) and one of the buddy algorithms (V-5). At higher memory utilizations its processing requirements were more favorable than any algorithm (except IV-1) that had a corresponding percentage of successful test cases. Although the 50 percent success rate on the 80 percent memory level trials left something to be desired, this was competitive with, or superior to, any of the algorithms except IV-1. The algorithm's main weakness is reflected on test loads having a large number of allocated blocks or having high memory utilizations, for these require more memory blocks to be searched in the course of making an allocation.

Table V. Performance Summary of Selected Algorithms on 18 Test Demand Loads

| | Maximum user memory = 50% | | | Maximum user memory = 66.7% | | | Maximum user memory = 80% | | |
|---|---|---|---|---|---|---|---|---|---|
| Algorithm | Minimum | Maximum | Successes | Minimum | Maximum | Successes | Minimum | Maximum | Successes |
| I-2 | 106 | 112 | 18 | 107 | 130 | 17 | 108 | 138 | 2 |
| I-5 | 76 | 87 | 18 | 79 | 113 | 18 | 79 | 198 | 9 |
| II-2 | 137 | 149 | 18 | 139 | 146 | 18 | 146 | 156 | 7 |
| II-6 | 90 | 114 | 18 | 103 | 166 | 18 | 133 | 287 | 10 |
| IV-1 | 41 | 42 | 18 | 41 | 42 | 18 | 41 | 169 | 15 |
| V-1 | 99 | 110 | 18 | 103 | 110 | 8 | 110 | 110 | 1 |
| V-5 | 76 | 77 | 18 | 76 | 76 | 3 | 76 | 76 | 1 |

Notes. Maximum User Memory—the maximum percentage of available memory theoretically required to satisfy allocation requests at any one point in time (ignoring internal fragmentation, external fragmentation, overhead space for pointers).

Minimum—the minimum average microseconds (on an IBM 360/65) required by an algorithm to allocate *and* deallocate one block of storage on any of the 18 test demand loads. Garbage collection time is included.

Maximum—the maximum average microseconds (on an IBM 360/65) required by an algorithm to allocate *and* deallocate on block of storage on any of the 18 test demand loads. Garbage collection time is included.

Successes—the number of test demand loads (out of 18) that the algorithm was able to process successfully without running out of available free memory space.

Table VI. Relative Performance of Selected Algorithms on 18 Test Demand Loads.

| Algorithm | Maximum user memory = 50% | Maximum user memory = 66.7% | Maximum user memory = 80% |
|---|---|---|---|
| IV-1 | 1.0 | 1.0 | 1.0 |
| V-5 | 1.8 | – | – |
| I-5 | 2.0 | 2.3 | – |
| V-1 | 2.5 | – | – |
| I-2 | 2.6 | 2.8 | – |
| II-6 | 2.6 | 3.2 | 4.3 |
| II-2 | 3.5 | 3.4 | – |

Notes. Algorithm execution performance is stated relative to the performance of algorithm IV-1 and is based upon the median average microseconds (on an IBS 360/65) required over the 18 test demand loads to allocate and deallocate one block of space. Garbage collection time is included. Maximum user memory is the maximum percentage of available memory theoretically required to satisfy allocation requests at any one point in time (ignoring internal fragmentation, external fragmentation, overhead space for pointers). The dash indicates that for half or more of the test demand loads the algorithm was unable to complete processing without running out of available free memory space.

The buddy algorithms (V-1 and V-5) worked well. However, memory utilization was again a problem. (The Fibonacci buddy system with appropriate modification may, however, offer a reasonable solution to the internal space fragmentation problem. See Section 6.) The V-5 algorithm was uniformly superior to V-1 in processing time, for on too many occasions V-1 recombined blocks upon deallocation only to resplit them again into smaller components. However, this recombination procedure (as opposed to garbage collection) did reduce external memory fragmentation sufficiently to permit a few additional test cases to be processed successfully.

The comparative performance of Algorithms I-2 and I-5 is interesting. Both algorithms are similar except that I-2 compacts space on deallocation while I-5 does so on allocation and then only if required. This results in greater execution times for I-2 (for combining blocks that are subsequently split apart). In addition, the greater space requirements of I-2 led to an additional eight cases of allocation failure because of insufficient space.

The two algorithms that used a memory ordered free space list (II-2 and II-6) were disappointing. With the exception of two test demand loads involving 25 demand sources at the 80 percent memory utilization level, the execution requirements of these algorithms exceeded those of the best memory order algorithm without a free list (I-5). All three algorithms had roughly comparable external space fragmentation.

## 6. Further Research

Further research in this area should proceed in at least two directions, including the validation of the current findings against allocation demands from actual simulations and the development and refinement of algorithms. The 18 test demand loads, used in this study, served to investigate algorithm performance under a wide variety of demand behaviors, pathological and otherwise. However, the experiments reported herein only reflect the basic capabilities of the various algorithms. It remains to apply these algorithms to demands arising from actual simulations, so that the likely or average performance of these algorithms over a broad range of real-world usage can be determined.

A second direction for further research includes the development and test of new memory allocation algorithms and the refinement of existing algorithms. For example, it would be very desirable to test the Fibonacci system of Hirschberg [8] when used with the recombination scheme of Cranston and Thomas [5]. It appears that this algorithm might mitigate the internal space fragmentation problem suffered by the buddy algorithms without sacrificing their computational efficiency.

## 7. Conclusions

The multiple free list algorithms appear to have performance advantages in a broad range of environments. The repetitiveness of a simulation program's requests for memory blocks of certain sizes contributes to the greater efficiency of these algorithms, even in situations where many different block sizes are requested or where the block sizes in use shift over time. A priori fears concerning external memory fragmentation by these algorithms appear to be unfounded.

A simple memory order algorithm can be surprisingly effective. Although its efficiency diminishes when a large number of blocks have been allocated and are in use simultaneously, it performs well in terms of both processing efficiency and memory utilization in many of the other types of demand environments tested.

The buddy algorithms were quite effective in terms of execution requirements. However, they made very ineffective use of memory. Thus, the use of these algorithms should be reserved for situations where memory is in plentiful supply or where most space requests are only slightly smaller than an integer power of two.

The algorithms that used a memory ordered free space list were surprisingly ineffective over the range of demand loads tested. The LIFO, FIFO, and size-ordered free list algorithms also performed poorly.

Although much work remains to be done in testing these algorithms in actual simulations and in developing new algorithms, the results achieved to date provide insight into the relative advantages and disadvantages of certain basic approaches to memory allocation in the context of computer simulation. Further, these findings are directly relevant to simulation languages in use today. Although the dependence on operating system functions for memory management in at least one implementation of both Simscript II and Simula precludes a simple substitution of algorithms for an operational test, the algorithms tested herein are applicable and could be used if the memory management functions were brought back into the simulation programs that are generated by these compilers.

*Acknowledgments*. The author wishes to thank the referees for the care with which they reviewed the paper and for their very constructive comments and suggestions.

Received October 1974; revised June 1976

**References**

**1.** Bailey, M.J., Barnett, M.P., and Burleson, P.B. Symbol manipulation in FORTRAN–SASP I subroutines. *Comm. ACM 7*, 6 (June 1964), 339–346.
**2.** Berztiss, A.T. A note on storage of strings. *Comm. ACM 8*, 8 (Aug. 1965), 512–513.
**3.** Campbell, J.A. A note on an optimal fit method for dynamic allocation of storage. *Computer J. 14*, 1 (Feb. 1971), 7–9.
**4.** Comfort, W.T. Multiword list items. *Comm. ACM 7*, 6 (June 1964), 357–362.
**5.** Cranston, B., and Thomas, R. A simplified recombination scheme for the Fibonacci buddy system. *Comm. ACM 18*, 6 (June 1975), 331–332.
**6.** Fenton, J.S., and Payne, D.W. Dynamic storage allocation of arbitrary sized segments. Information Processing 74, North-Holland Pub. Co., Amsterdam, 1974, pp. 344–348.
**7.** Hinds, J.A. An algorithm for locating adjacent storage blocks in the buddy system. *Comm. ACM 18*, 4 (April 1975), 221–222.
**8.** Hirschberg, D.S. A class of dynamic memory allocation algorithms. *Comm. ACM 16*, 10 (Oct. 1973), 615–618.
**9.** Iliffe, J.K., and Jodeit, J.G. A dynamic storage allocation scheme. *Computer J. 5*, 3 (Oct. 1962), 200–209.
**10.** IBM System 360/Model 65 Functional Characteristics. Form A22-6884-0, IBM, White Plains, N.Y., pp. 19–29.
**11.** Isoda, S., Goto, E., and Kimura, I. An efficient bit table technique for dynamic storage allocation of $2^n$-word blocks. *Comm. ACM 14*, 9 (Sept. 1971), 589–592.
**12.** Kiviat, P.J., Villanueva, R., and Markowitz, H.M. *The SIMSCRIPT II Programming Language*. Prentice-Hall, Englewood Cliffs, N.J., 1968.
**13.** Knowlton, K.C. A fast storage allocator. *Comm. ACM 8*, 10 (Oct. 1965), 623–625.
**14.** Knuth, D.E. *The Art of Computer Programming, Vol. I: Fundamental Algorithms*. Addison-Wesley, Reading, Mass., 1968, pp. 435–451.
**15.** Lewis, P.A.W., Goodman, A.S., and Miller, J.M. A pseudorandom number generator for the System/360. *IBM Systems J. 8*, 2 (1969), 136–146.
**16.** Margolin, B.H., Parmelee, R.P., and Schatzoff, M. Analysis of free storage algorithms. *IBM Systems J. 10*, 4 (1971), 283–304.
**17.** Markowitz, H.M., Hausner, B., and Karr, H.W. *SIMSCRIPT—A Simulation Programming Language*. Prentice-Hall, Englewood Cliffs, N.J., 1963.
**18.** Nielsen, N.R. Dynamic memory allocation in a simulation environment. Res. Paper 181, Stanford GSB, Stanford Res. Inst., Menlo Park, Calif., Aug. 1973.
**19.** Pritsker, A.A.B., and Kiviat, P.J. *Simulation with GASP II*. Prentice-Hall, Englewood Cliffs, N.J., 1969.
**20.** Purdom, P.W., and Stigler, S.M. Statistical properties of the buddy system. *J. ACM 17*, 4 (Oct. 1970), 683–697.
**21.** Randell, B. A note on storage fragmentation and program segmentation. *Comm. ACM 12*, 7 (July 1969), 365–369, 372.
**22.** Ross, D.T. The AED free storage package. *Comm. ACM 10*, 8 (Aug. 1967), 481–492.
**23.** Shen, K.K., and Peterson, J.L. A weighted buddy method for dynamic storage allocation. *Comm. ACM 17*, 10 (Oct. 1974), 558–562.
**24.** Shore, J.E. On the external storage fragmentation produced by first-fit and best-fit allocation strategies. *Comm. ACM 18*, 8 (Aug. 1975), 433–440.