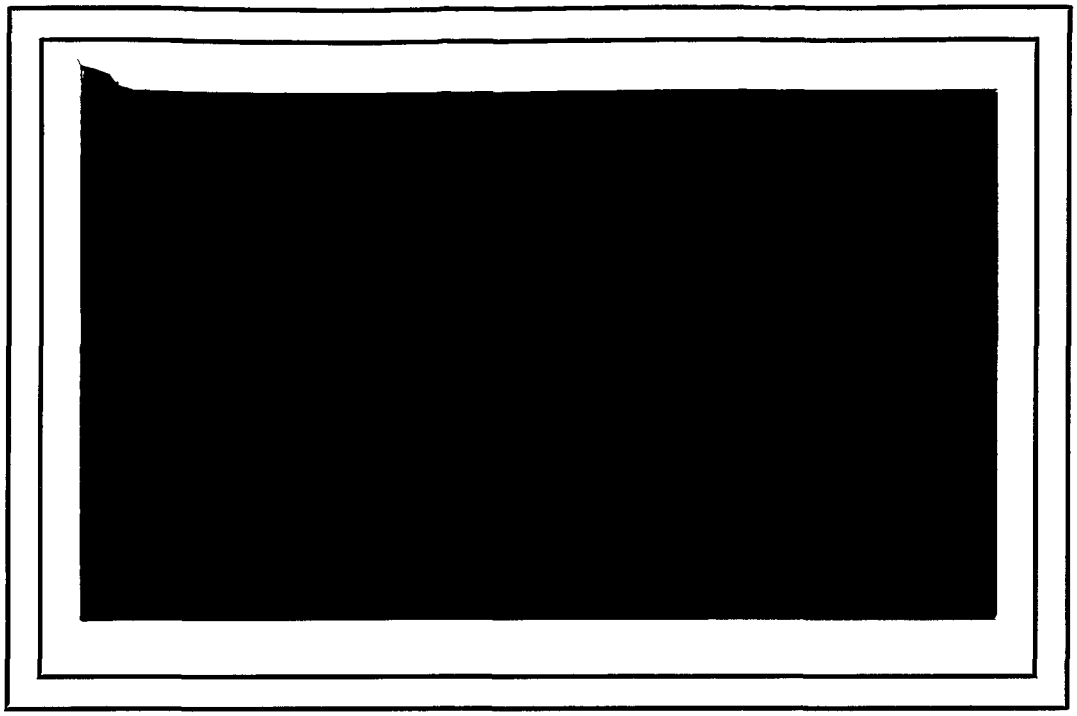


UNIVERSITY OF MARYLAND
COMPUTER SCIENCE CENTER

COLLEGE PARK, MARYLAND

An Adaptive Approach to the Dynamic Allocation of Buffer Storage

by
Sarah Catherine Crooke

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
1970

APPROVAL SHEET

Title of Thesis: An Adaptive Approach to the Dynamic Allocation of Buffer Storage

Name of Candidate: Sarah C. Crooke
Master of Science, 1970

Thesis and Abstract Approved: Dr. Jack Minker

Dr. Jack Minker
Associate Professor
Computer Science

Date Approved: May 20, 1970

ABSTRACT

Title of Thesis: An Adaptive Approach to the Dynamic Allocation of Buffer Storage

Sarah C. Crooke, Master of Science, 1970

Thesis directed by: Dr. Jack Minker, Associate Professor

Several strategies for the dynamic allocation of buffer storage are simulated and compared. The basic algorithms investigated, using actual statistics observed in the Univac 1108 EXEC 8 System, include the buddy method and the first-fit method. Modifications are made to the basic methods in an effort to improve and to measure allocation performance. A simulation model of an adaptive strategy is developed which permits interchanging the two different methods, the buddy and the first-fit methods with some modifications. Using an adaptive strategy, each method may be employed in the statistical environment in which its performance is superior to the other method.

ACKNOWLEDGMENT

The author would like to express her sincere appreciation to Dr. Jack Minker for his continuing guidance and helpful criticism throughout the preparation of this thesis.

The author also gratefully acknowledges the financial support given by the National Aeronautics and Space Administration under Grant NGR21-002-197 and Contract ~~NsG-398~~ which provided funds for the research performed and for the computer time required.

NGL-21-002-008

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

SCOPE OF THESIS

The subject of this thesis is the dynamic allocation of buffer storage which is a basic function of computer operating systems. The allocation methods investigated here are the buddy method and the first-fit method. This thesis presents the results from the simulation of the basic methods, modifications made to the basic methods, and the adaptive use of the modified methods.

The work for this thesis was carried out in essentially three phases: a bibliographic search, development of simulation models of basic algorithms for the dynamic allocation of buffer storage, and an investigation of the feasibility and possible advantage of employing an adaptive method for the dynamic allocation of buffer storage.

The first phase involved performing a bibliographic search of the computer science literature relevant to computer system evaluation techniques. A bibliography on the literature pertinent to the monitoring and analysis of computer operating systems was accumulated. A KWIC (Key Word In Context) index[25] was prepared for the bibliography. Chapter II of this thesis provides an overview of the papers that appear in the bibliography. The evaluation techniques discussed include simulation, mathematical modeling, software monitoring, and hardware monitoring.

In the second phase of the study algorithms suitable for

handling the dynamic allocation of buffer storage were analyzed. The analysis technique employed involved the digital simulation of models written in GPSS-II, a general purpose simulation language. The buddy method, which is implemented in the University of Maryland Univac 1108 EXEC 8 operating system, was analyzed. Buffer request distributions characteristic of the University of Maryland's 1108 EXEC 8 system were established from memory maps constructed from printouts of the buffer pool. The buffer request distributions obtained were used as input data to a simulation model of the buddy method. Validation of the simulation process was then possible by comparing the simulation outputs indicating the internal and external fragmentation, the number of searches for available buffers, and the number of collapses of adjacent buffers with the actual operating system characteristics.

The first-fit method for handling the dynamic allocation of buffer storage was then modeled and simulated. This was followed by modifications to the basic first-fit model which improved the performance of this method. Using the same request distributions in the simulation of these models, the results were compared with the outputs from the buddy method. It was found that the buddy method performance is best in view of the EXEC 8 operating system environment.

Underlying this result was the assumption that no significant internal waste is incurred due to the restriction in the buddy method that the size of all buffers allocated must be a power of two. There is no guarantee that the size of the buffer actually utilized by the requestor is close to but less than some power of two. There is the same probability that it will be close to but just greater than a power of two, in which case approximately one half of the allocated buffer will be unused.

The buddy model was run again under the assumption that requests were made for the exact size of buffer needed. It was found that internal memory waste is a significant factor and may well be unacceptable if the average size of the buffers requested is large. Comparison of the outputs of the first-fit method with those from the buddy method indicated that the first-fit method incurred less internal memory waste than the buddy method, specifically, whenever the average request size is greater than four times the average overhead of the first-fit method. Chapter III presents descriptions of the basic algorithms modeled, modifications made to the basic models, and the results obtained from the simulation process.

From the results found, it is clear that the performance of a given allocation scheme is a function of the buffer request distribution which is characteristic of the operating environment. It suggests that alternative strategies may be desirable when the characteristics of the request distribution changes. The realization of such a strategy in an actual operating system requires that alternative methods for performing a given function be made available in the system and that internal monitors be available in the operating system to detect and indicate the rate and direction of significant change in the operating environment; and, that a mechanism be provided for automatically replacing one strategy by another as a function of the environmental change. There is no indication in the computer science literature that such an adaptive strategy has been proposed or attempted in an actual system. In general, a system becomes fixed at system design time.

The initial problem associated with such a strategy is in making

two independent allocation algorithms compatible. It was found that modifications to the basic algorithms could be made which did not seriously degrade the allocation performance and at the same time permitted transition from one to the other automatically without interruption to system operations. A model of the adaptive method was constructed and the performance was determined through the use of digital simulation. The adaptive model and the outputs from the simulation process are discussed in Chapter IV.

The significant features and conclusions of this thesis are summarized as follows:

- Data obtained from an actual operating system are used in conjunction with digital simulation to analyze methods for the dynamic allocation of buffer storage.
- The allocation methods studied individually included the buddy method as implemented in the Univac 1108 EXEC 8 system and the first-fit method.
- Based on the request distribution for buffer storage found in the University of Maryland Univac 1108 EXEC 8 operating environment, the performance of the buddy method is better than that of the first-fit method in terms of allocation times and memory utilization.
- If the average buffer size requested is large, the internal memory waste introduced by the power of two buffer size restriction implicit in the buddy method may be unacceptable. Internal waste can be eliminated through the use of the first-fit method to allocate buffers of the exact size needed. However, some external waste is introduced due to the fragmentation of available space.
- An adaptive method is investigated where provision is made for

either the use of the buddy method, when speed is important and internal waste is acceptable, or the use of the first-fit method when this type of waste becomes a serious problem.

- An adaptive approach was developed, results from a simulation model of this strategy were obtained, and based on these results, it was concluded that if characteristics of the operating environment change significantly, such an approach should be considered seriously for implementation.

- It is recommended, based on the results of this thesis, that internal monitors be available on a selective basis to determine operating system characteristics, that alternative algorithms suitable for handling system functions be studied, and that the adaptive approach be considered whenever system performance can be improved (or maintained in unfavorable environments) through the use of alternative strategies. Note, selective system monitoring should be used only when the potential improvement in system performance exceeds the overhead and system degradation introduced by the monitoring process.

CHAPTER II

AN OVERVIEW OF COMPUTER SYSTEM EVALUATION APPROACHES

The need for evaluation arises initially when the need for a computer system is determined. The need for evaluation is never satisfied completely thereafter. The original plans for implementing a computer facility involve the following basic question: 'What configuration of hardware, software, and personnel is required to perform the anticipated data processing tasks and generate useful outputs within a required response time?'. It is clear that many different system configurations could satisfy the user requirements. The objective then, is to determine which configuration is 'optimal'. The optimal configuration must be considered relative to user requirements. This is the only context in which the term optimal as applied to computer systems has meaning. The situation is particularly difficult because user requirements may change with time. The system which is finally implemented may not be optimal, but rather a result of compromises made to best satisfy user requirements. In order to make meaningful decisions during the system design phase, standard measures of system capabilities must be employed. This leads directly to a consideration of the measures to be used in the evaluation of system performance. One is also led to a consideration of the techniques to be used for analyzing the system and assigning values to these measures.

The measures used in evaluating the system are a function of

user requirements. Some of the measures related to user requirements are turn-around-time, throughput, cost, system reliability, and combinations of these factors. Assume for the moment that the user is able to estimate his applications workload and to specify his requirements on the system. The problem then becomes one of adopting a technique or methodology for evaluating possible system configurations in terms of his requirements. A possible configuration here may be a standard off-the-shelf hardware/software system, or a configuration resulting from some suitable combination of available hardware/software components which can be integrated to handle the applications workload, or the design of a new system. Although it is difficult to evaluate the effect of the personnel within a system, an attempt must be made to take into consideration such factors as personnel experience level and expected competence. The capabilities provided for in a system design may be realized to a large extent or may be degraded significantly as a result of the personnel interacting with the total system.

A. Development of Computer Evaluation Techniques

A review of the brief existence of general-purpose computer systems may put into perspective the current concern for the need for system evaluation measures and techniques. As late as 1960, the problem of system configuration presented no serious selection problems. There were few equipments and few manufacturers. If a large scale processor were required and funds were available, a computer system could be installed necessitating relatively few decisions on the part of the user. The application determined whether a scientific

or commercial computer, i.e., binary or decimal, was needed. Standard software packages including O/S, compilers, and assemblers were furnished with the hardware. Having decided on a vendor, the hardware configurations were fairly standard. A few options could be exercised, e.g., the number of physical tape drives to be installed.

During the next few years, experience was gained in the use of the second generation computers. Among computer users, there was growing concern due to the lack of well-defined evaluation and selection techniques. By 1964, the year IBM announced their third generation computer, the IBM 360, it is significant that one full session of the AFIPS Spring Joint Computer Conference was devoted to computer system evaluation. The government, the largest customer of the computer industry was finding it more difficult to justify, in terms of value for cost, the purchase of one system as opposed to others. The number of vendors, the line of computers and options, the number of programming languages, and operating systems had all increased. The decisions regarding what computer system to select had increased accordingly. At this point several approaches were taken to get a handle on the seemingly unsurmountable task of computer selection.

In an effort to standardize computer system selection for a government project requiring the purchase of 150 computers, a method was proposed which involved assigning weights, that is, numerical values, to all items in a proposed system. This weighted factors selection method[1] recognized the need for evaluating 'extras' as well as standard items. The inherent weakness of the method lay in the use of absolute weights to score too many factors and to score details

within each factor in different ways. The result was that a given item, e.g., speed, might be weighted for many different reasons so that its true worth and influence in the final selection could not be determined accurately. A further objection to this selection method was that the decisions underlying the system evaluation were largely a matter of subjective opinion and were based on the evaluators' past experience. Evaluators are biased by their background, e.g., financial or engineering, and in the case of new systems, past experience may not be reliable as a basis for computer selection decisions. The value of this method was that it attempted to standardize the selection of computer systems so that particular vendor proposals could be treated impartially.

The cost-value selection technique[2] resulted as an outgrowth or extension of the weighted factors selection method. Only two categories of factors, costs and extras, were recognized. The costs included those associated with securing and maintaining the computer system equipment and the support necessary to satisfy the applications requirements. The 'extras', later translated to dollar cost, included items of value which were inherent in the costs of one system but not to all systems under consideration. Ideally, each item, i.e., each system attribute of value, was considered only once in the evaluation, either as a direct cost, an indirect cost via increased running time, or by its value as an 'extra'. The reduction of all items to a dollar cost produced a common denominator which was then used as a measure for all systems under consideration. The basic advantage of this technique over the weighted factors technique lay in the common denominator concept which allowed all item costs to be treated

independently. The cost-values derived for the various systems were applied as credits to offset the cost of the system and services. The system providing the most value for cost was then the system selected.

Obviously, this method does not solve all the problems involved in the selection of a computer system. Its primary shortcomings include its failure to consider interaction of personnel with system hardware and software, the system design integrity, and validation of proposed system characteristics. Further, in neither of these methods is there any attempt to utilize computers to automate the complex procedure of system evaluation and selection.

In view of the number of details involved in hardware and software description, it was clear that a library must be established and updated as new designs became available. Further, this library would be effective if it could be referenced automatically. The need for a complete library of EDP[3] information was not new. Auerbach Corporation very early in 1962 realized the need for standardized reports and information which could be readily accessed by computer users. The reports and information made available were and are valuable as a library resource; however, their role in system evaluation is limited to the extent that manual system evaluation itself is limited.

Perhaps, the first significant technical development is reflected in the initial efforts to automate system performance evaluation. This approach included the use of a tape library which could be accessed automatically in conjunction with an attempt to model and simulate the performance of proposed systems. The computer system

developed, SCERT (Systems and Computers Evaluation and Review Technique),[4,5] was designed to assist in making initial computer selection decisions, to aid in determining the adequacy of a given system, to evaluate modifications made to increase system capabilities, and to determine the effects of automating new applications and software. The development of this evaluation technique was well under way by 1964 and was reported at that time.

Since 1964, the original version of SCERT has undergone modifications and has been enlarged to permit evaluation of large complex systems as well as small special purpose configurations. More recently, CASE[30], a simulator comparable to SCERT has been developed by Software Products Corporation. Of some interest is the fact that both SCERT and CASE are maintained by the developers on a proprietary basis. Of more importance is the fact that the value of simulation in computer system performance evaluation is being recognized and that simulation techniques are being utilized.

B. System Measurement Tools

At the present time, the methods for computer system evaluation are still somewhere between an art and a science.[6] The scientific method involving observation, hypothesis, experimentation, and modification is difficult to apply to computer systems. This may be true because it is not possible to conduct controlled experiments on a complex and variable system or because to modify the physical system to perform experiments would be too costly and would require excessive time and effort. The problem of system evaluation has been attacked on several levels - analytical modeling, simulation, internal

software monitoring, and hardware monitoring. The applicability of any one of these techniques may be limited and the confidence to be placed in the final evaluation is a function of the level of understanding of the user.

1. Analytical Modeling. As evidenced in the recent literature, much work has been performed in the area of analytical or mathematical modeling. It is significant that the scope of the modeling studies has been limited to subsystems of the total system. Attempts to describe a total system mathematically result in complex unsolvable models or even if solvable, the models are not sufficiently flexible to permit modification and further analysis. Although the use of mathematical analysis has been restricted to logical subsystems of the total system, the results produced in many instances are directly applicable in making decisions during system design and later in formulating algorithms for system operational control.

Typical studies in mathematical modeling involve the analysis of I/O buffering requirements[7], paging characteristics[8], the phenomenon of thrashing associated with excessive paging[9], time-slicing algorithms for multiprogramming[10], queueing disciplines as applied to job scheduling[11], and dynamic allocation of system resources[12]. The models provide a means of thoroughly understanding specific critical aspects of a computer system. As indicated earlier, mathematical modeling is not a practical solution to the problem of total system evaluation. Its applicability should be viewed as local as opposed to global.

2. Simulation. A partial attack on the global problem is through simulation. The phrase 'partial attack' is used because to make the

most effective use of simulation, it should be used in conjunction with other techniques such as analytical models, software monitoring, and even hardware monitoring. A simulation model properly designed and implemented for a sizable system is expensive, but may be one of the best tools for accurately predicting and analyzing system performance. The proper use of simulation is not easy. If the level of simulation is too gross, not enough details are simulated and the resulting information content is low. If the level of simulation is too fine, the cost of performing the simulation due to run time may be prohibitive. Further, the results produced through simulation are no better than the assumptions underlying the construction of the model. The assumptions concerning the behavior of variables within the real system are perhaps most critical. In many cases the behavior of these variables can be represented only through random sampling of variables assuming a particular distribution. The results are then valid to the extent that the assumed behavior of the variables in the simulation approach the actual behavior of the variables in the system simulated.

To facilitate the expression of the components and logic of complex systems, special purpose simulation languages have been developed. The primary objective of such special purpose languages is to permit the user to concentrate more on the details of the system simulated than on the mechanics of the language in which the system is expressed. This is not to say that much simulation work has not been done in the past using available general purpose compilers such as FORTRAN, ALGOL, and PL/1. There is an advantage in using general purpose languages since communication of programs is facilitated due to

widespread use of these languages. A disadvantage of the use of these languages is that in order to simulate timing, interrupts, queues, and control functions accurately, more attention must be given to details of using the language than to details relevant to the simulation. The nature of the simulation languages developed varies from general purpose system simulators, e.g., GPSS[13] and SIMSCRIPT[14], to computer system simulators, e.g., CSS[15] and S3[16], to hardware simulators, e.g., Computer Design Language[17] and HARGOL[18]. Further, some of the languages were developed as independent assembly based languages and some as extensions of existing languages.

In deciding what language to use, certain factors may be critical - availability of the language for general use, i.e., proprietary or unrestricted, flexibility of the language, and prior experience with the use of the language. The simulation language, to a large extent, determines the scope of the simulation possible. Objectively, the language should be selected or developed to provide ease in representing the system to be simulated, to permit either general or detailed descriptions of system components as a function of the level of simulation required, and to make possible the use of mathematical models for characterizing alternative modes of system behavior. The outputs from a simulation study are equally important, i.e., the measures of system performance produced by the simulation which provide statistics relating to turn-around-time, throughput, hardware/software utilization and queueing processes. To be useful, the outputs should be a function of user need for detailed or general information at any desired frequency throughout the simulation run.

3. Software Monitoring. Internal software monitoring of an

actual computer system is another means of attacking the problem of assessing system effectiveness. System analysis, using this technique has been undertaken at the University of Michigan[19] and is also being used to monitor the MULTICS time-sharing system at M.I.T.[20]. Clearly, this technique is useful only in conjunction with an operational system. The monitoring discussed here is not necessarily connected with the collection of accounting type information. The function of the monitor is to gather statistics on actual system resource utilization, queue formation, job frequency, etc. The outputs then form the basis for identifying excessive queues if they exist, which in turn reflect bottlenecks in the system and need for improvement. The monitoring mechanism must appear to be operating in parallel with the normal operating system, causing essentially no interference which would alter the results of the standard mode of operation. Particular care must be taken in using this technique in that the monitoring is not actually performed in parallel, and the user must be assured that the interference, if any, is insignificant with respect to the parameters of interest.

Limited use has been made of this technique since the implementation of the monitoring mechanism is special purpose. Each computer installation invariably has its own unique operating system which means each new system monitored requires new routines and reprogramming to permit evaluation of system performance. Further, comparison of systems monitored may be difficult due to differences in system configuration and general operating procedures. It is my contention that each operating system must build in a monitoring capability of its own. This is true for any large system.

Very recent efforts in the area of software monitoring include the development of monitors by Boole and Babbage[28,29] and a software measurement technique, SIPE, (System Internal Performance Evaluation) developed by IBM[26]. Both of these monitoring devices have been designed for the IBM system/360 Time Sharing System. The use of either of these monitors results in some system degradation during the data collection and recording mode. The loss of system efficiency incurred is justified in that analysis of the operation of a large-scale complex operating system requires data that can be obtained only from 'inside' the system as it is operating. The basic feature of internal monitors is that they have access to, and can selectively record, system data. Subsequent analysis of the data recorded allows for locating the low efficiency portions (i.e., bottlenecks) of a configuration and permits determination and improvement of inefficient software.

Although the actual implementation of an internal monitoring device is special purpose, the results obtainable fulfill very general needs. Every operating system should have the capability of self-monitoring, particularly in areas where performance evaluation is critical and in cases where the workload characteristics and system utilization may vary over time. A logical extension to the self-monitoring concept is system self-modification, i.e., under certain conditions adjusting parameters within the system which govern system performance. Clearly, this step can not be taken until performance under manual control of parameter modification can be evaluated and understood fully.

4. Hardware Monitoring. The design and implementation of

special hardware monitoring devices has been limited due to cost of implementation primarily. The need for such devices has been realized as experience has been gained in the use of large multiprocessing and multiprogramming systems. In most cases, the system capabilities are unknown and means must be devised to determine the system operating characteristics such as I/O wait times, overlap of activities, resource utilization and idle or unproductive times. Hardware monitoring is especially attractive since, if properly designed, many signals can be monitored simultaneously, causing essentially no interference with the system monitored.

One of the earliest uses of hardware monitoring was the direct couple system implemented by IBM which permitted an IBM 7044 to monitor the IBM 7094 operating in stand alone fashion[21]. The 7044 acted as a big counter to obtain statistics on instructions processed in the 7094. This technique is currently being used by Univac to debug and evaluate the 1108 EXEC VIII operating system[27]. In this case, two 1108's are set up as a multiprocessing system, however, the only function of one processor is to gather information on the operations of the other processor. The cost of such monitoring precludes their general use by individual users attempting to improve system performance.

In 1967, the design of the SNUPER computer was reported[22]. The objective of the design project was to develop a monitoring device which would interface with a computer system, produce a record of significant events, and between significant events, provide for generation and maintenance of on-line displays. The ultimate goal of this study was to determine the class of instrumentation which could

give significant measures of system performance using a small, low cost SNUPER computer. If these objectives could be met, the computer then could be used at more than one computer installation. The most recent report on this project was given at the AFIPS 1969 SJCC[23]. The emphasis in this report was more on the class of parameters which could be monitored than on the hardware features required to handle the monitoring.

At the same time, IBM was working on a recording device, the Time-Sharing System Performance Activity Recorder (TS/SPAR) to be used in monitoring the class of TSS/360 computers[24]. Input to this device was via a specially engineered interface through which the internal states of the Model 67 system and I/O devices could be monitored. The report was non-committal as to the actual success realized through the use of the recorder. It was viewed more in terms of its potential for the future in the areas of multiprocessing, multi-tasking, data set organization in virtual and real storage, and I/O monitoring. A long range objective was to provide feedback capabilities and make the recorder a system monitor rather than merely a logger of information.

At the present time, any extensive hardware monitoring is special purpose, expensive and rather inflexible. As a consequence, hardware monitoring devices, developed and used, by computer system designers, have had limited use by the general user.

C. The Use of Multiple Measurements

In the preceding discussion, the major methods available for use in system evaluation have included mathematical modeling, simulation, internal software monitoring and hardware monitoring. Each of these

methods has its advantages and also its limitations. In the evaluation of system performance for a large scale multiprocessing or multiprogramming system, any one technique may not be a practical or satisfactory solution. Limiting factors may include cost, complexity of system, level of confidence in unavoidable assumptions made, inflexibility, or interference caused by the monitoring device. A more practical solution to system evaluation appears to be through the use of more than one technique.

D. Specific Applications

Perhaps the best example of the use of multiple measurement tools is found in the research now being conducted on the MULTICS time-sharing system[20]. At system design time, certain hardware features were provided to enhance software measurement. These included a central read-only system clock which produces a count per μsec, a time match interrupt, and a CPU memory cycle counter. When the system became operational, software modules were developed to use the hardware monitor features and to provide information on frequency and timing of missing page faults, missing segment faults, linkage faults, wall-crossing faults, and interrupts. By taking advantage of the built-in hardware features, the software required was not elaborate. For example, segment usage metering was performed through the use of the clock and the time matching interrupt. Every 10 μsec an interrupt occurred, at which time the core location was noted and recorded. Reduction of the data provided a histogram of segment usage and indicated most popular segments. The results permit localizing where time was being spent and further which procedures should be made

more efficient.

In order to conduct scientific type experiments, i.e., reproducible experiments as far as possible, bench marks were established for the MULTICS system. The bench marks took the form of script input which is essentially an established list of commands representing console users. During test periods, the system configuration is standardized and the use of the system is restricted, i.e., no other users are allowed to distort the experiment. One of two modes of operation then is possible - internal or external. In the internal mode, the script is read into the main computer. A simulation program is used to interpret the commands and to trigger the system functions just as if n consoles were driving the system. When the external mode is used, the script is interpreted by a PDP-8 computer and interrupts are produced at the main computer exactly as they would appear if produced directly from console users. A logical consequence of using bench marks for system evaluation is that optimization of system performance is in terms of the inputs used. The MULTICS project group considered this in setting up the script. The commands to the system included in the script were selected primarily from typical requests requiring extensive file maintenance and management. Optimization of the system in terms of these requests results in general system improvement since in a time-sharing system much time is spent in paging and file manipulation.

In summary, measurement tools being used in the MULTICS system include hardware monitoring (provided in system design), software monitoring, bench marks, and simulation. Evaluation of the data obtained through the use of these measurement tools is providing

insight into the operation of time-sharing systems and making system improvement possible through the analysis of effects produced by system modification.

E. Conclusion

Not all system analysts are fortunate enough to have integrated hardware instrumentation; however, extensive use of all available evaluation techniques should be considered. One attractive approach is through simulation, validated by actual system performance as determined using internal software monitoring. Further, the simulation process may be reduced through the use of results derived from mathematical modeling of subsystem behavior. The technique or combination of techniques to be selected and implemented for any given system will depend upon many factors including available hardware instrumentation, the scope of the evaluation, and the stage of system development. In any case, system evaluation must be a continuing effort - in the system design in order to meet user requirements and later in system operation to determine whether system capabilities have been exceeded, or the system is being used inefficiently, or simply to improve or to maintain system performance as user and application characteristics change with time.

CHAPTER III

ANALYSIS OF DYNAMIC ALLOCATION STRATEGIES

A. Scope of Analysis

The analysis undertaken makes use of simulation models and internal software monitoring of actual system performance. The scope of the simulation was restricted to analyzing the characteristics of dynamic allocation of buffer storage for temporary, unpredictable, and small storage requests. The Univac 1108 supervisory system, EXEC 8, allocation scheme was the subject of analysis. This system was selected because of its availability at the University of Maryland Computer Science Center for observation through software monitoring. The dynamic allocation schemes for buffer storage became the subject of analysis because this function is central to the allocation scheme implemented in the executive system and is a critical factor in system performance. From time to time the allocation scheme implemented in the EXEC 8 has come under close scrutiny of the system analysts. At these times attention has been directed more toward determining why system performance has become degraded or nonexistent than toward evaluating the merits of the implemented allocation scheme as compared with others which might be more effective under certain operating conditions.

It should be noted that the choice of buffer allocation schemes as the subject of study was made in view of the fact that the allocation of small buffers is relatively self-contained as compared with

dynamic allocation of user programs in a multiprogramming environment. In general, allocation of memory to user programs cannot be considered independent of a particular system design philosophy including scheduling procedures, priority schemes, and hardware restrictions. Further, allocation of memory to user programs may be extremely complex involving many variables and parameters which in themselves are not clearly understood. The interaction of these parameters is then another order of analysis. The unavoidable complexity and the magnitude of such a study dictate that experience should be gained in the use of the analysis techniques in understanding the basic elements of a system as a first step. The potential use of these techniques can then be realized in more extensive studies which should be undertaken.

1. Function Parameters. In the allocation of buffer storage, two factors, time and space, are important. In any given system one may be more critical than the other. If such is the case, time-space tradeoffs may be unavoidable. Ideally, the strategies implemented would be selected only after an analysis of potential schemes had been performed, which would indicate the strategy incurring the least penalty and best satisfying the critical space or time requirement. The two factors of interest in the dynamic allocation of buffer storage may be restated as the 'time to allocate and release buffers' and memory utilization or 'the percent of total reserved memory which is effectively used'.

The allocation time may be increased or decreased depending upon the allocation strategy adopted and the sophistication and complexity involved in the programming. The program complexity and possibly the running time may be increased if a premium is set on the memory use.

In any case, there is always some overhead time associated with the search and maintenance of available buffer storage lists. Contributing to memory loss are system overhead requirements and waste, so that the memory utilization factor is always less than 100%. Included in the system overhead is the amount of storage required for linkage, block sizes, and use tags. Contributing to the waste are two sources of unusable memory: external fragmentation of memory and internal fragmentation caused by fixed request size which requires that the request be equal to some specified buffer size. Whenever it is necessary to request a buffer greater than the buffer actually needed, some internal waste is incurred. The memory loss incurred by fixed request requirements may be acceptable and even desirable if space is not the prime consideration and the implementation is facilitated and/or the allocation time is reduced.

2. Pooled versus Private Buffers. Buffer storage allocation is a function common to most operating system executive routines. There are two ways to assign buffers: either buffers are acquired dynamically as needed from a pooled buffer, or each process requiring storage has its own private buffer which is sufficiently large to make the probability of overflow less than some number. The use of pooled buffers by an executive routine servicing many users through reentrant routines which require temporary buffers is essential if memory utilization is to be high. This is clear since otherwise for each routine the memory loss caused by each user is equal to the difference between the expected maximum buffer needed and the average buffer usage. A conclusion based on analysis reported by Denning[31] is that 'pooled buffers are far superior to private buffers, especially when the number of users is large'.

Another advantage of the pooled buffer lies in the fact that allocation of additional space for buffers regardless of which routines are temporarily active need be made only when the total memory allocated to the pool is near depletion. The term 'near depletion' describes the situation where a request is made for a buffer of size n and this request cannot be honored, however, the difference between the total memory reserved and the total memory allocated is greater than n. Restated, this means that if the used buffers were placed contiguously in the memory pool, n consecutive memory locations would be available to satisfy the buffer request. It is highly improbable that all available space will be used before apparent overflow occurs due to some degree of external fragmentation introduced in the allocation process. It is in the interest of maximum memory usage to implement an allocation scheme which keeps external fragmentation at a minimum or to provide for memory consolidation periodically. Because of the asynchronous nature of the executive functions and the many users operating concurrently in the computer system, buffer consolidation through memory rearrangement and relinkage would be unfeasible. The objective then is to evaluate allocation schemes in relation to the operating environment and decide upon one which keeps external memory loss within acceptable limits.

B. Buffer Allocation Algorithms

Basic schemes for dynamic allocation along with algorithms for implementation have been well defined in the computer science literature[32]. Some comparisons of the methods have been made on the basis of assumed operating environments. The schemes receiving most

widespread usage are the first-fit method, the best-fit method, and the buddy method. In the first-fit and best-fit allocation, a list of available storage is maintained. When buffers are released, they are returned to the list of available storage either separately or combined if the released block is contiguous with a block of available storage. The difference in the two methods is found in the allocation. In the first-fit method, a request for a buffer of size n is filled from the first block of available storage encountered on the list which is greater than or equal to n. In the best-fit method, if no block of size n exists, a search of the entire available storage list is made to find the block of storage which makes the available storage block minus n a minimum. In general, the best-fit method is implemented less often than the first-fit method because of the time factor involved in the available storage list search for each allocation made. It has further been found that the best-fit method does not necessarily reduce the problem of fragmentation[32].

The buddy system which is implemented in the EXEC 8 requires that the size of requested buffers be a power of two. It should be noted here that this requirement for standard request sizes may be an important factor in memory loss if the user must request buffers which are larger than actually needed. If no buffer of size 2^k is available, the smallest block 2^j which is greater than 2^k is split into blocks of $2^k,\ldots,2^{j-1}$ words each. Upon release of a buffer, halved blocks, called buddies, are recombined if both are available. More complete descriptions of the first-fit and buddy algorithms will be given later since these are the two basic schemes, with some modifications, which are evaluated.

As indicated earlier, the analysis techniques used included simulation and some software monitoring of the EXEC 8 operating system. The simulation permitted an evaluation of the allocation schemes in terms of time and memory utilization. The data obtained using internal software monitoring of the executive system provided request-release distributions representative of those seen by an actual operating system. Simulation taken alone is valid to the extent that the assumptions made about the actual behavior of the system parameters are valid. Software monitoring provides data representative only of the particular system monitored since, incorporating alternative schemes into an existing operating system for experimentation purposes is difficult, and in general, is not encouraged by system analysts responsible for maintaining an 'operating' system. Validation of the simulation models and increased confidence in the outputs from the evaluation process resulted through the combined use of the two techniques.

C. Simulation Language

The schemes for dynamic allocation of buffer storage were modeled using GPSS-II and processed using the Univac 1108 at the University of Maryland Computer Science Center. GPSS-II is a general purpose system simulator designed to permit the study of any system or process which can be reduced to a series of operations performed on units of traffic. The structure of the system simulated is described as a series of blocks, each block describing some step in the action of the system. A number of block types are provided, each corresponding to some basic actions or conditions that may occur in a system. In the

simulation process, units of traffic, or transactions, are created and processed through the system by the simulator.

The user of GPSS-II may control the volume of traffic, the action time in any block, transaction priorities, conditional entry or exit from blocks, and specify the outputs desired. The outputs may include information on the number of transactions, i.e., volume of traffic through portions of the system, the distributions of transit times for transactions between selected points in the system, the average utilization of system elements such as facilities and storage, and information on queue formation at selected points in the system. The outstanding features of the simulator include the facility with which continuous or discrete functions may be defined and used in the simulation process, the control the user has over the routing of transactions through the system, and the ease with which statistical data may be collected at critical points in the system.

The models developed to represent the dynamic allocation of buffer storage assumed the following correspondence between system components and the elements of the block diagram. Requests for buffer storage are treated as transactions, and the size of the buffer pool corresponds to storage capacity. The arrival of requests for buffer storage were generated assuming a Poisson distribution. The requests are serviced according to the allocation scheme modeled.

One of the more difficult aspects of the modeling involved controlling the locations in memory which were allocated for a given transaction. The GPSS-II language provides for defining storage capacity, and the simulator retains a record of used and unused storage, but does not record which specific transactions occupy the

storage. In order to realistically simulate the allocation process and determine the extent and type of memory fragmentation characteristic of each allocation scheme used, it was necessary to maintain a memory map in the models. Total buffer pool overflow was then determined as a function of whether n consecutive locations were available regardless of the total number of unused memory locations. In the buddy allocation model, the memory map of buffer storage was maintained using GPSS block types under the assumption that the available storage list would remain short, whereas in the first-fit model, the memory map was maintained using a Fortran subroutine which is permitted as a special GPSS block type. The provision for such routines is to permit the user to perform certain arithmetic and special operations in Fortran which cannot be performed conveniently by a combination of ordinary GPSS block types.

D. Simulation Models Developed

The dynamic allocation of buffer storage is an essential function in an executive program. In order to perform many utility functions within the system, e.g., input-output, and to maintain control over system operations, information must be maintained which reflects the current state of the system operations. Because of their frequent and asynchronous use, many system routines are coded to be reentrant. This, in turn, may require that each time a reentrant routine is executed, a buffer must be established to identify the source of the caller and to preserve any parameters modified by a call to the routine. In general, the size of buffers needed for maintaining system control are small, i.e., on the order of 2^2 to 2^8 words and the use time of a

buffer is relatively short. These two factors, size of buffers and use duration are important in evaluating alternative allocation schemes.

In the dynamic allocation of buffer storage, a method must be adopted for allocating and releasing variable size blocks of memory, maintaining a list of available or unused blocks, and extending the buffer pool when it nears depletion. In developing or selecting a suitable allocation scheme, decisions are necessarily made, either explicitly or implicitly, with respect to factors which could affect the efficiency of the allocation process. In adopting an algorithm, one, at the same time, adopts decisions such as whether to maintain one list of all available blocks or to maintain several lists; whether the blocks on the list should be ordered or unordered, and if ordered, whether they should be in increasing or decreasing order of size, or in order of memory address; and, whether requests for buffers must be a fixed size, one of several specified sizes, or a variable size. The execution time per allocation, the allocation routine complexity, and the amount of unusable space per allocated block are ultimately a function of the allocation process implemented. Through the use of simulation models, algorithms which are based on alternative approaches can be evaluated in terms of execution time and memory space tradeoffs. Initially, two basic allocation schemes were modeled, the first-fit and the buddy allocation method.

In the first-fit method, one list, essentially unordered, by size, of available storage blocks is maintained; the buffer request size is variable; the list is doubly linked so that, upon release of a buffer, adjacent available buffers in either the forward or backward

direction may be combined with the buffer being released; and two words in every allocated block are reserved for allocation control. Each time a buffer of size n is requested, the routine is entered. The list of available storage blocks is searched until the first block of at least n+2 words is found. The block from which the allocation is made is reduced by n+2 and the remainder, if greater than zero, is returned to the list of available storage. The address of the reserved buffer is then returned to the user.

The other basic algorithm selected for study is the buddy method. The buddy allocation scheme makes use of (m-1) locations which serve respectively as heads of the lists of available storage of sizes 4, 8,...,2^m. Circular lists, singly linked, are used for storing available blocks of storage. Before any storage has been allocated, list pointers are established so that AVAIL(i)=i, i=2,..., m-1 indicating these lists are initially empty and AVAIL(m) points to the location of the first available block of size 2^m. One word of overhead in each allocated block is used for allocation control. Implicit in the list definition is the fact that the maximum request size is 2^m-1 and the minimum request size is theoretically 1, although in the EXEC 8 implementation of the buddy method, the minimum is arbitrarily set at 3. Regardless of the exact buffer size requested, if it is between 1 and 2^m-1, a buffer of size 2^k is allocated, where k is the least power of 2 which is greater than the buffer size requested. It should be noted that although a request may be made for any size buffer within the specified range, the size of buffer allocated is always a power of two, representing essentially a restricted number of distinct buffer request sizes. As a consequence, the lists of

available storage are maintained by size. The basic request and release algorithms for the first-fit allocation schemes are taken from Knuth's The Art of Computer Programming, Volume I, entitled Fundamental Algorithms[32]. Certain modifications were made to the algorithms as given, in order to facilitate the implementation and reduce the simulation running time. For example, in the first-fit algorithm, the packing of the size, use tag, and link into one computer word was not actually performed in the simulation model. This reduces the number of operations to be performed in the simulation process which in turn reduces the simulation running time. As a result, two additional words in each allocated block are used for simulation control. Because it is a simulation, and no practical use is being made of the n-2 words in an allocated block, this modification does not logically change the basic algorithm. The only consequence of this change is that for the simulation model of this algorithm to function properly, the minimum buffer request must be two words, which is not an unreasonable restriction in view of the EXEC 8 requirement which may be viewed as typical of operating systems. The minimum buffer request size, plus the standard two words required for linkage and control guarantees that the four words of control used in the simulation are available. Minor changes, such as the reversal of the plus and minus boundary or use tags are a matter of programmer preference and in no way impose any additional restrictions on the allocation process.

The simulation models of the first-fit allocation and release algorithms are as follows.

1. Buffer Allocation (First-Fit). Let U point to the first available block of storage, and suppose that each available block with

address P contains the following information: SIZE(P), the number of words in the block maintained in the second and last word of each block; LINK(P), a pointer to the next available block on the list; LINKB(P), a pointer to the preceding available block on the list; and TAG(P), a sign on the size word which is used to control the release process. TAG(P) = '+' indicates a free block; TAG(P) = '-' indicates that the block is reserved. A 'roving' pointer, ROVER, is used so that the search for an available block begins in different parts of the available list, which avoids initiating the search with the first available block on the list for each buffer request. F is used in conjunction with ROVER to determine when all entries on the available list have been searched. Upon entry to the routine, F is set to zero. When U, the head of the list, is encountered, F is set to 1. If F=1 and the head of the list is encountered again, this means that the entire list has been searched without finding an available block of adequate size. Since ROVER may be positioned to any block in the list initially, some portions of the list may be searched twice. Note that if the search always begins at the first available block on the list at each request, there is a strong tendency for blocks of small size to build up at the front of the list, so that in general it may be necessary to search through many entries in the list before finding a block which will satisfy a buffer request.

A1: [First entry only, initialize.] Set U, P, and ROVER = address of first cell of buffer pool. Store size of buffer pool in the second and last word of block. Set LINKB(P)=0 and LINK(P)=Loc(U).

A2: [Initialize search.] Set P=ROVER, F=0.

A3: [Test end of search.] If P=Loc(U) and F≠0, no allocation is

possible. Otherwise, if P=Loc(U), set F=1, P=U.

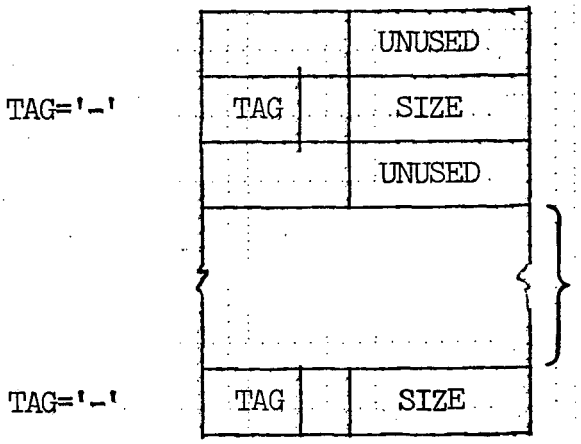
A4: [Search list.] If SIZE(P)$\geq$N, go to A5; otherwise set P=LINK(P) and go to A3.

A5: [Reserve N locations starting at L.] Set K=SIZE(P)-N. If K=0, set LINK(LINKB(P))=ROVER, set LINKB(ROVER)=LINKB(P). (This removes an empty block from the available list and sets L to the beginning of reserved block.) If K$\neq$0, set SIZE(P)=K. In either case, set TAG(P)='-' to indicate it is reserved and set L=P+K.

The algorithm terminates successfully, having reserved N locations beginning at P+K. The function of the allocation algorithm for the simulation process is to reserve buffers as requested and to insure that each block in the buffer pool has the form given in Diagram 3-1. Note here that since this allocation scheme is being used in a simulation process only, no attempt is made to reduce memory overhead, e.g., LINK, TAG, and SIZE will fit conveniently into one computer word if time is taken in the simulation to pack them. In general, then, two words of control are sufficient to maintain control of this data structure. When buffers are returned to the buffer pool, the release algorithm assumes that the blocks are in the form maintained by the allocation process.

2. Buffer Release (First-Fit). This algorithm puts a block of N locations starting at address L onto the available list. Whenever an upper adjacent block of locations is found to be available, it is deleted from the available list and collapsed into the block currently being released. If a lower adjacent block is found to be available, the block being released is combined with the block already on the list. If neither adjacent block is free, the block currently being

Reserved Buffer Format



Free Buffer Format

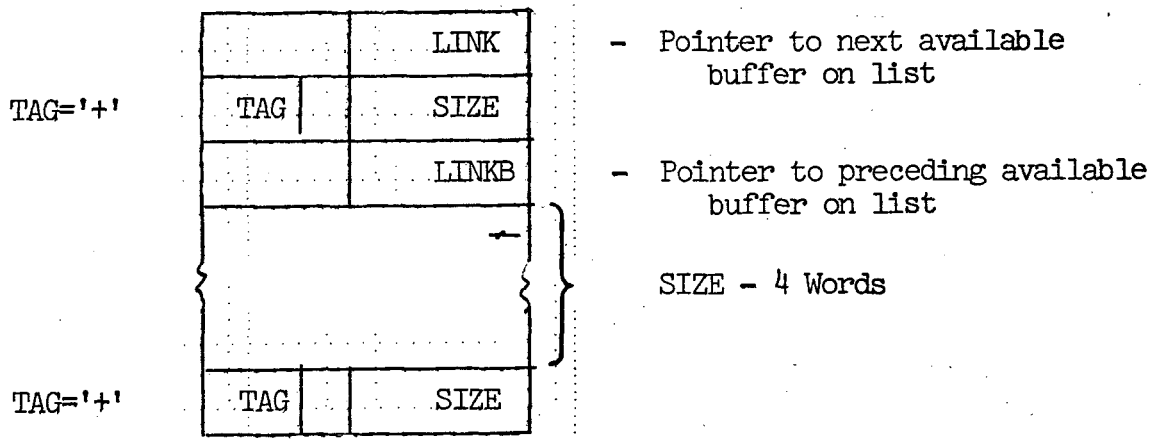


Diagram 3-1. Buffer Formats

Used in the First-Fit Simulation Model.

released is simply added to the front of the available list.

R1: [Check upper adjacent block.] Set P=L+N. If TAG(P)>0, go to R3.

R2. [Check lower adjacent block.] If TAG(L-1)>0, go to R4. Otherwise, set P1=U, P2=Loc(U), and go to R5.

R3: [Set up for deletion of upper adjacent block.] Set N=N+SIZE(P), P1=LINK(P), P2=LINKB(P), if P=ROVER, set ROVER=Loc(U).

If TAG (L-1)<0, go to R5, otherwise, set LINK(P2)=P1 and LINKB(P1)=P2.

R4: [Collapse current block with lower adjacent block.] Set N=N+SIZE(L-1), set L=L-SIZE(L-1), and go to R6.

R5: [Relink available list.] Set LINK(L)=P1, LINKB(L)=P2, LINKB(P1)=L, LINK(P2)=L.

R6: [Store size of block returned.] Set SIZE(L)=N, SIZE(L+N-1)=N, and return.

3. Buffer Allocation (Buddy). The buddy simulation model developed is based on the allocation and release algorithms presented in Knuth[32]. This method requires one word for control in each block and requires that the size of all blocks be a power of 2. This method keeps separate lists of available blocks of each size 2^k where $2 \leq k \leq m$, and 2^m is the largest permissible buffer size. When a buffer of 2^k words is requested, and no block of this size is available, then a larger block is split into two equal parts; at some point a block of the requested size is available. When one block is split into two equal blocks, these two blocks are called 'buddies'. If at a later time, both buddies are available, they may be collapsed into a single block.

The usefulness and practicality of this method lies in the fact that if the address and the size of a block are given, the buddy to

this block is easily found. Let $buddy_k(x)$ equal the address of the buddy of a block of size 2^k whose address is x. Then it is found that:

$$buddy_k(x) = \begin{cases} x+2^k & \text{if } x \bmod 2^{k+1}=0 \\ x-2^k & \text{if } x \bmod 2^{k+1}=2^k \end{cases} .$$

This function is easily computed with an 'exclusive or' instruction usually found in binary computer instruction repertoires.

When a block is reserved, only one word is needed to maintain control. This one word contains a 'use' tag and the block size. If the block is reserved, TAG(P)=0 and if the block is free or available then TAG(P)=1. When blocks are free, one link field may be used for maintaining a singly linked list, or two links may be used if doubly linked lists are desired. In the simulation model, singly linked lists are used. The buddy system algorithms are as follows.

Assume a request for a buffer of size 2^k .

A1: [Initialize, first entry only.] Set AVAIL(i)=i, i=2,...,m-1 and set AVAIL(m)=location of first buffer of size 2^m . Link all buffers of size 2^m and set link of last buffer on 2^m list = m, and set all sizes = m.

A2: [Search lists for first list with block size $\geq$ k which is non-empty.] Search AVAIL(i), where $k \leq i \leq m$ such that AVAIL(i)$\neq$i. If none, no allocation is possible for block of size 2^k.

A3: [Remove first block from list with available block.] Set L=AVAIL(i) and AVAIL(i)=LINK(L) where 2^i is first available block.

A4: [Test for i=k.] If i=k, return location L to user as starting address of reserved block.

A5: [Split 2^i block and put a block on 2^{i-1} list.] Set i=i-1,

$P=L+2^i$, LINK(P)=i, SIZE(P)=i, AVAIL(i)=P, and go to A4.

4. Buffer Release (Buddy). Assume a buffer of size 2^k starting at location L is to be released.

R1: [Calculate buddy address using function given earlier.] Set P=Loc(buddy). If k=m or block at buddy address is not available or has size < 2^k, go to R3.

R2: [Remove from list and combine with buddy.] Set AVAIL(k)=LINK(P), k=k+1. If P<L, set L=P and go to R1.

R3: [Place block on list k.] Set LINK(L)=AVAIL(k), AVAIL(k)=L, SIZE(L)=k, and return.

E. Inputs to the Simulation Models

The confidence to be placed in the outputs from a simulation model is a function of the extent to which the model represents the system function being simulated. Of equal importance are the assumptions necessarily made concerning the behavior of the parameters in the actual system. To test the models, statistics were needed on the behavior of the transactions in the model, where the transactions correspond to requests for buffer allocation and release in the executive system. In particular, statistics were needed on the request size distribution and on the rate of buffer request and releases. In order to test the models with realistic inputs, efforts were made to gather data characteristic of the EXEC 8 in an actual operating environment.

In order to approximate a request distribution, memory maps were constructed from printouts of the buffer pool, EXPOOL. From the memory maps, it was possible to tabulate the number of allocations of

each valid request size at the time the printout was produced. The distributions of buffer allocations by request size obtained from these memory maps are shown in Figure III-1. At this point, insufficient data are available to definitely correlate variations found in request distributions with particular system operating modes, e.g., batch or on-line. It could be significant in the evaluation of particular allocation schemes if such correlations are found to exist.

The buffer request and release rates are not available at this time. In the simulation process, buffer requests and releases are being generated assuming a Poisson arrival distribution and an exponential hold time. Under these assumptions, Figure III-2 then presents the distribution used as input to the simulation process and also the distribution constructed from a memory map at the end of the simulation run. Confidence was gained in the validity of the model since the distribution is not significantly altered as a result of the simulation process.

F. Outputs from the Simulation

Performance is being measured in terms of memory utilization and execution time required for the allocation process. In order to estimate relative execution times, data were collected on the time-consuming operations within the allocation processes. The following operations were tabulated for both the first-fit and buddy allocation models: the number of searches of the available storage list(s), the number of memory collapses, the number of searches required for releasing a buffer, and the number of splits required to obtain a buffer of

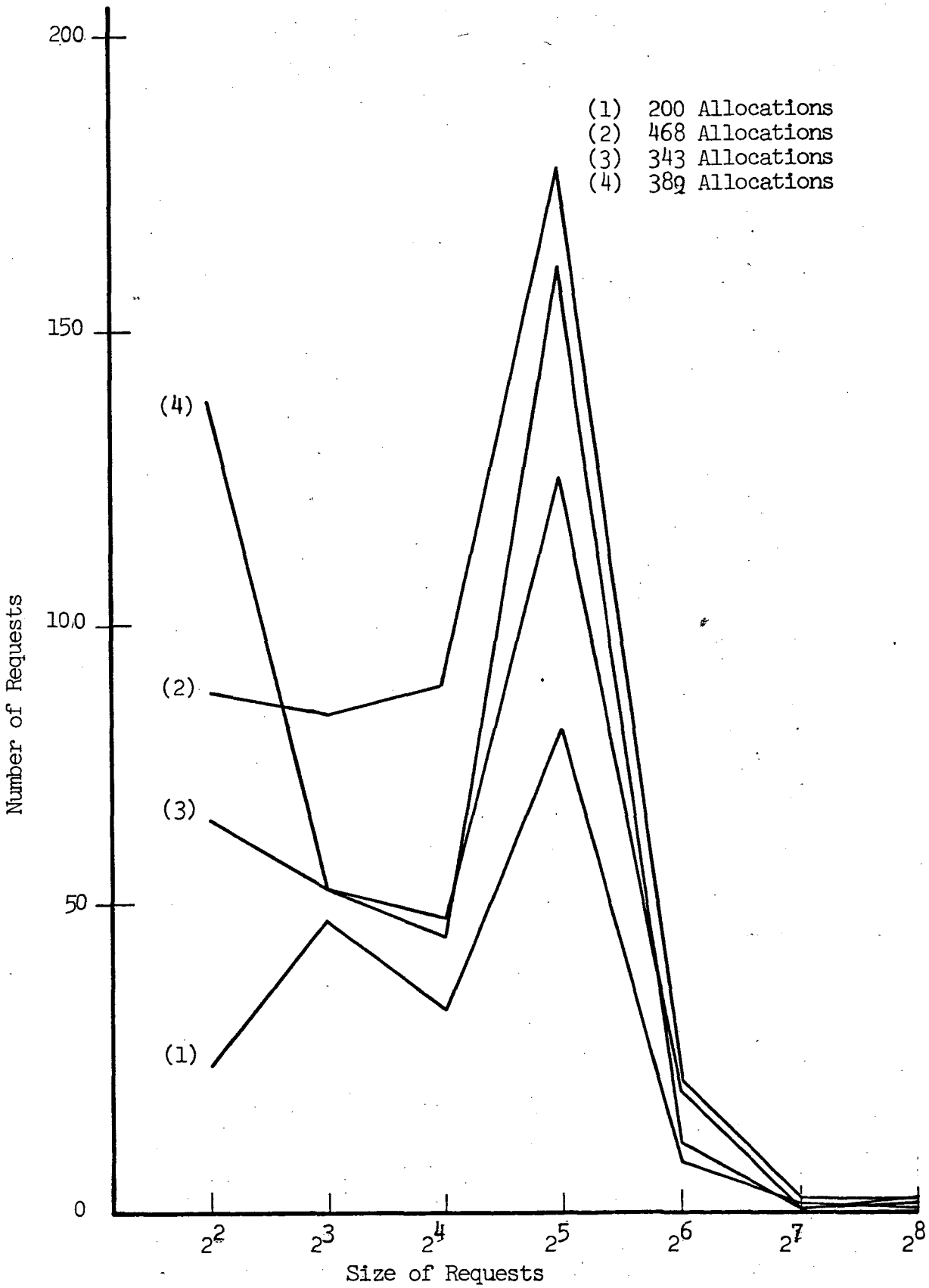


Figure III-1. Distribution of Buffer Requests by Size.

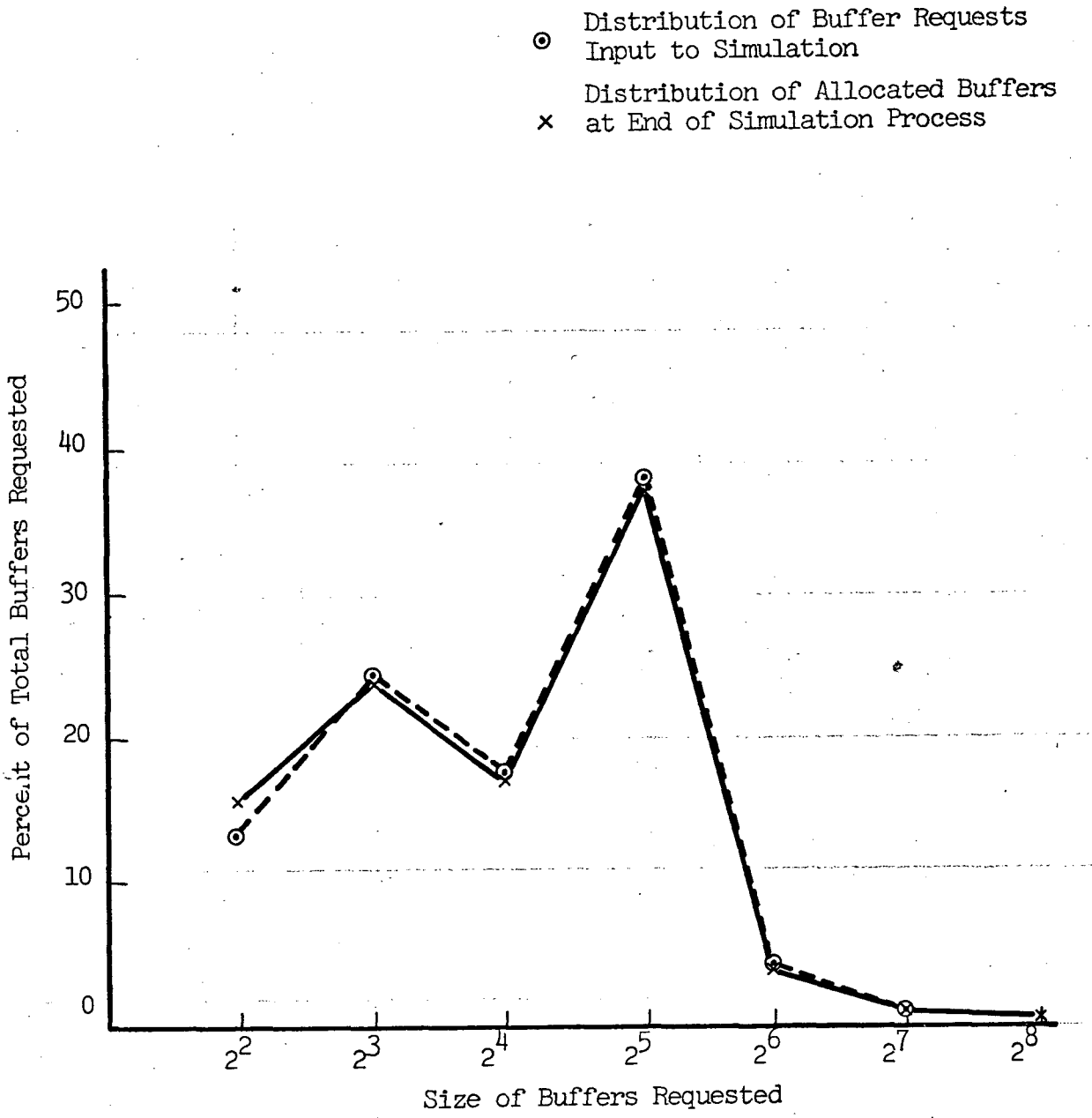


Figure III-2. Comparison of Buffer Request Distributions Input To and Output From the Simulation Model.

requested size. In order to estimate memory loss, data were obtained on the internal memory loss per allocation. This type of memory loss represents memory used for control and is tabulated in Table III-1. Also contributing to memory loss is external fragmentation, a relatively long term effect which is best seen through the use of memory maps. See Figures III-3 to III-6. The effect of this factor may be quite significant and contribute to the allocation time through an increase in the number of searches required to obtain a requested buffer. An estimate of the severity of this problem can be obtained both from a memory map obtained after the allocation process has been in progress for a period of time, and the number of search operations.

Both models, the first-fit and the buddy model, were executed using identical buffer request rate, size, and hold times. The total buffer pool was set at 13312 words of memory. Table III-1 gives a comparison of the operating characteristics of the two schemes.

In view of the results obtained, it seems clear that for the given distribution of requests, the buddy system is superior to the first-fit method if the prime consideration is either time or space. This is further substantiated by constructing and comparing the memory maps at the end of the simulation process. Figures III-3 and III-4 are indicative of the memory loss introduced by the buddy method and the first-fit method respectively. In the first-fit process, the problem of external fragmentation is so severe that although there is sufficient space to satisfy the buffer requests, this space is fragmented so there is insufficient contiguous space. As a result, the requests must be queued and satisfied as releases make memory available, or the total buffer pool is extended.

	BUDDY	FIRST-FIT
Mean Memory Loss Per Allocation	1	2
Total Memory Allocated	12200 (no queue)	12200 (requests queued for buffers of size 2^5 and greater)
Mean Number of Collapses	.012	.195
Mean Number of Searches	1.554	8.410

Table III-1. Comparison of Buddy and First-Fit Allocation Characteristics.

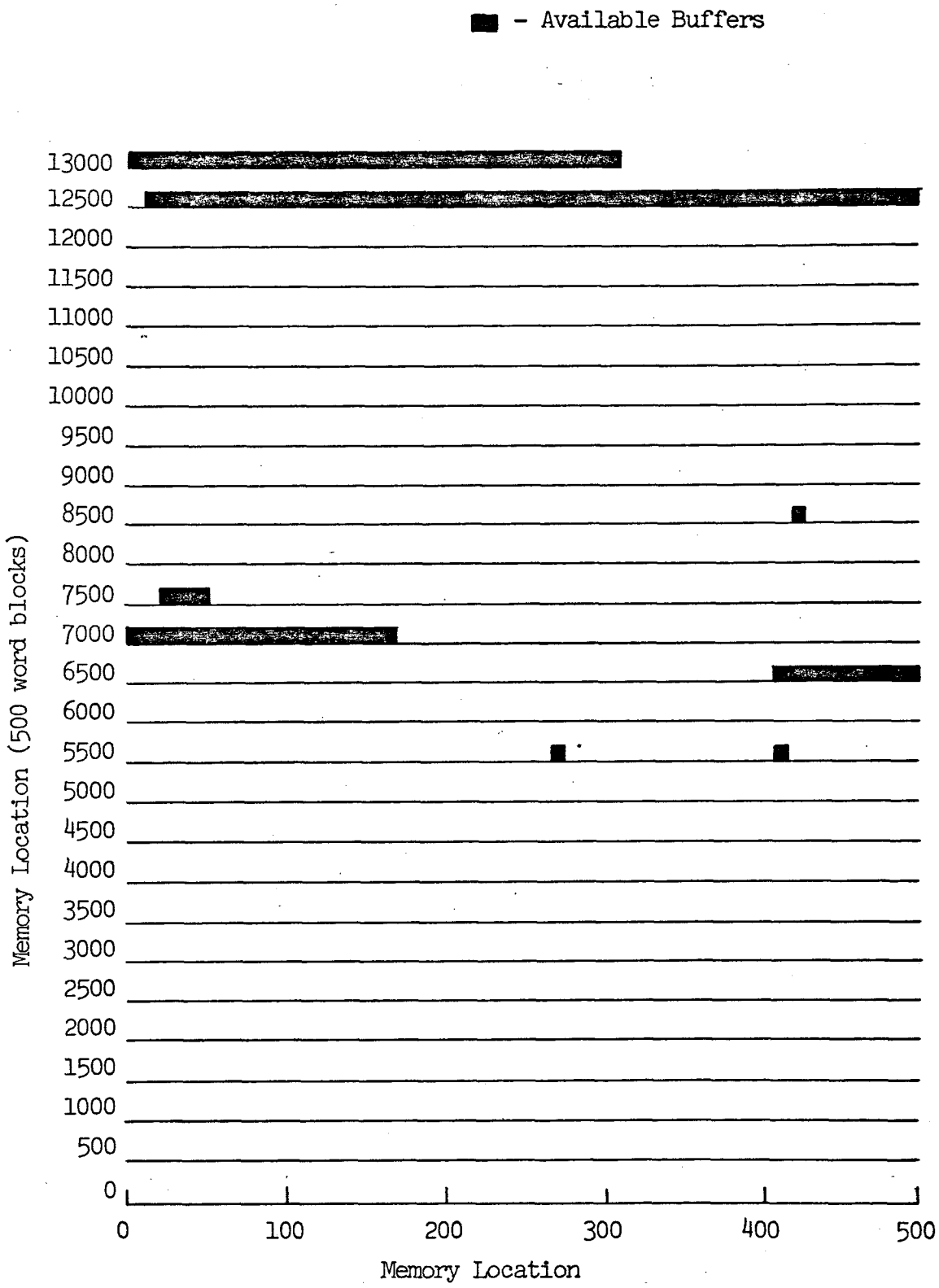


Figure III-3. Buffer Pool Memory Map Resulting from Simulation of Buddy Allocation Scheme. (Map constructed after 926 allocations and 400 releases.)

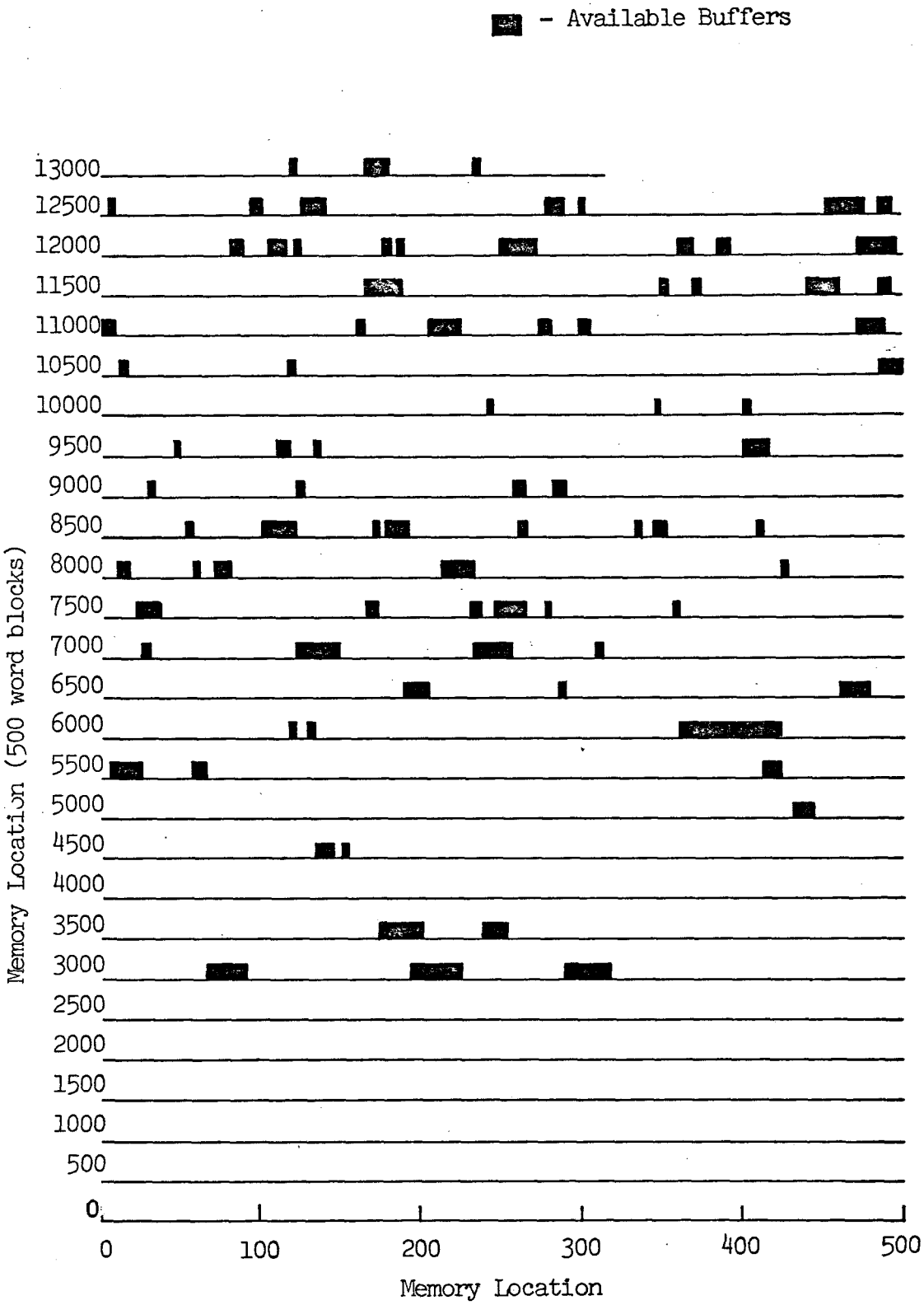


Figure III-4. Buffer Pool Memory Map Resulting from Simulation of First-Fit Allocation Scheme. (Map constructed after 934 allocations and 400 releases.)

Thus, the buddy method is found to be superior in this environment. The questions then are: 'Under what conditions could the first-fit method be comparable or superior to the buddy method?' and 'What modifications could be made to the basic first-fit algorithm to permit more efficient operation?'

G. First-Fit Model Modifications

In the original version of the first-fit model, the following statements characterize the allocation process:

a) the available blocks are maintained on one list.

b) the request sizes are identical to those used in the buddy method, i.e., request sizes are powers of two and it is assumed that no waste is incurred due to restricted request sizes.

c) two words of overhead in each block are used for control.

d) upon request for release of a block, an attempt is made to collapse this block with adjacent blocks in both the forward and backward direction.

1. Modification 1. Maintain Available Buffers by Size. The first modification to this algorithm provided for the same number of lists as used in the buddy method, i.e., one for each acceptable power of two. Since only a limited number of request sizes are made, the available blocks are maintained on lists by size. In Figure III-5, it can be seen from the resulting memory map, that the problem of external fragmentation has been reduced to the point that it is comparable to the buddy method. The results in Table III-2 indicate that in the first-fit method, the memory overhead per allocated block is still

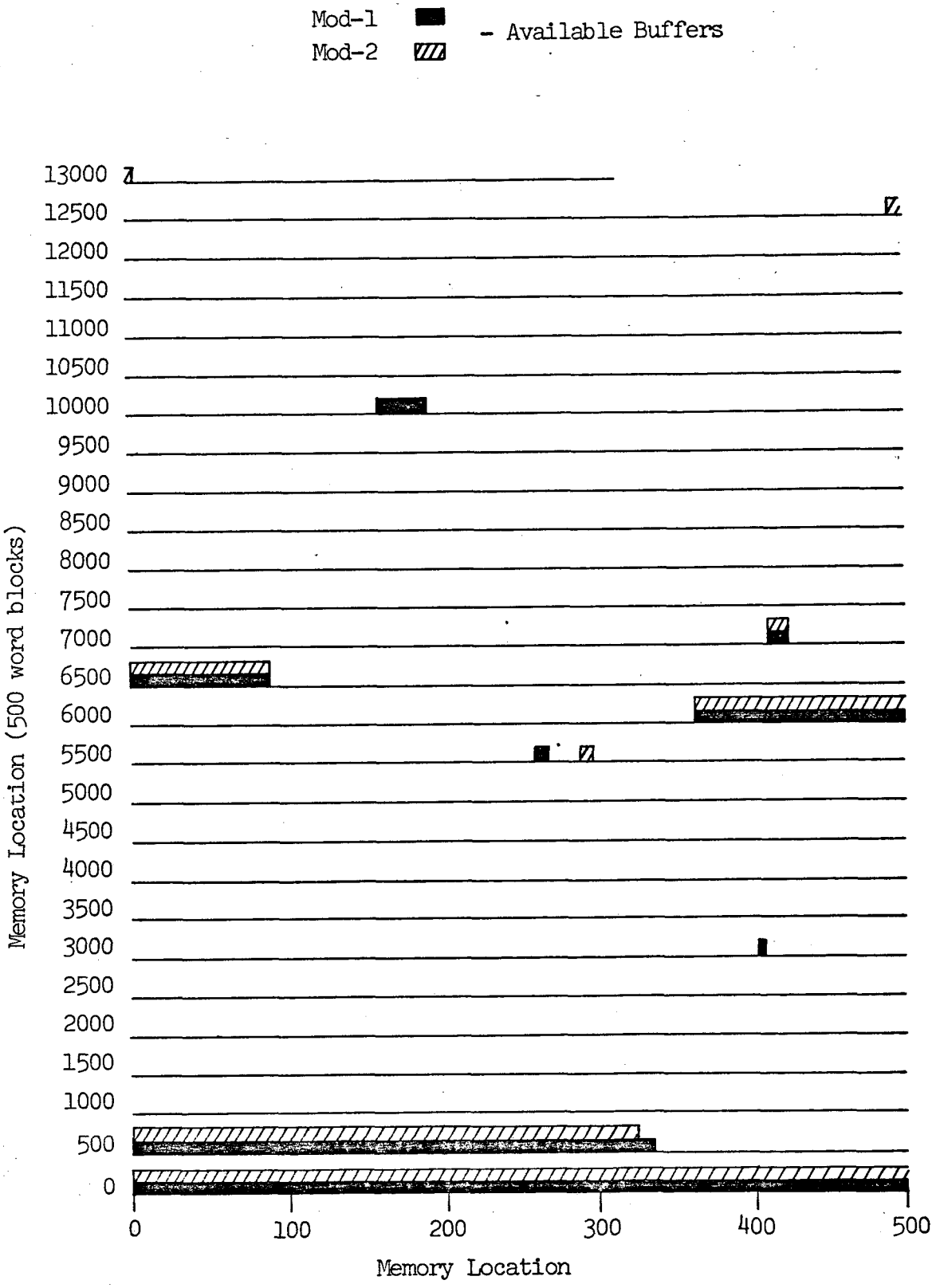


Figure III-5. Buffer Pool Memory Maps Resulting from Simulation of First-Fit Allocation Schemes - Mod-1 and Mod-2. (Map constructed after 926 allocations and 400 releases.)

twice that found in the buddy method; and, if execution time is important, the mean number of searches to find an available block is still significantly greater than that found in the buddy system.

2. Modification 2. Reduce Control Overhead. It was noted in both the buddy method and the first-fit method that the mean number of collapses per release is small. In the first-fit method, this represents collapses in two directions, forward and backward. By making a modification to the algorithm which permitted collapses in the forward direction only, several consequences were foreseen. First, the number of collapses would be reduced by a factor of two. Next, if collapses were attempted in only one direction, one word of overhead would be adequate for control since the last word in each block would not be used in the allocation process. This would make the two methods comparable with respect to memory overhead. Finally, a possibility of increased external fragmentation would be introduced due to the fact that adjacent blocks might be available and unusable because they were not coalesced into one block. From the memory map given in Figure III-5, it can be seen that no appreciable increase in external fragmentation resulted. The results in Table III-2 indicate an improvement in the overhead required and a reduction in the number of collapse operations. The mean number of search operations is essentially unchanged.

3. Modification 3. Permit Variable Request Sizes. In each of the foregoing tests, it was assumed that the number of words requested was the exact number of words needed by the requestor. Suppose this were not the case. Then the buddy method, as well as the first-fit method, have introduced internal memory waste which has not been apparent

	FIRST-FIT MOD-1	FIRST-FIT MOD-2	FIRST-FIT MOD-3	BUDDY MOD-1
Mean Memory Loss Per Allocation	2.0	1.0	2.454	5.962
Total Memory Allocated	12200	12200	9552	12200
Mean Number of Collapses	.035	.015	.045	.012
Mean Number of Searches	2.713	2.713	3.262	1.554

Table III-2. Comparison of Simulated Allocation Characteristics.

or considered in the preceding comparisons. In the case of the buddy system, it is impossible to eliminate this kind of memory loss, if it exists, since the block sizes are essential to the formulation of the buddy method. However, the first-fit algorithm imposes no restriction on the buffer size requested. The first-fit simulation model was then modified to generate exact buffer requests. The original distribution of request sizes was used to determine the range of a generated request size. A continuous function was used to obtain the exact number of words needed. For example, if a block of size 32, (2^5), were requested in previous runs, the block size generated in this test was some number between 2^4 and 2^5.

A further modification was made to the first-fit algorithm to handle a condition which had not been present up to this point. Since the buffer sizes were now permitted to be any size, a block returned to the available list could be so small that it would be virtually useless in satisfying future requests. For example, suppose a request size of n is allocated from a block of either n+1 or n+2 words. Then using the existing algorithm, a block of either one or two words is returned to the available list. Since request sizes were from the outset of this study assumed to be ≥ 2, it would be impossible to use available blocks of < 4 words if 2 words of overhead are assumed, or < 3 words if 1 word of overhead is assumed. In the interest of returning only useful buffers to the available lists, a constant was introduced. If the difference between the buffer size requested and the available buffer from which the allocation was made were less than some constant, the whole block was allocated. In the simulation model this constant was set at 4 with

the result that no block < 4 is placed on the available lists.

The results obtained using this model were viewed with mixed feelings. On the one hand, the total amount of memory actually allocated was considerably less than in any previous model and the internal memory waste per allocation was small. On the other hand, the external fragmentation problem is again significant as can be seen in Figure III-6. Also, in Table III-2, it should be noted that the number of searches to find an available block has increased.

Using the buddy method and fixed request sizes and assuming the same actual utilization of buffers requested, the mean internal memory loss per allocation was found to be close to six words per allocation. It is clear from the size of this number that this memory loss is rather severe. If the buffers needed are large, there is no guarantee that the size actually needed is close to but less than some exact power of two. There is the same probability that it will be close to but greater than a power of two, in which case approximately one half of the allocated buffer is unused.

There is the possibility that the requestor is careful to make his requests in segments if significant internal memory loss is incurred by a single request. For example, if a buffer of 70 words is needed, a buffer of size 2^7 may be requested resulting in 57 unused locations. The alternative procedure is to make two requests, one for a buffer of 2^6 and one for a buffer of 2^3 which results in no internal waste. If this procedure is followed, it is always possible to keep the internal memory waste small. It should be noted, however, that this is a very clear case of a space-time tradeoff, since in order to use memory effectively, it may be necessary to break one request into two or more

I A

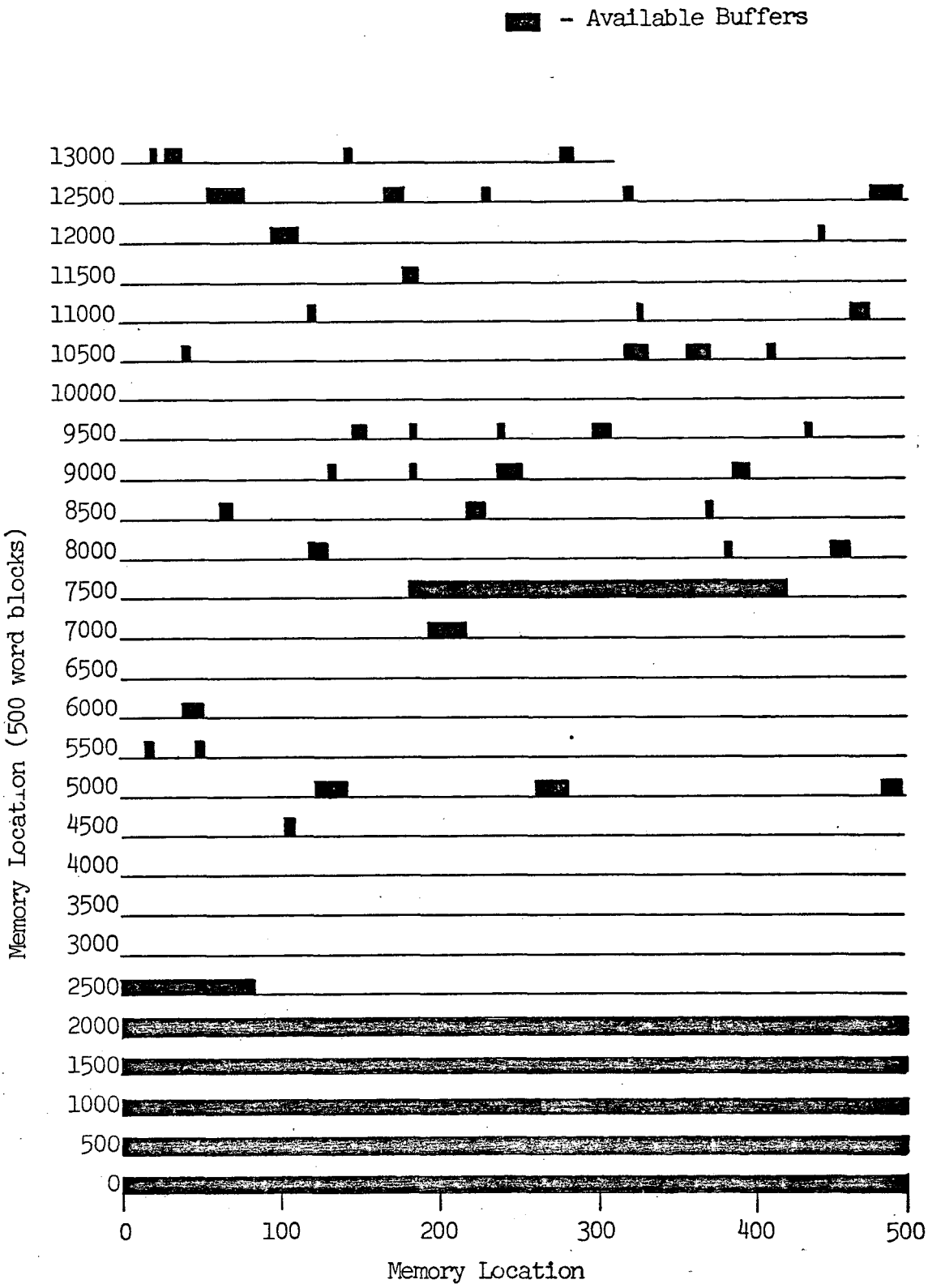


Figure III-6. Buffer Pool Memory Map Resulting from Simulation of First-Fit Allocation Scheme - Mod-3. (Map constructed after 926 allocations and 400 releases.)

requests. As a result, the number of buffers allocated and released is increased and the total allocation time is incremented accordingly.

H. Memory Waste Determination

In an effort to establish some measurable parameter which could be used to determine the choice of an algorithm, the data found earlier reflecting memory utilization was analyzed further. Of particular interest was a comparison of the memory allocated using the buddy method with the memory allocated using the first-fit method. Using the data presented in Table III-2, the following characterization of memory loss was formulated. The result is an indication of internal memory waste.

If, as in the buddy system, the size of each buffer is a power of two, and further, the requests are generated by sampling a given probability distribution, then the number of words allocated can be written as:

$$N \cdot \sum_{i=1}^{n} p_i 2^i = c_1$$

where,

N = the number of allocations

p_i = the probability that a request will be made for size 2^i.

c_1 = total memory allocated.

Then, if it is assumed that internal loss is incurred due to the fixed request sizes and if the loss is approximated, as in the simulation model, by using a uniform distribution, the space allocated using the first-fit method given exact request sizes is given by;

$$N[\sum_{i=1}^{n} p_i(2^i - \frac{(2^i - 2^{i-1})}{2}) + k] = c_2$$

where,

N = the number of allocations

p_i = the probability that a request will be made for size 2^i

k = constant overhead per allocated buffer

c_2 = total memory allocated.

Considering memory usage only, a cutoff point can be obtained which can be used as a factor in determining an internal loss criterion which is a function of the average request size. The result of rewriting the above equations is, for the buddy method,

$$N \cdot \overline{X} = N \cdot \sum_{i=1}^{n} p_i 2^i = c_1$$

and for the first-fit method:

$$N[\overline{X} - 1/2\ \overline{X} + 1/4\ \overline{X} + k] = c_2$$

where $\overline{X}$ is the average request size. Now setting $c_1 = c_2$ and solving,

$$\overline{X} = 3/4\ \overline{X} + k$$

$$\overline{X} = 4\ k.$$

This indicates that under the above assumptions whenever the average request size is greater than four times the average internal loss of the first-fit method, more memory will be used by the buddy allocation scheme than by the first-fit method. Since internal memory loss is not the only consideration, this factor would not be sufficient, taken alone, to determine that the allocation strategy should be changed. This is particularly true in view of the fact that the allocation time using the buddy method is less than that of the first-fit.

Ultimately, the criterion used should be based on the allocation and release times, memory usage, and possibly the measured rate of change in the request distributions. Further, in a particular application, the variables, time and space, may be assigned weights as a function of their relative importance in a given operating system.

In the interest of substantiating the conclusions of the analysis of relative internal memory loss incurred by the buddy and first-fit allocation schemes, special simulation runs were made. The request distribution used for this aspect of the study provided an average request size of 21.3 words per allocation. Both the buddy and the first-fit algorithms were used. In the first run the average memory loss in the first-fit method was 2.3 words per allocation. From the analysis above, if $\overline{X}$, the average request size, is greater than four times the average internal loss found in the first-fit method, then more memory will be needed using the buddy method than used by the first-fit method. The simulation outputs supported this conclusion. The first-fit allocation routine was then modified so that the average internal loss per allocation was increased to 4.3. Again the memory allocated using the first-fit method was somewhat less than that allocated using the buddy method. A final modification was made to the first-fit routine to make the average internal loss per allocation equal to 8.3 words per allocation which is greater than 21.3/4 which is the crossover figure. In this case, more memory was used by the first-fit method than in the buddy method. This series of simulation runs then validates the results of the analysis. See Table III-3 for a comparison of the simulation results produced in the sequence of the runs described above.

	Current Memory Used	Total Number of Entries	Average Memory Loss per Allocation	Average Allocation per Request
Buddy	9660	19800	5.575	21.3
First-Fit (1)	8248	16788	2.318	18.1
First-Fit (2)	9104	18552	4.277	20.4
First-Fit (3)	10952	22400	8.277	23.2

Table III-3. Comparison of Allocated Memory for Different Average Memory Loss per Allocation.

CHAPTER IV

INVESTIGATION OF ADAPTIVE ALLOCATION STRATEGIES

The remainder of this thesis is concerned with the results of an attempt to design alternative algorithms which are compatible and can be executed in turn as a function of the operating environment. The implementation of an adaptive scheme in a real operating system depends on the solution of two problems. The first involves selecting criteria which accurately reflect change or rate of change of conditions in an operating environment and providing a monitoring device which detects and signals the occurrence and direction of any significant change. The second problem involves devising alternative algorithms and determining the operating conditions under which they are most efficient. If having provided for a monitor which is capable of detecting operating environmental change, and in addition, if having determined which algorithm permits most efficient operation given the environment, the remaining objective is to provide a mechanism for automatically replacing one algorithm by another without interruption or serious degradation to the system operation. The basic algorithms for the dynamic allocation of buffer storage which were simulated individually are used.

In Chapter III-F, it was found that the buddy method is most efficient in time required to allocate and release buffers and in memory utilization if small buffers are predominant in the request distribution. On the other hand, it was found that the first-fit

method allows for economic use of memory in cases where the buffers requested are large and the size of the buffer allocated is unrestricted, unlike the power-of-two restriction implicit in the buddy method.

A. Comparison of Algorithm Characteristics

A brief review of the steps taken in simulating the basic algorithms is appropriate here since the characteristics of the allocation methods are determining factors in making the algorithms compatible. The buddy method requires that allocations be made in blocks which are powers of two. In this study all available buffer space is initially placed on one list in blocks of size 2^9. The range of acceptable requests was from 2^2 to 2^8 words in powers of 2. In turn, one list was maintained for each power of two, from 2 to 8. In the allocation phase the split operation insures that each available buffer on a given list is a power of two and that the start location, x, of that block is such that x modulo 2^i is zero. The proper release and collapse of adjacent blocks are also dependent on the size and start location of the buddy of the block being released.

In the first-fit method as simulated originally, requests for buffers were in powers of two and all available storage was maintained on one list. It was found that the number of searches required to find an available block of adequate size was large and also that core was fragmented to such an extent that requests for block sizes 2^5 and greater were queued. A modification was made to maintain the available storage on 8 lists as in the buddy system. The number of lists and the size of the blocks maintained is arbitrary.

The selection of this scheme was in anticipation of an investigation of compatible modes of allocation. The result was that fragmentation of core and the average number of searches to locate an available block of adequate size were reduced. Further modifications to the first-fit algorithm reduced the overhead per allocation to one word, making collapse of adjacent blocks possible in the forward direction only. Next, the request sizes were unrestricted, which resulted in the maintenance of some essentially unusable small blocks. Finally, to eliminate this latter effect, if the difference between the buffer size requested and the available buffer from which the allocation was made were less than 4, the entire block was allocated. The description of this last modification is correct but incomplete. The implementation of this modification also insured that the size of every buffer allocated would be an even multiple of 4, where 4 is the smallest useful block maintained on the available storage lists. The internal memory loss per allocation introduced by this modification is always less than 4. Here again the attempt to maintain the available storage blocks with start locations which are a power of two was deliberate. It was hoped that the transition from the first-fit method to the buddy method would be facilitated.

The algorithms at this point have the following characteristics in common. Available storage is maintained on eight lists, the start location of any allocated or available buffer is a power of 2, and requests are unrestricted as to size. The difference in the two methods are the following. The buddy method allocates blocks in powers of two while the first-fit allocates blocks in multiples of 4. In the buddy scheme the start location of every block on an available

list 2^i is a multiple of 2^i. In the first-fit method to the start location of a block on an available list 2^i is a multiple of four and only by coincidence is it a multiple of 2^i. The buddy allocation method does not place any limit on the internal memory loss per allocation. The first-fit method insures that the internal memory loss per allocation will be less than four. This upper limit on internal memory loss which is characteristic of the first-fit method is the basis for attempting to make the two modes of allocation interchangeable in an operating environment in which the power of two block size restriction produces a high average memory loss per allocation when using the buddy method.

B. Adaptive Strategies Considered

In view of the differences in the two methods, the transition from the buddy method to the first-fit method presented no difficulties The dependence of efficient split and collapse operations on the block size in the buddy method presented problems in going from the first-fit method to the buddy method. An analysis of these problems and attempts at their solution then become of primary importance.

Some of the alternative approaches considered to resolve these problems are discussed here prior to presenting the scheme which was simulated. For a solution to be acceptable the following conditions were used as guidelines. The efficiency of the allocation and release operations in the buddy method should be preserved. If either operation must be degraded, then it should be the release operation since the time required to satisfy a request for a buffer in a time-sharing environment is usually more critical than the time to

return buffers to the available storage lists. Further, if either the normal allocation or the release operation must be degraded, then it should be for a limited period of time after which the allocation process should return to its normal efficiency.

If the time to honor requests were no problem, then when the buddy method is initiated, all buffers on the available lists could be tested and modified to be acceptable to the buddy method. This direct approach to the problem guarantees the latter condition, that is limited interruption. This is not feasible since the time required to modify these lists is indeterminate and system operations for which the buffers are requested are very often time sensitive.

In both the first-fit and the buddy method, the same number of lists and list pointers to available storage are maintained. If the same list is used for both methods, the following difficulties are encountered. There is no assurance that a buffer on the list is the proper size or has an acceptable start location. This means that in the allocation process each buffer must be tested for size and start location. It may be found that a list is not empty, however, no buffer of adequate size is on the list. If this is the case, then the next higher list must be searched. When a buffer is found of adequate size, the procedure for allocating that block may be time-consuming. The following situations may exist. If the buffer start location is acceptable, an allocation is guaranteed, however, if the buffer is larger than requested, the remainder must be placed on the appropriate list which may be any list with buffers of size less than or equal to the buffer size requested. If the start location is not acceptable, then the lowest acceptable start location in the available

buffer must be determined. The initial portion of the buffer must be placed on another list and now another test on size is needed. If the size is adequate, then proceed as for the case with an acceptable start location outlined above. If the size is now inadequate, continue the list search until a block of adequate size is encountered and repeat the test, split, and return of unneeded portions of buffers to the appropriate available lists.

In view of the difficulties present in going from the first-fit to the buddy method, suppose then that the first-fit method is modified so that buffers are always allocated with acceptable start locations as required in the buddy method. This reduces the number of operations required to allocate in the buddy method but it is still necessary to return portions of buffers to the appropriate list and a test for correct size must be made on every buffer considered for allocation. Further, there is no convenient way of determining when return to normal buddy allocation can be made.

The next modification considered was that of maintaining separate lists of available storage in the two methods. Now when the buddy method is initiated, its lists are empty. Allocations can be made from the first-fit lists as outlined above until the lists are depleted. During this time the allocation of a buffer may be a lengthy procedure. The value of this approach lies in the fact that the duration of degraded performance is limited, that is, until the first-fit lists are empty, after which normal buddy allocation can be resumed.

C. Adaptive Strategy Simulated

The allocation scheme selected for simulation is as follows. Two

separate lists of pointers to available storage are maintained. When going from the buddy method to the first-fit method, the header list consisting of eight pointers to available buffers is transferred directly to the first-fit header list and the buddy header list is cleared. Allocations are then made using the first-fit allocation and release process with no further interruption to the system. When it is determined that the buddy allocation method should be initiated, a word is set equal to the largest buffer on the first-fit available list.

With each buffer request, the buddy routine is then entered. If the buffer requested is less than the flag word, the first-fit allocation routine is entered, and if possible the allocation is made by that routine. If the buffer requested can not be satisfied, the flag word is reset indicating that all future requests equal to or greater than the flag word should be allocated using the buddy method. Very rapidly the first-fit lists are depleted and all allocations are then made using the buddy allocation method. The time loss in this process is the time it takes for the one test which determines whether the allocation should be made using the first-fit or the buddy method. Memory loss is incurred during the transition phase since both allocation schemes must be present in core until the first-fit lists are depleted. A savings in memory space per allocation may be realized since the first-fit overhead per allocation is still limited and in general is less than that of the buddy method.

The foregoing discussion has been concerned with the allocation process primarily. The effect on the release process must also be considered. Since the buffer being released could have been

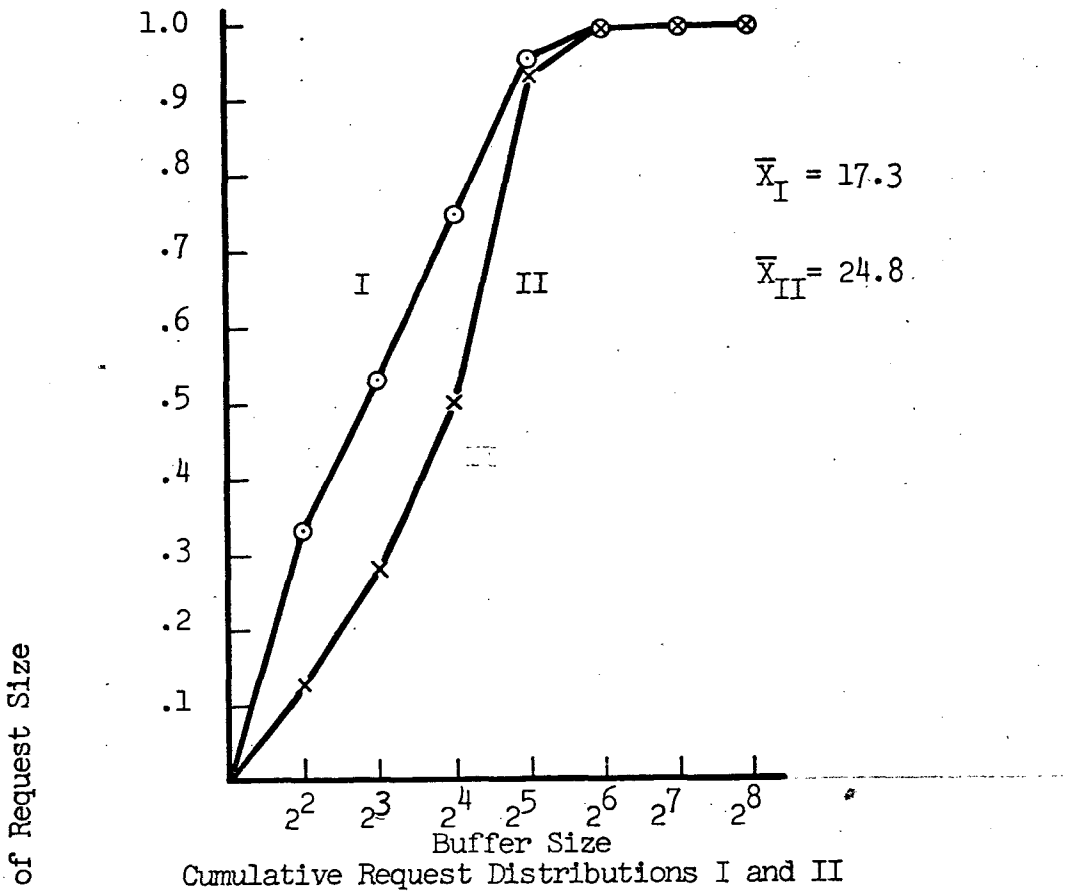
allocated using either the buddy method or the first-fit method, the size and start location must be checked prior to returning a buffer to an available storage list. If the buffer were allocated using the buddy allocation, it is clear that the normal release procedure could be followed. If the buffer were allocated using the first-fit method, then the release procedure becomes more involved. If the size is not a power of two, it is necessary to check the start location, determine the largest buffer for which that start location is acceptable, return a buffer to the list of that size, reduce the size of the returned buffer by that amount and continue this process until the entire buffer has been returned to the available lists. To eliminate the initial testing prior to release in the case where the allocation was made using the buddy method, it was decided that the buddy allocation should insert a negative sign on the size of the buffer when it is allocated. When a buffer is returned which was allocated using the buddy method, a sign test is the only additional operation introduced in the release process.

When buffers are returned which were allocated using the first-fit method, the splitting operations required to return the buffer to the appropriate available lists increases the number of small buffers on the available lists. This is not a serious problem since the primary reason for making a transition to the buddy method is because the average buffer size requested is decreasing. The only serious penalty paid is in the time required to split and return the buffers to the lists if they were allocated using the first-fit method and are not a power of two. These operations become more infrequent after the buddy allocation has been in operation for a period of time.

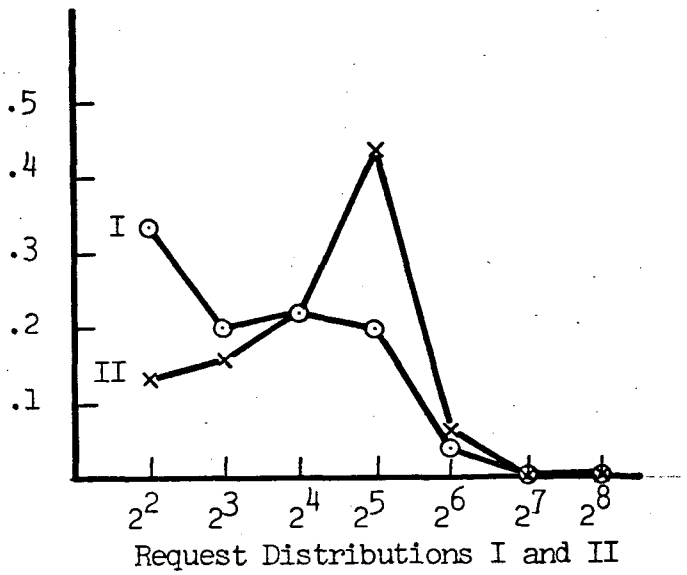
D. Results from Simulation of Adaptive Model

The following conditions were used for the final simulation runs in this study. Four request distributions were selected with average request sizes of 17.3, 24.8, 18.7, and 37.4 respectively. The distributions were used in pairs, 17.3 with 24.8, and 18.7 with 37.4. (See Figures IV-1 and IV-2.) The distributions selected are not representative of any actual operating system request distributions, but are used to represent changes in request distributions which could occur within an operating system. The total number of buffers which were both allocated and returned to available storage was set at 1000. The adaptive system was run using the buddy method to allocate requests generated using distribution I followed by the first-fit method to allocate requests generated using distribution II. The methods were then reversed so that the first-fit method was used with distribution I and the buddy method with distribution II. Two additional runs were made, one with the buddy method throughout and one with the first-fit throughout. The distributions were then interchanged and the same procedure was followed to produce four more runs. The second pair of distributions, distribution III and IV, were used in the same way to complete the simulation study.

Table IV-1 presents the memory usage results of the simulation runs using the request distributions I and II. Table IV-2 presents the results using distributions III and IV. In the first set of data, no queues of requests were formed. All buffers could be allocated as requested. In the second set of data, queues were formed in all runs. In an actual system, more memory would be allocated to the buffer pool so that queues would not be present. In the simulation this procedure

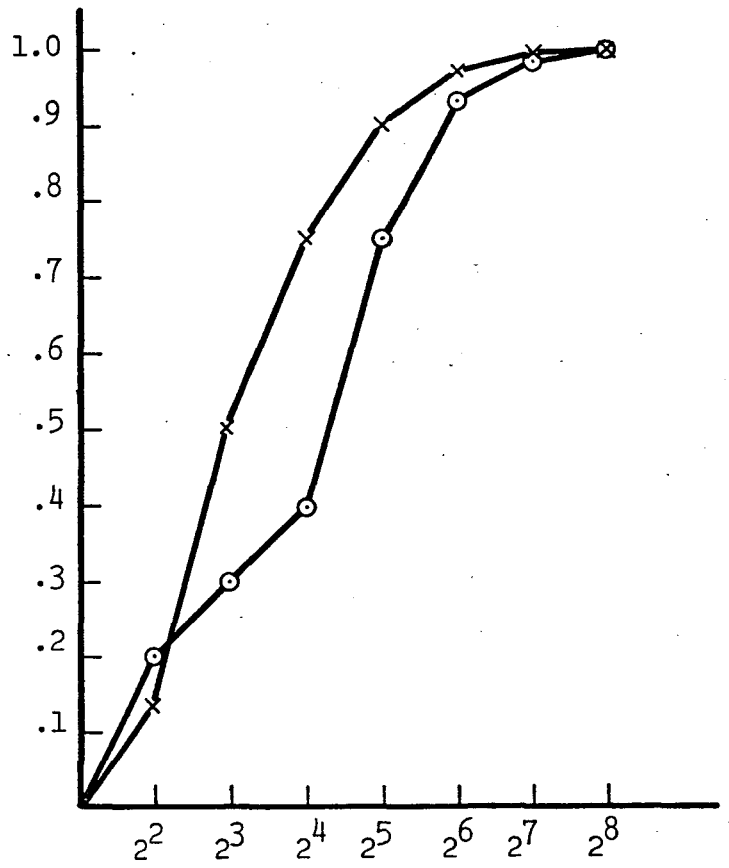


Cumulative Request Distributions I and II



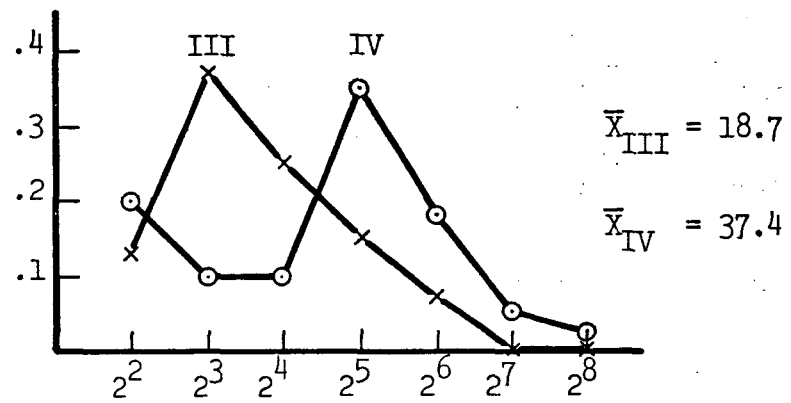
Request Distributions I and II

Figure IV-1. Buffer Request Distributions I and II Used in Adaptive Method.



Cumulative Request Distributions III and IV

Probability of Request Size



Request Distributions III and IV

Figure IV-2. Buffer Request Distributions III and IV Used in Adaptive Method.

	Current Memory Contents	Total Number Entries	Average Memory Loss	Queues
First-Fit → First-Fit	10064	24976	2.324	None
Buddy → First-Fit	10456	27404	3.862	None
First-Fit → Buddy	11712	27480	3.910	None
Buddy → Buddy	12104	29908	5.448	None

Distribution I followed by Distribution II.

First-Fit → First-Fit	8608	27004	2.373	None
First-Fit → Buddy	9556	28416	3.267	None
Buddy → First-Fit	9304	30908	4.845	None
Buddy → Buddy	10252	32320	5.740	None

Distribution II followed by Distribution I.

Table IV-1. Results of Simulation Runs
Using Request Distributions I and II,
with Average Request Sizes of 17.3 and 24.8 Respectively,
After 1579 Requests and 1000 Releases.

	Number of Requests	Current Memory Contents	Total Number Entries	Requests on Queue	Total Requests	Average Memory Loss
F → F	1574	13040	32160	640	32800	2.410
B → F	1584	13380	34980	2048	37028	4.340
F → B	1579	12220	32328	6368	38696	4.398
B → B	1567	14268	37584	4128	41712	6.941

Distribution III followed by Distribution IV.

F → F	1564	10928	37364	0	37364	2.398
F → B	1541	11388	38232	392	38624	3.360
B → F	1598	13592	44312	4608	48920	6.451
B → B	1643	14828	47076	4928	52004	8.159

Distribution IV followed by Distribution III.

Table IV-2. Results of Simulation Runs
Using Request Distributions III and IV,
with Average Request Sizes of 18.7 and 37.4 Respectively,
After 1000 Releases.

was not possible, so the length of the queues serve as a measure of the severity of the memory loss incurred using the alternative allocation schemes to handle different request distributions.

The results obtained are discussed in terms of memory loss only since the coding of the adaptive model was not optimized and, as a result, timing performance data were not available. In all cases, it was found that internal memory loss was at a minimum when the first-fit was used throughout. The maximum internal memory loss occurred when the buddy method was used throughout. These results are as expected and further substantiate the results of the analysis of Chapter III-H. Beyond this observation, these cases are of little interest.

Of special interest here is the performance of the two methods employed adaptively. The total memory allocated was found to be least when the buddy method was used to allocate the smaller average request sizes generated using distribution I followed by the first-fit method to allocate the larger average request sizes generated using distribution II. This can be seen from the data presented in Table IV-1. The total memory used in this case was 27404 words. Keeping the distributions in the same order and reversing the methods used with them respectively, the total memory used was 27480 words. The difference in this case was 76 words, a very small differential. When the first-fit method was used to allocate requests generated using distribution II followed by the buddy method to allocate requests generated using distribution I, the memory used was 28416 words. Again reversing the methods and keeping the distributions in the same order, the memory used was 30908 words. Here, the difference is 1492 words, or approximately a 5% memory increase, a slightly

larger differential but probably not significant.

Using distributions III and IV, queues were formed. Used in this comparison is the total number of words requested, that is the number of words actually allocated and the number of words represented by queued requests. The buddy method used with distribution III followed by the first-fit used with distribution IV resulted in requests totaling 37028 words of memory. By reversing the methods, 38696 words were requested. The difference is 1678 words or approximately a 5% memory increase. When the first-fit method was used with distribution IV followed by the buddy method with distribution III, the total memory requested was 38624 words. Reversing the methods resulted in memory requests of 48920 words. The difference here is 9296 words, an approximate increase in memory of 25%, which is quite significant. The reason for this is that the buddy method, used to allocate large average request sizes, had introduced internal waste and as a result had quickly exceeded the buffer pool. Subsequent large requests were then queued so that when the first-fit method was initiated to handle the smaller requests, there were many large request on queues which still needed to be serviced.

Table IV-3 presents data pertinent to the queue formation in these last runs. The presence of queues is important. The number of queue entries at the end of the run and the maximum contents of the queues indicate the interactive effect of the two methods when used adaptively. In going from distribution IV to III, the number of queue entries at the end of the simulation run is an indication of how well a given method recovers and handles queues once they have been formed. It should be pointed out that the buddy method used throughout gave

the worst recovery performance. This can be seen in Table IV-3. It was also evidenced in this case by the increased simulation run time.

In using distribution III followed by distribution IV, the queue formation was worst when going from the first-fit method to the buddy method. This is a result of the external fragmentation of returned buffers which were allocated using the first-fit method. Since some of the large buffers returned were split, there was an increase in the number of requests for large buffers which could not be satisfied.

In summary, it was found that in all cases, the total memory allocated was less when the buddy method was used to allocate the smaller request sizes and the first-fit was used to allocate the larger request sizes. When no queues are formed, the difference is minimal and the use of an adaptive strategy does not appear to be warranted. It is precisely the case where the memory pool is limited that an adaptive strategy is needed. When queues are formed and the buddy method is used with the larger average request sizes, the internal memory waste is significant and the duration of degraded allocation performance continues even beyond the point at which the request distribution again becomes favorable for use with the buddy method. It is quite clear in such cases that advantage is realized by using the allocation methods adaptively as a function of the request distributions.

In using such an adaptive structure where the method used is based on the request distribution, it appears that the use of the average request size at any given time may not be sufficient. The rate of change in request sizes may be equally important. Using the average request size and the rate of change in request size, it would

	Buffer Size	Maximum Queue Length	Total Requests Queued	Queue Contents at End of Run
F - F	2^5	2	2	0
	2^6	6	6	6
	2^7	2	2	2
B - F	2^5	5	6	0
	2^6	12	18	12
	2^7	6	7	6
	2^8	2	2	2
F - B	2^5	37	108	37
	2^6	41	79	41
	2^7	10	15	10
	2^8	5	8	5
B - B	2^5	21	101	21
	2^6	22	57	22
	2^7	6	18	6
	2^8	5	8	5

Distribution IV followed by Distribution III
with Average Request Sizes of 37.4 and 18.7 Respectively.

Table IV-3. Queue Formation Produced as Function of Adaptive Scheme Employed.

	Buffer Size	Maximum Queue Length	Total Requests Queued	Queue Contents at End of Run
F ⌐ F	2^6	1	1	0
	2^7	1	2	0
	2^8	1	2	0
F ⌐ B	2^6	2	7	0
	2^7	1	1	0
	2^8	3	8	3
B → F	2^2	2	21	0
	2^3	6	65	0
	2^4	16	69	0
	2^5	27	155	0
	2^6	21	79	0
	2^7	10	31	4
	2^8	16	21	16
B → B	2^2	14	242	4
	2^3	31	357	14
	2^4	24	285	8
	2^5	27	235	10
	2^6	21	104	8
	2^7	14	36	8
	2^8	13	19	11

Distribution III followed by Distribution IV

with Average Request Size of 18.7 and 37.4 Respectively.

Table IV-3 (continued). Queue Formation Produced

as Function of Adaptive Scheme Employed.

be possible to predict that alternative methods should be employed so that the methods could be interchanged prior to depleting the buffer pool and prior to experiencing degraded performance in the allocation process.

Assuming that the modifications made to the basic methods did not change their relative allocation times significantly, that is, the buddy method is faster than the first-fit method, and given the results of this study that the internal memory loss incurred by the first-fit is less than that of the buddy method, then it follows that an adaptive scheme should be employed. As a result, optimal space-time tradeoffs can be made as the system is operating. In an actual operating system, this requires that there be internal system monitoring which provides an estimate of average request sizes and the rate of change in the request size. With this information and an adpative strategy, such as that simulated in this study, the algorithms could be interchanged based on the system operating characteristics prior to system degration which results as a function of a given algorithm being used in an unfavorable environment.

BIBLIOGRAPHY

1. Rosenthal, S., "Analytical Technique for Automatic Data Processing Acquisition", Proc. AFIPS 1964 SJCC, 359-366.

2. Joslin, E. O., "Cost-Value Technique for Evaluation of Computer System Proposals", Proc. AFIPS 1964 SJCC, 367-381.

3. Auerbach Standard EDP Reports, Auerbach Information, Inc., Philadelphia, Pa.

4. Herman, D. J., Ihrer, F. C., "The Use of a Computer to Evaluate Computers", Proc. AFIPS 1964 SJCC, 383-395.

5. Ihrer, F. C., "Computer Performance Projected Through Simulation", Computer Autom., 17,4 (April 1967), 22-27.

6. Calingaert, P., "System Performance Evaluation: Survey and Appraisal", CACM 10,1 (January 1967), 12-18.

7. Shemer, J. E., "A Mathematical Analysis of Input/Output Interference in a Time-Sharing Information Processing System", Technical Information Series R63CD13, GE Co., Phoenix, Arizona, November 1963.

8. Shemer, J. E., Shippey, G. A., "Statistical Analysis of Paged and Segmented Computer Systems", IEEE Trans. EC-15 (December 1966), 855-863.

9. Denning, P. J., "Thrashing: Its Causes and Its Prevention", Proc. AFIPS 1968 FJCC, 915-922.

10. Coffman, E. G., "Analysis of Two Time-Sharing Algorithms Designed for Limited Swapping", J.ACM 15,3 (July 1968), 341-353.

11. Coffman, E. G., Kleinrock, L., "Feedback Queueing Models for Time-Shared Systems", J.ACM 15,4 (October 1968), 549-576.

12. Denning, P. J., "Resource Allocation in Multiprocess Computer Systems", (Ph.D. Dissertation), Tech. Report. MAC-TR-50, MIT, Cambridge, Mass., 1968.

13. General Purpose System Simulator II (GPSS-II). "Reference Manual", Univac Manual No. UP-4129.

14. Markowitz, H. M., Hausner, B., Karr, H. W., Simscript: A Simulation Programming Language, Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1963.

15. Computer System Simulator/360 Program Description and Operations Manual, (IBM Confidential), IBM Form No. Y20-0130.

16. Cohen, L. J., Associates, System and Software Simulator: S3, Technical Manual, (AD679-269 - AD679-272).

17. Chu, Y., "An Algol Like Computer Design Language", CACM 8,10 (October 1965), 607-615.

18. Grice, A., Hargol - A Hardware Oriented Algol Language, Internal Report No. VA5, August 1966, A/S Regnecentralen, Copenhagen, Denmark.

19. Pinkerton, T. B., Program Behavior and Control in Virtual Storage Computer Systems, (Ph.D. Dissertation), Technical Report 4, University of Michigan, Ann Arbor, Michigan, 1968.

20. Saltzer, J. H., "The Instrumentation of Multics", ACM 2nd Symposium on O/S Principles, October 1969, 167-174.

21. Conti, C., "System Aspects: System/360 Model 92", Proc. AFIPS 1964 FJCC, 81-95.

22. Estrin, G., Hopkins, D., Coggan, B., Crocker, S. D., "SNUPER Computer - A Computer in Instrumentation Automation", Proc. AFIPS 1967 SJCC, 645-656.

23. Russell, E. C., Estrin, G., "Measurement Based Automatic Analysis of Fortran Programs", Proc. AFIPS 1969 SJCC, vol. 34, 723-732.

24. Schulman, F. D., "Hardware Measurement Device for IBM System/360 Time-Sharing Evaluation", Proc. ACM 22nd National Conference, 103-109.

25. Crooke, S., Minker, J., "Key Word in Context Index and Bibliography on Computer Evaluation Techniques", University of Maryland Technical Report 69-100, December 1969.

26. Deniston, W. R., "SIPE: A TSS/360 Software Measurement Technique", Proc. of ACM 24th National Conference, 229-245.

27. Roek, D. J., Emerson, W. D., "A Hardware Instrumentation Approach to Evaluation of a Large Scale System", Proc. of ACM 24th National Conference, 351-367.

28. Systems Measurement Software (SMS/360), User's Guide for CUE-1, Boole and Babbage, Report No. 135, February 1969.

29. Systems Measurement Software (SMS/360), User's Guide for PPE, Boole and Babbage, Report No. 41, May 1969.

30. News Briefs in Datamation, March 1969, p. 109.

31. Denning, P. J., "A Statistical Model for Console Behavior in Multiuser Computers", CACM 11,9 (September 1965), 605-612.

32. Knuth, D. E., The Art of Computer Programming, Vol. 1, Fundamental Algorithms, Addison - Wesley, Menlo Park, California, 1968.

33. Minker, J., Crooke, S., Yeh, J., "Analysis of Data Processing Systems", University of Maryland Technical Report No. 69-99, December 1969, p. 103.