UNIVERSITY OF MARYLAND
COMPUTER SCIENCE CENTER

COLLEGE PARK, MARYLAND

Technical Report 69-99 December 1969
NGR 21-002-197

ANALYSIS OF DATA PROCESSING SYSTEMS

by

Jack Minker
Sarah Crooke
James Yeh

This research was supported by NASA Grant NGR 21-002-197 to the Computer Science Center of the University of Maryland.

TABLE OF CONTENTS

Chapter		Page
1.	INTRODUCTION	1
	1.1 Organization of Report	1
	1.2 Summary of Results	2
	1.2.1 Simulation Results	2
	1.3 Software Monitoring Results ..	4
	1.4 Analytical Modeling	6
	1.5 Bibliographic Research Results	6
2.	AN OVERVIEW OF COMPUTER SYSTEM EVALUATION APPROACHES	7
	2.1 Development of Computer Evaluation Techniques	8
	2.2 System Measurement Tools	12
	2.2.1 Analytical Modeling	12
	2.2.2 Simulation	13
	2.2.3 Software Monitoring	15
	2.2.4 Hardware Monitoring	17
	2.3 The Use of Multiple Measurements	19
	2.4 Specific Applications	19
	2.5 Conclusion	21
3.	ANALYSIS OF DYNAMIC ALLOCATION STRATEGIES	23

3.1 Scope of Initial Study 24

3.1.1 Function Parameters 25

3.1.2 Pooled Versus Private Buffers 26

3.2 Buffer Allocation Algorithms .. 28

3.3 Simulation Language 29

3.4 Basic Buffer Allocation Algorithms 31

3.4.1 Simulation Models Developed 34

3.4.2 Buffer Allocation (First-Fit) 35

3.4.3 Buffer Release (First-Fit) 38

3.4.4 Buddy System Allocation ... 38

3.5 Inputs to the Simulation Models 41

3.6 Outputs from the Simulation ... 43

3.7 First-Fit Model Modifications . 49

3.7.1 Modification 1. Maintain Available Buffers by Size . 49

3.7.2 Modification 2. Reduce Control Overhead 51

3.7.3 Modification 3. Permit Variable Request Sizes 53

3.8 Consideration of an Adaptive Approach 56

3.8.1 Provisions of a Self-Adaptive System 56

3.8.2 Proposed Extension to the Current Study 59

4. REVIEW OF SOME EXISTING HARDWARE AND SOFTWARE MONITOR TECHNIQUES 61

4.1 Hardware Measurement Techniques 61

4.1.1 IBM 7090 Hardware Measurement Technique 61

4.1.2 IBM System/360 Hardware Measurement Technique (TS/SPAR) 63

4.1.3 UNIVAC 1108 Hardware Measurement Technique 66

4.2 Software Measurement Techniques 68

4.2.1 GE GECOS II, GECOS III Software Measurement Techniques 69

4.2.2 CDC 6600 CHIPPEWA Software Measurement Technique 70

4.2.3 IBM TSS/360 Software Measurement Technique (SIPE) 70

4.2.4 IBM OS/360 Software Measurement Technique (SMS/360) 72

5. SYSTEM FUNCTION ANALYSIS USING SOFTWARE MONITOR TECHNIQUES 74

5.1 TRACE 74

5.2 ITFVA (Instructional Trace and Functional Value Analysis) 76

5.3 OPSDE (EXEC 8) Operating System Performance Data Extractor 80

5.4 Other Techniques Under Consideration 82

5.5 Performance Evaluation Analysis of EXPOOL..................... 83

5.5.1 The Buffer Pool 83

5.5.2 Request and Release Buffers from EXPOOL 85

5.6 Preliminary Results of an Analysis of EXPOOL 86

6. ANALYTIC STUDIES SUMMARY 97

6.1 Storage Requirements for Information Handling Centers (Abstract) 97

6.2 A Stochastic Model of an Information Center (Abstract) ... 98

7. TECHNICAL REPORTS AND PUBLICATIONS ... 99

8. QUARTERLY REPORTS 100

BIBLIOGRAPHY 101

1. Introduction

The work described in this technical report represents the final report on the work performed under NASA Grant NGR21-002-197 entitled, 'Analysis of Data Processing Systems'. Two major tasks were to be undertaken during the study. These were in the areas of problem definition and model development. The major function of problem definition was to characterize multiprogramming, multiprocessing computer systems in terms of hardware, software, personnel, and operating environment so that such systems or portions of them could be evaluated using analytic and simulation techniques. In the process, consideration was given to tools that could be developed and used to gather statistical data so as to permit practical measurement of the research results. In the area of model development, a portion or function of a multi-computer system was to be defined and analyzed.

1.1. Organization of Report

In this section, the results from the major tasks undertaken in this study are summarized. In Section 2, an overview is provided of the work performed and reported in the field of computer evaluation. The evaluation techniques included cover the topics of simulation, mathematical modeling, software monitoring, and hardware monitoring. In Section 3, work performed under this grant to analyze several alternative dynamic allocation strategies which may be implemented in an operating system is described in detail. The analysis focuses on a specific computer system so that the initial studies can be validated using realistic data. A simulation model has been developed which permits the use of a particular allocation scheme implemented in the Univac 1108 to be compared with other strategies. In Section 4, some of the existing hardware and software monitoring techniques that have been reported upon in the computer literature are reviewed in greater detail than was done in Section 2, and hence supplement aspects of the work reported on in that section. In Section 5, specific techniques are described which have been developed under this grant to permit the monitoring of software code. The general principles of the techniques which have specific implementation on the Univac 1108 are described. Section 6 summarizes the basic results obtained in the investigation of analytical models of portions of computer systems. The details of the work are presented in reports referenced in Section 7 which contains publications and reports issued under this grant. Finally, a bibliography of documents referenced directly in this report is provided.

1.2. Summary of Results

The work performed under this grant falls into four categories. These include simulation modeling, software monitoring, analytical modeling and bibliographic research.

1.2.1 Simulation Results

The simulation studies involved modeling a basic function of an operating system. In particular, the dynamic allocation of buffer storage was selected for study. The 'buddy' method (described in detail in Section 3.4.4), the strategy implemented in the Univac 1108 executive system, was modeled first. Subsequently, other allocation schemes were modeled for comparison and evaluation purposes. Details of the work performed and the models developed are given in Section 3 of this report. The basic results of this aspect of the work are summarized here.

. Simulation Effectiveness . Simulation was found to be an effective tool for varying the input data, for testing alternative design strategies, and for defining under what operating conditions one strategy would be expected to be preferable to others. (See Section 3.)

. Buddy Method vs. First-Fit Method . Outputs from the models indicated that the buddy method of dynamic allocation of buffers was more efficient than the first-fit method in terms of time and space for the type of buffer requests characteristic of the University of Maryland Univac 1108 operating system. (See Section 3.6.) The first-fit method is another strategy that may be used to dynamically allocate buffer space. The first-fit method is described in Section 3.4.1.

. Frequency of Memory Consolidation . It was found in the buddy system that the mean number of times that a released buffer could be combined with its buddy was .012. This indicates that a decrease in operating system overhead may be obtained by merely returning a buffer of 2^k words to the available storage list without attempting to recombine it with its buddy of 2^k words, or by limiting the attempts to recombine buddies to one per release. (See Section 3.6.)

. Improvement of First-Fit Method . Given the distribution of request sizes found in the Univac 1108 system, the basic first-fit model may be improved by maintaining available lists by size. The problem of fragmentation of the buffer space is minimal, and is essentially the same as that found as a result of allocation using the buddy system. (See Section 3.7.1.)

. <u>Conditions Under Which First-Fit Might be Preferrable</u> . The first-fit method compares favorably with the buddy system when requests for arbitrary space sizes are considered and when relatively large buffers are required. When the average request for buffer space is greater than 2^5 words, the overhead in the first-fit method becomes minimal, but the fragmentation of core increases; however, the loss of space introduced by the buddy system under these conditions may well be unacceptable. (See Section 3.7.3.)

1.3. Software Monitoring Results

The objectives of the software monitoring were (1) to provide a means of obtaining statistics characterizing an actual operating environment, and (2) to provide a tool for analyzing and evaluating the performance of system elements as a function of their implementation. Descriptions and details of the software monitoring tools which were developed are presented in Section 5. The results of general interest are summarized below.

. <u>Buffer Request Distribution of Univac 1108 EXEC VIII System</u> . The request distribution for buffers, obtained from the construction of memory maps taken at specified intervals of time show that 95% of the buffers requested lie between 2^2 and 2^5 words. The average request size is between 2^4 and 2^5 words. (See Section 3.5.)

. <u>Fragmentation Resulting from Use of Buddy Method</u> . Fragmentation of memory is not significant in the Univac 1108 executive system, which uses the buddy method of dynamic allocation. That is, the available memory is not fragmented to the extent that requests for space cannot be honored when the total available space exceeds the size of the requested buffer. (See Section 3.6.)

. Software Monitoring for Detecting Inefficient Code . The benefits of the best design approach may be negated by a poor implementation. After implementation, software monitoring may be used to detect potential areas of inefficient code. (See Section 5.6.) The value of software monitoring has become clear in this study through the use of an analysis routine, TRACE. This routine provided the means for monitoring the use frequency of code in the 1108 allocation routine. In the process, the effect of inefficient code was assessed.

. Inefficiencies in EXPOOL Implementation in EXEC 8 . Software monitoring of the Univac 1108 allocation routine detected inefficient code which, if modified, would reduce the number of operations from 103 to 74, the time to allocate a buffer from .180 msec to .130 msec. If the frequency with which this function is initiated is at least once per 25 msec, then by increasing the efficiency of the code, a saving of 28% in the overhead introduced by this function could be realized. The net result is that, in general, .016 hours per 8-hour shift of computer time now being wasted in overhead could be made available for useful work. The potential improvement in system efficiency applies to all Univac 1108 installations operating under the control of the EXEC 8 supervisory system. (See Section 5.6.)

. Development of a Function Performance Evaluation Tool . An analysis routine called TRACE has been developed which permits one to monitor the use frequency of code in a program. Specifically, the program to be monitored is not restricted by size, function, or special features required by the TRACE routine. (See Section 5.1.)

1.4. Analytical Modeling

Two mathematical models of a portion of a multi-computer problem were formulated and solved. The basic elements considered are a primary store, a secondary store, and a retirement policy whereby entries in the primary store are retired to the secondary store. In the first model the primary store was assumed to be infinite. In the second model this restriction was eliminated, a bounded store was assumed, and the model was solved. A paper presenting the results of the first model entitled, "Storage Requirements for Information Handling Centers" by H. M. Gurk and J. Minker will appear in the Journal of the ACM in January, 1970. A final report on the second model is contained in a University of Maryland Technical Report 69-90 entitled, "A Stochastic Model of an Information Center" by J. Minker and was delivered under the grant to NASA in July, 1969.

1.5. Bibliographic Research Results

A bibliography on the literature pertinent to the monitoring and analysis of computer operating systems has been accumulated. A KWIC (Key Word In Context) index has been developed and a technical report is being issued on this subject. The bibliography contains approximately 300 entries at this time. As new documents become available, updated versions of this bibliography may be obtained by submitting changes in subsequent computer runs. A preliminary KWIC index was included with the third quarterly report under this grant. The bibliography now being submitted updates the preliminary issue. The preliminary issue is available from the Clearinghouse for Federal Scientific and Technical Information, Report No. N69-30816.

2. 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.

2.1. 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.

2.2. 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.

2.2.1 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.

2.2.2 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.

2.2.3 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 our 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.

2.2.4 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 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 techniques 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.

2.3. 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.

2.4. 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.[25] 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.

2.5. 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.

3. Analysis of Dynamic Allocation Strategies

The basic objective of this phase of the study is to obtain a better understanding of the analysis techniques of simulation and internal software monitoring that might be applied to operating systems. Starting with the most elementary functions, analyses could be pursued to the more complex aspects of system design. As a case study, a basic system function, dynamic allocation of buffer storage, which allows for alternative strategies and implementation was selected. It was hoped that using the analysis techniques, it would be possible to determine under what operating conditions one strategy could be considered superior to another. Further, given an actual system with one of the alternative schemes implemented, decide, using characteristics of the environment in which the system is operating, whether another scheme would be more efficient and if so in what measurable respect. There are definite benefits derived by analyses even if it is found that an algorithm currently implemented in an operating system is best in terms of the environment in which it is functioning. This would indicate that this system function should remain unchanged unless it were found that the operating environment had changed significantly. Further, the analysis would provide a basis for determining which schemes should be considered seriously to provide a more effective system as a function of the nature of an environmental change.

This whole discussion and consideration of system or function evaluation leads back to the underlying objective which is to be able to attach values or apply measures to aspects of system design. Ultimately, the objective is to be able to make decisions concerning system design and modifications where the decisions are based on something more concrete and extensive than intuition and past experience. The latter may be invaluable in the creative stages of system design where ideas and alternative methods must be available for evaluation and consideration. However, the decision to implement a particular strategy should be a function of the system environment, the actual operating characteristics, and the interaction of system parameters.

3.1. Scope of Initial Study

The analysis undertaken in this study 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.

3.1.1 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 assocated with the search and maintenance of available buffer storage lists. Conbributing 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: fragmentation of memory and 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 memory loss 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.

3.1.2 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 fragmentation introduced in the allocation process. It is in the interest of maximum memory usage to implement an allocation scheme which keeps 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 memory loss caused by fragmentation at a minimum.

3.2. 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 2. 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 block $2^k, \ldots, 2^{j-1}$ 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 in this study.

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.

3.3. 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 for this study 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 generated per unit time has 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 of fragmentation of memory characteristic of each allocation scheme used, it was necessary to maintain the 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.

3.4. Basic Buffer Allocation Algorithms

In an executive system designed for multi-programming, two types of dynamic allocation are required. The first is for the allocation of user programs. In this case, the portion of available memory not required by the resident supervisor is available for user task programs. The size of user programs is variable and the need for large blocks of contiguous memory is quite common. As pointed out earlier, the problem of partitioning and allocating available memory among several users is not strictly a question of available space. Other complicating factors such as priorities and job scheduling strategies are involved.

The second type of allocation which is the subject of this initial study is internal to the executive routine itself. 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, requires 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 either type of allocation, a method must be adopted for allocating and releasing variable size blocks of memory, maintaining a list of available or unused blocks, and in the case of buffer allocation, 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, 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.

3.4.1 Simulation Models Developed

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 2 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.

3.4.2 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 search. 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, 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 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.

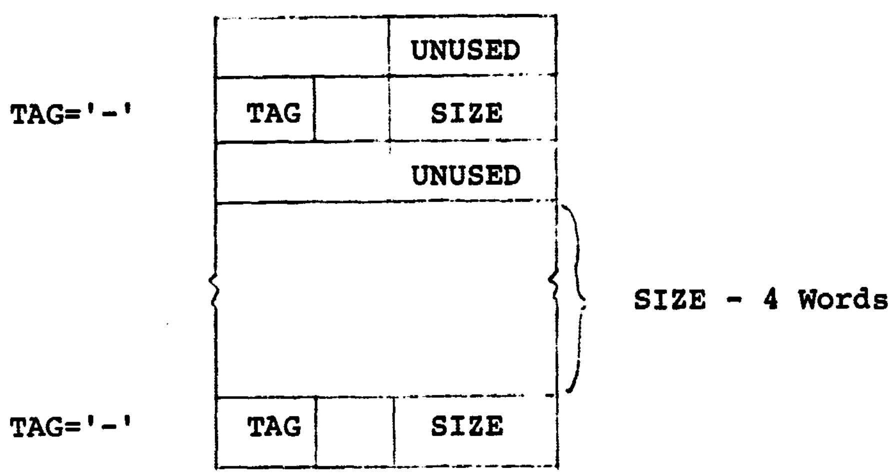
A3: [Test end of search.] If P=Loc(U) and F≠0, no allocation is possible. Otherwise, if P=Loc(U), set F=1, P=U.

A4: [Search list.] If SIZE(P)=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=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 have 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 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

Reserved Buffer Format



Free Buffer Format

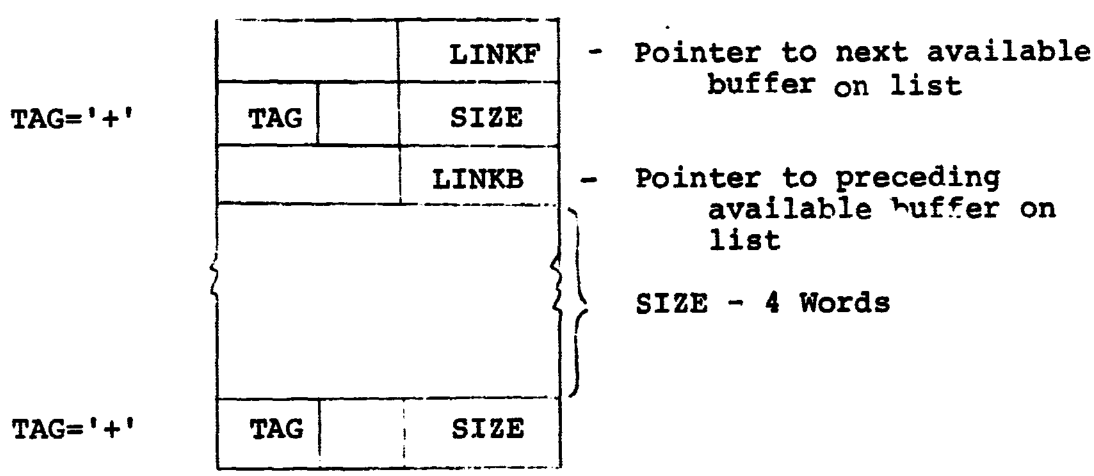


Diagram 3-1. Buffer Formats Used in the First-Fit Simulation Model.

the blocks are in the form maintained by the allocation process.

3.4.3 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 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), P=P+SIZE(P). If TAG(L-1)>0, go to R5, otherwise, set LINK(P1)=P2 and LINKB(P2)=P1.

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.

3.4.4 Buddy System Allocation

The second method of dynamic allocation is commonly referred to as the 'buddy system'. This method is implemented in the Univac 1108 executive system, EXEC 8. Again, the simulation model is based on the allocation and release algorithms presented in Knuth.[2]

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 $2 \le k \le m$, 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 available 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

---

* Note: If no block is available, no allocation is possible for block sizes 2^k and larger.

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.

Buffer Allocation (Buddy Method)

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.

Buffer Release (Buddy Method)

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.

3.5. 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 3-1. 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. A parallel effort to the simulation in this study is the modification of the EXEC 8 allocation routine which will permit monitoring the allocation process. Included in the data to be obtained are the time of a request or release, the size of buffer,

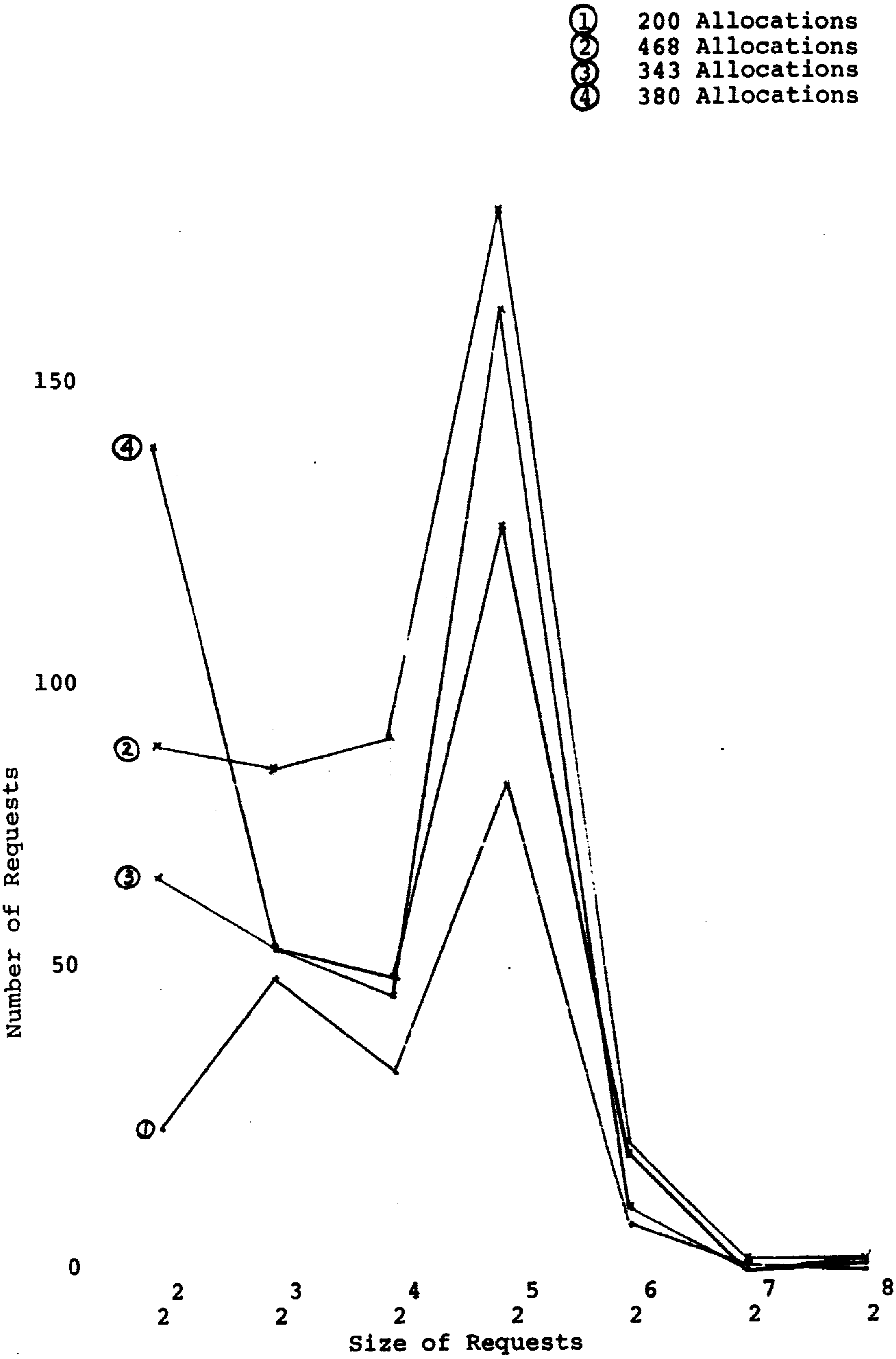


Figure 3-1. Distribution of Buffer Requests by Size.

the location, and when possible some indication of the buffer use. Analysis of these data will provide a rate and also another check on the distribution of requests and releases in the actual system. Actual rates will be used in the simulation process when they become available. At the present time, buffer requests and releases are being generated assuming a Poisson arrival distribution and an exponential hold time. Under these assumptions, Figure 3-2 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.

3.6. Outputs from the Simulation

Throughout this study performance is being measured in terms of memory loss and execution time required for the allocation process. Ultimately, the decision to implement a particular strategy in a particular system is one which is made by the system designer or analyst. Usually the decision is dependent on the premium set on time or space. In order to determine allocation times and space requirements, data must be analyzed either from an actual operating system or from a simulation model. In this phase of the study, data were obtained through the simulation 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,

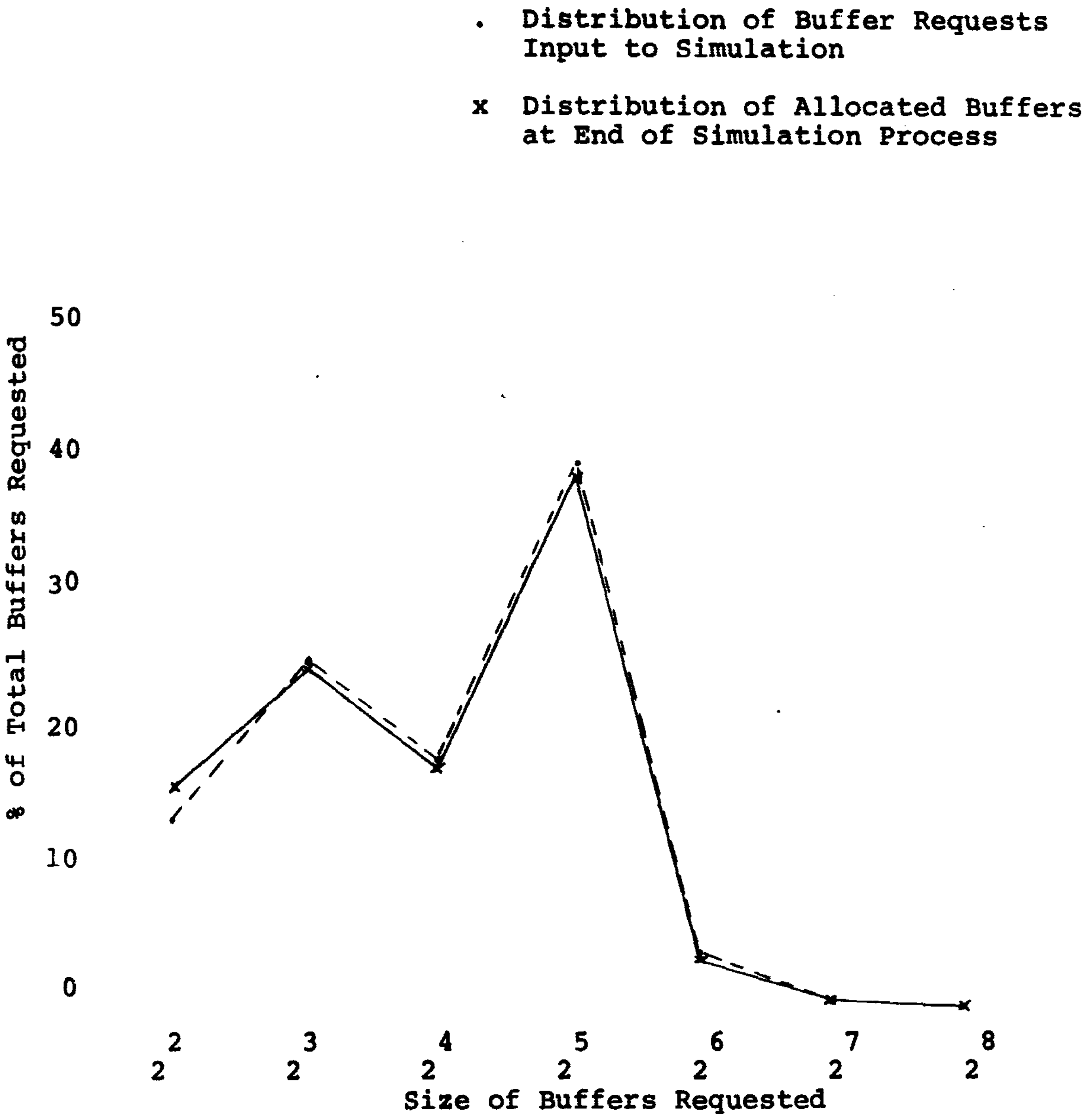


Figure 3-2. Comparison of Buffer Request Distributions Input To and Output From the Simulation Model.

the number of searches required for releasing a buffer, and the number of splits required to obtain a buffer of requested size. In order to estimate memory loss, data were obtained on the memory loss per allocation. This type of memory loss represents memory used for control and is tabulated in Table 3-1. Also contributing to memory loss is memory fragmentation, a relatively long term effect which is best seen through the use of memory maps. See Figures 3-3 to 3-6. 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 3-1 gives a comparison of the operating characteristics of the two schemes.

In view of the above results, 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 3-3 and 3-4 are indicative of the memory fragmentation introduced by the buddy method and the first-fit method respectively. In the first-fit process the problem 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

	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 3-1. Comparison of Buddy and First-Fit Allocation Characteristics.

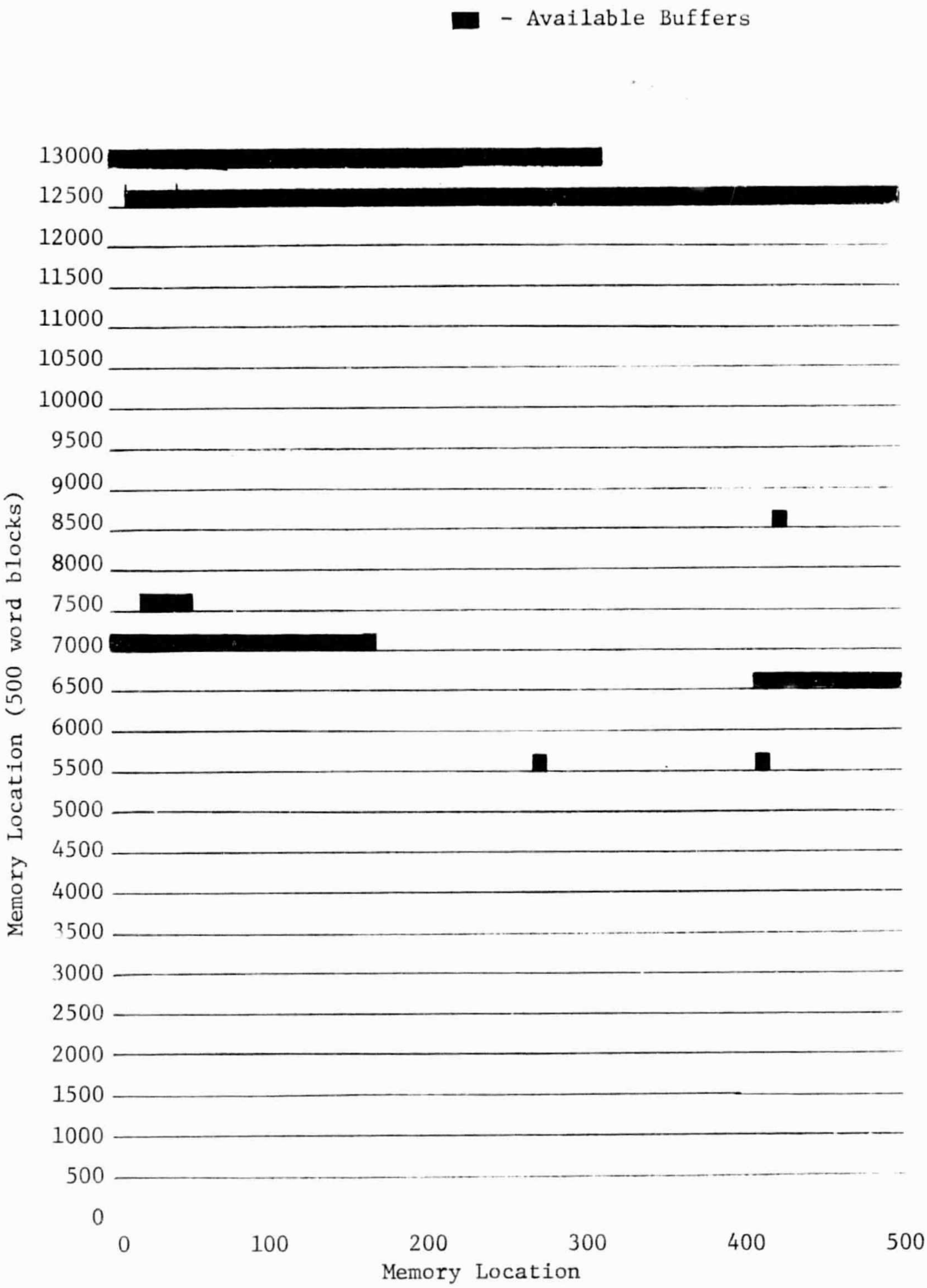


Figure 3-3. Buffer Pool Memory Map Resulting from Simulation of Buddy Allocation Scheme. (Map constructed after 926 allocations and 400 releases.)

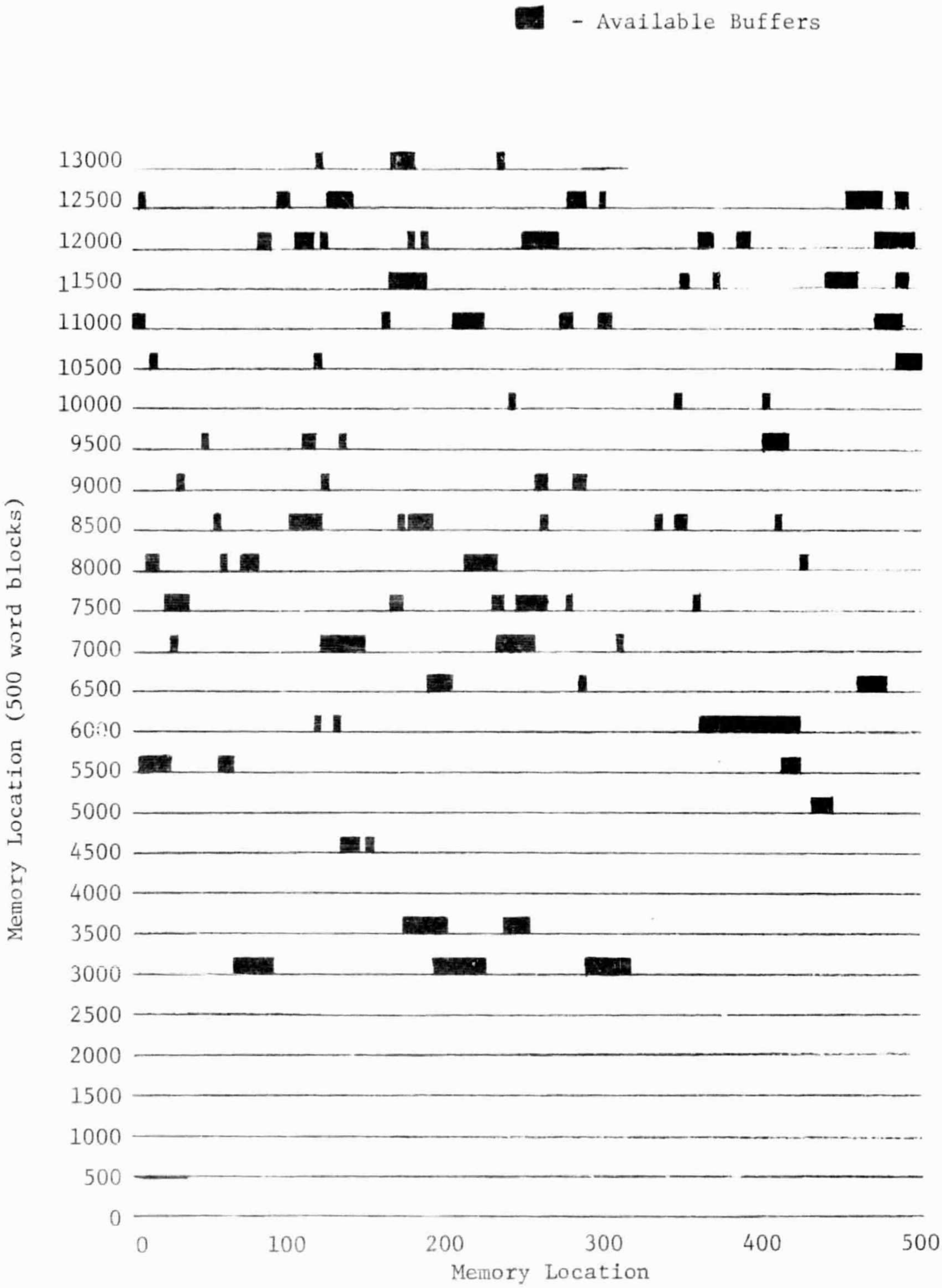


Figure 3-4. Buffer Pool Memory Map Resulting from Simulation of First-Fit Allocation Scheme. (Map constructed after 934 allocations and 400 releases.)

pool is extended.

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?'.

3.7. 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.

3.7.1 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 3-5, it can be seen from the resulting memory map, that the problem of fragmentation has been reduced to the point that it is comparable to the buddy method. The

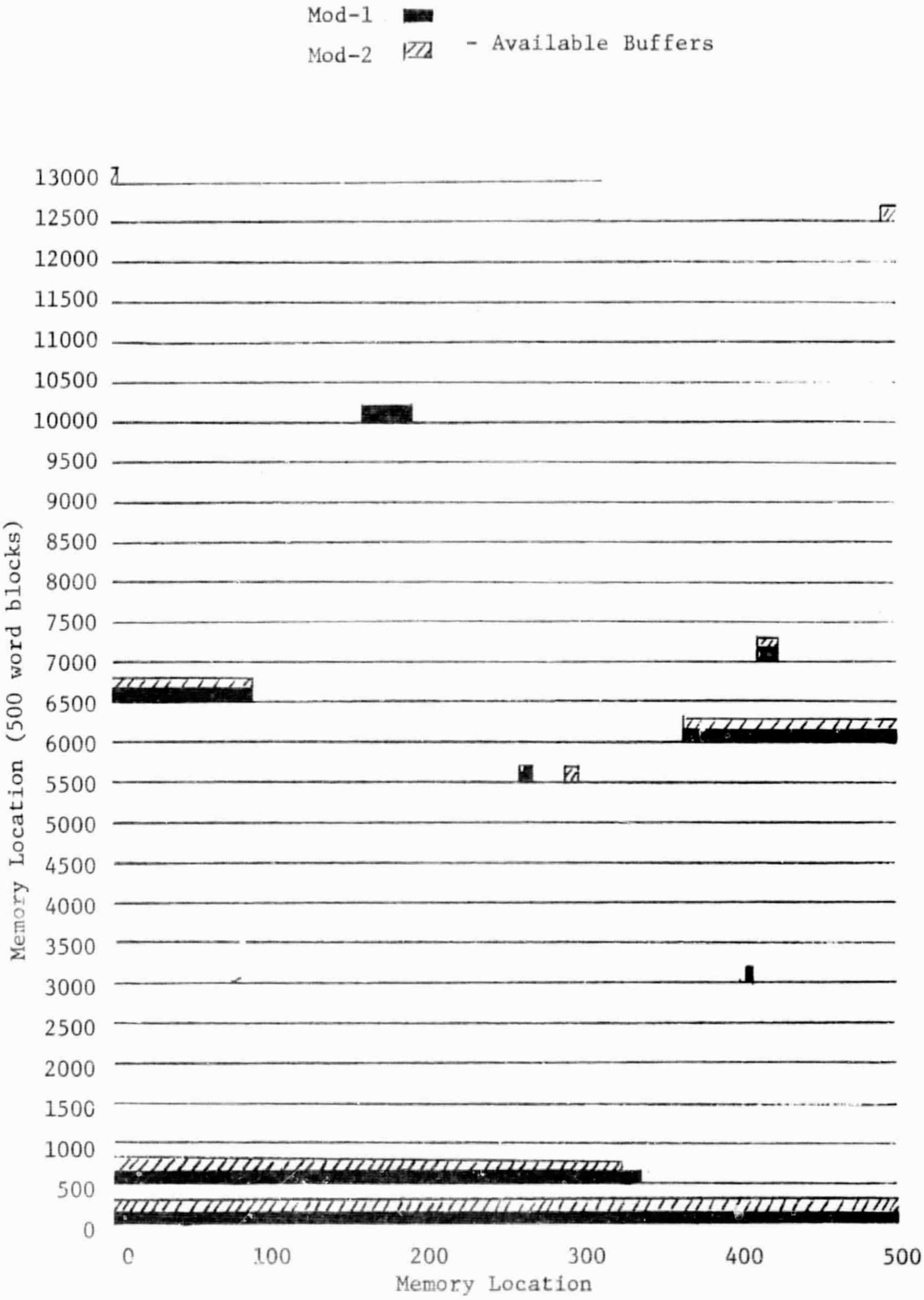


Figure 3-5. Buffer Pool Memory Maps Resulting from Simulation of First- Fit Allocation Schemes - Mod-1 and Mod-2. (Map constructed after 926 allocations and 400 releases.)

results in Table 3-2 indicate that in the first-fit method, the memory overhead per allocated block is still 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.

3.7.2 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 fragmentation would be introduced due to the fact that adjacent blocks might be available and unusable because they were not coelesced into one block. From the memory map given in Figure 3-5, it can be seen that no appreciable increase in fragmentation resulted. The results in Table 3-2 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.

	FIRST-FIT MOD-1	FIRST-FIT MOD-2	FIRST-FIT MOD-3	BUDDY MOD-1
Mean Memory Loss Per Allocation	2.0	1.0	1.434	16.281
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 3-2. Comparison of Simulated Allocation Characteristics.

3.7.3 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 memory waste which has not been apparent 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 memory loss per allocation was small. On the other hand, the fragmentation problem is again significant as can be seen in Figure 3-6. Also in Table 3-2, 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 memory loss per allocation was found to be between 16 and 17 words per allocation. It is clear from the size of this number that the memory loss is quite 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 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 memory wasted. If this procedure is followed, it is always possible to keep the memory waste small.

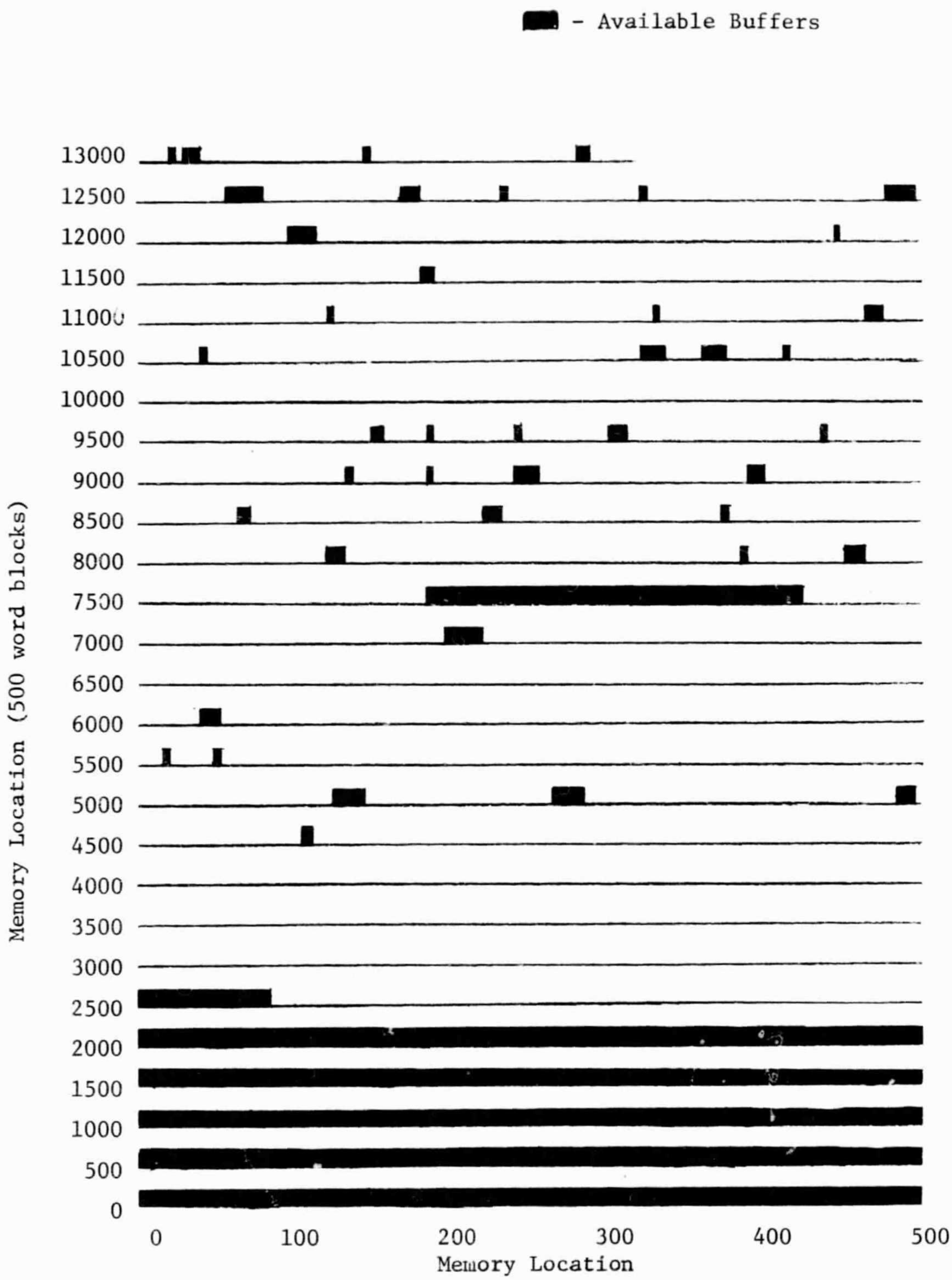


Figure 3-6. Buffer Pool Memory Map Resulting from Simulation of First-Fit Allocation Scheme - Mod-3. (Map constructed after 926 allocations and 400 releases.)

It should be noted, however, that this is a very clear case of a space-time tradeoff, since in order to reduce memory loss, it may be necessary to break one request into two or more requests. As a result the number of buffers allocated and released is increased and the total allocation time is incremented accordingly.

3.8. Consideration of an Adaptive Approach

In the foregoing discussion, if the buddy system were implemented, the requestor, i.e. the user of the system, was left implicitly with the responsibility of making efficient use of the buffer pool or of keeping the execution time at a minimum. It is possible that more direct control of system efficiency should be maintained from within the system itself. In other words, it might be advisable to have the user make requests for the exact size of buffer needed in every call to the allocation routine. The extent of memory loss which is a function of the mean request size would then determine whether a particular strategy should be used in the allocation process. As noted earlier, it could be that, if significant variation in request sizes is noted during different modes of system operation, different allocation schemes should be available and interchangeable by the system as warranted by the request distribution.

3.8.1 Provision for a Self-Adaptive System

In order to recognize the need for system modification, software monitors should be available which can be used to gather statistics which indicate what environmental changes occur. For example, the data collected may indicate that queues are forming at some point in the system, excessive time is being spent in performing certain functions, a distribution of requests for a certain function has changed, or memory available for program or buffer storage is frequently exceeded. If the information gathered on the system indicates that the changes which have occurred over a period of time are becoming stabilized, but different from the original operating characteristics, and if the operating efficiency is being impaired as a result, a system revision is warranted. A study should then be made to find alternative strategies which permit more efficient operation in view of the changes.

The above discussion describes the situation where the changes are uni-directional over a relatively long period of time. If this is the case, at some point strategies for handling certain functions may be replaced by more efficient ones. There is the possibility, however, that over relatively short periods of time, significant changes may occur in the operating system characteristics. This could very well happen in a system which is batch oriented but is capable of operating in an on-line environment. In this situation it would not be feasible to terminate operations and load an alternative system which is designed to handle either an on-line or batch workload efficiently. Alternative approaches to this problem are to design a system which favors one or the other of these environments and accept a reduction in system efficiency when operating in the other mode, or design the system so that it is not really efficient for either one, but is not seriously impaired in either environment.

It is fairly clear that operating characteristics, and as a consequence, operating strategies, may vary significantly for batch processing and on-line operation. It may not be quite so clear when the functions to be performed are common to most systems and the basic algorithms are already implemented in the system to handle them. At system design time, if consideration for system performance were a factor, the algorithms would be developed and implemented to permit optimum performance in terms of the expected operating environment. For example, a buffer allocation scheme might be implemented which performed best if the frequency of requests for small buffers were large. The algorithm would also handle large buffer requests so that even if the distribution of request sizes changed, the algorithm would still handle the allocation but perhaps not so efficiently. In order to evaluate the performance of basic algorithms within any operating system, it is first necessary to gather statistics which define the actual system operating characteristics. Then some experimental work must be performed, either through simulation or through actual system modification, to determine what improvement in handling the function, if any, can be produced by alternative strategies.

If it is found that alternative algorithms produce more efficient operation for particular distributions, and that the distributions vary from one type to another fairly consistently, then one might consider providing for the implementation of alternative algorithms for handling the function. If a distribution drifts over time or with modes of operation, then it might be advantageous to implement a system monitor which determines the distribution during operation. Then, when a threshold established as a result of measurement and analysis is crossed, a signal for phasing out one algorithm and initiating an alternative strategy would be produced. The mechanics for phasing one algorithm out and the other in must guarantee that the changeover be automatic and that the system operation be uninterrupted. The notion of implementing a system with self-monitoring, self-analysis, and self-adaptive features is very attractive, however more experience must be gained in the analysis of data obtained from system monitoring and more experimentation and analysis must be performed to define the relation between algorithm performance and the conditions under which a particular algorithm is most efficient.

3.8.2 Proposed Extension to the Current Study

As a first step in exploring the feasibility of such a self-adaptive system, the algorithms for the allocation of buffer storage which were analyzed individually are being considered for this proposed study. The objective is to implement two algorithms and then define and simulate the mechanics required for passing from one allocation strategy to the other.

In order to implement the adaptive system, it was decided that in all cases, requests should be made for the exact size of buffer actually needed. This permits the system to determine, based on the distribution of requests and memory loss, which algorithm should be used to allocate buffers. In the system a record must be kept of the frequency of requests by size so that periodically the memory loss can be estimated. When the percent of memory loss exceeds a preset cutoff, the alternative allocation strategy will be initiated as requests continue to be made. Taken alone, the implementation of the alternative modes of buffer allocation are relatively straightforward. Further, if planned for at the time of system design, the implementation of software monitoring devices for gathering statistics is not a major undertaking and the periodic computation of the significant parameters should not be time consuming.

The more difficult aspects of the processing techniques being proposed here, lie in the design of compatible modes of allocation and insuring a smooth transition from one to the other. The result of attempting to make the allocation modes compatible is that probably neither strategy is implemented in its basic form. It would be remarkable if the restrictions and modifications made to the buddy system and the first-fit allocation method to make them compatible actually result in either taken alone, being more efficient. Hopefully the disadvantages will be more than offset by the advantages realized. Only a careful analysis of a given set of conditions can determine the net result.

4. Review of Some Existing Hardware and Software Monitor Techniques

In previous sections, we have described the role of analytical techniques and simulation to aid in the process of computer system evaluation. In this section, we shall consider both hardware and software monitoring of a computer system. A summary of some of the significant attributes of the monitoring techniques discussed in this section are given in Table 4.1.

4.1. Hardware Measurement Techniques

Within the normal standard hardware features of a digital computer, such functions as address stop switches, trap transfer modes, and normal error-faulting procedures are important for measurement purposes. In addition, some special hardware devices have also been developed and added to systems so as to perform hardware monitoring of a computer's performance. Devices can be attached to a central processor so as to passively examine each instruction as it is executed. Hardware monitor devices have built-in counters and self-contained output devices to record the occurrence of any given data pattern. It will be useful to review several approaches to perform hardware measurement of a computer.

4.1.1 IBM 7090 Hardware Measurement Technique[33]

This device is designed to record information from the CPU while the CPU is processing data. The recorded data is then used to analyze the basic nature of the program and to measure the performance of the hardware. The hardware measurement device consists of a control unit, a control panel, and an

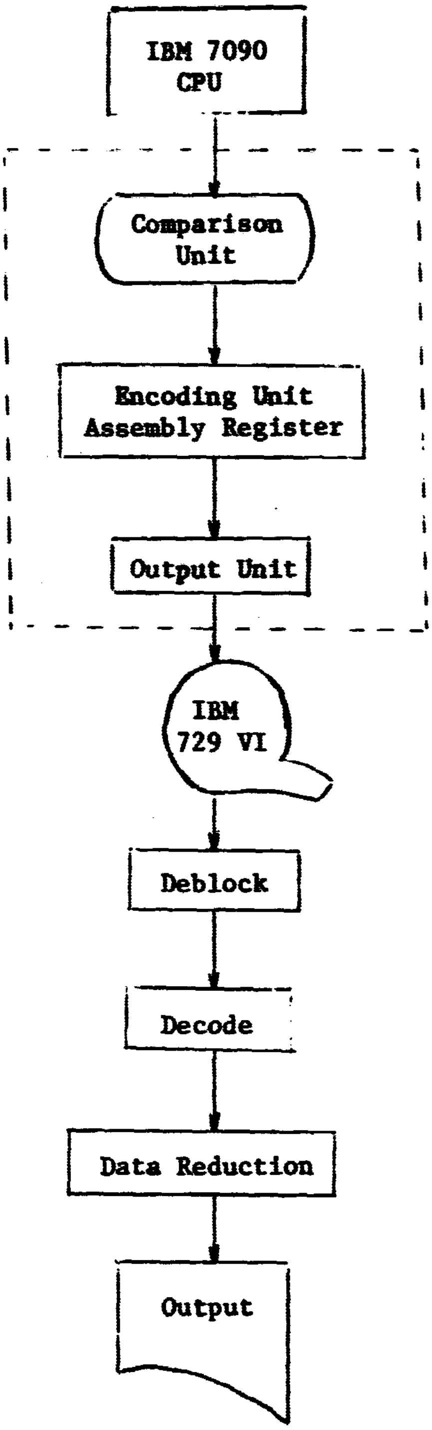
Attributes \ Techniques	Hardware Measurement Technique	Performance Data Recording	Instruction Trace
Degradation Effect on Measured System	None	Low	Very High
Level of Detail Recorded	Low	Medium	Very High
Special Hardware Required	Yes	No	No
Cost	High	Medium	Low
Flexibility	Very Low	Medium	High
Purpose	Overall System Analysis	Overall System Analysis	Implementation Analysis

Table 4-1. Comparison of Measurement Techniques.

IBM 729 VI tape drive. There are three internal sections of the control unit: (1) An input unit, which contains 40 lines from the monitored CPU, six 24-bit data buffers, and one comparison unit. Of the 40 lines, there are 24 data lines which are used to transfer 20 bits of the contents of the instruction counter, and 4 bits specifying the channel in-use to one of the data buffers; 15 selector lines which transfer the 15-bit op-code to the comparison unit; and 1 stroke line which contains the status of the input lines. The comparison unit compares the 15 selector input lines with each of five sets of switches manually set by the operator from the control panel. Data are recorded if there is a match between the 15 selector lines and one of the five sets of switches. (2) An encoding unit and assembly register, which encodes the 24-bits of data to a variable length string, packs the string into 6-bit groups, and transfers the string to the output buffer one group at a time. (3) An output unit, which contains eight 6-bit output buffers and one tape controller. A block diagram of the operation of the device is shown in Figure 4-1.

4.1.2 IBM System/360 Hardware Measurement Technique (TS/SPAR)[24]

TS/SPAR (Time-Sharing System Performance Activity Recorder) is a hardware-measuring device used to collect performance data for measuring the dynamic operations of an information handling system. It can be used to measure the external effects of internal software and hardware operations, and to measure the internal operational characteristics of software or hardware units. It can also be used to count the frequency of an event, to clock its duration, and to record the gross time. A block diagram of TS/SPAR is shown in Figure 4-2. Electronic counters



Recording Machine

. Compare the 15 Selector Lines with the 5 Sets of Switches.

. Encodes the 24-bits of Data into a Variable-Length String and then Packs the String into 6-bit Words.

. Writes Inter-record Gaps between each Logical Record .

. Decodes the Variable-Length String to 24-bits of data and Adds Time Information.

. Produces a Trace-like Printout or Generates CRT Graphical Display.

Figure 4-1. Functional Diagram of the IBM 7090 Hardware Monitor Device.

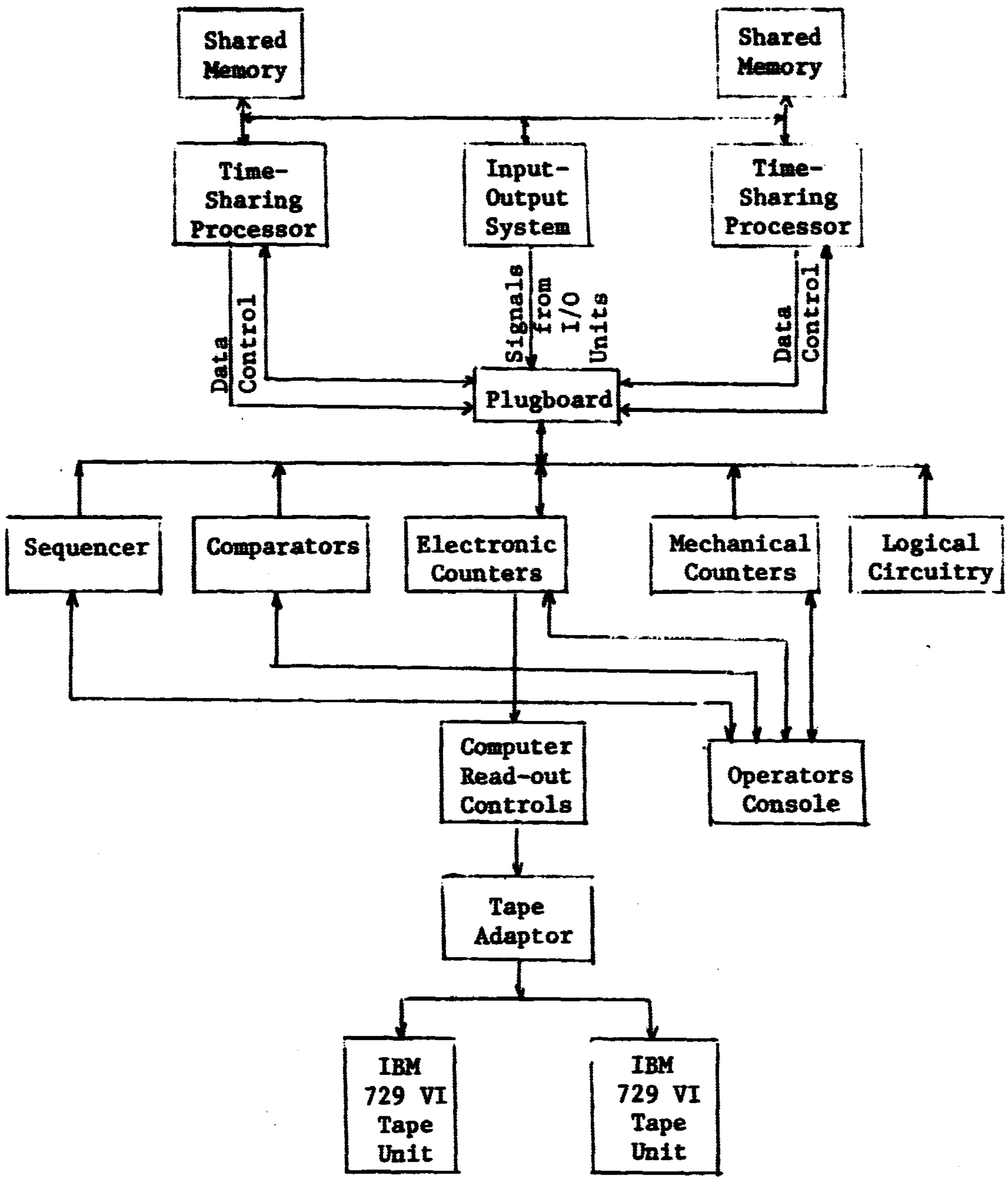


Figure 4-2. Functional Diagram of TS/SPAR

within the device provide accumulative storage for up to 48 measurable parameters of 3 decimal digits length. Mechanical counters are activated when overflow occurs from the electronic counters. Comparators are used to dynamically monitor data paths in the interface and to compare them with fixed values indicated by switch settings. These switches are used to indicate to the monitor a unique address, an operation code, or some contiguous memory locations. The sequencer can be used to detect any three-event sequence. An event may be a reference to a real or virtual memory address, an instruction counter, an op-code, a control signal, etc. The time interval between the occurrence of events is not considered, only the event sequence is of interest. The plugboard receives the interface signals and transfers the data and control to the various functional areas in the recorder. The logical circuitry is accessible from the plugboard to logically combine interface signals so as to form complex events or to generate control signals.

Input to TS/SPAR is through a specially engineered interface which can handle 256 predetermined signals and strokes. These interface signals reflect certain key states (internal or external) of the system to the recorder.

4.1.3 UNIVAC 1108 Hardware Measurement Technique[27]

A Univac 1108 is used to measure the performance of another 1108 system. The hardware measurement system uses a special hardware device interface as a recording processor to gather live data. (See Figure 4-3) It contains a hardware monitor, data collection software, and data reduction software. The monitor creates and records data each time a jump instruction is executed in the monitored processor. The collected data is

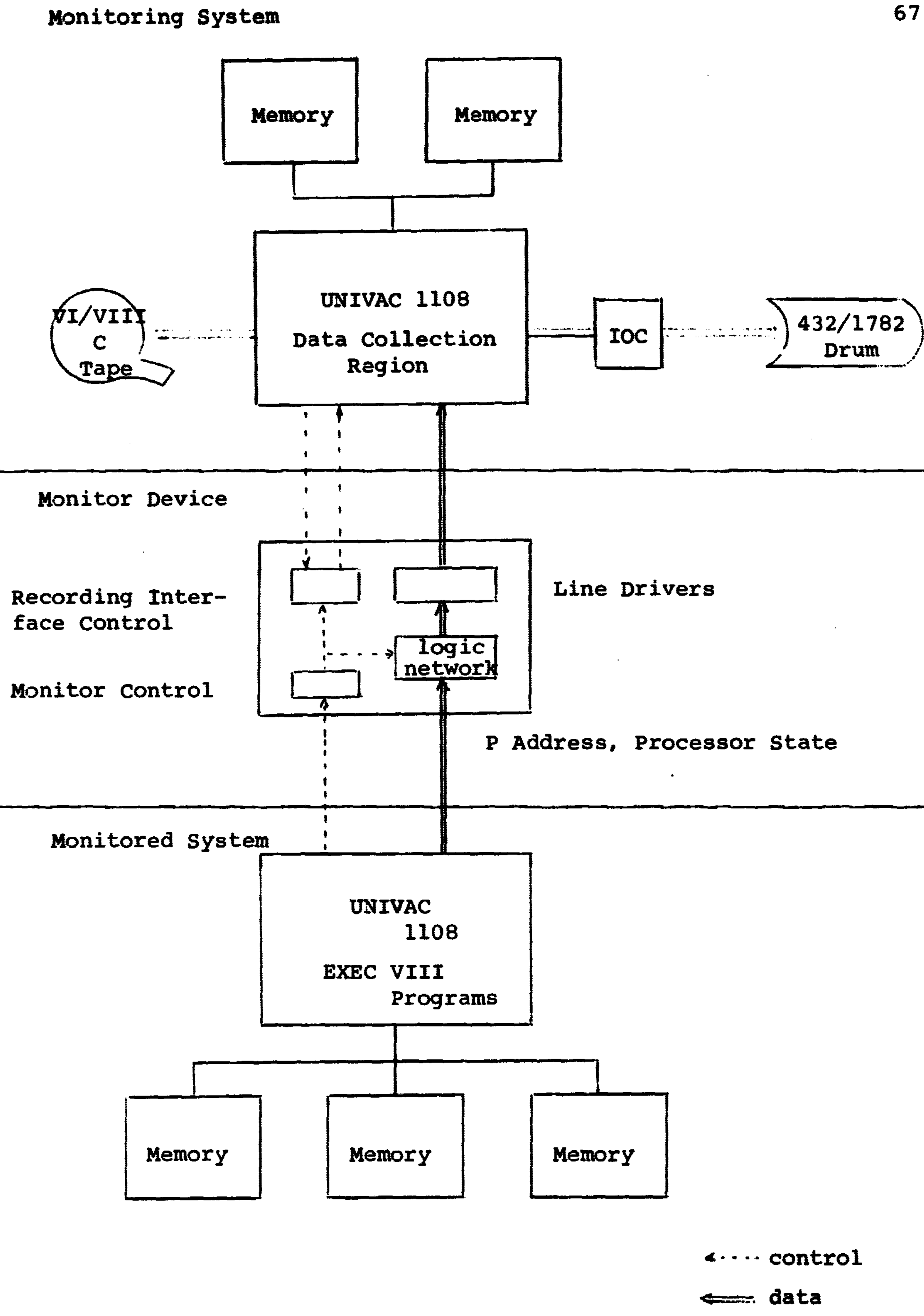


Figure 4-3. Block Diagram of the UNIVAC 1108 Hardware Monitor Device

transferred to a drum via two large core storage buffer areas. When the drum is filled, the data are transferred to tape. A special data reduction software package reduces the data into either graphic or statistical form to provide a perspective of the performance analysis of the monitored equipment.

4.2. Software Measurement Techniques

Software measurement techniques can generally be divided into three classes:

(1) Tracing and Sampling of System Operations.
To analyze the performance of individual programs, tracing, or high density sampling methods may be used to obtain the distribution of the CPU and I/O time for the program.

(2) Software Recording.
A software recording mechanism that operates within the operating system to collect important events and decisions made within the system. Such a mechanism can reveal the exact sequences and paths of events that occurred during execution.

(3) Analysis of Recorded Data.
The recorded internal performance data and/or the standard system accounting data may be used to provide a long period performance analysis. The output of the analysis could lead to information that could help to maintain a system at top efficiency.

There have been several developments in the field of applying software techniques to monitor systems. Four of these developments are described below.

4.2.1 GE GECOS II, GECOS III Software Measurement Technique[34,35]

The overall performance of a computer system depends on the efficiency of both the hardware/software environment and the programs which operate in that environment. The software monitoring device used in GECOS II is designed to permit analysis of the system performance and also of individual programs. The system analysis includes user program accounting analysis, overhead analysis, and trace analysis. To provide for individual program analysis, i.e. functional value analysis, high density sampling is used. By frequently interrupting the system at random or periodic times, the fraction of the total time spent in a particular instruction sequence is found to be proportional to the number of samples taken while in that sequence. The results of the periodic sampling are used as the basis of I/O and program execution time profiles. Several software measurement techniques were applied during the development of GECOS III. Software measurement of processes internal to the system were developed. Event counters were included in all functions of the system so that they could be analyzed and studied separately. Internal system auditing was provided to check on new entries in each of the system queues, to checksum critical tables each time they are referenced, and to checksum all system files as they are loaded into core for execution. Event tracing is used to detect the occurrence of important events. Decisions made within the system are monitored and made available for subsequent analysis by recording, in a circular list, each intermodule transfer. The total data collected on function usage, queue formation, table and file manipulation, and event occurrences is sufficient to summarize system operation and performance. The total analysis uses as input, standard system accounting data, the recorded trace entries, and other parameters made available from the system.

4.2.2 CDC 6600 CHIPPEWA Software Measurement Technique[25]

The Lawrence Radiation Laboratory uses a PPU (Peripheral Processor Unit) as a programmable hardware monitor to record and to analyze the activity in the CDC 6600 central processor and other peripheral processors. Two monitoring routines, MR SEE and MR EYE, are used. MR EYE gathers information on CPU activity, central memory utilization, channel activity, PPU activity and control disposition. MR SEE furnishes data on the disk utilization and the job profiles.

4.2.3 IBM TSS/360 Software Measurement Technique (SIPE)[26]

SIPE is an on-line software recording technique used to collect the data necessary to measure and to evaluate the performance of the IBM System/360 Time Sharing System (TSS/360). SIPE is a selective, event-driven recording mechanism that operates within TSS/360. The activating mechanism of SIPE is called a 'hook'. (See Figure 4-4) Hooks have been implemented at various points throughout the resident supervisor code. Each hook includes an identifier code. Based on this code, SIPE collects the applicable data. The degradation of the operating system with the SIPE monitor is proportional to the number of times SIPE hooks are activated. It is also affected to some degree by the volume of the output data. To compromise between resolution and degradation, a selective option function (delta-data-set) has been implemented. The delta-data-set is input to SIPE as a parameter at the start of a run. The given delta-

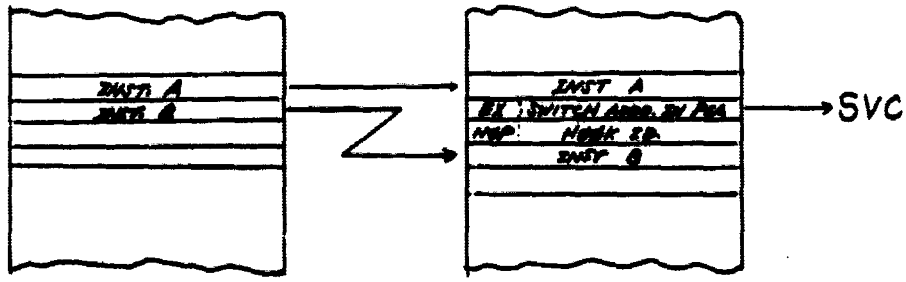


Figure 4-4. The "Hook" Structure of SIPE

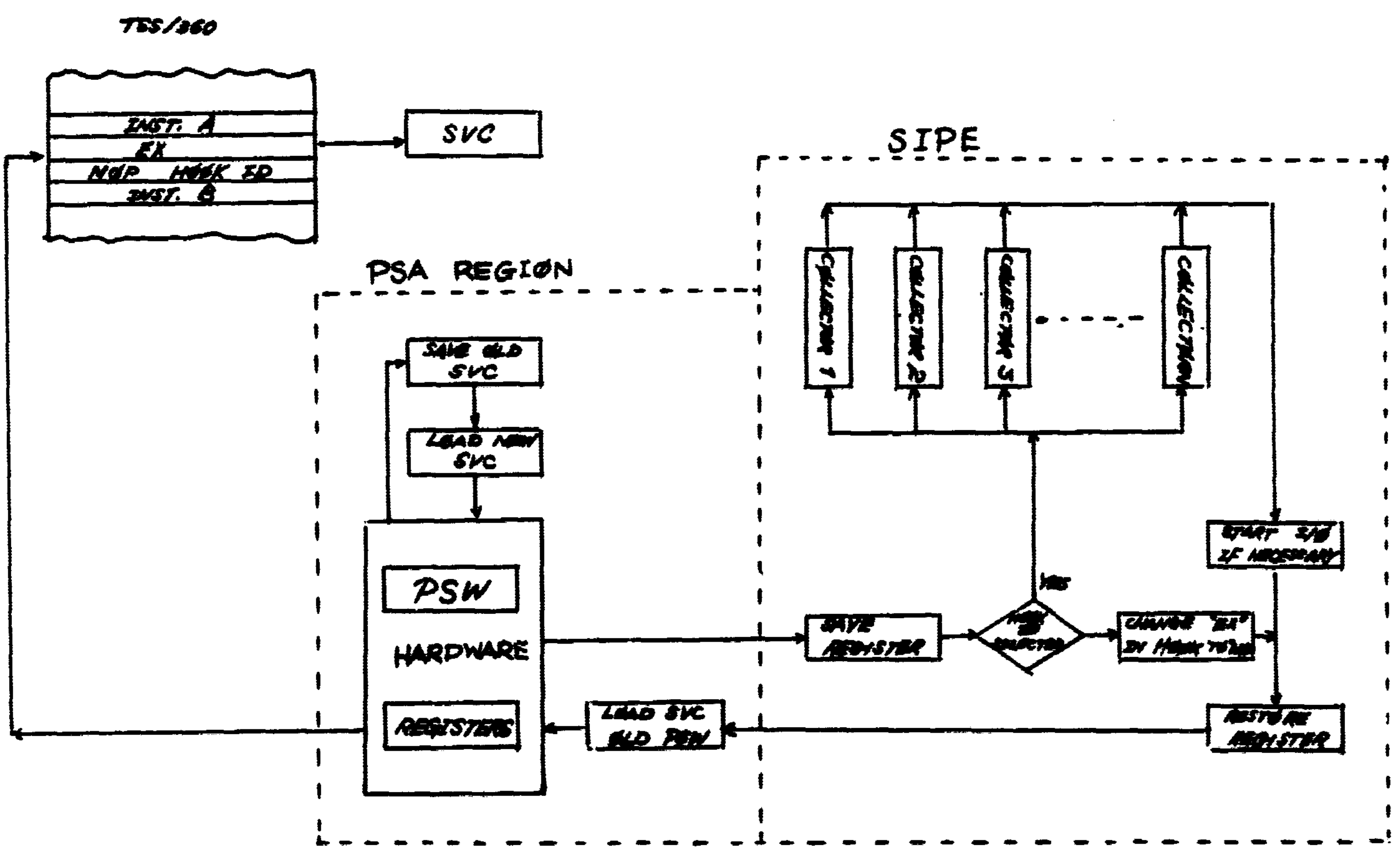


Figure 4-5. Functional Diagram of Interface between TSS/360 and SIPE

data-set instructs SIPE to 'turn-off' any hook or group of hooks for that run. In order to derive meaningful information from the data collected by SIPE, a library of data reduction programs has been developed. These programs convert the SIPE data to a simple or elaborate form for use in performance evaluation, system analysis and debugging as requested by the analyst. A functional diagram of the interface between TSS/360 and SIPE is shown in Figure 4-5.

4.2.4 IBM OS/360 Software Measurement Technique (SMS/360)[28,29]

SMS/360 is a system measurement software package developed by Boole and Babbage, Inc. Two components of the SMS/360 are described below. These are the PPE-2 and the CUE-1 components.

The PPE-2 (Problem Program Efficiency) component is concerned with the efficiency of the user's problem program. The output of the PPE provides the distribution of CPU and I/O time spent by the user's program. The PPE consists of two elements: the Extractor program and the Analyzer program. The Extractor program randomly samples the problem program during its execution and collects statistics for later analysis. Each time the Extractor records a sample, one of two events has taken place, either the instruction address falls within sample bounds, or a SVC (supervisor call) has been invoked from within the sample bounds. The Analyzer uses the collected data to generate reports which indicate where and how the program spends its time and how the program is balanced between being compute bound and being input/output bound. The reports generated include a number of tabular displays and one graphic display called the Histogram.

The CUE-1 (Configuration Utilization Efficiency) component is used to aid in maximizing system throughput by determining the configuration utilization and by showing specific hardware, software relationships which contribute to configuration utilization. CUE is also divided into two programs, the Extractor and the Analyzer. The Extractor collects data on hardware usage, disk head movement, data cells, and transient supervisor call routine usage. The Analyzer generates a configuration report, an equipment usage sub-report, a head movement sub-report, and a SVC sub-report. The quantitative information given in these reports can assist in locating bottlenecks in a configuration which might otherwise be overlooked.

5. System Function Analysis Using Software Monitor Techniques

The objective of the software monitoring efforts conducted under this grant was to develop techniques to permit the collection of data from the operating system as it was running. A quantitative study of an operating system using data on the behavior of that system is an effective approach to permit one to locate and to examine defects that may exist in the structure and utilization of the operating system. In the design of a system monitor technique, the following capabilities were desired: (1) To provide a technique that would permit one to study the logic and behavior of programs so as to define and locate significant events that occur within a program; (2) To provide a technique which would permit analysis and evaluation of the implementation of a program, so that local performance errors could be detected and possibly avoided; (3) To provide a technique to collect the applicable data of the total operating system in order that the interaction of system functions can be analyzed and evaluated; and (4) To provide a technique to continuously report the performance summary on a display or on an on-line printer at specified periods of time. To meet some of these objectives, several programs were designed and implemented on the 1108. These programs are described below.

5.1. TRACE

TRACE is a special simulation tool which has the ability to simulate itself. It is written and developed for the purposes of studying the logic and behavior of a program. It is sometimes very difficult to obtain documentation and descriptions of system routines. This has been found to be the case with the 1108 Executive routine. TRACE can provide useful information concerning the operation of a program, such as the location of the instruction, the data in the operands of the instruction itself, and the contents of all registers used by the instruction. The TRACE routine records data at every instruction, or at selected instructions, and then prints out a step-by-step account of the behavior of the program. From the printout developed by TRACE, the programming technique of the traced program can be observed and evaluated. The scheme is particularly useful since the 1108 has a complicated set of registers. Some of the registers are altered by certain operations while others are not. This is also true for memory words used by the program. When programmers perform coding, redundant operations such as those used to load a register or to store a memory cell are generally prevalent in the code. By applying the trace technique to a program, these wasteful instructions can be detected and, at times, avoided.

In the TRACE program we contrive to let the machine execute most of the instructions as the instruction appears in the program. The exception is that TRACE modifies jump or conditional jump instructions before execution so as to insure that control will return to the TRACE routine after the jump has taken place. Inside the TRACE routine a memory word is maintained to simulate the hardware instruction counter which points to the current instruction to be traced. TRACE copies the traced instruction into its own work area. Before execution of the instruction, a subfunction is called to analyze the op-code so as to identify whether this is an unconditional or conditional jump instruction. If the instruction is not a jump type instruction, the simulated instruction counter is increased by one and the traced instruction is executed. However, if the instruction is a jump type instruction, the address field of the jump instruction is saved first and then replaced by a specified address. If a jump occurs, i.e., the condition of the jump is satisfied, the control then goes to the specified location instead of to the successor instruction. In this fixed location, the simulated instruction counter is replaced by the saved address field. In this way the exact program instruction sequences can be traced. A general flow chart of the TRACE program is shown in Figure 5-1. An output from the TRACE program is also given in Figure 5-2.

5.2. ITFVA (Instruction Trace and Functional Value Analysis)

The purpose of a functional value analysis is to try to improve the efficiency of a program. In analyzing a program to achieve this improvement, the payoff between the time spent in analysis, debugging, and the total possible machine time gained should be considered. A technique is described that will indicate to the user the most frequently executed code within his program. Since it is executed frequently there is a higher payoff if this portion of the code is improved.

Either in a high level language or in a machine language program, a jump instruction represents the end of a sequence of operations. Those contiguous sequential operations can be considered as a single macro-instruction. In this way, a program can be divided into several macros, each terminated by a jump instruction. By 'Kirchhoff's Current Law', the number of times the control flows out of a macro-instruction must equal the number of times control is transferred to the macro-instruction. Hence, if we record the information when a transfer is made to a special instruction (location), then we can get

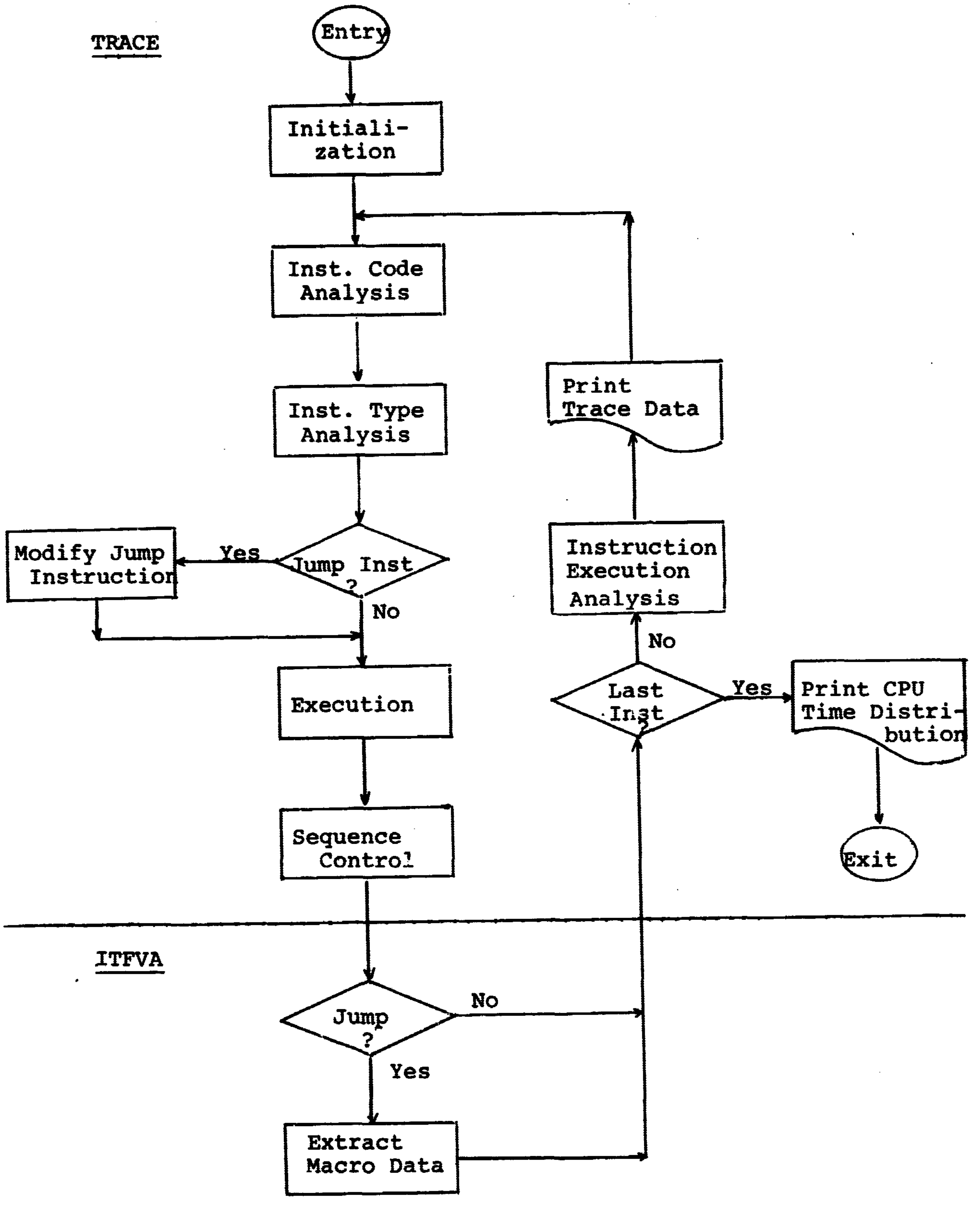


Figure 5-1. Functional Diagram of ITFVA and TRACE.

```
43115  53 02 04 14 0 043071  A 000000000017  000200000000  000000000012  777777777777  777777777777
43117  25 16 14 00 0 000001  X 000000000011  777777777777  777777777777  777777777777  777777777777
43120  72 02 05 00 0 043115    777777777777  777777777777  777777777777  777777777777  777777777777

43115  53 02 04 14 0 043071  A 000000000017  000400000000  000000000011  777777777777  777777777777
43117  25 16 14 00 0 000001  X 000000000010  777777777777  777777777777  777777777777  777777777777
43120  72 02 05 00 0 043115    777777777777  777777777777  777777777777  777777777777  777777777777

43115  53 02 04 14 0 043071  A 000000000017  001000000000  000000000010  777777777777  777777777777
43116  74 04 00 00 0 043122    777777777777  777777777777  777777777777  777777777777  777777777777

43122  27 01 14 14 0 043071  X 000000000004  777777777777  000000000004  777777777777  777777777777
43123  10 13 04 13 0 000000  A 000000000000  001000000000  011530057010  777777777777  777777777777
43124  10 16 05 13 0 000000  A 000000057010  000000011400  011530057010  777777777777  777777777777
43125  74 04 00 00 0 043130    777777777777  777777777777  777777777777  777777777777  777777777777

43130  74 13 13 00 0 043234    777777777777  777777777777  777777777777  777777777777  777777777777

43234  46 16 14 00 0 000000  X 000000000004  777777777777  777777777777  777777777777  777777777777
43235  50 13 00 00 0 043706  A 000000000004  000000011530  777777777777  000000000000  777777777777
43237  53 16 00 00 0 000011  A 000000000004  000000011530  777777777777  777777777777  777777777777
43241  27 01 15 00 0 043713  X 000000000000  777777777777  777777777777  000000000000  777777777777
43242  54 01 01 00 0 043712  A 000000000000  040075413506  777777777777  022000033000  777777777777
43243  74 04 00 00 0 043246    777777777777  777777777777  777777777777  777777777777  777777777777
 (1)          (2)            (3)    (4)           (5)           (6)           (7)           (8)
```

Figure 5-2. Sample output from the TRACE program.

1 The absolute address of the traced instruction.
2 The instruction code being traced.
3 An indicator of what type of control register is being used by the traced instruction, i.e. A,X, or R register.
4 The content of the register referenced or a code of 777777777777.
5 The contents of the next sequential register or a code of 777777777777.
6 The contents of the index register referenced or a code of 777777777777.
7 The contents of the operand of the traced instruction before execution or a code of 777777777777.
8 The contents of the operand of the traced instruction after execution or a code of 777777777777.

the exact number of times that the macro-instruction has been executed.

This functional value analysis program is formed by modifying the TRACE routine described above by adding a sorted, linked list to record the transfer information. (See Figure 5-1) After the recording is complete, another analysis routine is called to print the distribution of CPU time for each macro-instruction. An analysis of EXPOOL on the Univac 1108 that resulted from the use of ITFVA is presented in Section 5.5 as a case example.

Another technique most frequently used for functional value analysis is the high density sampling method which was described in Section 4.2. The advantages of using the TRACE routine are: (1) The TRACE routine is easily modified to permit recording information of every instruction traced or to record the trace data only when a jump occurs; (2) It provides a high level of information detail since the recorded data contains the exact number of instructions executed in each macro, and if desired, provides the exact sequence of each macro-instruction performed.

The disadvantage of using TRACE is that it will greatly slow down the execution of a system. Hence, TRACE is best suited for the analysis of short input-data independent programs. An analysis of the ITFVA routine indicates that the time required by using ITFVA within a system results in the need for an increase of 18 times the normal execution for a non-jump type of instruction, and an increase of 60 times for a jump instruction.

The above disadvantage can be avoided to a certain extent by using the TRACE technique in conjunction with event counters. That is, set a counter in every basic system function which is to be monitored. It is relatively simple and straight-forward to implement. According to the contents of these counters, the most frequently executed function can be detected. The procedure then is to analyze only frequently executed functions with the TRACE technique. This provides a very simple and useful tool to improve the implementation and efficiency of either a system routine or a user program.

5.3. OPSDE (EXEC 8) Operating System Performance Data Extractor

The purpose of evaluating an operating system is to determine and to substantiate the capabilities and the limitations of that system. The problem is to find out what is going on inside the system and where the CPU spends the majority of its time. To solve this problem requires that data be obtained 'inside' the system as it is running. OSPDE is developed so as to provide a software recording technique to extract internal system performance data. Such data provides the exact sequence and patterns of events that occurred during execution. It can be used as an input to a simulation model to provide a realistic calibration and feedback to the system designer. This provides a good, quantitative measure of the existing system which permits pinpointing 'performance bugs' - the results of errors in programmer evaluation and judgment on performance optimization. Under this grant, the program OSPDE has been designed, but has not yet been implemented. The structure of the data item and the data block of OSPDE is shown in Figure 5-3. The major objectives of the design were: (1) To minimize the system degradation by providing a selective option, which permits the user to be selective in the system events to be monitored at any

Data Item

JOB NO.	DATA ID	DATA LENGTH	TIME

Data Block

INITIATED TIME OF THIS BLOCK	
# OF ITEMS LOST IN PREVIOUS BLOCK	# OF WORDS IN PREVIOUS BLOCK
DATA BLOCK NAME	
DATA ITEMS	

Figure 5-3. A Data Item and Data Block of OSPDE.

given time; (2) To share a tape path with the system, use a variable data length structure and a unique data collection macro-instruction to get additional generality and flexibility; and (3) To use the mechanism of a double output buffer, i.e. while one buffer is transferring data to tape, the other buffer is being filled with data. The CPU is forced to wait when the second buffer is full and the first buffer has not yet transferred data to tape. With this arrangement the loss of data is possible.

5.4. Other Techniques Under Consideration

If the OSPDE recording rate is approximately one millisecond, there will be sixty thousand data items recorded every minute, and 3.6 million data items recorded every hour. It is obvious, from these huge volumes of data, that a process to reduce data must be done on a computer to give meaningful information to the user. Hence, a data-reduction and reporting routine is needed. This routine should have the capability to receive parameters from the user, to select any combination of events of the recorded data, and to output the analysis results in tables or graphs.

The standard system accounting routine provides data concerning the resources and the elapsed time used by a program. The accounting data can be used to measure gross performance, and can be combined with OSPDE recorded data to summarize the overall system performance during long periods of computation time. As described above, such a technique is required to provide continuous measurement analysis to the user.

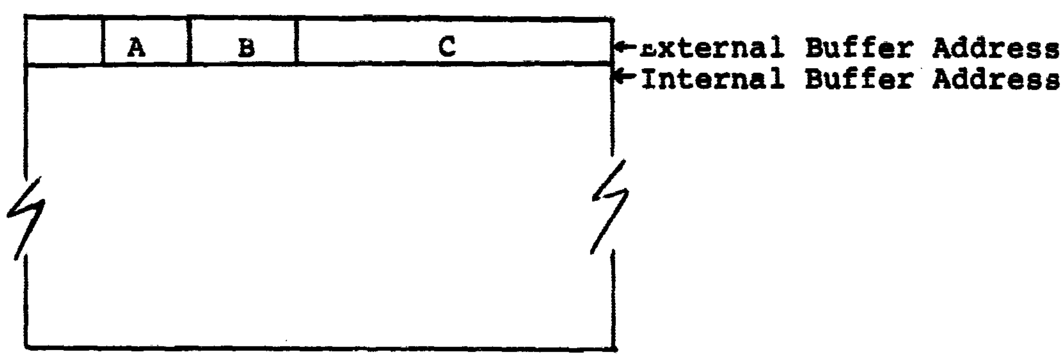
5.5. Performance Evaluation Analysis of EXPOOL

EXPOOL is a core resident element within the EXEC 8 operating system that contains a buffer pool and two routines to maintain this pool. EXPOOL is one of the most active elements in the EXEC 8 supervisor. All system tables, queues, and control words are located in the EXPOOL buffer pool. Because of its central role, the frequency of use within the system, it was chosen for detailed analysis using the techniques developed during this study.

5.5.1 The Buffer Pool

The common buffer pool within EXPOOL is maintained in order to provide a maximum number of buffers with a minimum amount of overhead. The 'buddy' system storage allocation technique is used here with permissible buffers of 2^n-1 words, where $2 \leq n \leq 9$. The buddy system has been described in Section 3. The structure of a buffer is shown in Figure 5-4.

The EXPOOL buffer pool initially contains 27 blocks of 2^9 words each as implemented in the University of Maryland EXEC 8 Operating System. Of the 27 blocks, 10 blocks are generated at assembly time and 17 blocks are given to the EXPOOL buffer pool by linking 17 blocks of no-longer-needed core to the end of the available chain upon termination of system initialization. When all space within EXPOOL has been allocated, the buffer pool may be expanded by calling CRQED to get a block of 2^9 words from system D-bank. The borrowed core space will be released for user program use as soon as it is no longer needed in the buffer pool. When the total unused space is less than 4000 memory words, the buffer pool is set to a tight mode. In the tight mode, only critical requests, i.e. those with the flag set, can be allocated space. All other requests are linked



A = { 0 if the buffer is used.
B if the buffer is free. }

B = the internal size index.

C = { the link to the next buffer if the buffer is free.
the function ID if the buffer is used by a function.
the switch ID if the buffer is not used by a function.
the return point if the buffer is used by the EXEC main interlock code. }

Figure 5-4. Structure of a One Block Buffer of Size 2^B.

to the EXPOOL request chain and the requestor is deactivated by EXPOOL.

5.5.2 Request for and Release of a Buffer from EXPOOL

To request a buffer storage area from EXPOOL, the following calling sequence is used:

```
LXI,U        X11,P
LMJ          X11,EXPOOL
```

On exit from the request, the program leaves the external buffer address in the AO register, the return address in index II (XII), and the address of the word that contains the user specified parameters, P, in the Al register. The information indicating the exact nature of the buffer request is made available to EXPOOL in the following format:

P:

SIZE	N		F	C	ADDRESS

where: SIZE = number of words in the buffer desired.

N=0 : needs a buffer when it becomes available
  1 : must receive the buffer immediately to continue processing.

F=0 : add to the end of chain
  1 : add to the front of chain

C=0 : no chaining
  1 : chain as specified in F

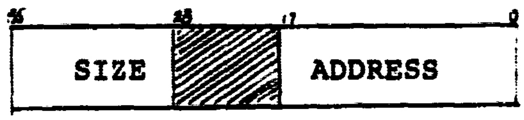
ADDRESS = a pointer to the control word if C=1; or the address of the buffer to be assigned if C=0.

To release a buffer storage area from EXPOOL, the routine EXREL is initiated by providing the following calling sequence:

```
L          A0,P
LMJ        X11,EXREL
```

where, P has the following format:

P:



5.6. Preliminary Results of an Analysis of EXPOOL

The efficiency of a function or program depends both on the algorithm used, and the effectiveness of the code used to implement the algorithm. In evaluating EXPOOL both the algorithm and the implementation have been analyzed. As described in Section 3, a simulation model of the buddy system storage allocation technique, as well as several other allocation schemes have been constructed and run on the Univac 1108.

Several core memory dumps of the EXPOOL buffer pool have been taken. The distribution of used buffer size was calculated according to the results obtained from the memory dumps, and has been used as the input source to ITFVA (Instruction Trace and Functional Value Analysis) described in Section 5.2. The time interval between a buffer being allocated and released is assumed to be an exponential distribution. Under ITFVA requests and releases are called. Figures 5-5 & 5-6 show the analysis result of the original EXPOOL program. We see 23.7 percent of of the allocation time has been spent in looking through the table, TAB2, to convert the external request size into the internal buffer size index. It is interesting to note that within EXPOOL, the table TAB2 is ordered randomly as shown in Figure 5-7. That is, there is no rationale for the sequence of entries in the table. It is of interest to calculate the average time required to search for an entry in the table. If we let E be the average search time to find a matching entry in TAB2, N(i) be the number of instructions needed to access the ith entry in the table, and P(i) be the probability that

CODE EXECUTION FREQUENCY FOR EACH INTERVAL

LABLE	RELATIVE LOCATION START	END	TOTAL INST. EXECUTED	PERCENT OF RUN TIME
EXPOOL	0015	0030	3141	16.87
EXP2	0031	0037	500	2.69
EXPEXT	0040	0070	1300	6.98
INLK	0071	0142	1300	6.98
REQUES	0143	0154	976	5.24
NOMORE	0155	0171	1441	7.74
REQ2B	0172	0204	630	3.38
MCORE	0205	0277	3	.02
EXREL	0300	0316	3341	17.94
ER22	0317	0330	800	4.30
EXREXT	0331	0343	500	2.69
ER23A	0344	0356	0	.00
RELEAS	0357	0363	1680	9.02
REL1.1	0364	0413	1378	7.40
REL1.2	0414	0434	811	4.36
REL2	0435	0442	900	4.83
REL3	0443	0446	0	.00
REL56	0447	0473	0	.00
OTHER	0000	0000	2	.01

TOTAL 18619 INSTRUCTION EXECUTED DURING THIS ANALYSIS.

① ② ③ ④ ⑤

Figure 5-5. Code Execution Frequency for Each Labeled Block of the Accessing Routines (EXPOOL/EXREL) as Implemented in EXEC 8.

1. The block symbolic name, i.e. label.
2. The relative location of the label to the start of the routine.
3. The relative location of the instruction preceding the next label.
4. The total number of executed instructions within each labeled block of the routine.
5. The percentage of total run time spent in each labeled block of the routine.

THE MOST FREQUENTLY EXECUTED INTERVALS

LABLE (EXREL) STARTING LOCATION 0300 TOTAL 3341 INSTRUCTION EXECUTED.

MACRO INST. LOCATION START	END	MACRO INST. LENGTH	EXECUTION FREQUENCY	TOTAL INST. EXECUTED	PERCENT
0300	0311	9	100	900	26.94
0312	0313	2	100	200	5.99
0312	0315	3	747	2241	67.08

LABLE (EXPOOL) STARTING LOCATION 0015 TOTAL 3141 INSTRUCTION EXECUTED.

MACRO INST. LOCATION START	END	MACRO INST. LENGTH	EXECUTION FREQUENCY	TOTAL INST. EXECUTED	PERCENT
0015	0023	7	100	700	22.29
0024	0025	2	100	200	6.37
0024	0027	3	747	2241	71.35

LABLE (RELEAS) STARTING LOCATION 0357 TOTAL 1680 INSTRUCTION EXECUTED.

MACRO INST. LOCATION START	END	MACRO INST. LENGTH	EXECUTION FREQUENCY	TOTAL INST. EXECUTED	PERCENT
0357	0403	16	60	960	57.14
0357	0405	18	40	720	42.86
(1)	(2)	(3)	(4)	(5)	(6)

Figure 5-6. Analysis of Most Frequently Executed Labeled Blocks of the Accessing Routines (EXPOOL/EXREL) as Implemented in EXEC 8.

1. The relative location of the first word of each macro-instruction to the start of the routine.
2. The relative location of the last word of each macro-instruction to the start of the routine.
3. The number of instructions in each macro-instruction.
4. The number of times the macro-instruction was executed.
5. Total instructions executed in each macro-instruction.
6. The percentage of labeled block execution time spent in the macro-instruction.

<u>TAB2 as Implemented in the EXEC 8</u>

```
. Table of External and Internal Buffer Sizes

.        + External Size, Internal Size Index

TAB2.
                  +        3,2
                  +        6,3
                  +        28,5
                  +        56,6
                  +        224,8
                  +        127,7
                  +        15,4
                  +        7,3
                  +        31,5
                  +        63,6
                  +        255,8
                  +        511,9
```

<u>TAB2 Reordered to Optimize Table Lookup Process</u>

```
. Table of External and Internal Buffer Sizes

.        + External Size, Internal Size Index

TAB2.
                  +        511,9
                  +        127,7
                  +        224,8
                  +        255,8
                  +        56,6
                  +        63,6
                  +        6,3
                  +        7,3
                  +        15,4
                  +        28,5
                  +        31,5
                  +        3,2
```

Figure 5-7. Structure of TAB2 as Used in EXEC 8 and Structure of Reordered TAB2.

the ith entry in the table is requested, then

$$E = \sum_{i=1}^{12} N(i)*P(i).$$

If N(i) = n*i, where n is a constant, the value of E is minimized if $P(i) \leq P(j)$ for all $j \geq i$. That is, a minimum search time can be obtained if the table entry is given in decreasing order according to its probability of occurrence. In Figures 5-8 & 5-9, the result of reordering the table, TAB2, according to the size usage distribution obtained in Section 3 is shown. The percentage of CPU time spent in this table lookup is still high, but an average of 15.5 percent of allocation time has already been saved.

An additional saving in time may be obtained by recalling that the buddy system storage allocation technique is so defined because each buffer request made for a block of size n, where $2^k \leq n < 2^{k+1}$, is allocated a block of exactly 2^{k+1} words providing 2^{k+1} is less than or equal to the maximum block size permitted. In most allocation schemes, to convert an external request length to the internal size index, a table lookup is used. Actually, the feature of the buddy system provides a very easy way to handle the conversion. The simple formula is that the internal buffer size index k equals the number of bits in the machine word minus the number of bits with leading zeros. For this, a single shift and count instruction can get the size index immediately. Now the average search time E is decreased substantially. For, in this case, N(i) becomes a constant, c, the time to perform the shift and count instruction. Hence E = c. Figure 5-10 shows the result of the above change in the time required to access the appropriate word. An average of 29.1 percent saving for each request (or release) is gained over the code currently implemented in EXEC 8.

CODE EXECUTION FREQUENCY FOR EACH INTERVAL

LABLE	RELATIVE LOCATION START	END	TOTAL INST. EXECUTED	PERCENT OF RUN TIME
EXPOOL	0015	0030	1485	9.70
EXP2	0031	0037	500	3.27
EXPEXT	0040	0070	1300	8.49
INLK	0071	0142	1300	8.49
REQUES	0143	0154	976	6.38
NOMORE	0155	0171	1441	9.41
REQ2B	0172	0204	630	4.12
MCORE	0205	0277	3	.02
EXREL	0300	0316	1685	11.01
ER22	0317	0330	800	5.23
EXREXT	0331	0343	500	3.27
ER23A	0344	0356	0	.00
RELEAS	0357	0363	1680	10.98
REL1.1	0364	0413	1378	9.00
REL1.2	0414	0434	811	5.30
REL2	0435	0442	900	5.88
REL3	0443	0446	0	.00
REL56	0447	0473	0	.00
OTHER	0000	0000	4	.03

TOTAL 15307 INSTRUCTIONS EXECUTED DURING THIS ANALYSIS.

(1) (2) (3) (4) (5)

Figure 5-8. Code Execution Frequency for Each Labeled Block of the Accessing Routines (EXPOOL/EXREL) after Re-ordering the Table, TAB2.

1. The block symbolic name, i.e. label.
2. The relative location of the label to the start of the routine.
3. The relative location of the instruction preceding the next label.
4. The total number of executed instructions within each labeled block of the routine.
5. The percentage of total run time spent in each labeled block of the routine.

THE MOST FREQUENTLY EXECUTED INTERVALS

LABLE (EXREL) STARTING LOCATION 0300 TOTAL 1685 INSTRUCTION EXECUTED.

MACRO INST. START	LOCATION END	MACRO INST. LENGTH	EXECUTION FREQUENCY	TOTAL INST. EXECUTED	PERCENT
0300	0311	9	100	900	53.41
0312	0313	2	100	200	11.87
0312	0315	3	195	585	34.72

LABLE (RELEAS) STARTING LOCATION 0357 TOTAL 1680 INSTRUCTION EXECUTED.

MACRO INST. START	LOCATION END	MACRO INST. LENGTH	EXECUTION FREQUENCY	TOTAL INST. EXECUTED	PERCENT
0357	0403	16	60	960	57.14
0357	0405	18	40	720	42.86

LABLE (EXPOOL) STARTING LOCATION 0015 TOTAL 1485 INSTRUCTION EXECUTED.

MACRO INST. START	LOCATION END	MACRO INST. LENGTH	EXECUTION FREQUENCY	TOTAL INST. EXECUTED	PERCENT
0015	0023	7	100	700	47.14
0024	0025	2	100	200	13.47
0024	0027	3	195	585	39.39
(1)	(2)	(3)	(4)	(5)	(6)

Figure 5-9. Analysis of Most Frequently Executed Labeled Blocks of the Accessing Routines (EXPOOL/EXREL) after Reordering the Table, TAB2.

1. The relative location of the first word of each macro-instruction to the start of the routine.
2. The relative location of the last word of each macro-instruction to the start of the routine.
3. The number of instructions in each macro-instruction.
4. The number of times the macro-instruction was executed.
5. Total instructions executed in each macro-instruction.
6. The percentage of labeled block execution time spent in the macro-instruction.

CODE EXECUTION FREQUENCY FOR EACH INTERVAL

LABEL	RELATIVE LOCATION START	END	TOTAL INST. EXECUTED	PERCENT OF RUN TIME
EXP001	0000	0004	900	7.12
EXP2	0005	0013	100	.79
EXPEXT	0014	0044	1300	10.29
INLK	0045	0112	900	7.12
REQUES	0113	0124	976	7.72
NOMORE	0125	0141	1441	11.40
REQ2B	0142	0154	630	4.99
MCORE	0155	0247	3	.02
EXREL	0250	0254	800	6.33
FR22	0255	0266	600	4.75
EXREXT	0267	0302	300	2.37
FR23A	0303	0311	0	.00
RELEAS	0312	0316	1680	13.29
REL1.1	0317	0347	1378	10.90
REL1.2	0350	0367	811	6.42
REL2	0370	0375	900	7.12
REL3	0376	0401	0	.00
REL56	0402	0426	0	.00
OTHER	0000	0000	6	.05

TOTAL 12637 INSTRUCTION EXECUTED DURING THIS ANALYSIS.

① ② ③ ④ ⑤

Figure 5-10. Code Execution Frequency for Each Labeled Block of the Accessing Routines after Eliminating the Table, TAB2.

1. The block symbolic name, i.e. label.
2. The relative location of the label to the start of the routine.
3. The relative location of the instruction preceding the next label.
4. The total number of executed instructions within each labeled block of the routine.
5. The percentage of total run time spent in each labeled block of the routine.

In the 1108 executive system there will be essentially the same number of releases as requests for buffer storage after the system stabilizes, so that, in the following discussion, no attempt is made to distinguish the type of action requested in the allocation process. In the EXEC 8 version of the allocation routine, by using the TRACE routine it was found that the average number of instructions required for an allocation was 103. In the 1108, the average time per instruction is 1.75 μsec. Therefore, the time spent in one allocation process is 1.75 μsec times 103 instructions or .180 msec.

By reordering the table, TAB2, so that the order of the entries in TAB2 are given in decreasing order according to their probability of occurrence, the average number of instructions required for an allocation was found to be 87. The time spent in the allocation process is then .152 msec, a reduction of .028 msec per allocation. By introducing a shift and count instruction to replace the table lookup process, the average number of instructions was reduced to 74. The time spent in the allocation process is then .130 msec. This represents a reduction of .050 msec over the EXEC 8 version or a reduction of .022 over the version with a reordered TAB2.

The significance of this reduction in the number of instructions executed per allocation can be seen only in relation to the frequency with which this routine is executed. If the routine is executed infrequently, this reduction in instructions executed is of little consequence. If, however, it is found that a buffer request occurs every k milliseconds for a particular installation, the percent of total running time and the actual time spent in performing this function may be significant and can be calculated directly. A table has been prepared indicating the reduction in total running time per 8 hour shift which can be realized as a function of the execution frequency of the allocation routine. The results are given in Figure 5-11.

The frequency with which this routine is executed is a function of the installation operating environment and the executive system activities, in particular input and output activities. It has been estimated by the systems staff at the University of Maryland that the allocation process is executed at least every 25 msec. Assuming an operating expense of $500/hour, the consequence of the implementation is a loss of between $5000 and $10,000 per year for the University of Maryland Computing Center on a three shift basis. As indicated above, the loss experienced will vary from installation to installation. An internal software monitor could be used at each particular installation to determine the exact frequency with which the allocation routine is executed. The loss could then be assessed. The results obtained would then determine the expected improvement in system performance through the modification of this routine.

It may be that the results found by monitoring the execution frequency of the allocation routine would not warrant system modification if this were the only installation with this 'performance bug'. Considering the fact that this is not a special purpose operating system, but rather one which is utilized at many computing centers throughout the country, the composite loss appears to be such that system modification and improvement in system performance is imperative.

1108 Running Time Spent in O/S Buffer Accessing Routines

Accessing interval in msec.	(1) EXEC 8 Accessing Technique		(2) Reordered TAB2 Accessing Technique			(3) Shift and count instruction Accessing Technique			
	%	time/8 hr. in hours	%	time 8/hr. in hours	time saved in hours over (1)	%	time 8/hr. in hours	time saved in hours over (2)	time saved in hours over (1)
3	6.0	.48	5.1	.408	.072	4.3	.344	.064	.136
5	3.6	.288	3.04	.243	.037	2.6	.208	.035	.080
10	1.8	.144	1.52	.121	.023	1.30	.104	.017	.040
25	.72	.0576	.60	.048	.0096	.52	.0416	.0064	.016
50	.36	.0288	.30	.0243	.0045	.26	.0208	.0035	.0080
100	.18	.0144	.15	.0121	.0023	.13	.0104	.0017	.0040
200	.09	.0072	.075	.006	.0012	.065	.0052	.0008	.0020
500	.036	.00288	.03	.00243	.00045	.026	.00208	.00035	.0008
1000	.018	.00144	.015	.00121	.00012	.013	.00104	.00017	.0004

Figure 5-11. Comparison of Alternative (EXPOOL/EXREL) buffer Accessing Techniques as a function of average accessing Intervals in msec.

6. Analytic Studies Summary

Two mathematical models were developed in an effort to characterize information handling centers. In both cases, the information center was assumed to be a two level store with a primary and a secondary store. The difference in the models lies in the assumptions made about the stores. In the first model, the primary store is assumed to be infinite. In the second model, this restriction is removed and the model is solved for a bounded primary store.

Abstracts of the two reports are given here. The complete reports have been submitted to NASA as independent documents.

6.1. Storage Requirements for Information Handling Centers

By H. M. Gurk and J. Minker

In this paper the authors investigate a stochastic model relevant to certain kinds of information handling centers, best typified by computer utilities and document storage and retrieval. The growth characteristics of an information center are evaluated for a retirement policy that governs when items are retired from a two level auxiliary store (disc or drum in the case of the computer utility and document files in the case of the document center) to a less accessible store. A retired document, or segment in a utility environment, may be reactivated and brought back into the primary store provided that a sufficient number of requests have been made for it.

For a given retirement and reactivation policy, an integral equation is derived for the expected number of items in the primary store. This equation depends upon the arrival distribution for documents, the request distribution, and the parameters associated with the retirement policy. No particular limiting assumptions have been made with respect to the form of the distributions. Explicit solutions to the integral equation are derived for document arrivals that follow a Poisson distribution to determine the expected steady-state size of the primary store, and the standard deviation of the size.

6.2. A Stochastic Model of an Information Center by J. Minker

In the earlier paper, the author investigated a stochastic model relevant to information handling centers best typified by computer utilities and document storage and retrieval centers. The growth characteristics of information centers were evaluated for retirement policies that govern when items are retired from a primary store to a less accessible store. The results obtained assumed that the primary store was of unbounded capacity. In this paper we remove this restriction and consider the case where the primary store has a finite capacity.

A set of integral equations is derived for the expected number of items in the primary store. The integral equations depend only upon the arrival distribution for documents, the request distribution, and the parameters associated with the retirement policy. No particular limiting assumptions have been made with respect to the form of the distributions.

The set of integral equations are solved for document arrivals that follow a Poisson distribution. The expected value of the size of the store approaches the result given in the first paper as M, the size of the primary store, becomes unbounded.

7. Technical Reports and Publications

Documents resulting from work performed under this grant are as follows:

(1) Minker, J., 'A Stochastic Model of An Information Center', University of Maryland Technical Report 69-90, July 1969.

(2) Gurk, H.M., Minker, J., 'Storage Requirements for Information Handling Centers'. To be published in Journal of the ACM, January 1970.

(3) Crooke, S., Minker, J., 'Key Word In Context Index and Bibliography on Computer System Evaluation', University of Maryland Technical Report December 1969.

8. Quarterly Reports

Brief reports that describe the direction of research have been issued on a quarterly basis. The following quarterly reports have been submitted to NASA under this grant.

(1) First Quarterly Report (October 1, 1968 - December 31, 1968) NASA Grant NGR21-002-197.

(2) Second Quarterly Report (January 1, 1969 - March 31, 1969) NASA Grant NGR21-002-197.

(3) Third Quarterly Report (April 1, 1969 - June 30, 1969) NASA Grant NGR21-002-197.

(4) Fourth Quarterly Report (July 1, 1969 - September 30, 1969) NASA Grant NGR21-002-197.

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 Info., 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", Comput. 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, Nov. 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. Rpt. 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, N.J., 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 Mich., Ann Arbor, Mich., 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 Conf., 103-109.

25. Stevens, D.F., "System Evaluation on the Control Data 6600", IFIP International Conference 1968, pp.C34-38.

26. Deniston, W.R., "SIPE: A TSS/360 Software Measurement Technique", Proc. of ACM 24th National Conf.,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 Conf., 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, Cal., 1968

33. Apple, C.T., "The Program Monitor - A Device for Program Performance Measurement". ACM 20th National Conf. 1965, pp.66-75.

34. Cantrell, H.N., Ellison, A.L., "Multiprogramming System Performance Measurement and Analysis". AFIPS SJCC 1968, pp.213-221.

35. Campbell, D.J., Hefener, W.J., "Measurement and Analysis of Large Operating Systems during System Development". AFIPS FJCC 1968, pp.903-914.