THE UNIVERSITY OF ALBERTA

ADAPTIVE PAGE FAULT CONTROL: USE OF THE WORKING SET PARAMETER

by

© BRIAN J. WESLEY

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF MASTER OF SCIENCE

DEPARTMENT of COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL, 1974

ABSTRACT

The properties of the Working Set Model of Program Behavior are used to analyse the relationship between page demand and memory utilization on virtual, paged computers. The study defines possible operating system implementations of the working set parameter or "window size" for purposes of page fault control. Topics discussed are adaptive control and size determination of the parameter plus solutions to the problem of working set recognition. Additional possibilities for adaptive control of paging are investigated by relating empirically derived program working set behavior to source languages and instruction types executed.

The working set concepts are extended to apply at the page level. An investigation of working set page characteristics is used to determine the feasibility of adaptive memory management based on individual page behavior.

ACKNOWLEDGEMENTS

I wish to thank Professor T. A. Marsland, my supervisor, for his advice, criticism, and guidance during the research and preparation of the thesis. I would also like to thank Mr. A. R. Davis of the University of Alberta Computing Services for the information on MTS, and fellow graduate student, Mr. S. Sutphen, for his criticism and assistance during data accumulation and analysis.

The financial assistance received from the National Research Council of Canada, in the form of a scholarship and busary, and from the Department of Computing Science, in the form of a teaching assistantship, is gratefully acknowledged.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (cont'd)

I: Introduction

Because of the operational complexity of virtual memory computers, it is extremely difficult to understand how paging rates and storage utilization are related. A major goal of the thesis is to investigate this relationship by examining measurements made of missing page rates and expected working set sizes of individual programs, as described in Denning's Working Set Model of Program Behavior [D2]. Fifteen different program runs representing six different programming languages are analysed. Page reference strings were gathered by executing interpretively one million instructions per program. They provide the data necessary to construct average working set size and missing page rate curves for each program.

The study also proposes extensions to the original working set concepts to include the behavior of individual pages. Three selected programs are examined to determine empirically whether or not a working set approach can be applied at the page level.

The working set curves provide information about the value of the working set parameter, T, desired by an individual program for efficient execution. This parameter or "window size" is the length of process execution time that determines whether or not a page is part of a program's

working set. In general, as the window size increases so does the number of pages allocated to a program in main memory; while the same program's average paging rate decreases. The thesis not only investigates the size of the parameter T, but also discusses adaptive control of the window size and means of effecting such control. Current implementations of working set concepts are used to illustrate different adaptive procedures.

The working set curves are also used to gather information on program characteristics. In particular, an exponential curve is fitted to each missing page rate curve to determine quantitatively any differences or similarities between the various programs. The resultant coefficients obtained from the exponential fits are compared to those determined in a previous study [R4].

To provide a more detailed description of the content of this study, a brief discussion on the subjects within each of the remaining chapters follows.

The early computer environments consisted of a single user, operating a small primitive device executing a program written in machine code. Today, a far different situation exists with the availability of diverse high-level languages, multiple users having "simultaneous" machine access, and sophisticated operating systems controlling

computer resources. Chapter II describes some of the improvements in memory utilization techniques between these two eras, including the development of virtual memory.

Chapter III presents the results of some previous and current studies of program behavior on virtual, paged computers. Included in the description of each study are the procedures used for gathering and analysing the program data.

Chapter IV, the Working Set Model, summarizes Denning's theoretical model of program behavior including the basic properties of locality of reference and the characteristics of the working set, W(t,T). The program behavior studies in Chapter III are related to the properties of the model. By the addition of several new properties, the model is extended so that its concepts are applied at the page level. The chapter concludes with a discussion of the limitations of the model.

Determining how the working set parameter or window size is implemented on an operating system is the subject of Chapter V. Should each program's working set be recognized or is it sufficient to recognize only a "system" working set? Should the chosen window size be fixed or variable in time? Should the window size be the same for all programs or should each program have its own individual working set

parameter? The assumptions necessary and the reason for each type of approach are presented. Several implementations of adaptive windowing are described. The feasibility of extending window size control to individual program pages is discussed.

Chapter VI presents the empirical results of the research; the determination of the program and page behavior curves. These results are related to the findings of Chapter V. Also included is a discussion of previously published empirical working set results and a description of the investigative procedures used in this study.

The final chapter presents further conclusions, a summary of all results, and ideas for further research.

II: Development of Memory Utilization Techniques

Prior to 1956, computers were primitive by today's standards. Usually, main memory was less than 12K words residing on either a William's tube or mercury delay line. Computers also existed which used a magnetic drum as the main storage unit for the CPU. The very limited amount of real memory necessitated the use of very succinct programming code. The structure of any program was also extremely important on drum memory computers so that instruction access time could be minimized. Since programming was done using machine code and absolute addressing, programmers had to have a very intimate knowledge of the problem, their program, and their computer.

Despite the development of some simple assemblers, essentially no software system support for computers was available until the appearance of the first operating systems in 1956. They consisted of a monitor, used for control purposes, plus various, frequently used, subroutines. The operation of the system was batch-oriented. Many programs would be placed on a single tape which was then used as the input job stream. When the last job on the tape had been executed, the system would call for a new "batch" of programs so that processing could continue. The system improved CPU and memory utilization by providing the

commonly used loaders, compilers, I/O routines, and reducing the physical switching time between the serially executed jobs. However, even this simple operating scheme would have been impractical without the advent of magnetic core memory which allowed greatly expanded, rapid, random access storage for programs.

During the period of 1956-58, the FORTRAN programming language was developed. It and the many other languages which have followed have greatly simplified programming. Since they provide the means of stating more easily the algorithmic solution to a problem for machine application, even casual users can now have access to computing. But the success of computer languages has also increased demand for the use of computing resources, including memory. The desire for more computer power has prompted continued improvement in existing software and development of new hardware and software capabilities.

Two improvements developed in this period were relocatable code and the use of overlays. Relocatable code for user's programs simplified the use of absolute system code. The technique also enabled the system to exercise more control over where and when problem programs were loaded into memory. Overlaying allowed a programmer to construct a program that during execution could overlay parts of itself

with code from a secondary storage device. By removing the execution requireme. of an entirely resident program, machine storage capacity limitations could be circumvented. During the period of 1959-61, transistorized circuitry was introduced which replaced vacuum tubes used on earlier production computers. Because of the increase in electronic switching speeds, much faster CPU operation was realized. Equally important was the development of the asynchronous I/O channel. Essentially a small computer, it controls the actual transfer of data between the external I/O devices and the internal buffers of the CPU. When equipped to interrupt the main frame activity of the CPU, the channel provides the computer system with the ability to perform fully overlapped I/O. Instruction execution and the input and output of data can proceed simultaneously, not only improving the utilization of memory but also enhancing the processor capabilities.

During the middle 1960's the third generation of computers was marketed. These machines, exemplified by the IBM /360 line, introduced the use of integrated circuitry and increasing amounts of hardware protection for the system supervisory program. Because of the increasing number of casual programmers, the development of programmable interrupting clocks, supervisory and user modes of computer operation, and memory read/write protection was necessary to

guarantee system integrity for all users.

Multiprogramming, the sharing of memory between a number of tasks, each concurrently competing for system resources was a major software development. There are several advantages in using this technique. Having multiple jobs on the system, the influence of I/O wait time can be reduced since another task can be executed if the currently running program blocks for I/O. By using a system software scheduler, the "turn-around" time for a job can be controlled by manipulating its execution priority. Supervisory control of program execution can be further enhanced by allocating execution time to individual tasks in time slices or quanta. A more uniform approach to system design can be taken since all I/O processors can be treated as tasks in the multiprogramming environment.

The simplest multiprogramming systems are batch-oriented, with each user program memory resident until completed. A modification to this approach is known as time-sharing. The system employs the use of on-line terminals developed under the following assumptions. It is desirable that programmers can interact with programs, that each interaction would require only small amounts of CPU time and memory, that a program's space requirements can be swapped easily in and out of main memory, and that the time between

each user's interaction is relatively long. Thus by giving each interactive user quite limited CPU time, and swapping programs when required, even larger numbers of concurrent users could be serviced without inordinate expenditures in expanding computer memory resources.

Although time-sharing as described can be implemented using a fixed memory allocation, it inherently advocates a dynamic approach to memory management. Thus, time-sharing is often seen in context with virtual memory, a concept which depicts the user's address space as one-level storage [D3,K4]. Each program appears to be running on a computer with the same operating characteristics as the real one except for a greatly expanded memory. In actuality, the real machine must have a hardware mapping device which automatically translates, during execution, a program's virtual address to the machine's real address [I5]. Every task's excess addressable memory must reside on a secondary storage medium. It is transferred to main memory by the system when referenced by the task and removed whenever the system requires additional real memory. This automatic folding of the program is transparent to the user [S2].

The two basic schemes used to translate virtual to real memory addresses are segmentation and paging. Using a segmented approach, a program's virtual address space is

divided into variable size, content-related blocks of contiguous code called segments [B1,B7,D7,R1,V3]. Each virtual address consists of s+d bits, where the s bits identify the particular segment; and the d bits, the displacement of the addressed location within the segment. The maximum size of any segment is 2 to the power d addressable locations. Since a segment can have any length less than the maximum, its size must be known by the system for memory allocation purposes. Typically, real memory consists of active segments interspersed with "holes" of unallocated storage space. Since these holes are not executable code, they contribute to the system overhead. When a segment is referenced and must be loaded into memory, the storage allocation routine must determine if there is a hole available to fit the segment. If there is, the segment is transferred to memory producing another hole, smaller than the original one. The proliferation of small holes which normally will not hold a segment is a form of storage loss known as external memory fragmentation [D4,R3]. When memory becomes fragmented, transferring a new segment to memory may require either a re-grouping of the active segments to create one large hole or removing some segments according to a segment replacement scheme (or perhaps both). The managing of variable block sizes is both a costly and complex process and is the chief disadvantage of using a

segmented translation scheme.

A second approach, paging, reduces memory allocation difficulties by dividing a program's virtual memory into fixed sized blocks of code called pages. External memory fragmentation becomes nil since all "holes" are available for use. The only real waste of memory space occurs when the code available for loading does not completely fill a page. Such memory loss is called internal memory fragmentation [R3]. Normally, this situation occurs only in the last page of a program unless the system loader pays particular attention to page boundaries. For example, the loader on the Michigan Terminal System (MTS) [A2,M5,M6] system attempts to place a program module on a page boundary when not enough space remains in a previously loaded page [D1]. Under these circumstances internal fragmentation may be more pronounced. Because of the inherent simplicity of memory allocation using paging, the approach has become widely adopted in the computer industry [A1,A3,C3,I5,K6,O2,W1]. For this reason, the remainder of the thesis is restricted to memory management via paging.

A paging environment raises two prominent questions concerning the management of the memory resource. What algorithm will be used to fetch pages into memory and what one will be used for removing pages when more holes are

required? Nearly all systems currently in use employ "demand" paging for deciding which pages are to be brought into memory. A page, not in memory, is demanded whenever it is referenced by a program. A page fault is said to occur and the system prepares to fetch the demanded page. Some form of pre-paging is not used because there has been little evidence of any advantage of using this technique [D4]. However, there has been at least one recent investigation into pre-paging which reports an overall improvement in drum utilization [G2].

Deciding which page to remove from memory requires a page replacement algorithm. Many different algorithms have been employed or suggested for storage management purposes [A1,B2,B3,C3,C4,D2,D4,I2,I5,K3,S3,T1,T2]. A random page replacement policy simply removes a page at random. A first-in, first-out (FIFO) algorithm removes the oldest page in memory when space is required. Although both of these policies are easy to implement, they do not accurately reflect program behavior. Programs do not usually reference pages at random, and the oldest page may not be the least likely page to be referenced next. A far more sophisticated algorithm was implemented on the Atlas Time Sharing system, but was also not very successful [B3,K3]. This algorithm was based on the assumption that programs always use looping structures. A more successful algorithm is the least

recently used (LRU) replacement policy [C4]. Since it can also be easily approximated by hardware, it has become the most widely used approach. The LRU algorithm recognizes that most programs are characterized by a locality of reference among the more recently used pages. All pages not recently referenced by a program will have their hardware "use" bits off. The operating system considers these pages as prime candidates for removal. Unfortunately, this policy can lead to disastrous system performance if applied without concern to the relationship between memory utilization and program paging levels [D3]. Denning, in his Working Set Model of Program Behavior, attempts to provide this relationship and suggests yet another page replacement algorithm [D2]. This model is discussed in Chapter IV and provides the focal point for this study.

It is obvious that many of the early page replacement algorithms were based on misconceptions of program behavior. The next chapter gives a brief review of empirical studies of program characteristics in a paging environment.

III: Studies of Program Behavior in a Paging Environment

The objective of Chapter III is to present a short review of several studies of program behavior in a paging environment. Five separate investigations are described covering a period of 1966 to 1973. They are by Fine, Jackson, and McIsaac [F1]; Coffman and Varian [C1]; Freibergs [F3]; Brawn and Gustavson [B5]; and Lewis and Shedler [L1]. Each description briefly details the investigative procedure used to gather the program data, how this data was manipulated, and the results produced. None of these studies involve program working set behavior or related concepts.

The study by Fine, Jackson, and McIsaac was one of the first studies done on program behavior under paging. Object programs were interpretively executed to obtain the execution and data page addresses, plus the current instruction count. These page references were recorded for a service interval, a period of time terminated either by a call to the system or by the execution of 80,000 instructions. This number of instructions represents approximately four hundred milliseconds of CPU time or the time slice length for the Q-32 Time Sharing System (for which the reports was originally done). One hundred and

eighty-two of these intervals were studied. They represented thirty-five requests for CPU service, having total instruction lengths ranging from 7 to as many as 1,281,504 instructions. At the beginning of any service interval, the instruction count was set to zero and all pages of the program were considered inactive (not in memory). During simulated execution, a page once referenced was considered available for the remainder of that interval. The results indicate that until a program obtained a "sufficiency" of pages, it demanded pages at a very rapid rate. This "sufficiency" of pages was a considerable fraction of a program's total declared page requirements. Finally, a program would not execute long between calls for a new page even after acquiring a sufficiency of pages. In summarizing these results, the study was very pessimistic concerning the benefits of a demand paging strategy on a time-sharing system:

> " the usual concept of a high-speed memory filled with a page or two from each of many programs desiring processing does not look as though it will stand up subject to the page call rates observed in this study. The page-fetching mechanism seems likely to congest within a few milliseconds; until some of the programs acquire a sufficiency of pages there would be little chance of processing-fetching overlap; and a sufficiency of pages for some programs means that others must be squeezed out of core and deferred " [F1,p.227]

The study also concluded that re-organizing programs as a solution to the apparent inefficiency of demand paging was

"unrealistic" since it meant attempting to fit the work to the system instead of vice versa.

Despite this grim report, development and investigations of paging systems continued. In 1968, the next three studies to be described were published.

Coffman and Varian presented further experimental data on programs in a paging environment. They also used interpretive execution of object programs to obtain execution and data addresses. These addresses were mapped into page addresses which were analysed to gather paging statistics. These statistics were collected during the simulated execution of the program. The procedure was to vary either the page size (the mapping imposed on the object code) or the number of pages allowed to be active (in memory) during different interpretive program executions. Typically, 125,000 instructions were executed per program. A result, consistent with the Fine study, was that program executing when substantially less than memory resident induced excessive page turning in light of the existing design of paging systems. The LRU page replacement strategy was studied indicating an operational performance of 30-40% of that of the optimal page removal scheme [B3]. A final result concerned program behavior and page sizes. If program execution was confined primarily to small areas within a

page, performance was improved more by increasing the number of pages in memory rather than increasing page size [H1,H2].

Freibergs' study of programs was more generalized. Although object programs were interpretively executed, an actual trace of execution and data addresses was produced which was then examined by an analyser program. Supervisory calls to the system were not executed interpretively since they were thought to be too system-dependent. The program code was divided into pages of 1024 words or less where each functional part of the program began on a new page boundary. Freibergs felt that simply dividing the IBM 7044's (the machine used in study) memory into 32 equal size pages would result in an over-estimation of the number of page boundary crossings during execution. He also felt that this structure would more acccurately reflect the memory organization of a paged computer system. A total of 10.35 million instructions were traced from the execution of FORTRAN, string processing, GPSS, FORTRAN compilation, and COBOL programs. The results of the study showed that most SVC's are due to I/O requests. (An SVC is an instruction which when executed by the problem program initiates a supervisory controlled operation. An example of one such operation is any input/output on a large computer system). Assuming overlapped I/O channel operation and a blocking factor of ten, the study determined that the number of instructions

executed by a program would be 10^3 to 10^4 before halting for an input/output operation. This would be true about 50% of the time. Finally, all the jobs executed over a 24 hour period at the McGill University Computing Centre were analysed. It was found that 50% of all programs studied required 4-6 pages of memory of which only 2-3 of these pages were instruction code. The study concluded that any scheduling policy should allocate each program at least six pages of memory before any execution activity proceeds.

The final 1968 study is by Brawn and Gustavson. Unlike the previous studies of paging, the behavior measured in this work is that of the influence of coding techniques on program performance in a paging system. This study used a specially modified IBM 7044 which provided each user (or program) with a virtual machine of 2^{21} words of 10 microsecond memory. A selected program was re-written using a particular programming practice. When this program version was executing on the machine, a non-disruptive hardware monitor collected data. A software analyser then used this data to produce summaries of total execution time; idle time; system execution overhead; page fetches and writes; as well as other additional run data. Execution of both single and multiple programs on the system was investigated. The study concluded that improving the programmer's coding style was advantageous to system performance regardless of what

page replacement policy was being used. The following programming practices were suggested as a means of improving program behavior in a paging environment. Arrays and matrices should be accessed in the physical order in which they are stored in memory. Large matrices should be reduced to smaller sub-matrices each residing on a single page. Data bases should be made as small as possible and ordered according to time of use. Program subroutines should be ordered in memory in a contiguous fashion according to time of use. Efforts should be made to eliminate or reduce the existence of global, randomly used, unordered variables. All of these techniques define a type of program organization beneficial to performance in a paging environment. However, they all are examples of attempting to fit the work to the system. The degree of awareness that the average programmer requires to use an operating system effectively is not intended to be a topic of this study. But the subject is of considerable merit and should be investigated in greater detail.

The final study to be described is that by Lewis and Shedler. This work is basically a statistical analysis of program paging behavior as is indicated by its title, "Empirically Derived Micromodels for Sequences of Page Exceptions." The main data for the study is a nine million long reference string traced from the execution of a 512

page program. From the study of transitions in the LRU stack distance strings (each reference to page K in the string is replaced by the number of distinct pages accessed since the last reference to page K) the program's paging behavior can be interpreted as consisting of two types. The program either tends to stay in only one page and references a few others or the program references pages almost at random. The interreference distribution of page exceptions is a convex function, initially decreasing rapidly, having a discontinuity at the interreference distance equal to the page size, followed by a long exponential tail. The distribution, plus correlation coefficients between the distances themselves has resulted in a two-state semi-Markov model of the sequence of page exceptions for the program. It is actually a micromodel, since the parameters for the model hold only for the period of time over which the page exception process was shown to be stationary. (The page exception process is stationary if the stochastic mechanism underlying the exception rate is independent of absolute time orgin.) This work is still continuin in an effort to obtain a more generalized model.

The orientation of the Lewis and Shedler study towards building a behavioral model of programs is significant. Except for this work, none of the others have attempted to place their results within a framework. The Working Set

Model of Program Behavior, discussed in the following chapter, attempts to provide organization by defining the relationship between memory utilization, process efficiency, and program paging level.

IV: The Working Set Model of Program Behavior

The traditional multiprogramming operating system [I3,I4] allowed a number of programs to be executed together by sharing the memory of the computer. Each program received a fixed amount of contiguous storage, enough to make the program's code entirely memory resident. When the program currently executing "blocked" for I/O, another program which was "ready", could be executed immediately. In this manner, the central processing unit could be run in a more continuous fashion and the I/O operations of the currently executing programs could be "overlapped" in time. Unlike this traditional approach to memory sharing, paging systems have the property of allocating not only CPU service but also memory to programs on the basis of demand. This enables these operating systems to accept and run "simultaneously" programs which taken together would require more real memory than is totally available at any given moment. However, this characteristic also makes paging system performance subject to the impact of program memory reference behavior. Severe performance degradation can occur as a result of "thrashing", a paging system phenomenon, usually characterized by high paging activity and low CPU utilization [D3]. The increased complexity of paging systems in comparison to the traditional multiprogramming approach has prompted the need for relating program behavior and

memory demand. One such attempt at answering this need is the Working Set Model of Program Behavior by P.J. Denning [D2].

The model has its basis in the notion of "locality", a concept of program behavior indicating that the most recently referenced pages are related to one another. It is derived both from the intuitive feeling that programmers tend to code sequential and looping structures, group data into content-related blocks, as well as from actual studies of program structure [B5,C1,C2,F1,F3,H1,H2,L1,P3,S2]. Denning has summarized locality in the following statements:

> " A program distributes its references non-uniformly over its pages, some pages being favoured over others...the density of references to a given page tends to change slowly in time...two reference string segments are highly correlated when the interval between them is small and tend to become uncorrelated as the interval between them becomes large " [D4,p.938]

In addition, the model also makes two basic assumptions concerning the page reference strings of programs.

The first assumption, that the reference strings are unending, allows the limit to be taken as time tends to infinity, and simplifies the notation. The error associated with the assumption is small since reference strings produced by program execution are very long. For example, assuming that the average 360/67 machine instruction takes 1.5 microseconds [R4] and references 1.5 pages (see results

Chapter VI), then only one second of execution produces a reference string 10^6 long.

The second assumption is that a reference string's underlying stochastic mechanism is stationary (independent of absolute time origin). Since stationarity can be expected only within substrings, an analysis applied over many such substrings (or "localities"), can determine only the average behavior of a program. The results may not indicate meaningfully the behavior of any given locality. However, the importance of this restriction does depend on the analysis and this study agrees with Denning's statement:

> " Since our primary interest is understanding the behavior of the working set model as an adaptive estimator for use in memory management, this limitation is not severe " [D6,p.131]

A very brief description of the working set properties follows. For further information concerning working set concepts, refer to references [D2]-[D6].

The behavior of a program can be studied in machine-independent terms by analysing its reference string $r_1 r_2 \ldots r_t \ldots r_K$. Each r_t represents a reference to a page of the program. That is, if $r_t = i$, then page i is the t^{th} reference where t is measured in process time and is discrete.

The working set, W(t,T), is defined as the set of

distinct pages referenced by a process in the time interval (t-T, t). The working set size, S(t,T), is defined as the number of pages in W(t,T). The average value of S(t,T) over the first K references of a program is given by

$$S_K(T) = \frac{1}{K} \sum_{t=1}^{K} S(t,T) \qquad (1)$$

If K is allowed to tend to infinity, then the <u>average working set size</u> of is defined. It is denoted as $\bar{S}(T)$ and is independent of time according to the two initial assumptions.

The re-entry or missing page rate represents the average rate at which pages enter the working set. Consider the following definition:

$$\Delta_t(T) = \begin{cases} 1 & \text{if } r_{t+1} \text{ is not in } W(t,T) \\ 0 & \text{otherwise} \end{cases} \qquad (2)$$

where $r_1, r_2, \ldots, r_K$ are the elements of the reference string. Then the average rate at which pages enter W(t,T) over the first K references is given by

$$M_K(T) = \frac{1}{K} \sum_{t=0}^{K-1} \Delta_t(T) \qquad (3)$$

If the limit of (3) is taken as K tends to infinity, the re-entry or <u>missing page rate</u> is defined:

$$\bar{M}(T) = \lim_{K \to \infty} \frac{1}{K} \sum_{t=0}^{K-1} \Delta_t(T) \qquad (4)$$

Again, this value is independent of t because of the initial

assumptions.

The properties of the working set are illustrated in Figure 1. The curve of $\overline{S}(T)$ is bounded by $\overline{S}(T)=T$ and N. The first limit can be reached only if $\Delta_t(T) = 1$ for all t; that is, each successive reference is to a page outside the working set. The second limit represents the total number of pages in the program.

Since no fewer pages can be referenced in longer intervals of time, S(t,T) is a monotonically non-decreasing function of T. Then by definition

$$W(t,2T) = W(t,T) \cup W(t-T,T) \tag{5}$$

But this implies

$$S(t,2T) \leq S(t,T) + S(t-T,T) \tag{6}$$

and since S(t,T) behaves on the average like S(t-T,T) (under the initial assumptions of the Working Set), then

$$S(t,2T) \leq 2S(t,T) \tag{7}$$

Thus, the curve of S(t,T) is non-positively accelerated, as well as monotonically non-decreasing. By the definition of the average working set size, the curve of $\overline{S}(T)$ also has the same characteristics and is shown in Figure 1.

By definition, the following holds

$$S(t+1,T+1) = S(t,T) + \Delta S \tag{8}$$

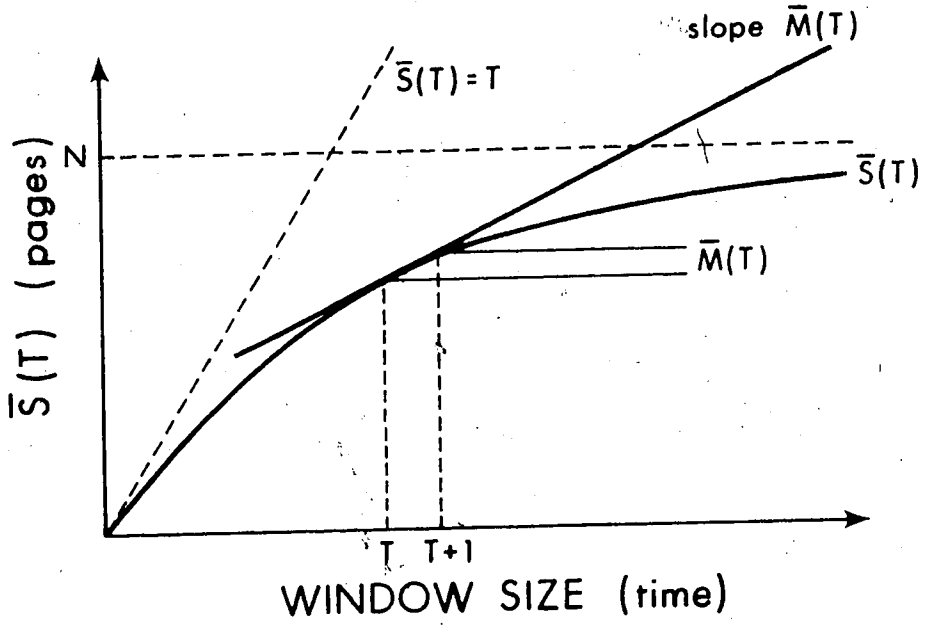


Fig. 1 : Curve of the Average Working Set Size, $\bar{S}(T)$

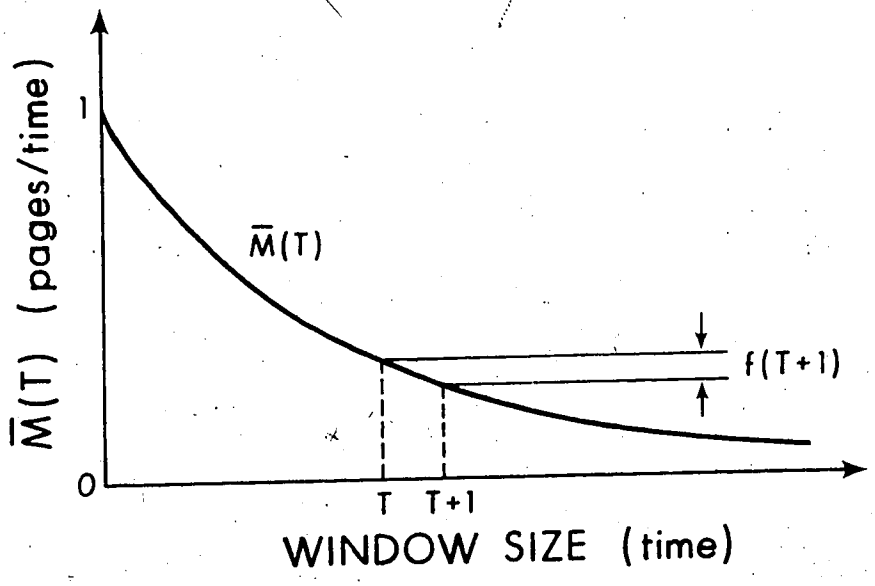


Fig. 2 : Curve of the Missing Page Rate, $\bar{M}(T)$

where $\triangle S$ represents the change in working set size. However, by taking the expected value of both sides of (8), the following is obtained

$$\bar{S}(T+1) = \bar{S}(T) + \overline{\triangle S} \qquad (9)$$

But, the expected change in the working set size is the missing page rate, $\bar{M}(T)$. Therefore, (9) can be re-written as

$$\bar{S}(T+1) = \bar{S}(T) + \bar{M}(T) \qquad (10)$$

where $\bar{M}(T)$ is monotonically non-increasing and non-negatively accelerated as indicated in Figure 2.

The direct result of these properties is that the larger the value of T or "window size" for a working set, the greater the number of program pages that will be memory resident and the lower the paging rate. The implied result for the operation of any paging system is the working set principle. It states that a program may be active and receive time slices on a processor only if its working set is loaded into main memory. That is, if there are n active programs, the main memory capacity is P pages, and the ith working set contains $S_i(t,T_i)$ pages, then the relation

$$S_1(t,T_1) + S_2(t,T_2) + \ldots + S_n(t,T_n) \leq P \qquad (11)$$

must be true with high probability at all times [D5]. The problem of thrashing should cease to occur in a paging environment, if the working set principle can be used to

ensure that main memory is not over-committed.

Two further results can be drawn from the properties of the working set. Memory dispatching algorithms should choose replacement pages only from those program working sets which have been "de-activated". These are the programs which have blocked for I/O or have been denied CPU service for scheduling or memory dispatching purposes. Secondly, the current working set size of a program can be used as an estimate of its future memory demand. Job schedulers can use this information to determine which tasks should be run and when. Memory dispatching algorithms can use predicted program page requirements to decide how many pages should be freed for re-allocation.

In the previous program behavior studies [C1,F1], the excessive paging which characterizes the research results is probably due to the violation of the working set principle. Similarly, the study by Freibergs, which advocates allocating at least four to six pages per program before execution, reflects the conclusions drawn from the working set properties. All the techniques expressed in [B5] point to the necessity of making the working set of a program more compact and less subject to a high missing page rate. In the study by Lewis and Shedler, one of the two types of program behavior was consistent with the notion of locality.

However, the referencing of pages in an apparently random manner does not conform to basic formulation of the working set. Perhaps, these random accesses represent transition periods between localities and are relatively short with respect to the entire reference string. Only further study of the rate of change in $S(t,T)$ can reveal the practical validity of $\overline{S}(T)$ as an estimator of a program's memory demands.

Unfortunately, the Working Set Model neglects any consideration of the internal behavior of the program. In particular, how do the individual pages contribute to the working set behavior of the program as a whole. One of the purposes of this study is to extend Denning's concepts by applying them at the page level. A second purpose is to study empirically the validity of these extensions to the Working Set Model. The following discussion presents the definitions for the extended model. The definitions themselves are close analogies to those in the original model.

A page's working set, $W(t,T,i)$, at time t is the set of references to page i in the time interval $(t-T,t)$. The working size of a page, $S(t,T,i)$ is defined as:

$$S(t,T,i) = \begin{cases} 1 & \text{if } W(t,T,i) \text{ is non-empty} \\ 0 & \text{otherwise} \end{cases} \qquad (12)$$

Therefore, the average value of $S(t,T,i)$ over the first K references is given by

$$S_K(T,i) = \frac{1}{K} \sum_{t=1}^{K} S(t,T,i) \qquad (13)$$

By letting K tend to infinity, $\bar{S}(T,i)$, the average working size of page i is defined.

The missing page rate is a measure of the average rate at which page i enters $W(t,T,i)$. Define the following:

$$\Delta_t(T,i) = \begin{cases} 1 & \text{if } r_{t+1} = i \text{ and } W(t,T,i) \text{ is empty} \\ 0 & \text{otherwise} \end{cases} \qquad (14)$$

where $r_1, r_2, \ldots, r_K$ are the elements of the reference string. Then the average rate that page i enters $W(t,T,i)$ over the first K references is

$$M_K(T,i) = \frac{1}{K} \sum_{t=0}^{K-1} \Delta_t(T,i) \qquad (15)$$

The missing page rate for page i is then defined:

$$\bar{M}(T,i) = \lim_{K \to \infty} \frac{1}{K} \sum_{t=0}^{K-1} \Delta_t(T,i) \qquad (16)$$

Note, that for a complete program the missing page rate with $T = 0$ is equal to 1, since the working set is always empty [D6]:

$$M_K(0) = \frac{1}{K} \sum_{t=0}^{K-1} \Delta_t(0) = 1 \qquad (17)$$

The definitions for the working set page behavior are consistent with (17) since the summation of the missing page rates over all pages of a program with T = 0 is also equal to 1. The following shows this result for an n page program:

$$\sum_{i=1}^{n} M_K(0,i) = \sum_{i=1}^{n} \{ \frac{1}{K} \sum_{t=0}^{K-1} \Delta_t(0,i) \}$$

$$= \sum_{i=1}^{n} (\text{probability of referencing page } i) = 1 \qquad (18)$$

Result (18) also defines the value of $\bar{M}(T,i)$ for $T = 0$. Since the probability of referencing page i is the same as the expected frequency of reference to page i, the following can be stated:

$$\bar{M}(0,i) = \text{probability of referencing page } i$$

$$= \lim_{K \to \infty} \frac{1}{K} \sum_{t=0}^{K-1} \Delta_t(0,i)$$

$$= e(i) \qquad (19)$$

where e(i) is the expected frequency of reference to page i. A more complete proof of the consistency of program and page working set properties is given in Appendix 1.

A page's interreference distribution is defined as

$$F(x,i) = \lim_{K \to \infty} \left[\frac{\text{no. of } x_i \text{ in } r_1 \ldots r_K \text{ with } x_i \leq x}{\text{no. of } x_i \text{ in } r_1 \ldots r_K} \right] \tag{20}$$

The variable x_i is called an interreference interval for page i. A page has an interreference interval $x_i = q$ when two successive references to the page are at times t and t+q. The interreference density function is then

$$f(x,i) = F(x,i) - F(x-1,i) \tag{21}$$

All of these definitions can be used together to prove the following properties concerning the working set behavior of individual pages. The proofs of these properties are given in Appendix 1. (The properties P1 through P9 are developed by Denning in [D6] and this labelling notation is continued here).

P10 : $e(i) = \bar{S}(1,i) \leq \ldots \leq \bar{S}(T,i) \leq \bar{S}(T+1,i) \leq 1$

P11 : $\bar{S}(T+1,i) - \bar{S}(T,i) = \bar{M}(T,i)$

P12 : $0 \leq \bar{M}(T+1,i) \leq \bar{M}(T,i) \leq \ldots \leq \bar{M}(0,i) = e(i)$

P13 : $\bar{M}(T,i) = e(i) - e(i)F(T,i) = \sum_{y>T} e(i) f(y,i)$

P14 : $\bar{M}(T+1,i) - \bar{M}(T,i) = -e(i) f(T+1,i)$

P15 : $\bar{S}(T,i) = \sum_{z=0}^{T-1} \bar{M}(z,i) = \sum_{z=0}^{T-1} (e(i) - e(i) F(z,i))$

$$= \sum_{z=0}^{T-1} \sum_{y>z} e(i)\, f(y,i)$$

P16 : $\bar{S}(T+1,i) + \bar{S}(T-1,i) \leq \bar{S}(T,i)$

P17 : $\lim_{T\to\infty} \bar{S}(T,i) = 1$

P18 : $\lim_{T\to\infty} \bar{M}(T,i) = 0$

P19 : $\lim_{T\to\infty} \sum_{i=1}^{n} \bar{S}(T,i) = \lim_{T\to\infty} \bar{S}(T)$

The properties of the Extended Working Set Model indicate that the average working size curve of a page, $\bar{S}(T,i)$, is monotonically non-decreasing and bounded above by $\bar{S}(T,i) = 1$. The missing page curve for each page reflects the "slope" of the $\bar{S}(T,i)$ curve, and therefore is monotonically non-increasing in T and non-negatively accelerated. The direct result of these properties is that as T, the "window-size", increases so does the probability that a page is within W(t,T) and that its corresponding missing page rate is lower. This result is not too surprising since the properties of the extended and original model formulations should be consistent with one another. Property 19 proves the intuitive notion that the behavior of the entire program is the same as the "summation" of the behavior of the individual pages. The most important aspect of the extended model is that individual page behavior can now be quantified and analysed in terms of working set

concepts. This is revealed in the results presented in Chapter VI.

There are several limitations on the Working Set Model, both practical and theoretical. The first one concerns the assumption that the stochastic mechanism underlying the reference strings is stationary. In practice, this is not the case since references to any page i tend not to be uniformly distributed over the K references of the string as assumed by the model. Instead, references to a page tend to exhibit a locality. The expected interreference distance of page i tends to be less than

$$\bar{x}(i) = \frac{1}{e(i)} \tag{22}$$

assumed by the model (where e(i) is the expected frequency of reference to page i). If g(i) is the number of references to page i, then this locality has an expected length of $\bar{u}(i)g(i)$ references where

$$\bar{u}(i) = \sum_{x \geq 0} x f(x,i) \tag{23}$$

is the actual expected interreference distance for page i. Therefore, by the definition for the calculation of $\bar{S}(T,i)$,

$$\bar{S}(T,i) = \lim_{T \to \infty} \sum_{z=0}^{T-1} \sum_{y>z} e(i) f(y,i)$$

$$= e(i) \lim_{T \to \infty} \sum_{z=0}^{T-1} z f(z,i)$$

$$= e(i)\ \bar{u}(i)\ , \qquad (24)$$

the lack of stationarity implies that as T tends to infinity the limit of $\bar{S}(T,i) < 1$ for most i. Because of the procedure for calculating $\bar{S}(T,i)$, the contribution of a page's individual locality to the program working set size is averaged over the entire reference string. For example, pages exhibiting a small locality and reference count contribute very little to a program's working set size, despite having a full page memory demand for some particular T and t. To compensate for this situation, the calculation of $\bar{S}(T,i)$ may be modified to

$$\bar{S}(T,i) = \sum_{z=0}^{T-1} \sum_{y>z}^{---} a(i)\, f(y,i) \qquad (25)$$

where

$$a(i) = \frac{1}{\bar{u}(i)} \qquad (26)$$

(Note that as $\bar{u}(i)$ tends to $\bar{x}(i)$, $a(i)$ tends to $e(i)$, $\bar{u}(i)g(i)$ tends to K, and the stochastic mechanism tends to become more stationary). What this modified calculation of $\bar{S}(T,i)$ accomplishes is to force the limit as T tends to infinity of $\bar{S}(T,i) = 1$ as predicted by the model. Although, this is strictly an a posteriori calculation, it fulfills the intuitive notion that as the window size T increases, the residency of an individual page should also increase. The increase continues up to the critical T where the window

size becomes longer than the actual expected interreference distance for the page. At this T, the page has an average working size equal to one.

A second limitation of the Working Set Model is that despite the extensions, it does not fully represent the internal processes of a program. In particular, the page-to-page reference behavior within the working set is neglected which eliminates the Model as a theoretical basis for studying program structure.

A final shortcoming is that no consideration is given to the I/O characteristics of programs. A more complete model of program behavior should include this facet, which has a very critical impact on any practical computer application.

Although the Working Set Model, plus extensions, does not represent complete program behavior, it does provide a good approximation of the relationship between program memory requirements and paging demand on a computer. The next chapter investigates the application of working set concepts for the adaptive control of an operating system's paging environment.

V: The Working Set Window Size

The most unifying characteristic of the Working Set Model is that all the properties of program behavior are based on one parameter, the window size, T. Understanding this parameter is vitally important in any practical application of working set concepts in a multiprogramming, paging environment. This chapter considers the system design problems of determining how to control the value of T, how to recognize the working set, and how to determine the size of T. Examples of actual implementations are given to illustrate the various applications of the window size parameter within computer systems.

The first problem that confronts the system designer is whether the working set is to be recognized at the "program" or "system" level. Although the working set model relates to individual program behavior, a systems approach assumes that the working sets of all programs executing on the computer can comprise a single "system" working set. The advantage of using this approach is that total memory demand and paging level can be controlled with relative simplicity. However, the flexibility of detecting memory requirements and paging activity of individual programs is removed. The knowledge of such parameters can be very significant to the operation of job schedulers and memory dispatching algorithms. The second

problem concerns the actual working set parameter itself.

When controlling the window size, it must be decided whether T be fixed or variable. In other words, should the operating system attempt to change the size of T to match current program "demands" or are these demands such that a fixed window size is suitable? If a variable T is used, should control of its value be applied globally or locally to the executing programs? That is, should T be the same for all programs or do programs vary in their behavior enough to warrant their own "local" window size?

A fixed window size approach assumes that if T is large enough, then at any given time t, a mix of programs executing on a computer will have relatively the same paging rate. The approach further assumes that this window size will be small enough to provide "acceptable" memory utilization. These assumptions are graphically indicated in Figure 3. The curves, A and B, represent the average total missing page rates of a mix of programs executing at two different values of t on the same computer. Note, that for window size T, both curves indicate essentially the same paging rate. The greatest advantage of this method for controlling T is the simplicity of implementation.

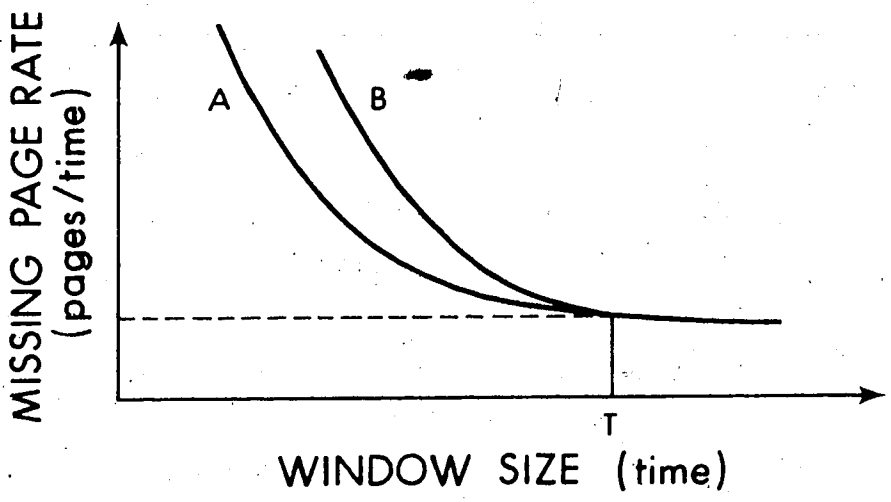


Fig. 3 : A Graphical Argument for a Fixed Window Size

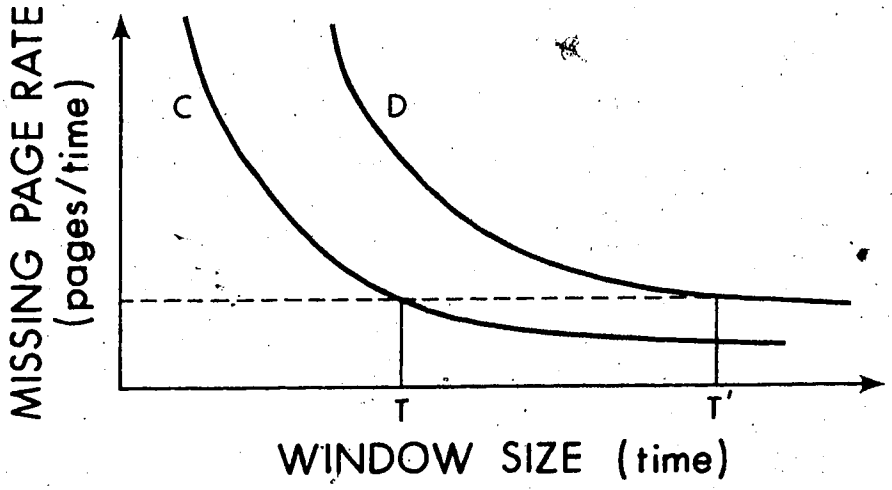


Fig. 4 : A Graphical Argument for Variable Window Sizes

The reason for choosing a variable window size approach is basically that the assumptions for using a fixed T are not valid. Instead, the assumptions for using a variable T are illustrated by Figure 4. Here, the curves C and D again represent the total average missing page rates of a mix of jobs executing at two different times t on the same computer. However, note that the curves do not converge as the window size increases. Rather, only for two quite different values of the window, for example T and T', are the missing page rates the same for paging loads C and D. The primary advantage of using this approach is that better control of the total program paging and hence memory demand is possible.

Whether or not an operating system uses global or local control of the variable window size depends first on how the working set is recognized. Since a "system" working set by definition does not differentiate between individual programs, it is possible to exercise only global control over the value of T. However, if program working sets are being recognized, the decision to use a global or local procedure is based on how much control of individual program behavior is desired.

Local control has the expense of increased complexity, but provides the operating system with the capability of

manipulating the execution of each job. This is shown graphically by Figure 5. Curves X, Y, and Z represent the missing page rates of three programs running under similar background load conditions. The example in this figure has been constructed so that the total page rate for the global window T, is equal to the sum of applying local windows, T', T'', and T'''.

$$\bar{M}_X(T)+\bar{M}_Y(T)+\bar{M}_Z(T) = \bar{M}_X(T')+\bar{M}_Y(T'')+\bar{M}_Z(T''')$$

$$a'+a''+a''' = b'+b''+b'''$$

In addition, $\bar{M}_X(T') = \bar{M}_Y(T'') = \bar{M}_Z(T''') = b' = b'' = b'''$. Despite the inherent simplicity of a global window, the local approach forces (in this example) all programs to page at the same rate regardless of their different $\bar{M}(T)$ curves. Thus, these programs would be treated equally in terms of elapsed running time. Obviously, a priority scheme could be employed to increase or decrease the individual program window size (hence the missing page rate) to respectively increase or decrease a program's "turn-around" time. However, the use of local window control may not be in the best interests of total system efficiency. One of the advantages of a global window parameter is that it effectively rewards compact, well-structured programs by reducing their elapsed running time.

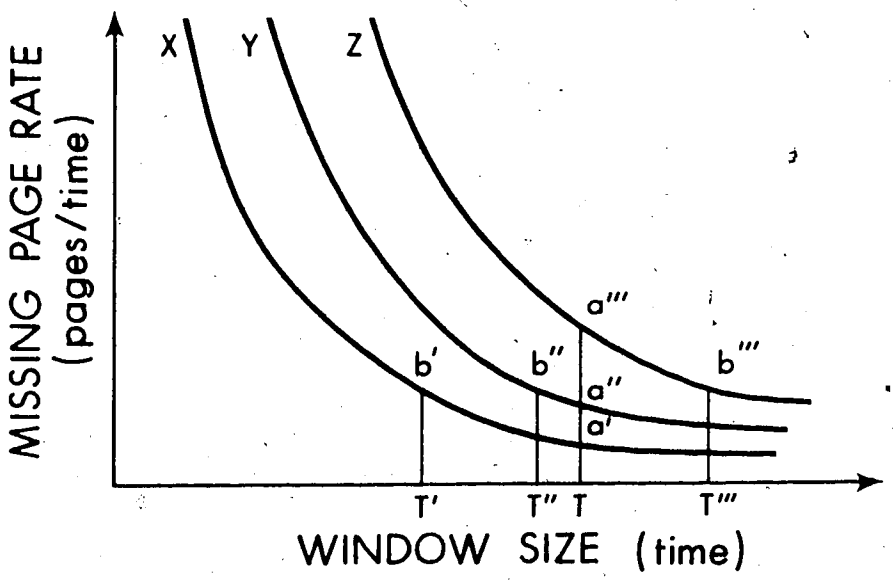


Fig. 5 : Local vs. Global Window Size Control

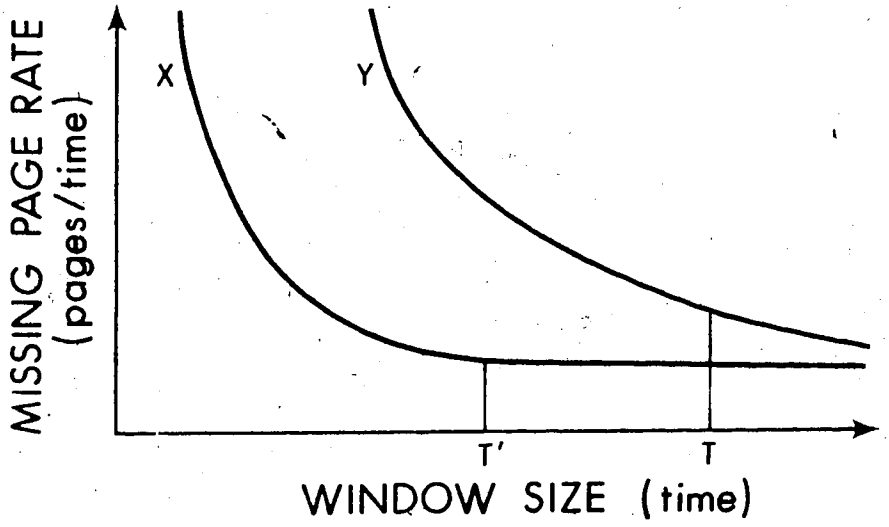


Fig. 6 : A Graphical Argument for Individual Page Windows

Executing such a program with a window size designed for more page avid programs results in the compact program having relatively more of its working set resident and hence a lower $\bar{M}(T)$. This is illustrated in Figure 5 by curves X and Z. Since a' is less than a''', program X has comparatively more of its working set in memory than program Z.

Manipulating the working set parameter, T, offers the designer a wide choice of operating system characteristics. Assuming there is a means of effectively implementing a variable window, the decision to use fixed or variable windows can be decided by analysing the basic assumptions of each approach. This study suggests that further research be done to determine the variation in the actual paging rate of job streams.

The decision of local versus global windows is not nearly so clear-cut since it appears to be a trade-off of flexibility against efficiency.

Whether a system designer choses to use a system or program approach for recognizing working sets depends on the "power" of the system desired and the amount of software and machine effort available to construct a working system. Although the program approach is inherently more complicated, it is not immediately obvious that it would be

less efficient. The systems approach appears to lack in sophistication what it gains in simplicity. However, it is one of the assets of the Working Set Model that the preceding discussion does allow a very concise description of options available to the system designer. These are indicated in Table I.

TABLE I: Design Options for Implementing the Working Set Parameter, T

Window Type	Window Control	Working Set Recognition	Implementations
Fixed	n/a	System	none
Fixed	n/a	Program	Working Set Dispatcher, TSOS
Variable	Global	System	none
Variable	Global	Program	MANIAC II
Variable	Local	Program	Balanced Core, TENEX

The next section of this chapter investigates the various ways in which T is implemented to recognize the working set of a program. In the discussion, several technical terms will be used and these are defined at this time [R5]. 'Any job within the running set is actively competing for the computer's resources and has already been allocated some important resources such as memory. Programs in this set maintain their working set of pages, even during

a page fault. If a job is in the ready set, it is waiting for system resources to be freed for allocation. Although programs in this set could execute, they are currently denied service for job scheduling or memory dispatching reasons. When a job is a member of the blocked set, it neither is allocated nor demands memory or CPU time. Typically, a job will move from the ready to the running set when allocated memory and CPU time and enter the blocked set by incurring an I/O wait (other than a page fault) during execution. A quantum is the amount of CPU time allocated to a job when it enters the running set. When a quantum assigned to a job expires, the process is demoted from the running set to the ready set, freeing resources for other jobs. Each quantum is usually divided into a number of equal time units called time slices.

The author is not aware of an operating system using a system-oriented fixed window. The "systems" approach assumes that the total pages in the working sets of the running set programs would comprise a "system" working set. All other pages within the computer would be considered available for re-allocation. Such a system could be implemented by having a timer associated with each hardware page in the machine. If a program's page is referenced, its associated timer is reset to zero. Otherwise, the timer is incremented until a

pre-determined count is reached. The time taken to reach this count would be the window size T for the system. Since it is the system working set which is being determined, these timers would run in real time rather than in individual program process time. The choice of T is critical, for if it were too short, the page replacement process would become FIFO-like in its operation.

There are at least two different ways of implementing an operating system using a fixed window applied to programs. The most uncomplicated are those which use the quantum or time slice completion interrupt hardware to assist in the determination of the program working sets. The TSOS software system, implemented on the RCA Spectra 70 series time-sharing computers, uses a window size of the form T=q where q is the quantum size [W1]. Once a process references a page, that page remains resident in memory until the quantum expires or the job enters the blocked set. The working set size is calculated at the end of q to determine future memory dispatching requirements. The Working Set Dispatcher [R5], is another software implementation of this type, except T=q/k where k is the number of time slices in a quantum. Therefore, only those pages referenced by the job during its latest time slice are ineligible for replacement. The Dispatcher calculates future

memory requirements at the end of each time slice. Both of these implementations measure T during individual program process time and use standard page replacement algorithms.

The second approach to fixed-valued program-applied window sizes is the method originally suggested by Denning [D2]. It requires the use of a shift register associated with each hardware page. The window, T, is divided into k bursts of CPU time. Every time a page is referenced, the left most bit of the register is turned "on". At the end of each burst of processing, the register is shifted right one bit position. The bit shifted off the right end is lost and an "off-bit" is introduced into the left most position. When all k bits of the shift register are "off", its associated page is no longer considered within the working set of the program and is eligible for replacement. Ideally, the window size T is measured only during the individual program execution time.

The remaining discussion centers on implementations of T using variable window sizes. This approach appears to have greater flexibility; but with increased system complexity.

To the best knowledge of the author, there are no implementations having variable window sizes, recognizing a

system working set, and using global parameter control. It can be argued that LRU algorithms implemented on current paging computers are an example of such an approach. Those pages with their "use" bit on, can be considered as members of the system working set. Typically, as the load on such a system increases, so does the amount of paging. This reflects the decrease in the length of time a page is within the working set. Such a result can be desirable since it indicates the system's attempt to satisfy all users. However, since these same systems make no explicit attempt to control the working set, continued reduction in the implicit window size results in programs paging excessively. When this occurs, the system is overloaded and thrashes. A working set approach is designed to prevent deterioration in machine performance by maintaining acceptable bounds on program loading. The number of users of the computer is not necessarily limited. Instead, total throughput is maintained at acceptable values with only the degradation in response time indicating the heavy demand for the computer's resources.

The next group of implementations considered are those that recognize program working sets and use a variable window size.

One such implementation is the Maniac II computer described by Morris [M3]. It varies the working set parameter in a global manner and the approach is hardware-oriented. Each hardware page on the Maniac II has an associated counter which is reset whenever that page is referenced. Each program in the running set has an associated binary state register having one bit for every real page in the machine. The presence of an "on-bit" in this register indicates that the corresponding hardware page belongs to that program. When the program is executing, the register is used to "gate" timing pulses to the counters. Since only those pages belonging to a particular program have their associated counters incremented, the working set is measured in process time. The number of timing pulses required to increment a counter from reset value to when the counter expires, is the value of T, the working set parameter. Pages with expired counters are no longer members of the working set and are eligible for replacement. The value of T can be varied globally by the system software. The criterion for changing T was not identified, but is probably similar to one of the following approaches.

There are at least two different implementations of systems using program working sets and variable window sizes controlled locally. Each uses a different decision process

for modifying T. The first is one described by Denning [D2,D6] and takes advantage of the property:

$$1 \geq \bar{M}(1) \geq \ldots \geq \bar{M}(T) \geq \bar{M}(T+1) \geq \ldots \geq 0$$

The property implies that as the window size of a working set increases, the paging rate decreases and vice versa. This "implicit" windowing can be accomplished by monitoring the paging rate of a program (or perhaps of an entire system as in Morris' case). By defining a range of paging rates for a job, the window sizes are also implicitly defined. An example of an actual implementation using this approach is the TENEX system [B6].

In the TENEX system, each program is required to fault at PAV, the defined average paging rate. When a fault occurs and a program is paging at a rate less than PAV, some of its pages are made eligible for removal. If a job is faulting at a rate higher than PAV, then a page fault results in that page being added to the job's working set. Page removal is accomplished via a standard LRU page replacement algorithm.

The second implementation of a variable window approach is described by Doherty [D8]. The time slice or implicit window size is varied under the assumption that a change in working set size of a job indicates a change in the working set. When the time slice is changed, it is varied inversely

in proportion to the process' working set size. Larger programs also receive their shorter time slices at intervals longer than for "normal" sized programs. This procedure is refered to as the "Principle of Balanced Core Time". Its basic intent is to reduce large program influence on system multiprogramming level, reward the behavior of compact working set programs, and reduce the dependence of scheduling as a function of program running time.

Having presented the different approaches for controlling the working set parameter and several means of recognizing the working sets of programs, how is the size of T chosen for a physical implementation? If T is to be fixed, what value should it be? If T is to vary, within what range of values should it be limited?

The original determination of the size of T was suggested by Denning [D2]. He felt that T should be chosen using the expected residency of a page. If X is the average interreference time for a page and Q is the average transfer time between main and secondary memory, then a curve representing page residency is shown in Figure 7. If X is less than T, then the page resides in memory 100% of the time. However, if $T<X\leq T+Q$, the page will be referenced during the period it is being transfered to the secondary

storage medium. Once the page reaches this device, it would immediately begin the return trip to main memory. The page is then resident in memory $T/(T+2Q)$% of the time. If $T+Q<X$, then the residency of a page is a decreasing function in X of the form $T/(X+Q)$. The page reappears in memory every $X+Q$ seconds and will be resident for T seconds before again being sent to secondary storage. Denning reasoned that T should be at least 2Q to reduce the intial drop from 100% residency to that of 50%. Assuming that Denning's approach is realistic, Table II indicates the average access times of IBM manufactured discs and drums and their approximate working set parameter.

TABLE II: I.B.M. Secondary Storage Device Access Times and the Corresponding Working Set Parameter Suggested by Denning for 50% Residency

DEVICE TYPE	2314	3330	2301	2305I	2305II
DATA RATE / SECOND	312Kb	806Kb	1.2Mb	3.0Mb	1.5Mb
AVERAGE ACCESS TIME	60ms	30ms	8.6ms	2.5ms	5.0ms
ON-LINE CAPACITY/PACK	29.7Mb	100Mb	4.1Mb	5.4Mb	11.25Mb
ROTATIONAL PERIOD	25ms	16.7ms	17.1ms	10ms	10ms
SUGGESTED WINDOW SIZE	120ms	60ms	17.2ms	5.0ms	10.0ms

Kb = Kilobytes, Mb = Megabytes, ms = milliseconds

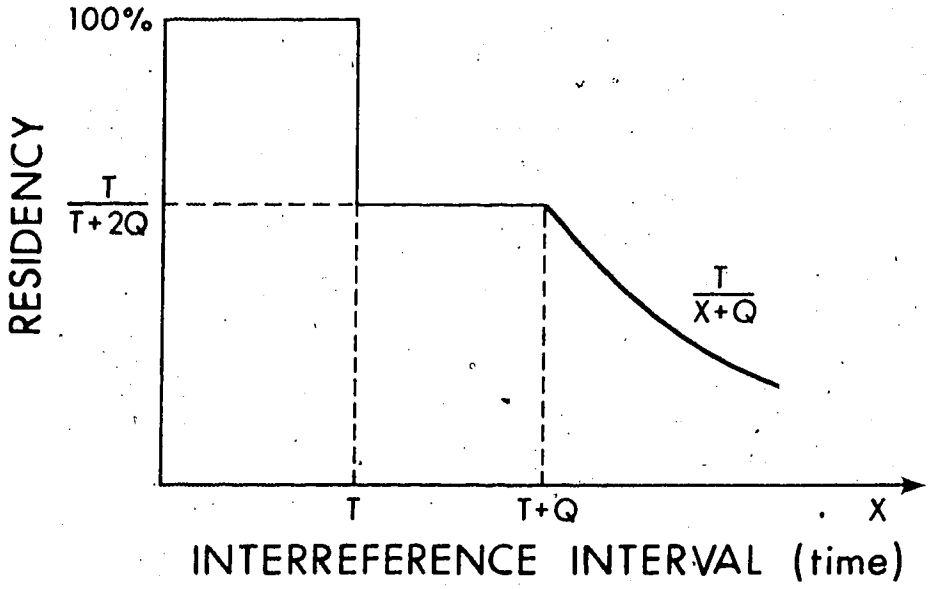


Fig. 7 : Denning's Graph of Page Residency

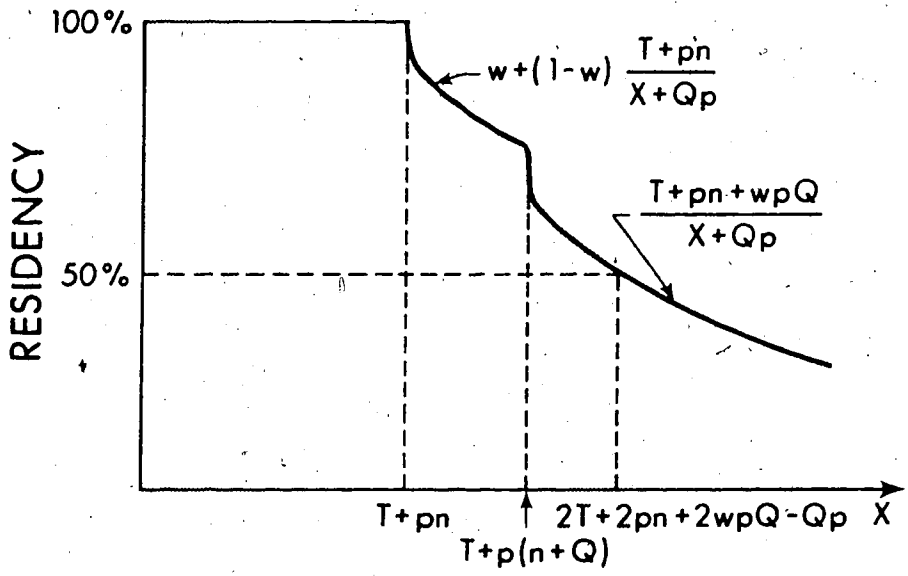


Fig. 8 : Prieve's Residency of an "Average" Page

Prieve [P4] has indicated several inaccuracies in the residency argument for determining T. Basically, Denning's argument assumes that all expired pages are written to the secondary device, that freed pages are immediately reassigned, and that Q is the same in process time as real time. Since Prieve suggests that these assumptions are generally not true, he has presented a more "complete" curve of page residency, as shown in Figure 8. The parameters in the curve are defined as

p: the percentage of real time devoted to executing the process

w: the percentage of released pages which need to be written

n: the number of seconds after a page is available for reassignment that it is reassigned. The value of n can be calculated as

$$n = \frac{\text{no. of pages in freelist}}{\text{no. of pages requested/second}}$$

From Figure 8, it can be seen that if all pages are written to the secondary device, the residency of a page will be 100% for $X \leq T+p(n+Q)$. The curve of residency also becomes $(T+p(n+Q))/(X+Qp)$. If no pages are written to the secondary device, the curve of residency is simply $(T+pn)/(X+Qp)$. Unlike the conclusion reached by Denning concerning T, Prieve states:

> " ...to decrease the residency from 100% to 50% takes almost the same increase in the interreference interval as required to cause any decrease from 100 percent residency for reasonable values of p, Q, n, and w. Therefore,

> we must conclude that the residency arguments do not yield much information on the selection of T. " [P4, p.620]

Unfortunately, Prieve does not provide any examples of reasonable values of p,Q,n, or w. Therefore, the following values for these variables have been determined:

p: 3%, during the afternoon, the MTS operating system at the University of Alberta usually averages approximately 33 "simultaneous" users. This value for p assumes that CPU use is uniformly distributed over all users.

Q: 10ms., the approximate access time of an IBM 2301 drum.

n: 1.0 sec., the MTS operating system at the University of Alberta attempts to provide 30 free pages to satisfy the request for 30 pages/sec which is the typical paging rate.

w: 90%, [F3] indicates that one out of eleven pages is not "changed".

Substituting into Prieve's expression, $2T+2pn+2wpQ-pQ$, for the 50% residency level, the result value for x is:

$$2T+2(.03)(1.0)+2(.9)(.03)(.01)-(.03)(.01)$$

$$= 2T + .06024 \text{ seconds}$$

Since Denning suggests only T=2Q or .020 seconds as the value X for a page to be 50% resident, there is at least a factor of five difference between the two models of residency. It can perhaps be seen why Prieve reached his conclusion. However, this new model of page residency may not be entirely accurate either, since it may be that n essentially equals zero in some systems. That is, once a page is successfully written to the secondary storage unit,

it is immediately released by the system. Since the 2pn contribution is very significant in Prieve's expression, its removal once again makes page residency a vital factor in determining T. The expression for X becomes 2T+2wpQ-pQ. By substituting the previously determined values for the variables w,p, and Q, the value of X is:

$$2T+2(.90)(.03)(.01)-(.03)(.01)$$

$$=2T+.00024 \text{ seconds}$$

If T is [illegible] 2Q as suggested by Denning, then the 50% level usi[illegible]ve expression is approximately 4Q or 40 ms. It [illegible]sting to note that PAV for the TENEX system is 67 ms. or approximately 4Q (2 drum revolutions for that system).

Obviously, if the 2pn is a significant factor, the page residency argument for determining T is invalid. For example, assume that once a job enters the running set, it remains there using a relatively high percentage of the real CPU time (although its average CPU use could be the smaller p value as calculated in the original example). On the MTS operating system, the running set approximately averages 10 jobs. If p is set equal to 10%, then the 50% residency level occurs when X=2T+.2008 sec. Comparing this value to the original one suggested by Denning (.020 seconds) at least an order of magnitude of difference is found. This result

certainly justifies Prieve's conclusion (the 100% page residency level would occur with X= T+pn = T+.100 which is greater than the largest window considered in the present study!). Unfortunately, which representation of p is the "correct" one (in the examples given), is beyond the scope of the study. Perhaps both are correct, each indicating a different loading situation on the computer. The larger value for p may indicate a lower competition for the CPU whereas a smaller p may be typical of a heavily loaded system. Further research will have to be done to determine more accurately how the factors represented by p (and for that matter n) influence the calculation of page residency based on page interreference distances. However, T must still be determined so that software schedulers and dispatchers can make meaningful assignments for memory allocation.

Chapter VI will present some reasonable values for T, given the considerations already presented. The means for deciding the values are empirical; by studying the missing page rate and working set size of programs with respect to varying window sizes.

Chapter VI also presents an investigation into whether pages can be divided into "paging" and "non-paging" types

and whether the frequency of reference to a page relates to its "page" window size. These investigations will be used to decide whether adaptive window size control can be extended to the individual page level and can this control be exercised through a page's frequency of reference. In particular, if in Figure 6 the curves X and Y represent the missing page rates of two pages, then X could have a window T' which would be just as effective as T. Then, if X does leave the working set, memory is released T-T' seconds sooner than if T were the window. The results of an examination of the situation depicted in Figure 6 are also presented in the chapter.

Chapter VI: The Empirical Study of Working Sets

What work has been done on the empirical study of program working set behavior? As previously mentioned in Chapter V, there are several papers discussing the implementation of working set concepts; yet, only the work by J. Rodriguez-Rosell [R4,R5] gives information on actual process working set characteristics. One of the purposes of his research was to determine a fixed window size (or time slice length) and the number of these windows that should form a quantum. Data was gathered on dynamic program behavior by doing a fully interpretive simulation of the execution of selected problem programs on an IBM 360/67. The programs used were IBM/360 software; a FORTRAN compiler, a PL/1 compiler and an assembler. The simulation procedure enabled most of the page references to be gathered. An additional program was employed to examine the I/O channel programs for their page references. From the results of his study, Rodriguez-Rosell has indicated a value of 30 milliseconds for T and that 35 such windows compose a quantum. Using these values, his operating system (a modified version of CP-67) was stable with a mean multiprogramming level of three [R4].

In this study, program reference strings are collected

using a modified version of the *TALLY program [M4] available on the MTS operating system. The program employs an instruction-by-instruction interpretive execution of problem programs to gather page reference strings. The MTS operating system at the University of Alberta loads all the code for the user problem programs into virtual segment five of the IBM 360/67 computer. Only the page references to segment five are used in the results of this study. Other virtual segments are used for re-entrant system routines, system tables, and for program I/O considerations. Since system code is available to all programs executing and I/O processes vary from machine to machine, these pages were not included in a problem program's calculated average working set size and missing page rate curves. However, in order to retain realistic system timing considerations, references to these pages are involved in the working set calculations.

The reference string analysis was based on an algorithm suggested by Denning [D3]. The actual algorithm used is given in Appendix 2. Also in that appendix, is another algorithm, modified according to the properties of the extensions to the Working Set Model to produce the average working sizes and missing page rates for individual pages.

Both algorithms used differ from the one suggested by

Denning. He set the time for the first reference to a given page equal to L+1, one more than the maximum window size studied. Then, the working set calculation includes this initial L+1 distance as part of a page's interreference instruction behavior. Thus, the first occurence of any page in the reference string effectively produces a page fault. In the present study, the interreference distance for the first reference to a page is set to one. That is, the first occurence of a page does not constitute a page fault. The algorithms are inherently similar, but express different approaches to the problem of a practical calculation intended for unending reference strings. To illustrate the difference, consider the following example. If a page is referenced only once in a total of K program references, Denning's algorithm calculates the page's average working size to be T/K. The algorithms used in Appendix 2 calculate this same page's average working size to be 1/K. Since in theory the limit of T/K is zero, this author feels that the 1/K value is more representative. However, to convert to Denning's algorithm, simply add n*(T/K) to any total program working set size produced in this study (where n is the number of different pages referenced by the program).

The empirical results gathered here are divided into two parts. The first part concerns the data gathered about

program behavior. The second section presents the results obtained by applying working set concepts to the behavior of individual pages within a program.

TABLE III. Description of the 15 Program Runs

PROGRAM NUMBER	SOURCE LANG.	PROBLEM TYPE	PROBLEM NAME	REMARKS
1	ASSEMBLER	Compiler	*FORTG	
2	LISP	String Processing		Interpreted
3	SNOBOL4	String Processing	ELIZA	Interpreted
4	FORTRAN	Simulation		
5	ALGOLW	Chess	WITA	
6	PL360	Compiler	*ALGOLW	Terminal I/O Bound
7	ALGOLW	Parser		Terminal I/O Bound
8	PL360	Compiler	*ALGOLW	
9	ASSEMBLER	Compiler	*FORTG	
10	ASSEMBLER	Compiler	*PLI	
11	ALGOLW	Chess	WITA	
12	ASSEMBLER	Library Search		
13	ASSEMBLER	Compiler	*PLI	
14	FORTRAN	Matrix Processing		
15	FORTRAN	Chess	COKO	CPU Bound

TABLE IV: Numerical Description of the 15 Program Runs

PROGRAM NUMBER	INSTRUCTIONS SIMULATED	REFERENCES ANALYSED	% REFERENCES IN USER REGION
1	952,256	1,000,000	96.3
2	1,048,938	1,000,000	97.5
3	1,065,408	1,000,000	95.9
4	1,049,004	1,000,000	97.3
5	1,060,399	1,000,000	98.9
6	1,064,466	1,000,000	23.3
7	972,048	1,000,000	41.9
8	1,054,456	1,000,000	92.5
9	1,055,094	1,000,000	100.0
10	1,059,717	1,000,000	31.0
11	1,059,949	1,000,000	92.0
12	1,013,678	1,500,000	63.9
13	1,059,101	1,500,000	60.3
14	1,049,820	1,500,000	97.2
15	1.052,458	1,500,000	83.7

Program Behavior

Fifteen different reference strings, each representing the instruction-by-instruction simulation of approximately one million instructions form the basic data investigated. Eleven unique programs were used to produce these strings. They vary in problem type from compilers to chess-playing programs. Table III provides a concise representation of all the reference strings employed in this analysis. The source language column indicates the language the problem program was written in, while the problem type represents its particular application. To distinguish programs having the same application, the problem name column identifies the particular program used when gathering the data. The remarks column indicates any special comments concerning the particular reference string. Six of these strings originated from compilers. The source language for all these compilers is 360/assembler except for the *ALGOLW compiler which is written in PL360, a macro language using 360/assembler. Two different chess programs were run; one written in ALGOLW and the other in FORTRAN. WITA, the Algol program, was run twice under different initial conditions; but had very consistent working set characteristics. Both the string processing programs were run using an interpreter. Since the source language used by these

interpreters is probably assembler, the characteristics of languages, LISP and SNOBOL, may have been obscured. An extension to this study would be to gather and analyse data using compiled LISP or SNOBOL code. The remaining four programs were provided by Computing Science students at the University of Alberta.

The first two columns of Table IV indicate the total number of instructions simulated for each program execution and the total number of references used in the actual working set analysis. Eleven strings were analysed for one million references while the remaining four were investigated for 1,500,000 references. This deliberate inconsistency ensures that there are no gross discrepancies in the analysis between strings of the two different lengths.

The final column, percentage of references in the user region, is a measure of the number of references in the data string which actually occur within the problem program (in segment 5). When the data was collected, the entry point address of the problem program was subtracted from each reference. Thus, only those references which are non-negative are attributed to the problem program being analysed. All other references are assumed to be within the

operating system's service code and I/O buffers. Often, this procedure meant that the problem program was link-edited to ensure that all its routines were loaded beyond the entry point address. The non-program page references are included in the analysis so that interreference distances between pages of the problem program reflect the program's use of the service code. Generally, those routines most used by a problem program involve input/output activity. This is indicated by the correspondence between the low percentage of problem program references and the high terminal input/output activity for programs 6 and 7 (see Table VIII). Over all the programs studied, the average percentage of references to pages not in segment five is 21.9%. In the MTS operating system, all service code and I/O buffers are pageable. Because of the initial limitation placed on this study, the contribution of these pages to the paging rate of the system has not been investigated. However, since an average of one out of five references is to one of these system or I/O pages, a more complete study of program behavior should determine the paging activity involving this memory. Two other programs, the PL/1 compiler and the library search program, have lower reference counts in the problem program region. Since both programs have low terminal I/O activity, the results suggest that both programs probably make extensive use of disc

operations during execution.

TABLE V: Working Set Characteristics of the 15 Program Runs

PROGRAM NUMBER	TOTAL PAGES REFERENCED	MAXIMUM WORKING SET SIZE	MINIMUM PAGING RATE	WINDOW SIZE AT MIN RATE
1	20	10.4	.62	100
2	23	11.8	.89	50
3	58	47.8	6.80	100
4	10	10.0	.086	60
5	40	30.3	1.20	100
6	14	12.7	.38	75
7	15	11.0	.48	75
8	34	16.5	1.15	95
9	12	11.7	.24	100
10	51	11.4	1.48	85
11	35	26.3	1.00	100
12	12	7.66	.00	40
13	49	13.9	2.10	100
14	8	5.21	.35	65
15	40	18.6	5.20	100

Maximum working set size at 100,000 references
Minimum paging rate in 10^{-5} pages / references
Window size measured in 10^{3} page references

TABLE VI: "STEADY-STATE" Working Set Characteristics and the Exponential Fit Parameters A and B for the 15 Program Runs

PROGRAM NUMBER	WINDOW SIZE AT START OF STEADY-STATE BEHAVIOR	WORKING SET SIZE FOR THIS WINDOW	PAGING RATE FOR THIS WINDOW	A	B
1	50	10.1	.82	.065	-0.29
2	25	11.2	1.01	.020	-0.11
3	70	45.5	10.8	.860	-0.28
4	40	9.70	.67	.22	-0.68
5	60	29.8	2.93	.70	-0.46
6	45	12.5	.67	.12	-0.44
7	60	10.9	1.08	.11	-0.38
8	50	16.5	1.45	.060	-0.20
9	70	11.6	.84	.24	-0.44
10	60	10.8	1.98	.21	-0.33
11	70	26.1	3.10	.66	-0.43
12	20	7.62	.43	.021	-1.60
13	70	13.2	2.53	.19	0.26
14	65	5.11	.35	.031	-0.26
15	40	15.3	7.27	.17	-0.14

Paging rates measured in 10^{-5} pages / references
Window size measured in 10^{3} page references
Exponential parameter A measured in 10^{-3} references
Exponential parameter B measured in 10^{-4} references

Tables V and VI present the results determined by applying a working set analysis to the reference strings. The first data column of Table V indicates the number of different pages referenced by the individual problem program. A total of 421 pages is referenced over the fifteen program executions. This represents an average of 28 pages per program. The second column shows the maximum average working set size of the programs. The value is maximum since it is calculated at the largest window size used in the study (100,000 references). The average working set size of a program will increase with window size as long as the average paging rate is not zero. Only program 12 reached its maximum average working set size with a window of less than 100,000 references. The average value of the maximum average working set size for all programs is 16.4 pages. Thus, the working set size calculations indicate that these programs on the average require only 58.5% of their referenced pages to be resident at any given time. The next two columns of Table V indicate the minimum missing page rate of a program and the window size at which this rate occurred (given that the maximum window size was 100,000 references The average minimum missing page rate for all programs was 1.46×10^{-5} pages per reference. This minimum rate occurred at an average window size of 83,000 references. If it is assumed that the mean time to execute

a 360/67 machine code instruction is 1.5 microseconds, then this missing page rate is approximately 10 pages per second. If it is further assumed that every page is written to a secondary device when its window size expires, then the total average paging rate is 20 pages per second per program. However, this is a maximum paging rate. Since expired pages of a program can be referenced again, they can be "re-claimed" before being removed from memory. It should be emphasized that these calculations are based on the average program for this study. The results may or may not be representative of the typical program executing on any given computer at any given moment in time.

The first three data columns of Table VI measure the characteristics of an apparent "steady-state" behavior for each program. First, inspect Figures 9 to 23, the average working set curves. An outstanding characteristic of almost all $\overline{S}(T)$ curves is the occurrence of a "knee"; and of the $\overline{M}(T)$ curves an "elbow". This knee (or elbow) separates window sizes where the change in missing page rate is rapid from those windows where the rate appears to be practically constant. Although this flattening of the missing page rate is visually striking, the magnitude of the actual change in rate in this region can vary considerably between programs. Thus, the window size chosen to indicate the onset of

steady-state behavior was visually determined and indicated in column one. Column two gives the average working set size of the problem program at this window size and the next column indicates the corresponding missing page rate. The average window size for the start of this behavior was determined as 53,000 references. The corresponding average missing page rate was 2.39×10^{-5} pages per reference with an average value of the average working set size of 15.7 pages. In comparison to results shown in Table V, the average paging rate has increased approximately 65% whereas the memory utilization has been reduced by less than 5%. This implies that a program's average working set size is quite invariant in the window size range of 50,000 to 100,000 references. Yet, higher paging rates can be the result for smaller window sizes within this range without any concomitant memory saving.

The columns marked A and B represent the coefficients found when an exponential fit was applied to the missing page rate curves. The least-square method was used to obtain a straight line through the log-transformed missing page rate data points. The the "missing page rate" intercept gave the value of ln. A and B was simply the slope of the line. This fit has the form :

$$Ae^{-Bt}$$

where A is expressed in units of 10^{-3} references and B in units of 10^{-4} references. The coefficient A tends to be an indicator of the magnitude of the paging characterizing a program whereas B indicates the compactness of a program's working set. A relatively large value of B indicates a steeper "slope" on the fitted exponential curve. The steeper this slope, the more likely the interreference distance between pages of its working set is shorter, than for a program having a smaller B coefficient. In general, as both the value of A increases and the value of B decreases, the more likely the program is to have a higher missing page rate. Having either one of the above true, is not by itself, a completely valid indicator of a program's relative paging rate.

The fit of the exponential curve to the data was quite good for window sizes greater than 20,000 references. This is graphically shown by the Figures 9 to 23. However, for window sizes less than 20,000 references, the actual missing page rate tended to rise far more steeply than that of the fitted exponential curve. Perhaps a better fit for the entire data curve may be obtained by considering the missing page rate curves to be conic.

Exponential fits were applied to these curves to compare the results of this study to that of Rodriguez-Rosell. The average values obtained for A and B in that investigation was $.45 \times 10^{-3}$ instructions and $-.40 \times 10^{-4}$ instructions, respectively. From the results indicated in Table VIII, the average number of references per instruction is 1.58. Therefore, converting Rodriguez-Rosell's coefficients to the same units used in this study gives a value of A of $.28 \times 10^{-3}$ references and B equal to $-.25 \times 10^{-4}$ references. The average value of A and B determined over the fifteen program runs is $.23 \times 10^{-3}$ references and $-.31 \times 10^{-4}$ references, respectively. The agreement between the two sets of values is quite close. Since Rodriguez-Rosell considered the pages a program used for I/O, including channel programs, at least some of the discrepancy between the coefficient values can be explained. The parameter, A, which tends to indicate paging magnitude, would be larger in his study. Rodriguez-Rosell may have included the influence of channel program execution time on the interreference distance. If so, his B coefficient would tend to be larger than that determined in the thesis. It is recognized that some of the discrepancy between the two sets of coefficients may be due simply to differences in the source data programs and algorithms used to calculate the working set curves.

TABLE VII: Average Expontial Fit Parameters of the Expected Missing Page Rates For Different Program Types

PROBLEM PROGRAM TYPE	AVERAGE A	AVERAGE B
Non-Student	.28	-0.31
Student	.09	-0.44
Chess Playing	.44	-0.33
String Processing	.43	-0.27
FORTRAN	.11	-0.22
ASSEMBLER	.13	-0.32
Compiler	.14	-0.31
ALL PROGRAMS	.23	-0.31

Exponential parameter A measured in 10^{-3} references
Exponential parameter B measured in 10^{-4} references

Table VII indicates the average values of A and B determined over the various problem types and source language groupings of the programs. The group of student programs has the smallest value of A and the largest value of B. This agrees with the intuitive belief that most student programs are relatively small in comparison to non-student programs. Despite the fact that the chess programs have the highest average A value, the size of B was only slightly larger than average. These programs tend to access more pages outside the established working set, but accessing patterns within a working set are more compact

than the average program. The same working set behavior cannot be said of the string processing programs since their interreference distances tended to be a little longer than the average program. Coupled with a larger than average A coefficients, the result is a highly paged program. The FORTRAN programs have a low value for A but also the lowest value for B. The initial indication of relatively low program paging activity is offset by the working set tending to be much less compact than the average program. The assembler and compiler programs have almost identical values for A and B. Since six of the seven assembler programs studied are also compilers, the result is not surprising. An average value of A indicates that these programs tend to have lower paging activity than average.

Rodriguez-Rosell found that the value of B was practically the same for all programs. He also determined that the A values for assembler and FORTRAN programs (converted to units of this study) were $.28 \times 10^{-3}$ references and $.19 \times 10^{-3}$ references, respectively. The results presented by the study have indicated a variety of values for B, but that A is essentially the same for both assembler and FORTRAN programs (approximately $.135 \times 10^{-3}$ references). Both these results seem to imply that differences between individual programs of a particular type have a greater

influence on A and B than the unifying characteristics of similar source code. Because of the small sample size, the success of applying working set concepts to classify programs according to functional characteristics (compilers, string processing programs, student programs) cannot be completely ascertained from the results presented here. Only further empirical research will reveal whether programs can be functionally differentiated by their working set characteristics. Regardless of the differences between programs, working set concepts do provide a way of quantifying the [illegible] program. It may be possible to obtain a distribution of programs running on a computer according to A's and B's. This presents a reasonably simple and meaningful approach to describing an input job stream. If an operating system utilizes a working set approach, such a program load description would provide information for tuning the computer for more efficient operation.

Table VIII gives a detailed account of the instruction types which were executed during each of the fifteen program data runs. The first four data columns indicate for each program the percentage of type G, P, R, and B instructions which were executed. The G type instructions are those requiring a memory fetch, the P type are those requiring a memory store, the R type are register-to-register

operations, while the last type indicate branching instructions. The fifth column indicates the percentage of instructions which reference memory (sum of P and G types).

TABLE VIII: Instruction Type Execution and I/O Characteristics for the 15 Program Runs

PROGRAM NUMBER	%INSTRUCTION TYPES % G	% P	% R	% B	RATIO DATA/ INSTR WORDS	SVC's	TERMINAL I/O's
1	39.23	14.47	16.12	30.12	53.70	65	0
2	32.35	10.61	21.53	35.51	42.96	33	-
3	46.65	16.01	11.28	26.06	62.66	164	19
4	60.61	8.75	9.52	20.89	69.36	14	5
5	46.34	16.47	19.52	17.67	62.81	56	17
6	39.08	13.64	16.79	30.49	52.72	1804	240
7	40.79	13.29	13.99	31.93	54.08	1519	250
8	39.81	10.84	18.25	31.10	50.65	105	0
9	42.43	10.12	29.10	18.35	52.55	2	0
10	41.89	11.63	14.40	32.08	53.52	639	-
11	46.25	16.73	19.97	17.05	62.98	37	3
12	38.68	10.13	21.57	29.62	48.81	352	4
13	40.93	14.23	12.23	32.61	55.16	531	9
14	63.37	9.93	10.21	16.49	73.30	13	5
15	56.88	14.39	12.27	16.46	71.27	0	0

Is there any relationship between the type of instructions a program executes and its working se

behavior? There appears to be little relationship between a program having a lower percentage of memory access instructions and having a higher B coefficient. Those programs having a higher percentage of R type instructions do appear to have a higher B value, yet the converse does not necessarily hold. The characteristic of a lower percentage of B type instructions again tends to indicate a higher B coefficient value. However, there does not seem to be any relationship between instruction types and the actual minimum missing page rate determined for a program. It should be noted that these conclusions are not drawn from rigid statistical calculations but rather from observation of the collected data. Further rigorous investigation should be applied in this area to settle completely questions of instruction type execution and program properties. However, at least at this level of analysis, it appears that there is little to be gained from using instruction execution types for adaptive control purposes.

As indicated earlier, the working set model of program behavior does not include any input/output considerations. However, in any practical investigation of program behavior, this aspect must be at least considered. Would it be sensible to suggest a window size, T, for a paging system when the average time between I/O operations causing a

program to block is T', where T' is much less than T? To investigate this question, two types of information were gathered in this study. The first type of information collected is indicated by the column marked SVC's in Table VIII. The final column in that table, the number of terminal input/output operations, is the second type of I/O information accumulated.

An SVC is an instruction which when executed initiates a supervisory controlled operation. At the time of this study, it typically took two SVC's to initiate an I/O operation (a start I/O and a wait for I/O completion). Unfortunately, SVC's could also be executed for non-I/O functions. In addition, not all I/O SVC's executed would cause the program to block. Because of system internal I/O buffering, control may be returned to the program without its execution being interrupted. These factors all tend to make the average blocking distance between I/O operations longer in process time than indicated by the SVC count. What is required is a more complete analysis of the relationship between I/O activity and SVC execution. It is suggested that either an a priori distribution of SVC types executed be determined or that during reference string accumulation the type of SVC's executed be gathered as well. In either case, the impact of I/O buffering on SVC execution

must also be obtained. It would have to be determined whether or not this information could be gathered during program simulation. Depending on the operating system, real time I/O buffer behavior of the problem program may or may not be possible under software monitoring.

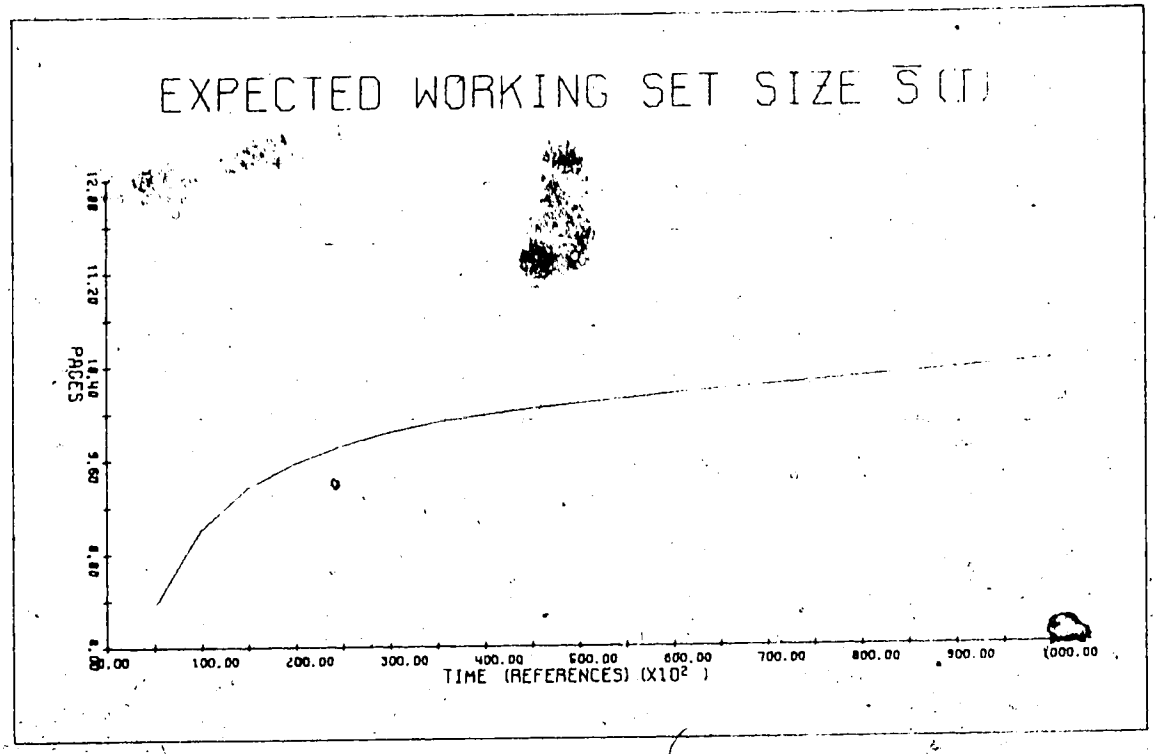
The count of the number of terminal I/O operations was also gathered to obtain information on the frequency at which programs block for I/O. If a request is made for terminal input, the program normally enters a wait state. Depending on the operating system, the program's pages are either released immediately for system use or are gradually released according to the standard aging algorithm (as in MTS). Terminal output may or may not result in the program blocking. If output buffering is done to the terminal, then this count of terminal I/O operation will not reflect accurately the program's blocking behavior. The accuracy of this count is further reduced by programs which do disc I/O. This activity may cause blocking of program execution in addition to that at the terminal. Again, from the data collected, it is not possible to determine accurately a program's I/O blocking behavior. Only further study, as previously suggested, will provide more reliable results on program input/output characteristics.

What conclusions can be drawn from the collected I/O data given the limitations presented? If it is assumed that all SVC's initiate an I/O operation, then the average number of I/O's per program is approximately 350 with one occurring every 3000 instructions. However, if only terminal input/output's are considered the number of I/O operations drops to an average of 37 per million program references. This implies that the average number of instructions executed between an input or output has increased to 27,000 instructions or approximately 43,000 references. If only those programs having a relatively small percentage of instructions executed within service routines are considered, the average number of terminal I/O operations drops to only six. This gives an average of 167,000 instructions or 250,000 references between I/O operations. However, for those programs which have higher percentage of references outside the segment five code, the average number of I/O operations at the terminal increases to 126 giving an average of 8,000 instructions or 13,000 references between input/output's. It should be repeated that the influence of disc I/O and output buffering to the terminal have not been considered in these calculations. They should be slight on "non-I/O" programs. Assuming no output buffering to the terminal and that each I/O activity results in a program entering the wait state, the "I/O-bound" program has an I/O

blocking distance much smaller than any window size that would be suggested for an operating system. This is so, even neglecting the influence of disc input/output. Before output buffering would be a meaningful influence in reducing the degree of program blocking with respect to window sizes, it would have to quadruple the number of instructions executed between I/O waits. This would then move the instruction distance between I/O blocking into the window size region, found in the present study, for the start of steady state or stable program paging behavior.

The answer to the original question is that a window size T for an operating system should be chosen on the basis of program paging demands. It is found that "I/O-bound" programs appear to have an inter-I/O distance, T', which is much smaller than T. If a system is running in batch mode, the degree of input and output buffering may be enough to increase T' to a value comparable to T. Unfortunately, a strictly terminal oriented system may not be buffered enough to accomplish a comparable increase in T'. However, T' is an average. Thus, there will be extended periods of uninterrupted processing where the longer window size advantage will apply even for I/O dependent programs.

PROGRAM 1 ASSEMBLER



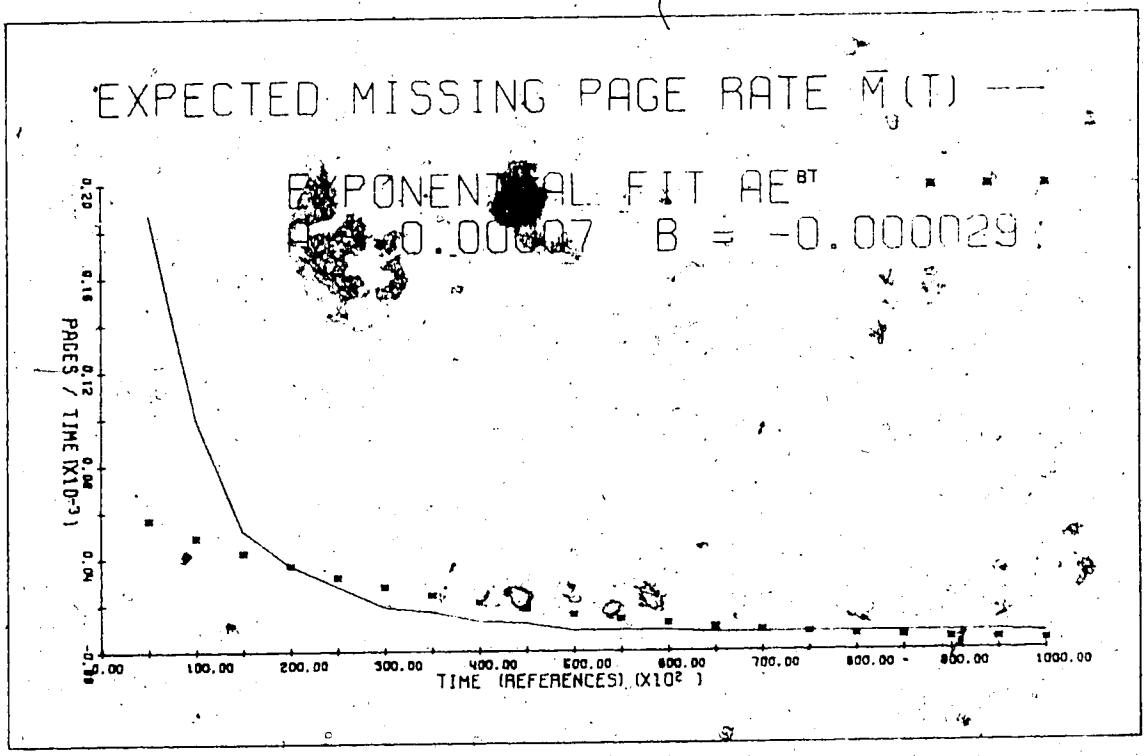
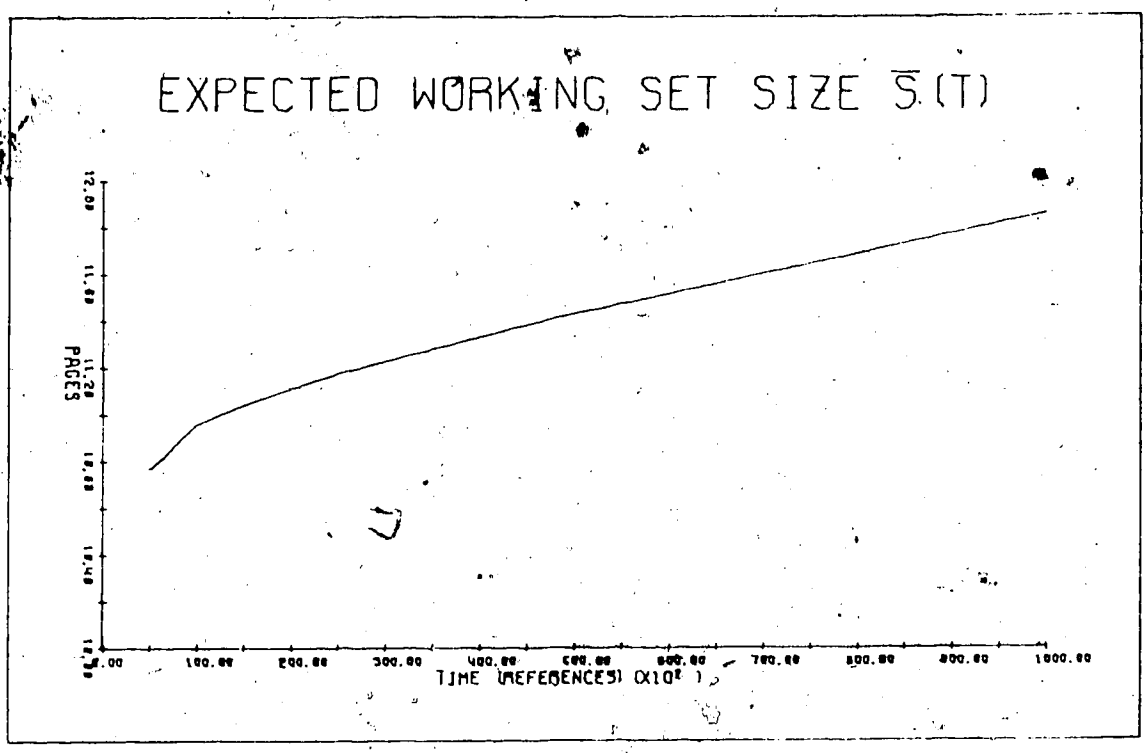


FIGURE 9 : $\bar{S}(T)$ AND $\bar{M}(T)$ CURVES

PROGRAM 2 LISP



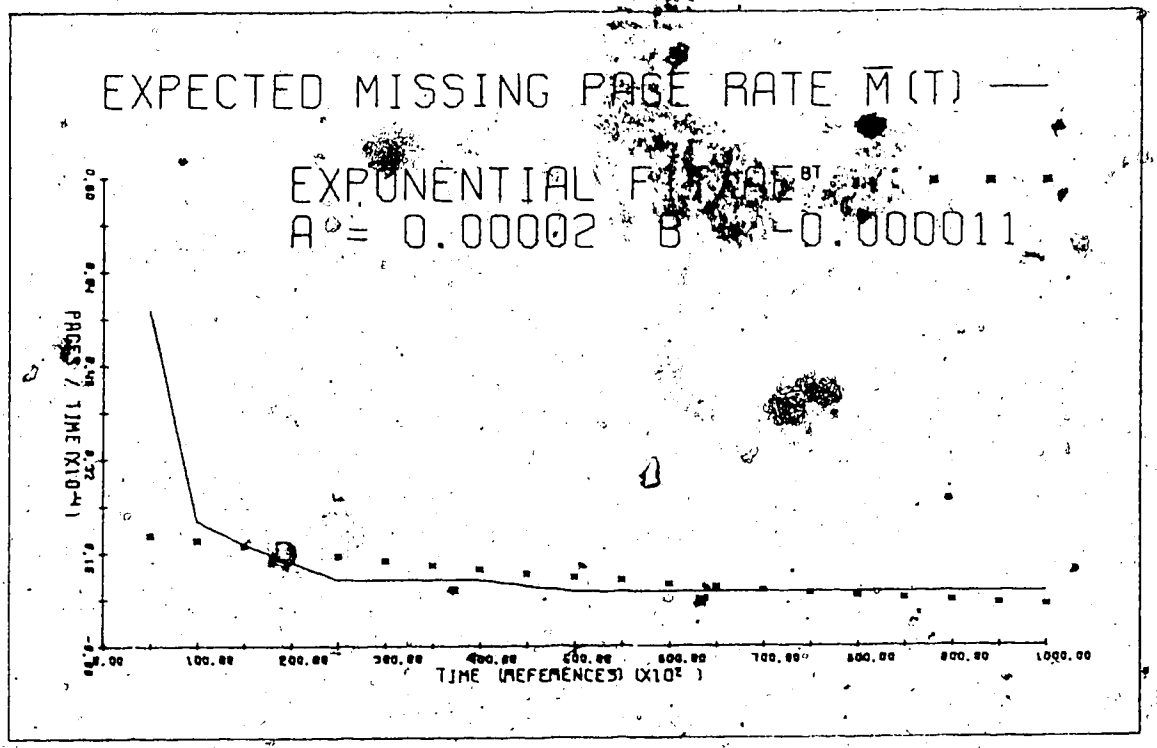
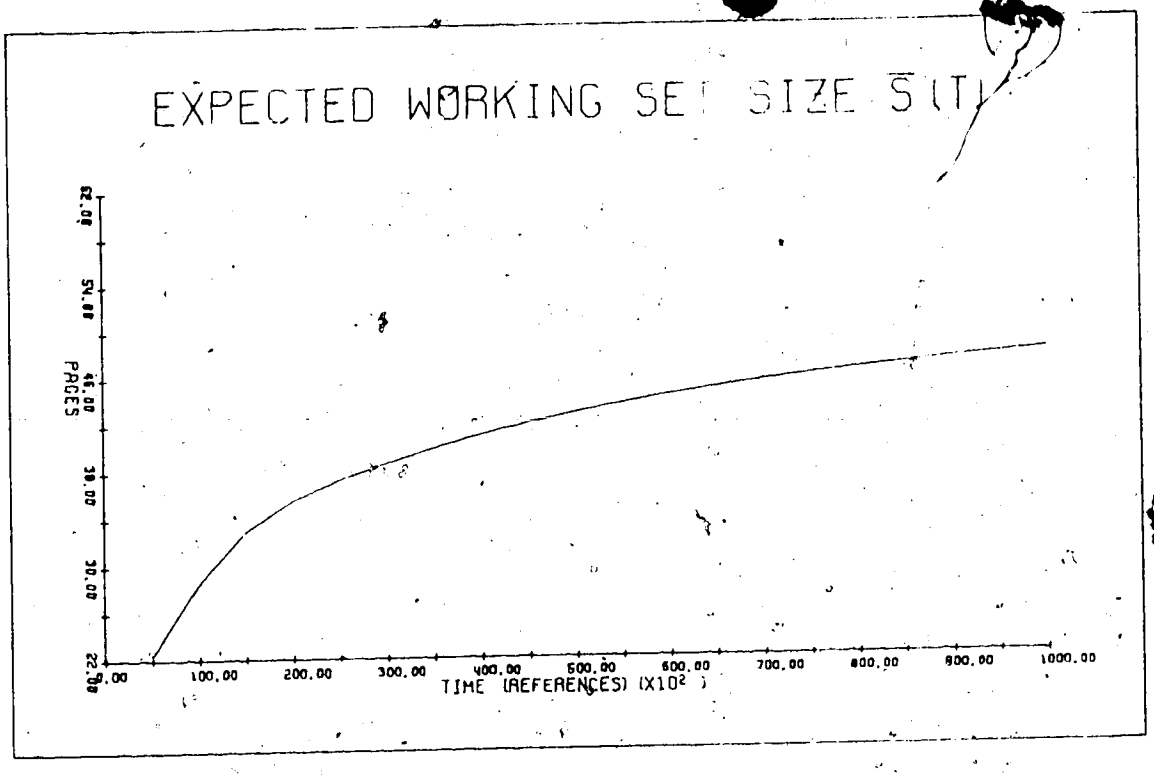


FIGURE 10: $\overline{S}(T)$ AND $\overline{M}(T)$ CURVES

PROGRAM 3 [redacted]OL



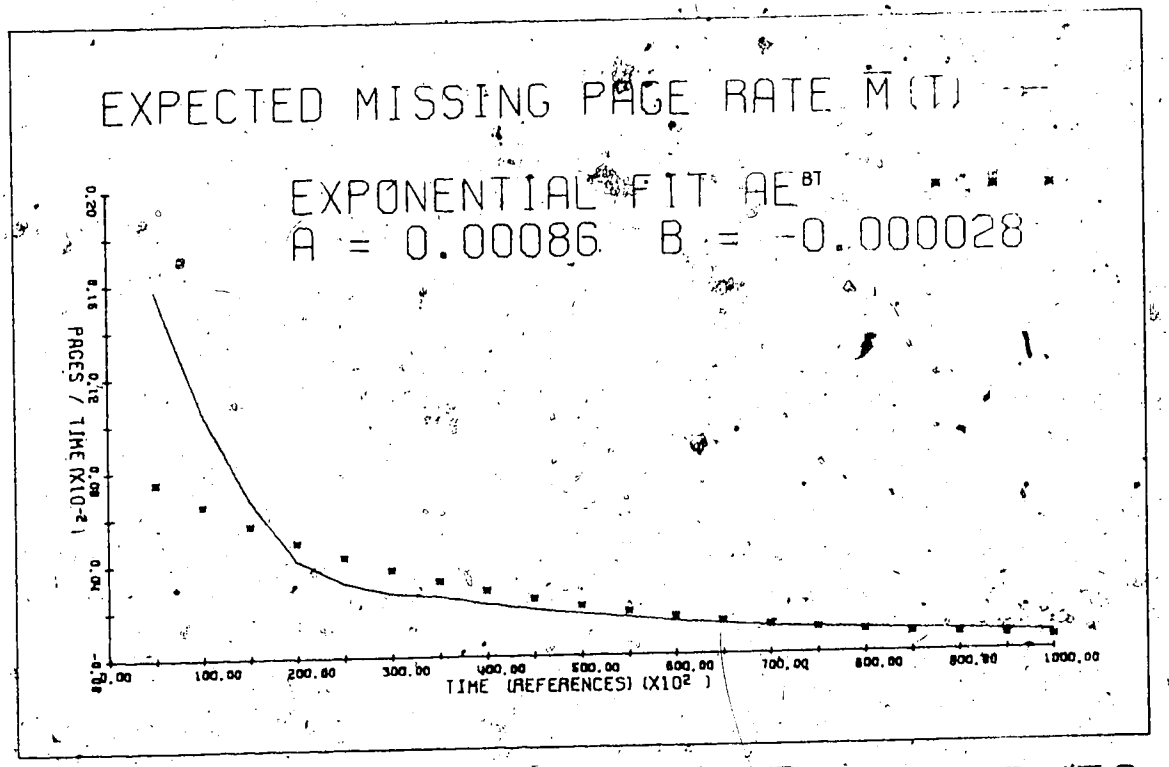
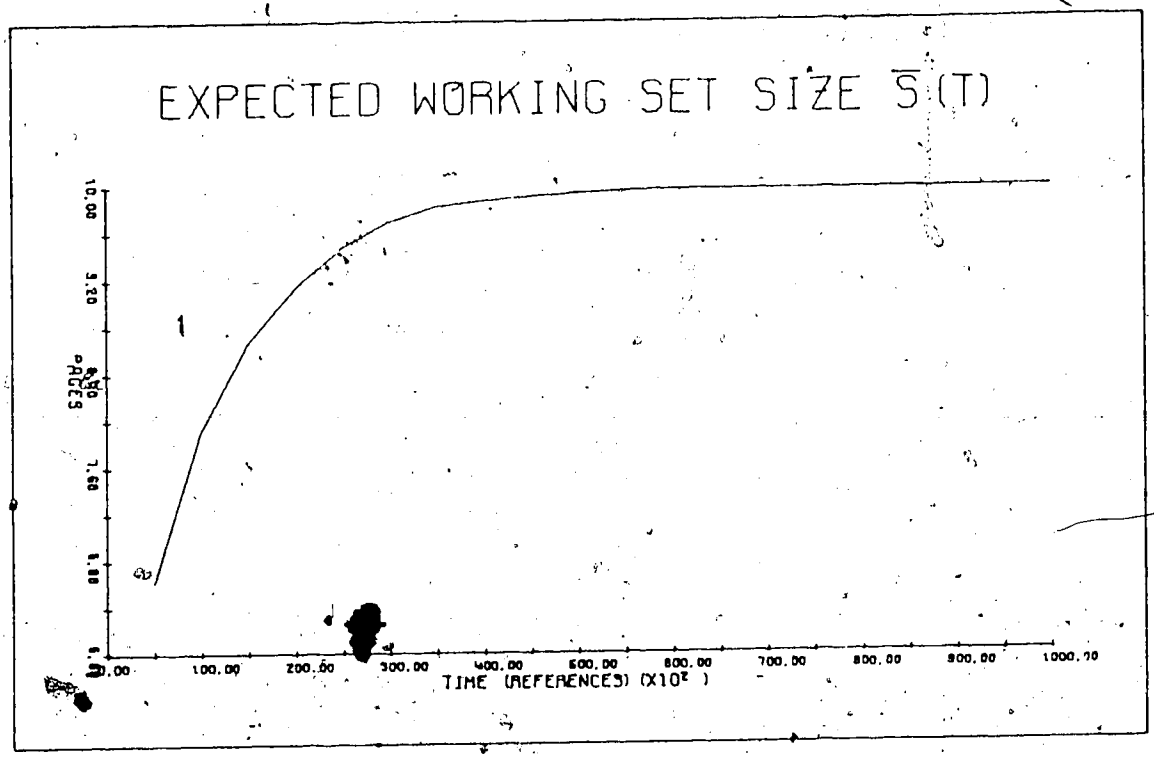


FIGURE 11 : $\bar{S}(T)$ AND $\bar{M}(T)$ CURVES

PROGRAM 4 FORTRAN



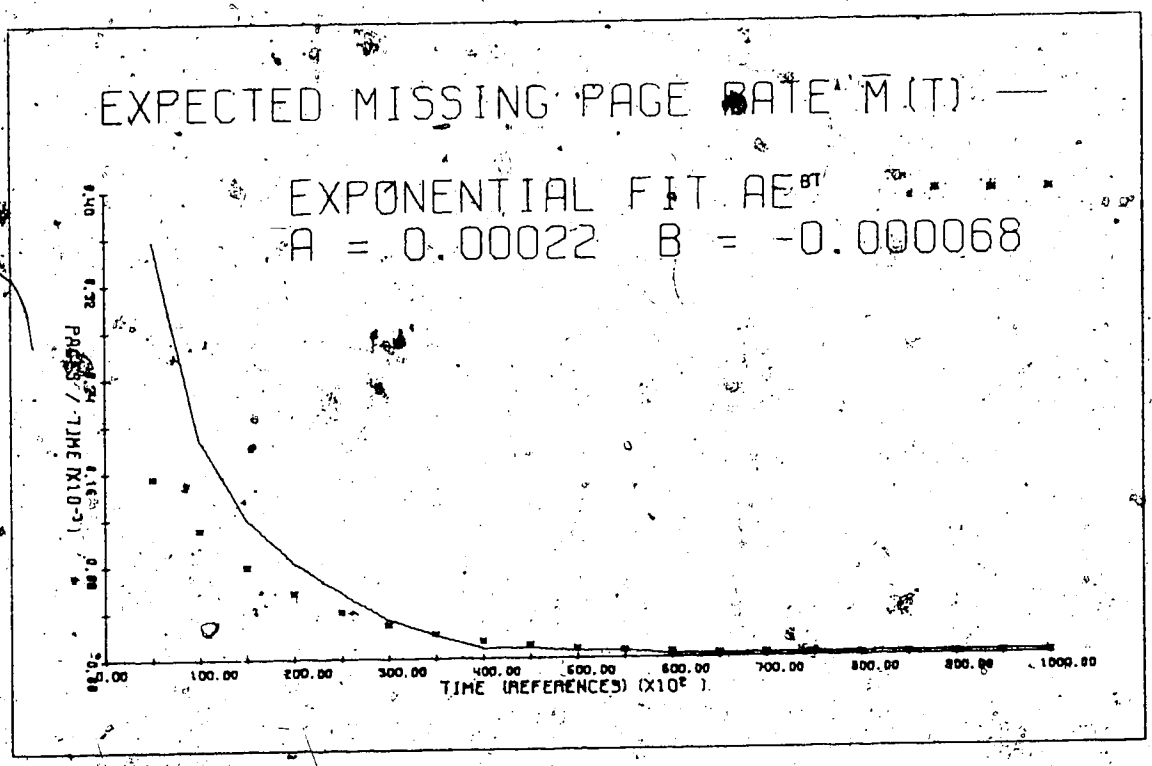
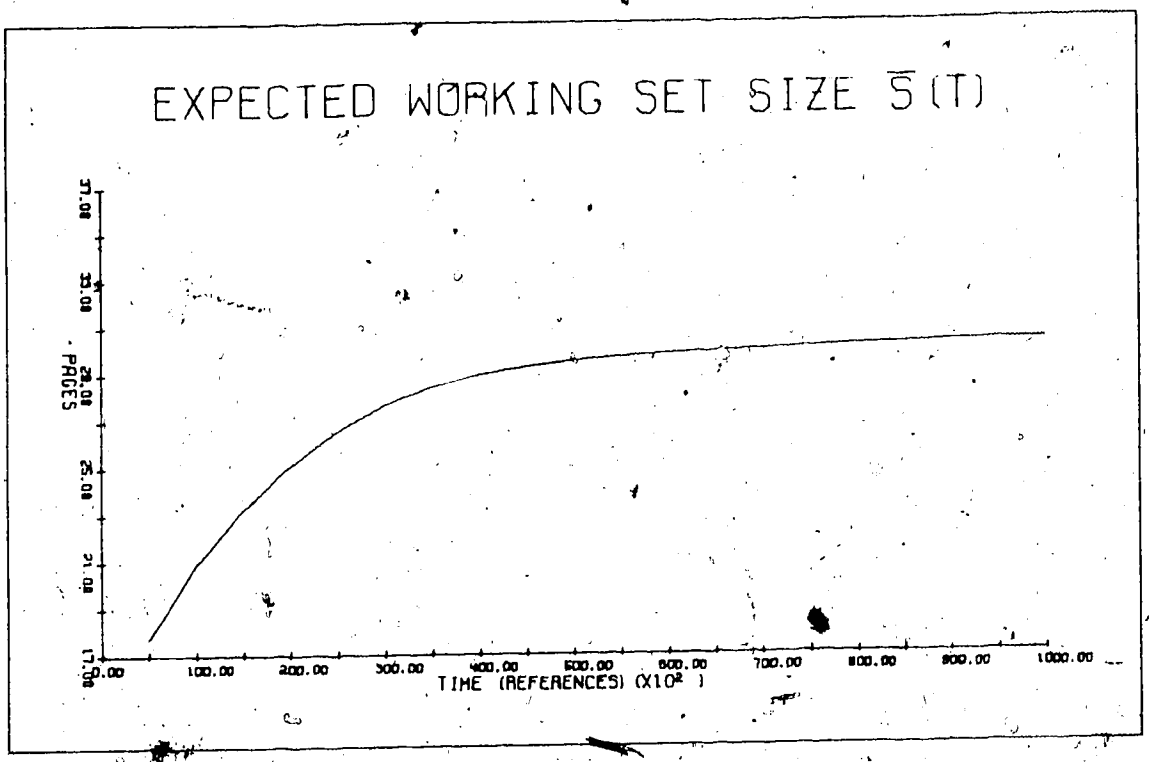


FIGURE 12. $\overline{S}(T)$ AND $\overline{M}(T)$ CURVES

PROGRAM 5 ALGOL



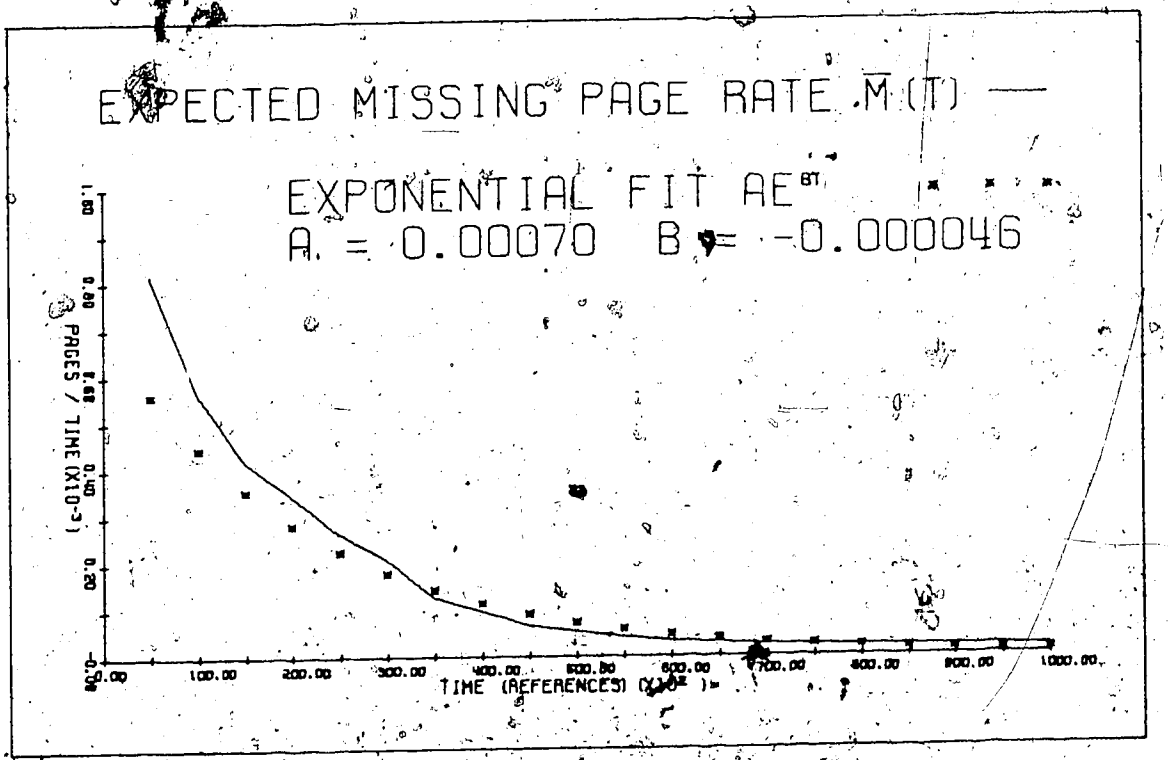
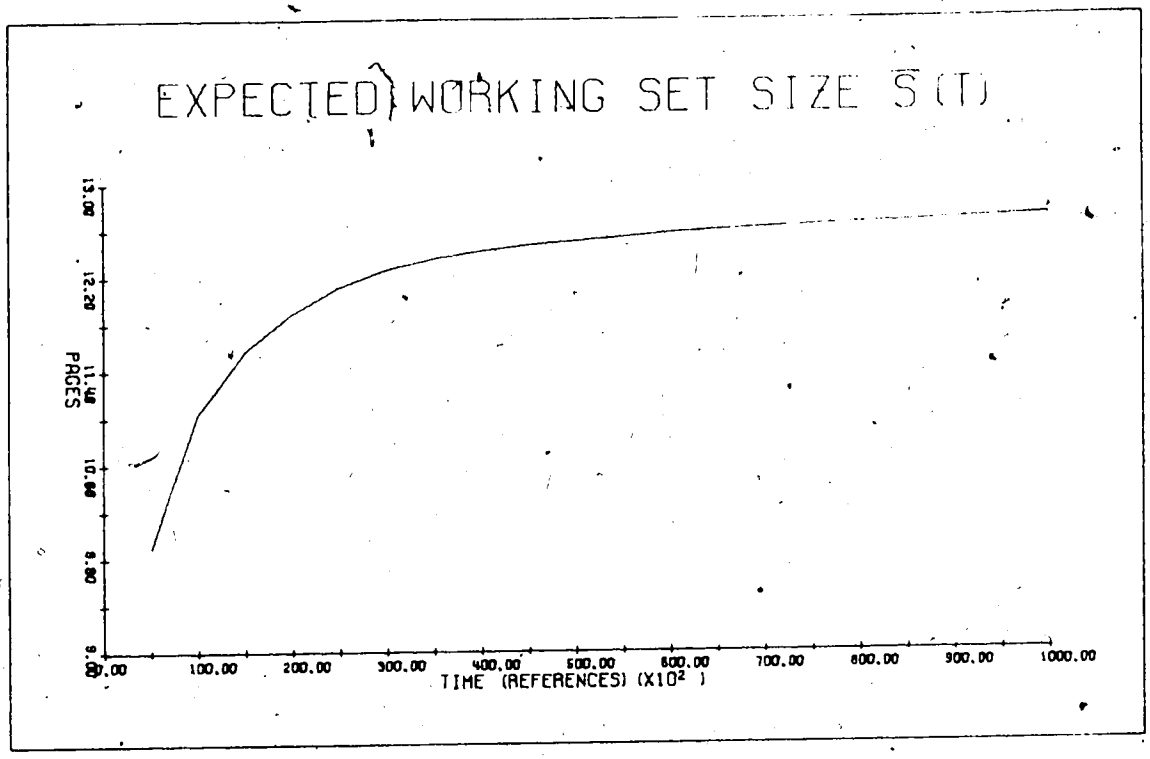


FIGURE 13: $\overline{S}(T)$ AND $\overline{M}(T)$ CURVES

PROGRAM 6 PL360



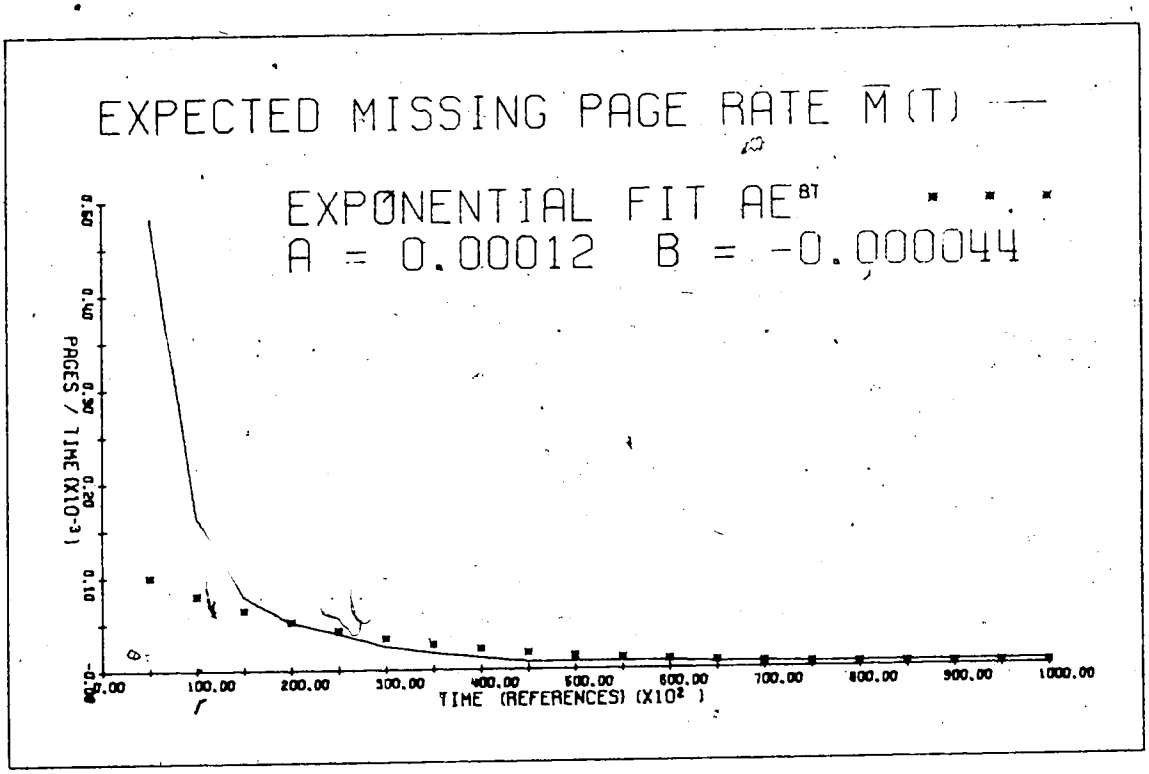
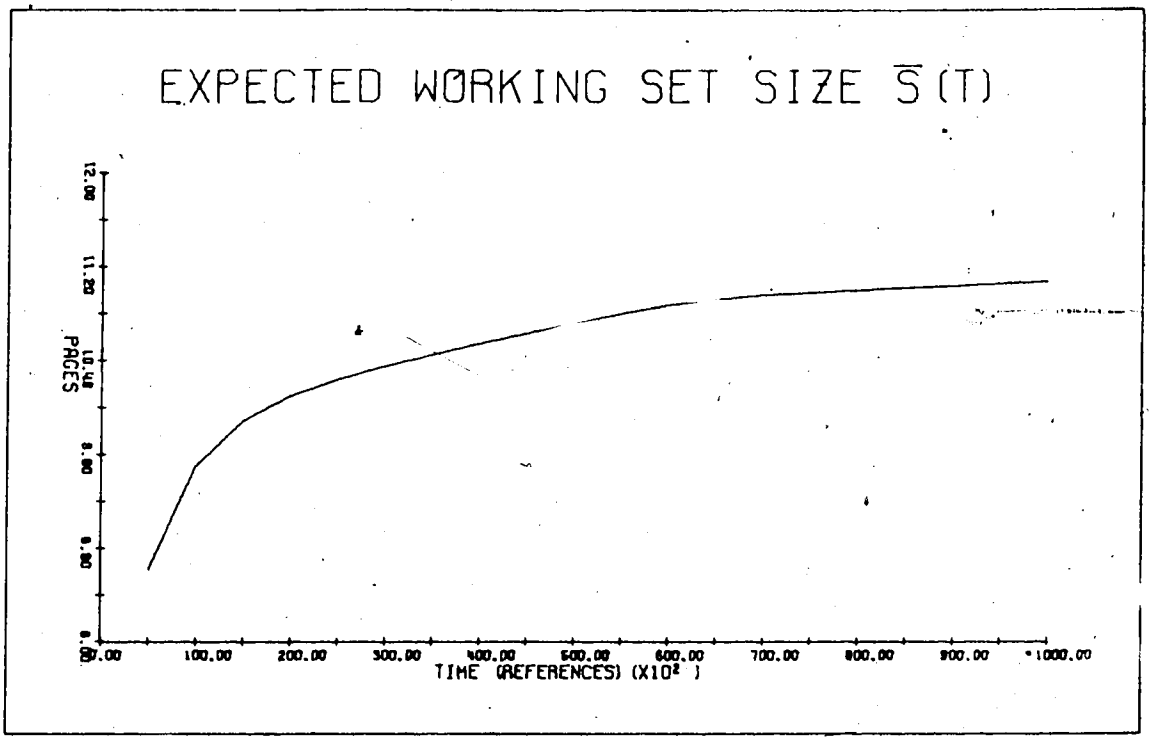


FIGURE 14: $\overline{S}(T)$ AND $\overline{M}(T)$ CURVES

PROGRAM 7 ALGOL



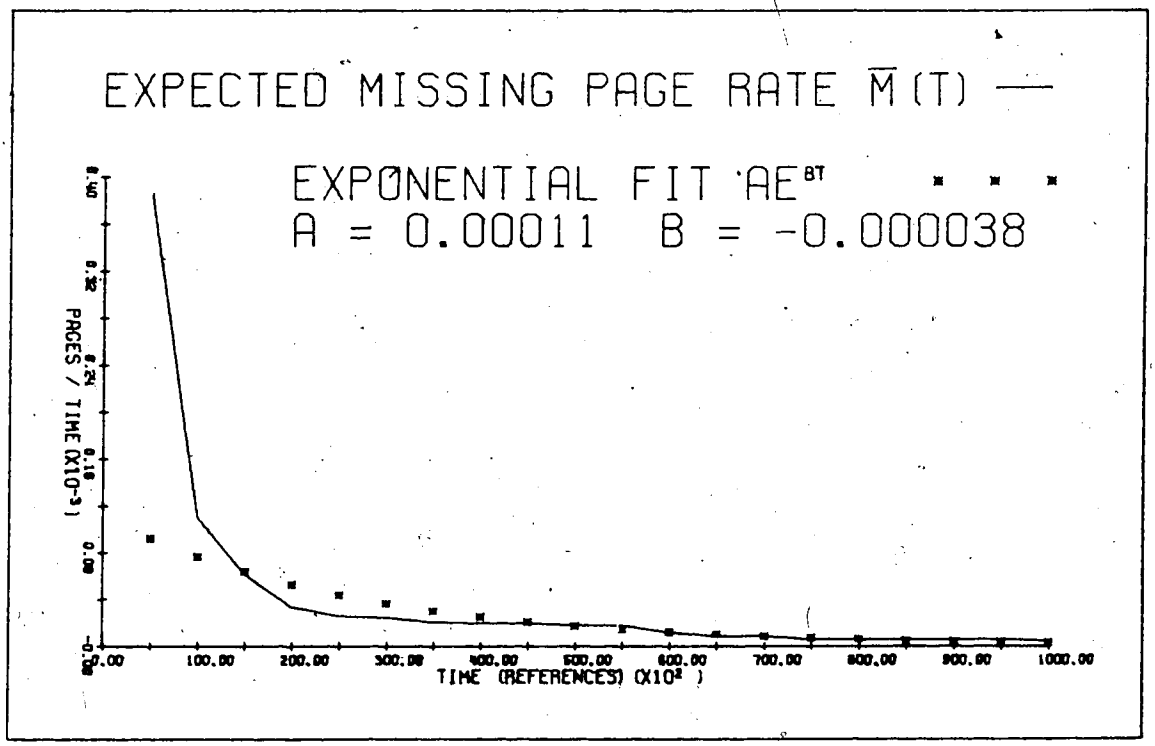
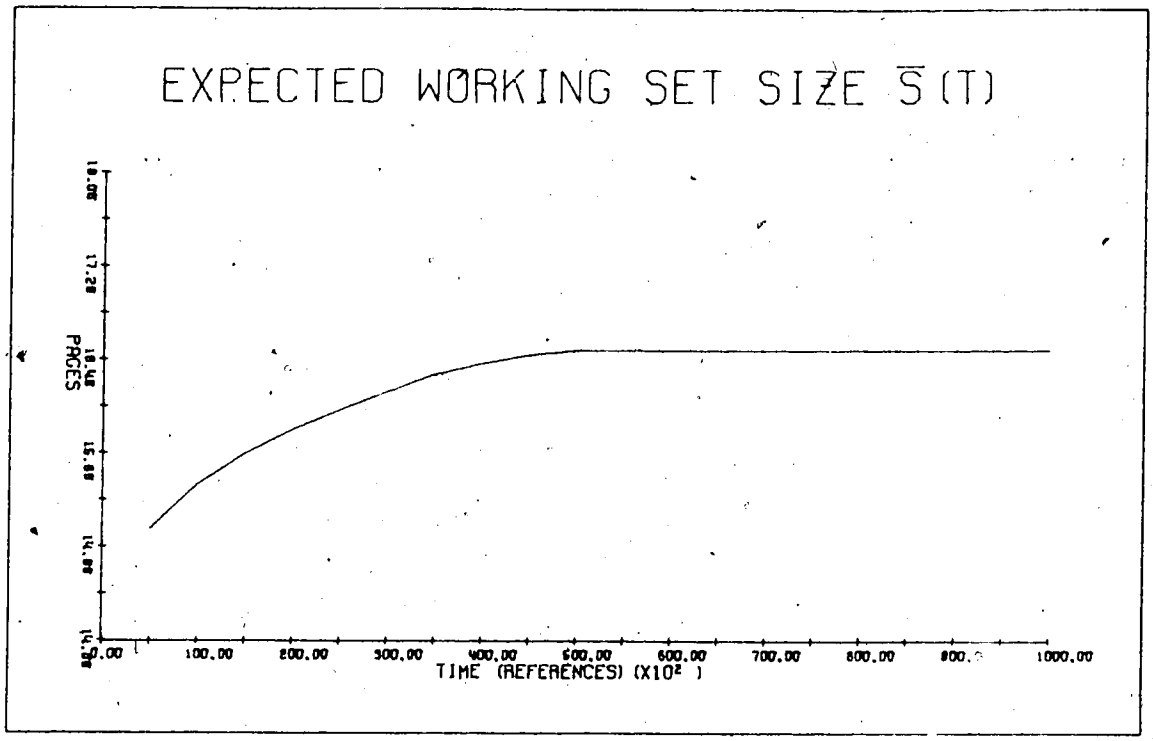


FIGURE 15: $\overline{S}(T)$ AND $\overline{M}(T)$ CURVES

PROGRAM 8 PL360



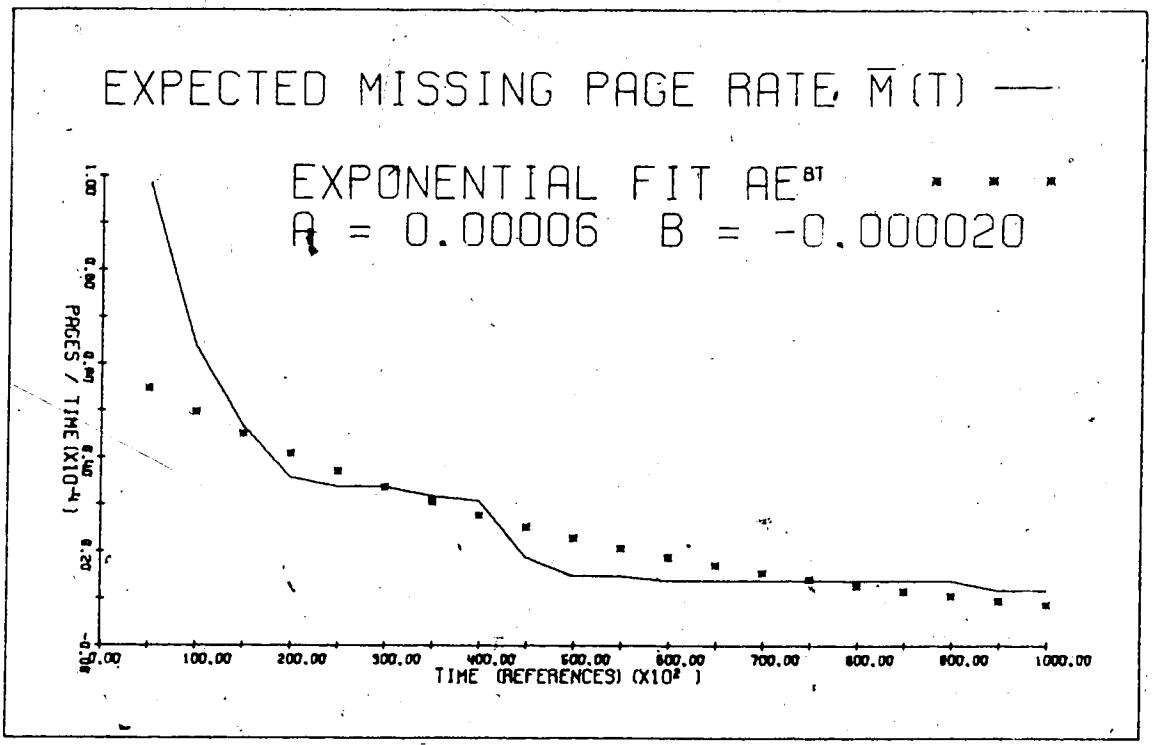


FIGURE 16: $\overline{S}(T)$ AND $\overline{M}(T)$ CURVES

PROGRAM 9 - ASSEMBLER

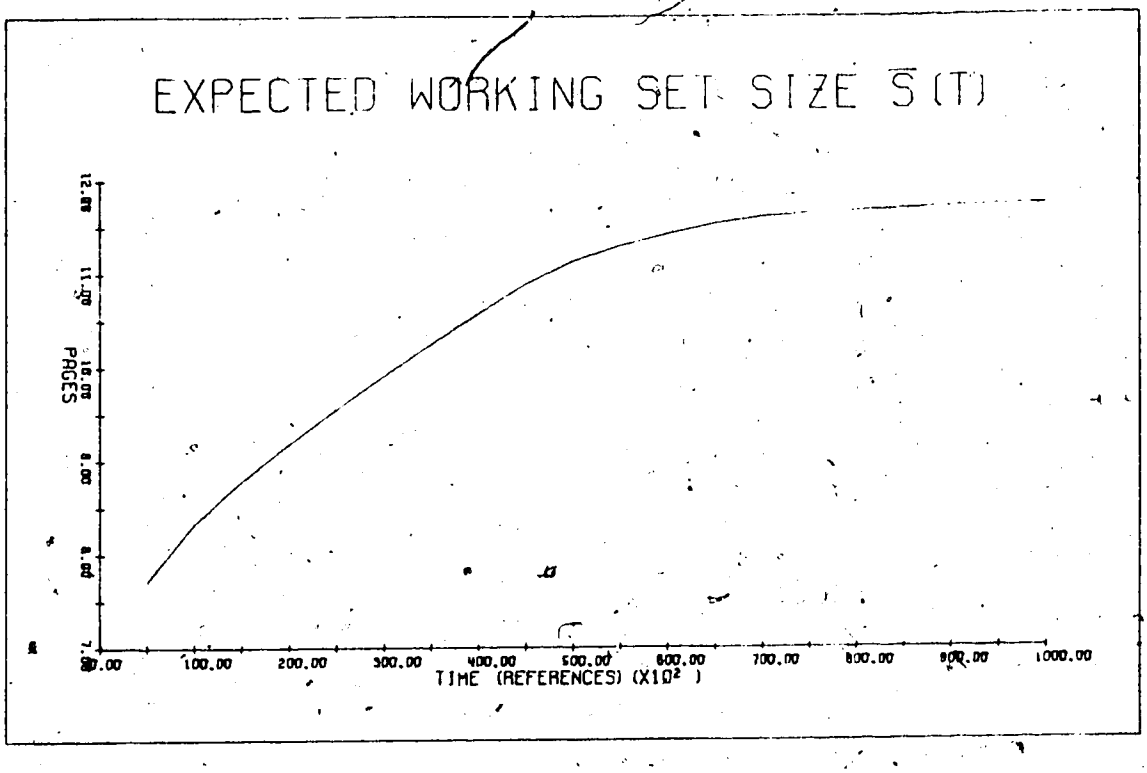
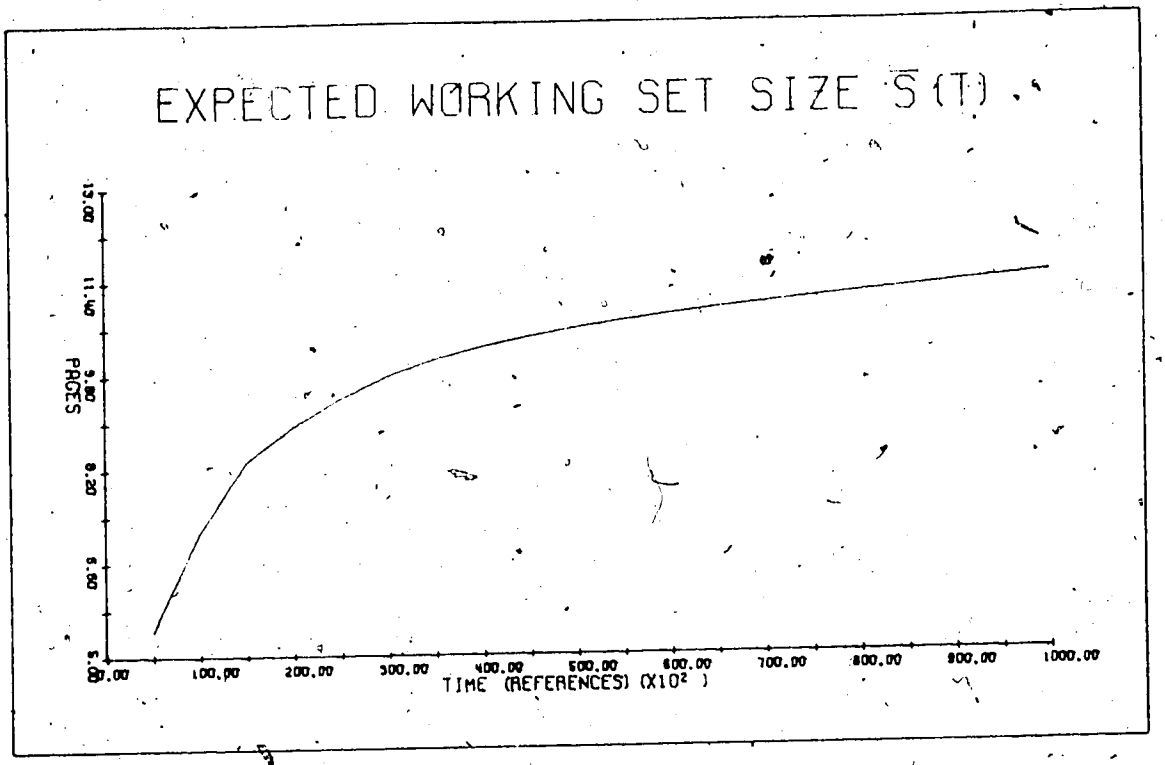


FIGURE 17: $\overline{S}(T)$ AND $\overline{M}(T)$ CURVES

PROGRAM 10 ASSEMBLER



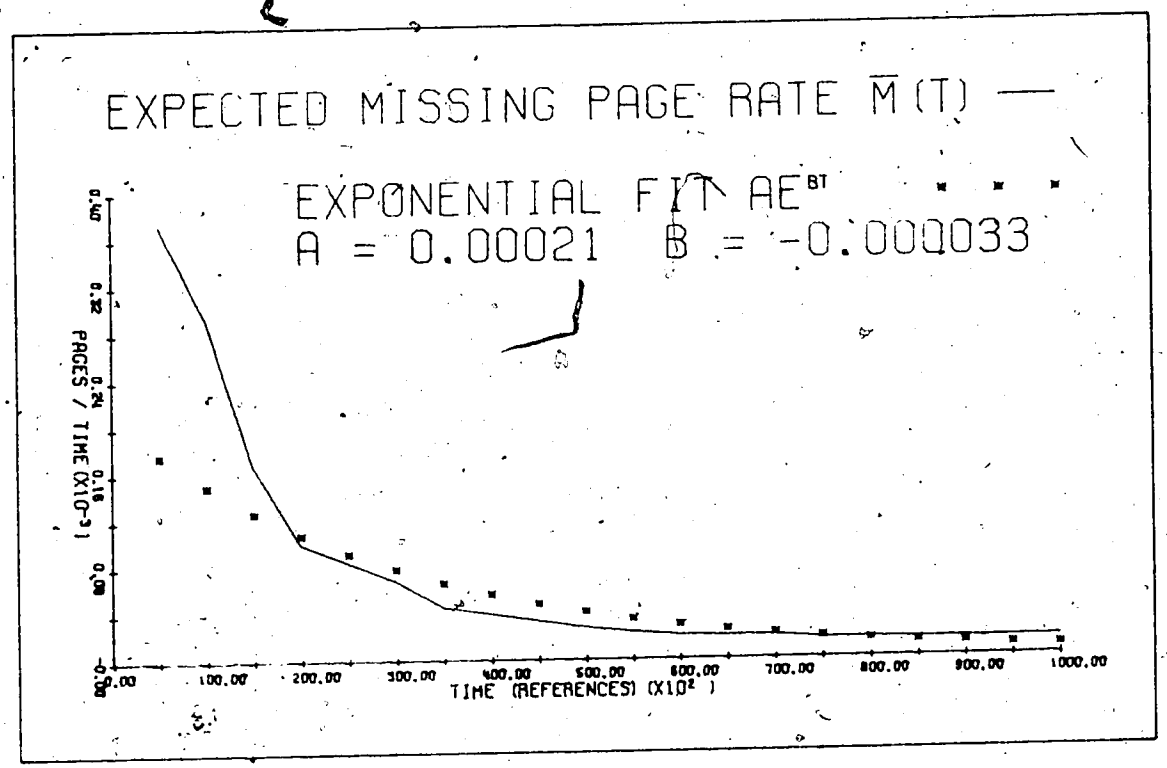
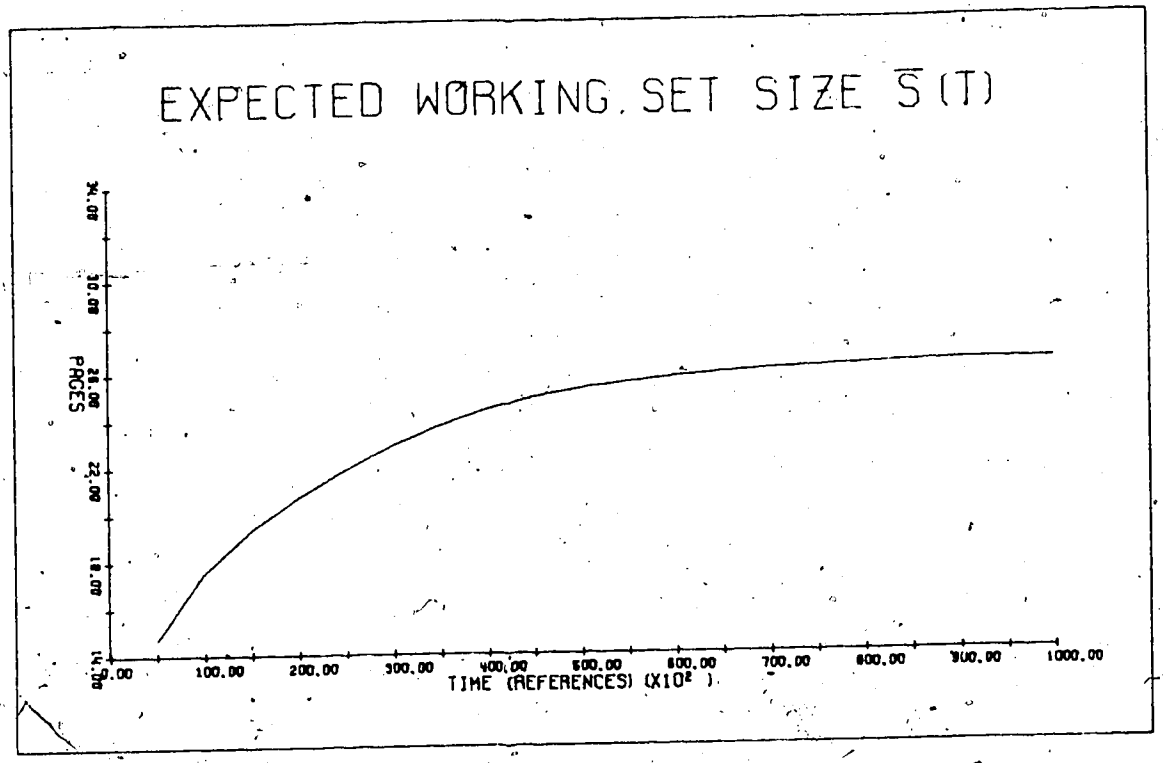


FIGURE 18: $\bar{S}(T)$ AND $\bar{M}(T)$ CURVES

PROGRAM 11 ALGOL



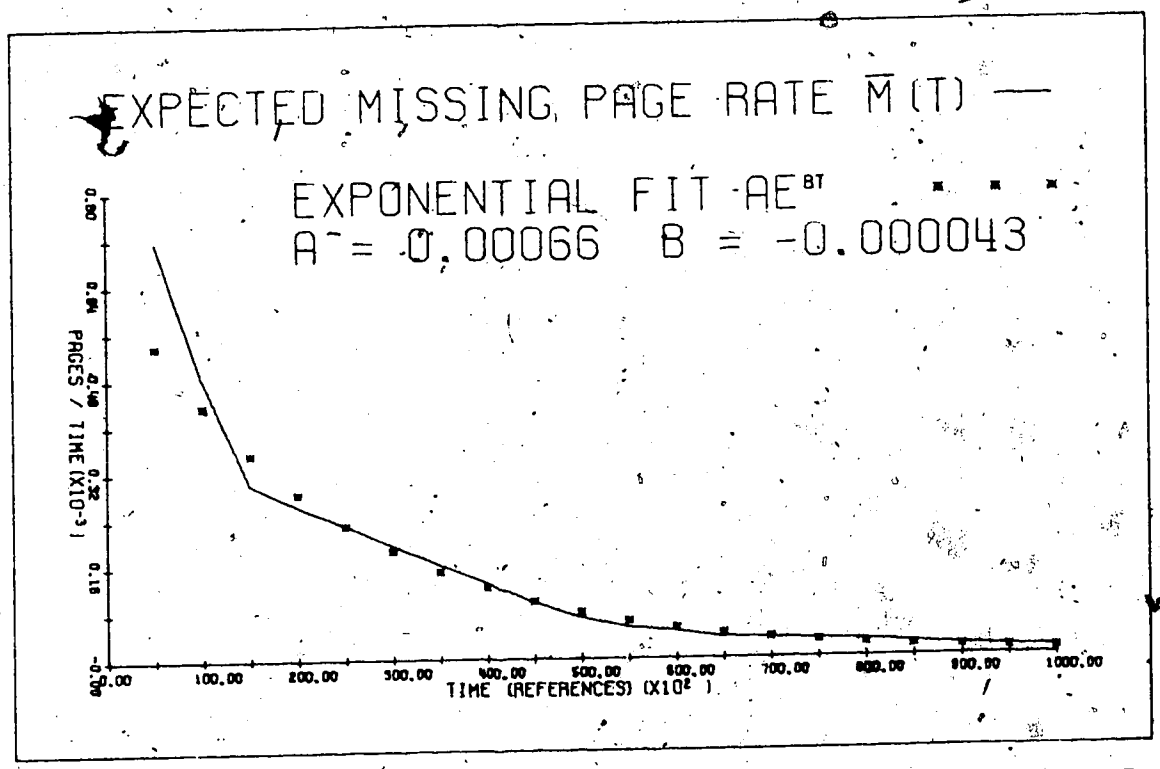
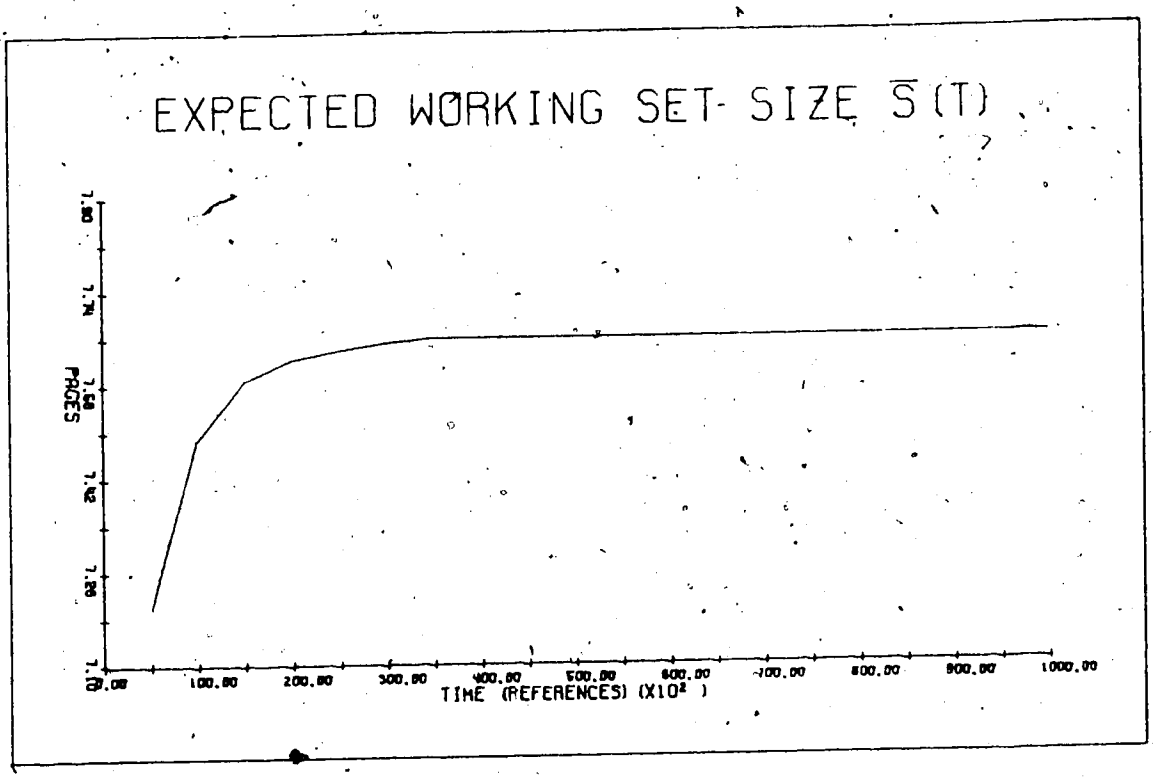


FIGURE 19: $\overline{S}(T)$ AND $\overline{M}(T)$ CURVES

PROGRAM 12 ASSEMBLER



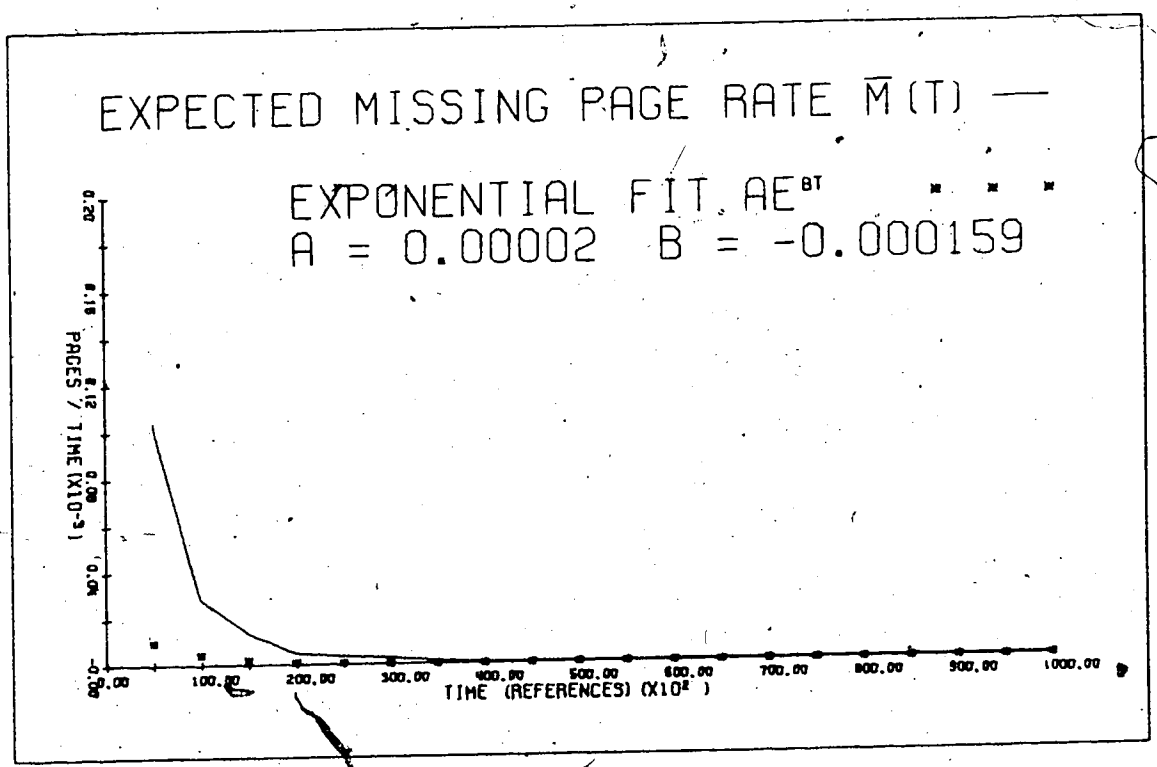
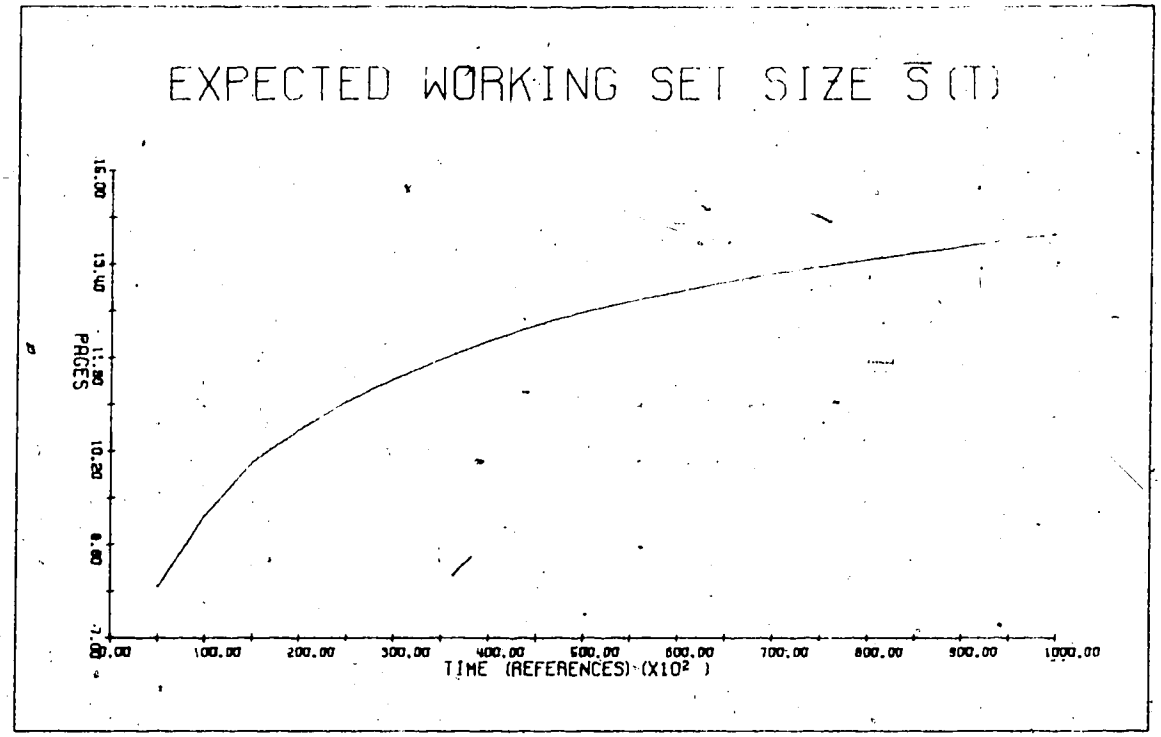


FIGURE 20: $\bar{S}(T)$ AND $\bar{M}(T)$ CURVES

PROGRAM 13 ASSEMBLER



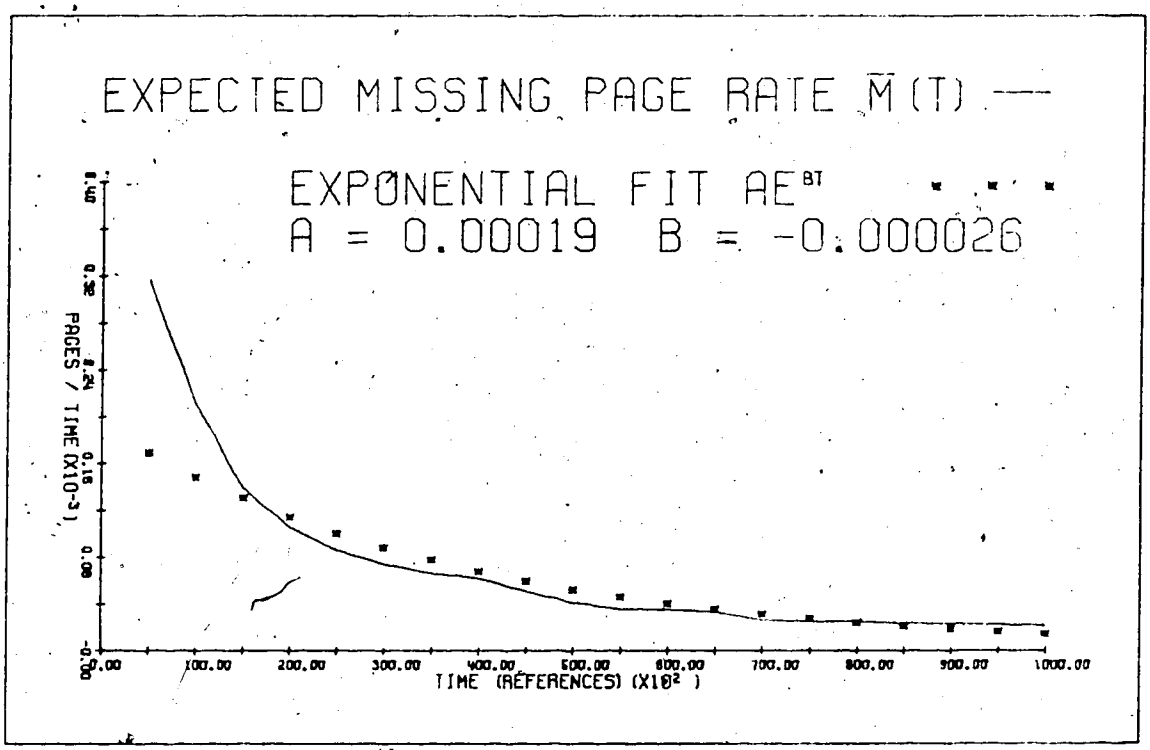
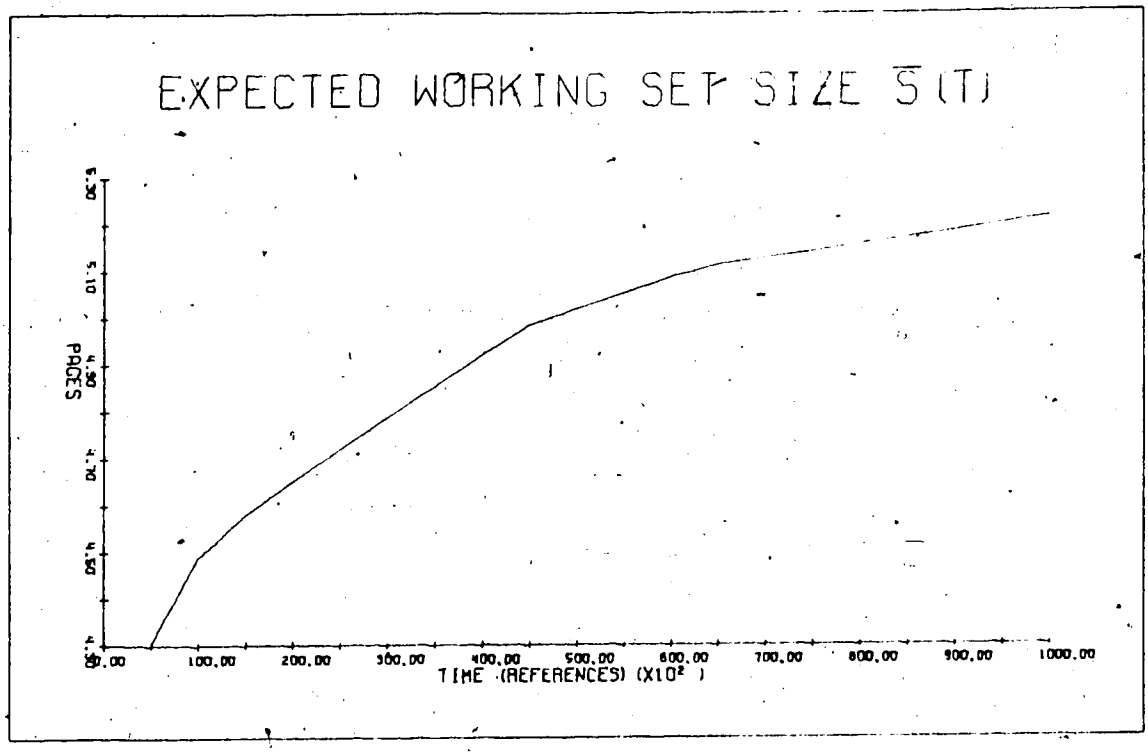


FIGURE 21: $\overline{S}(T)$ AND $\overline{M}(T)$ CURVES

PROGRAM 14 FORTRAN



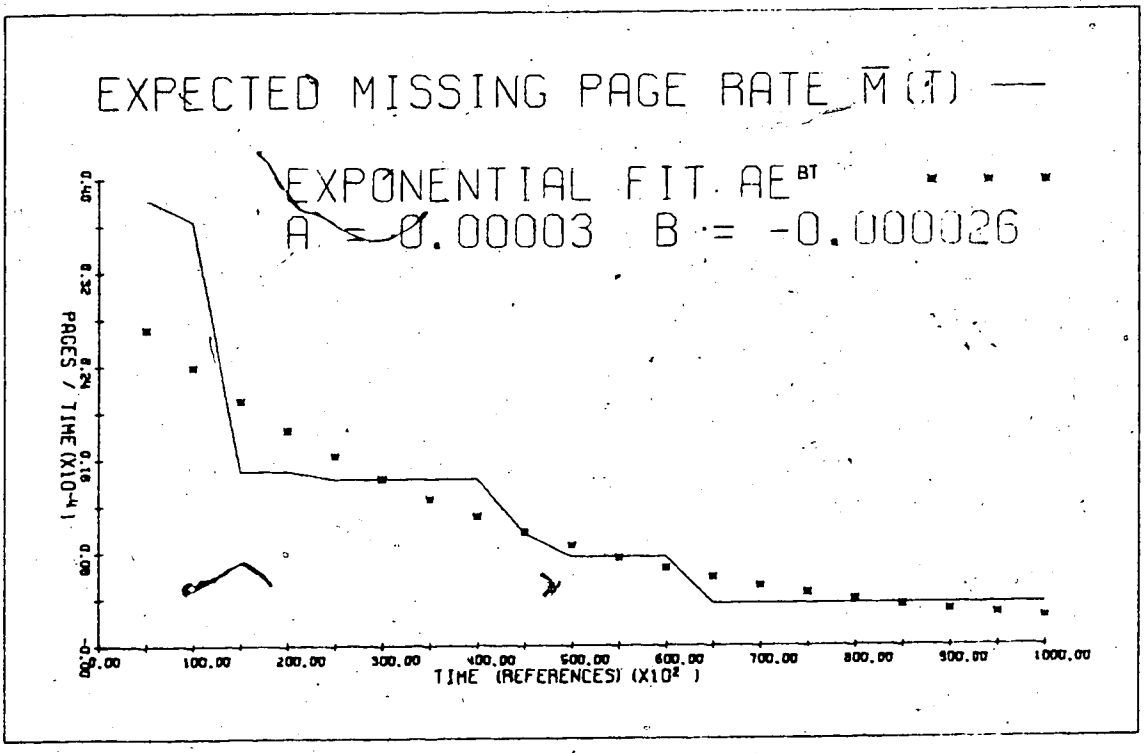
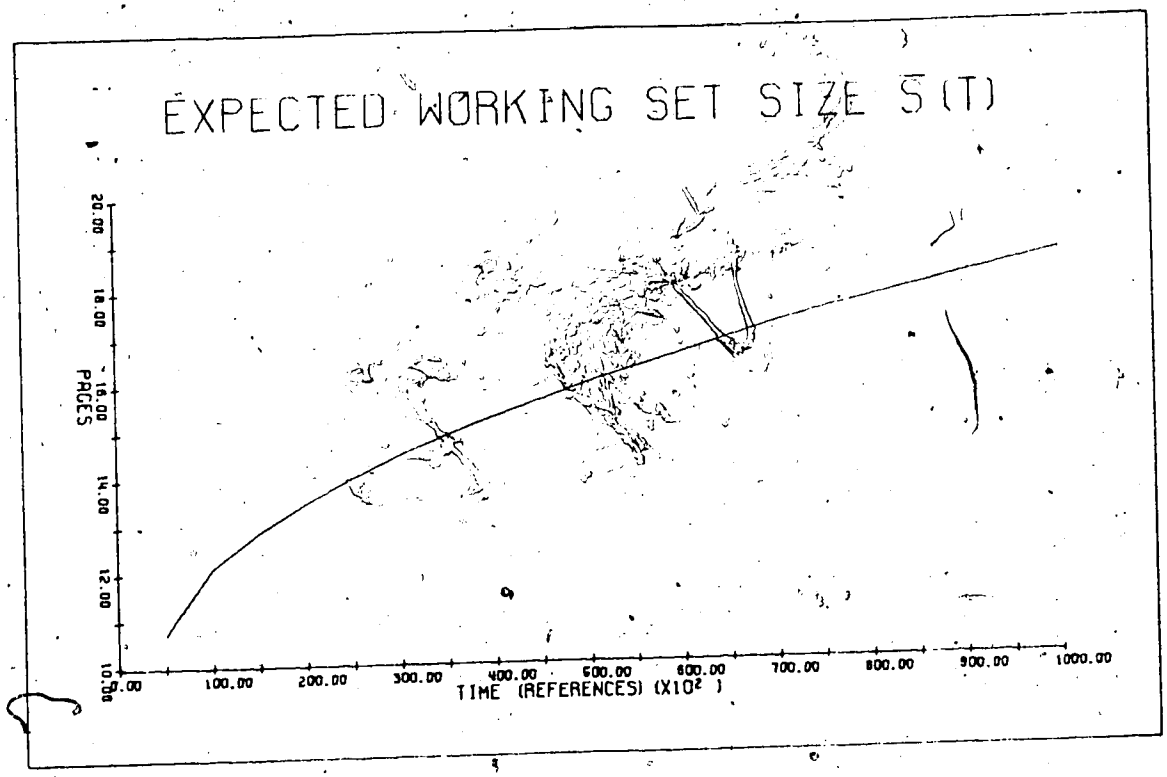


FIGURE 22: $\bar{S}(T)$ AND $\bar{M}(T)$ CURVES

PROGRAM 15 FORTRAN



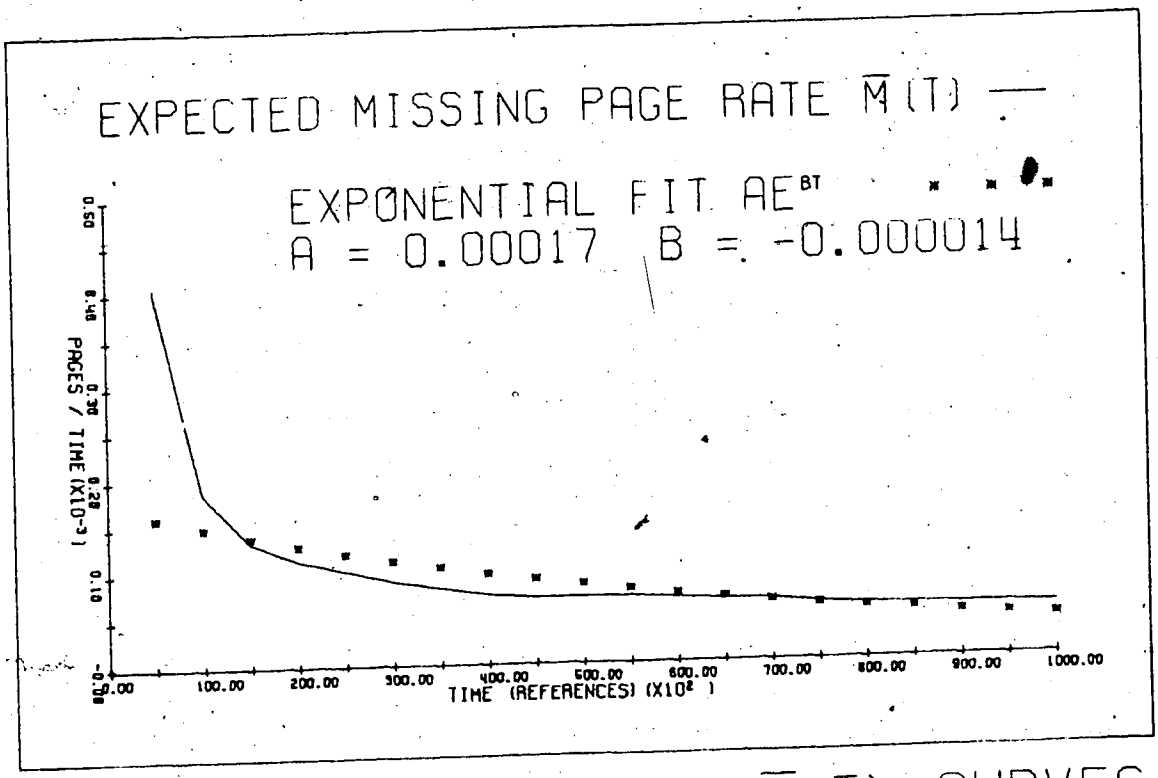


FIGURE 23: $\bar{S}(T)$ AND $\bar{M}(T)$ CURVES

Page Behavior

Working set concepts were applied to reference strings of individual pages to obtain information concerning the internal behavior of a program. The page analysis was done on programs 8, 9, and 11. Together, they represent three million references to a total of 81 pages. The results listed in Table IX present the general characteristics of the pages of the three programs. Data column one gives the number of pages accessed by the individual programs. The maximum average working size of a program's page is given in column two (given that the largest window size is 100,000 references long). The average working size over all pages is .67 pages. The final column of this table gives the percentage of pages which caused page faults when the program reached its minimum missing page rate. The average percentage of these "paging" pages per program was 18.5% of the total pages referenced. Example graphical plots of individual pages' missing page rate and working size curves are illustrated in Figures 24 and 25, respectively.

PROGRAM 9 TYPE = ASSEMBLER

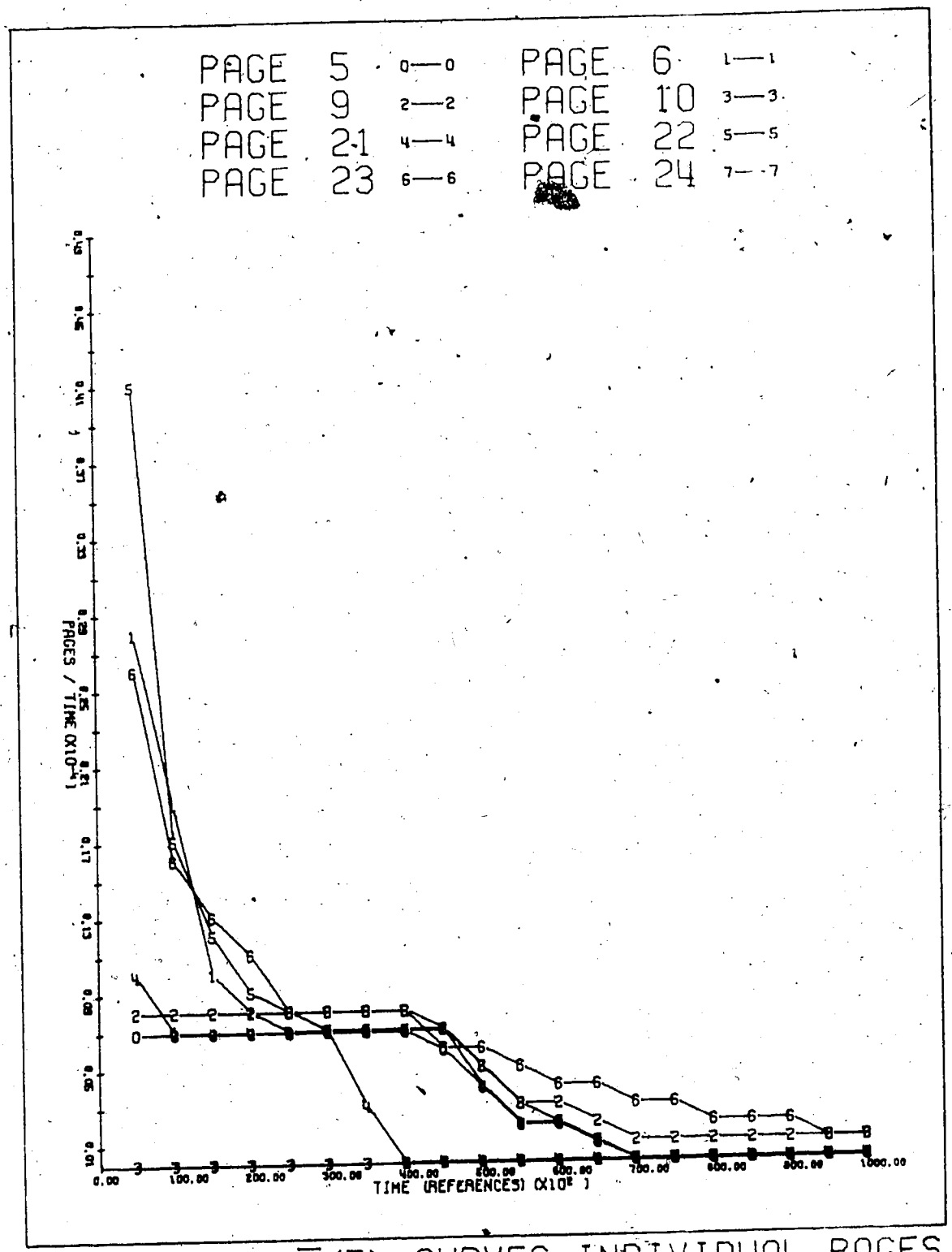


FIG. 24: $\overline{M}(T)$ CURVES INDIVIDUAL PAGES

PROGRAM 9 TYPE = ASSEMBLER

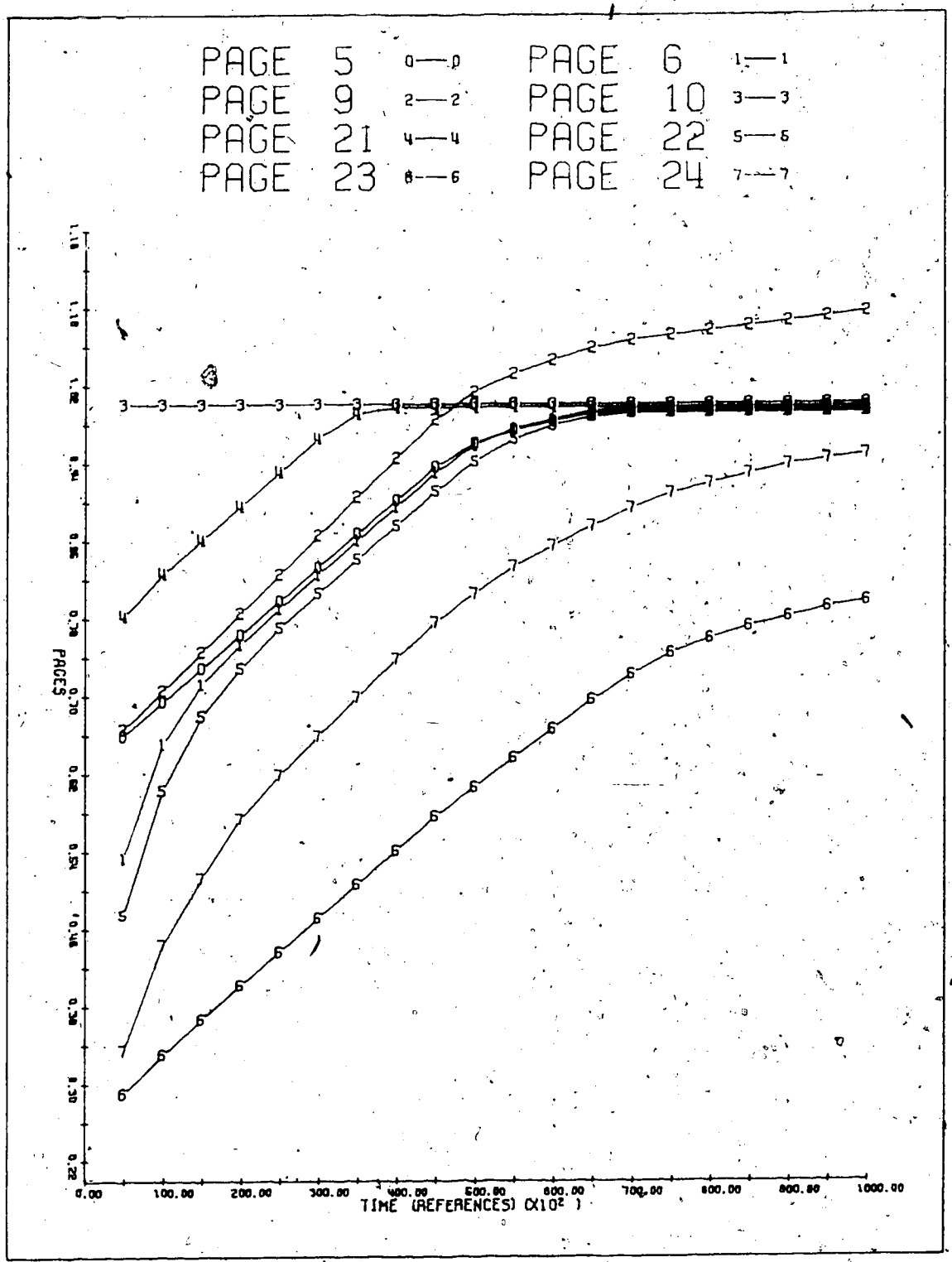


FIG. 25: $\overline{S}(T)$ CURVES INDIVIDUAL PAGES.

TABLE IX: Page Working Set Characteristics of Programs 8,9, and 11

PROGRAM NUMBER	TOTAL PAGES REFERENCED	MAX. AVERAGE PAGE WORKING SIZE	% of PAGES PAGING AT MINIMUM PAGING RATE
8	34	.485	20.6
9	12	.974	25.0
11	35	.752	14.3

TABLE X: Characteristics of Paging and Non-paging Pages for Programs 8,9, and 11

PROGRAM NUMBER	AVERAGE REF'S TO PAGING PAGE	ADJUSTED AVERAGE REF'S TO PAGING PAGE	AVERAGE REF'S TO NON-PAGING PAGE	PAGING PAGE AVERAGE WORKING SIZE	PAGING PAGE ADJUSTED AVERAGE WORKING SIZE	NON-PAGING PAGE AVERAGE WORKING SIZE
8	32,850	16,355	25,750	.344	.156	.528
9	107,370	53,685	75,320	.947	.477	.973
11	25,350	14,600	26,600	.855	.322	.720

All the pages of the three programs are separated into two groups; "paging" and "non-paging" pages. Paging pages are those which did not have a zero missing page rate given the largest window size studied. Non-paging pages did reach this zero paging rate under the same circumstances. Table X

presents a more detailed description of the characteristics of the two page types. The first three data columns indicate average references received by a page type. The final three columns indicate average working sizes of the different page types. Over the three million references, an average of 44,900 references/page was made to paging pages whereas this number was 32,900 for non-paging pages. The average working size for paging and non-paging pages was .63 and .74 pages, respectively. However, the calculations for paging pages are misleading since they are based on page residency over the entire reference string. From the data collected in the study, each paging page was found to re-enter memory an average of 1.8 times after being initially accessed and then removed. The working size, which is a measure of the total locality of a page, can be adjusted to reflect the multiple residencies of a paging page. This is done by simply dividing a given paging page's number of total references or working size by the number of times the page entered memory. The adjustment lowers the average working size of a paging page to .28 pages with an average number of references of 20,900 per residency. This result also lowers the average working size of a page in these programs to approximately .60 pages. Given that a page is resident, the frequency of reference to non-paging and paging pages is 6,550 and 7,450 references per 100,000

program references, respectively.

From these results, the differences distinguishing paging and non-paging pages appear to be in the window size required for complete residency and the average number of references to the two page types. The locality or working size of both groups of pages are nearly the same over the whole reference string. However, the analysis indicates that both groups of pages have essentially the same reference frequency over 100,000 program references. That is, although paging pages have a larger average number of references per page than non-paging pages, this distinction disappears when the rate of reference over the individual page residencies is considered.

Figure 26 presents a graph of the accumulated percentage of non-paging pages as a function of their working size or locality over the entire reference string studied. The most striking aspect of this graph is that the non-paging pages appear to group around a given locality. Program 9 has 100% of its non-paging pages at 100% memory residency. Program 11 has 25% of its pages at less than 10% residency, whereas the rest are close to 100% residency. The remaining program, number 8, has groupings of pages having residencies 300,000; 700,000; and one million

references. This study did not investigate whether or not those pages grouped together in locality also formed a temporal grouping during execution. The graph also indicates that non-paging pages can be further sub-divided into groups; "resident" and "non-resident" non-paging pages. It is these non-resident pages which do not page that will be further investigated as to whether or not adaptive techniques can be applied to reduce their memory utilization.

The graph in Figure 27 presents the percentage of non-paging pages which fault as a function of window size. These lines indicate that, except for 10% of program 11's non-paging pages, page faulting ceases for window sizes larger than 70,000 references. Thus, it may be possible to adaptively adjust window sizes for non-paging pages so that they are smaller than those for paging pages. However, the graph gives the result's for all non-paging pages. Figure 26 indicates that many of these pages are essentially memory resident and so no advantage can be gained by adjusting their window size. What is necessary is to give the accumulated percentage of pages that fault as a function of window size for all three defined page groupings. Figure 28 presents this result for all pages of the three programs studied. Paging pages have been depicted on the graph by

considering the minimum paging level a page attains as the zero paging level. This allows the paging pages to be treated in the same manner as the non-paging pages. The plot of the lines for the paging pages really shows that these pages are biased towards larger window sizes. The plot for the resident non-paging pages indicates that this group of pages is composed almost equally of pages requiring smaller window sizes and larger window sizes to be entirely resident. Only the non-resident non-paging pages show a distinct bias towards a smaller window size. This is a good result for adaptive control purposes, since only these pages offer any advantage for using a smaller window. However, how much of an asset is this reduction in window size in terms of memory saving? Assume that these pages can be detected adaptively with 100% accuracy. The non-resident non-paging pages represent a total of 29 out of 81 pages or 36%. From data collected, their average working size was calculated to be .35 pages. Suppose also that the window size for the system were set at 100,000 references. The total memory utilization for the average program would be 28x.67 pages = 18.8 pages per second of execution. Approximately ten pages of this average program would be non-resident non-paging. If they are all given a window size of 60,000 references and all removed from memory after one second of execution a total memory saving of .04 pages

per second for each page will accrue. This represents a total maximum saving of .4 pages per second or less than a 3% reduction in memory requirements over using the same window size for all pages. The saving hardly recommends any adaptive procedure even assuming it is 100% effective.

However, Figure 28 does indicate the great variety of window sizes that can be desired by a page to give it no-fault characteristics. In order to improve memory utilization, it may be possible to adaptively assign each page a window size. One such means of accomplishing this purpose may be by monitoring the frequency of reference to a given page. Unfortunately, the graph of Figure 29 indicates little substantiation for such an approach. This graph is a scatter diagram showing the window size at which non-paging occurs as a function of the average number of references a page receives per 100,000 program references. A visual investigation of the graph shows that the frequency of reference does not relate in any consistent manner with window sizes.

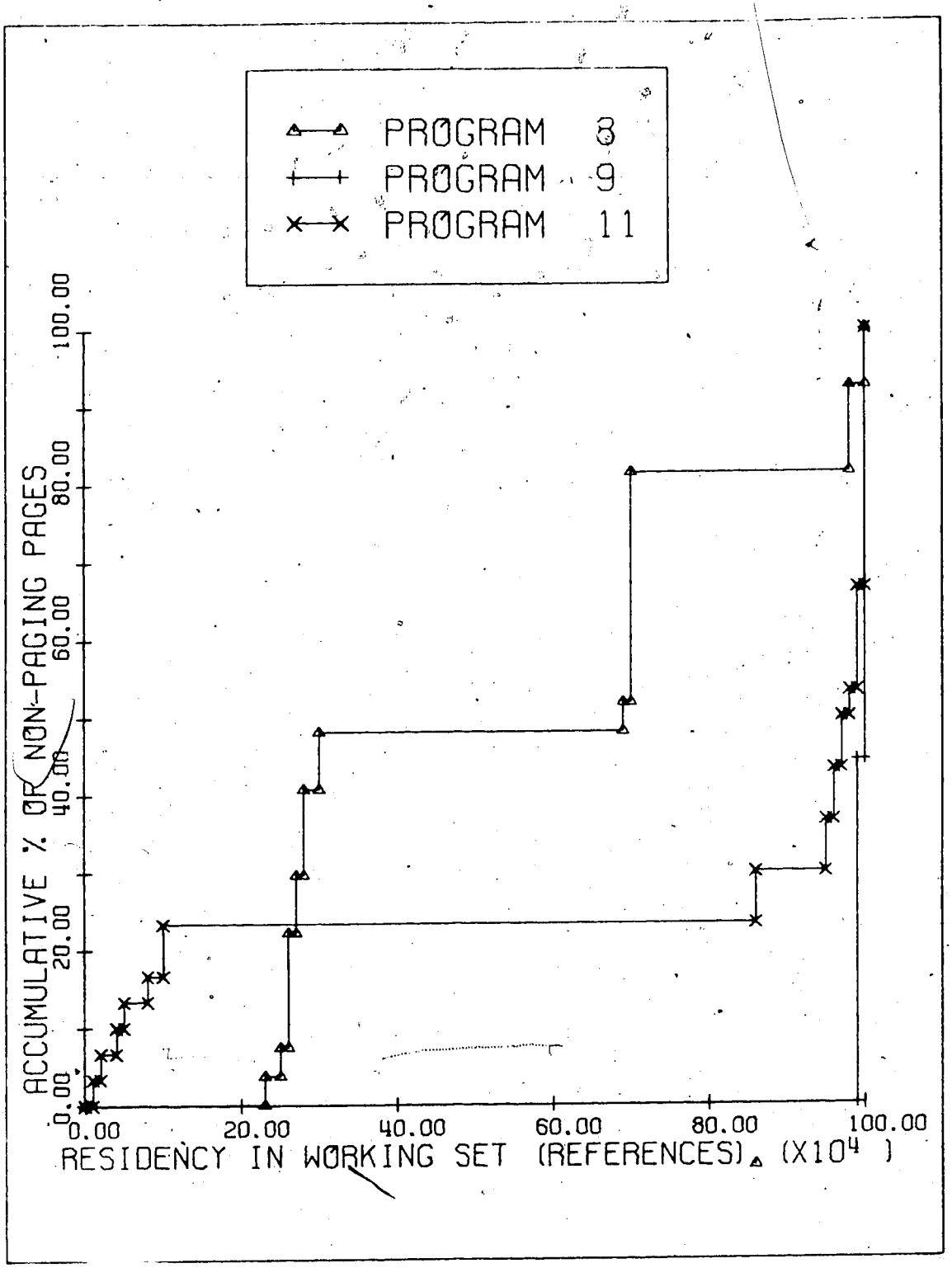


FIG. : EXPECTED RESIDENCY TIME IN THE WORKING SET FOR NON-PAGING PAGES OF PROGRAMS 8, 9, AND 11

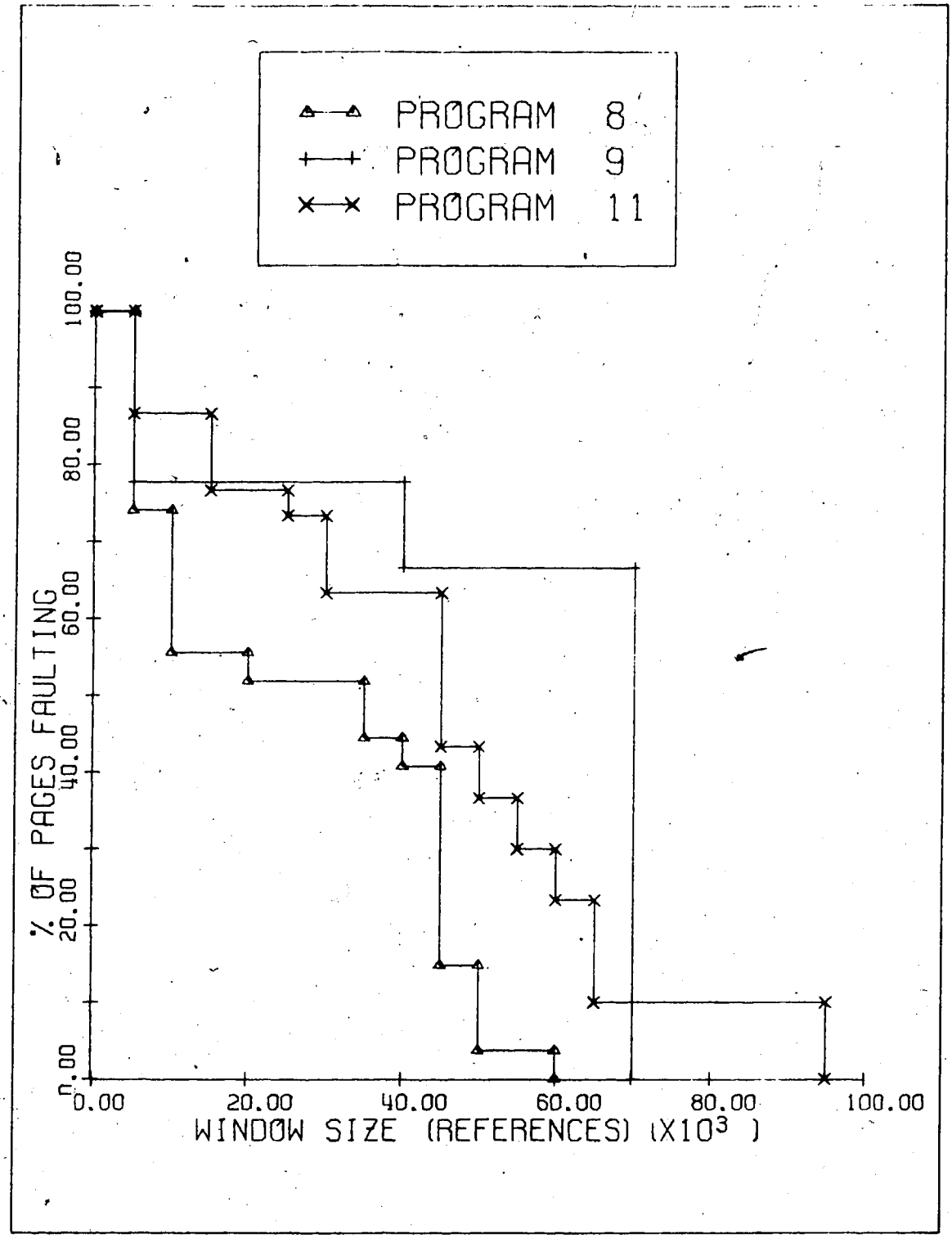


FIG. 27 : RELATIONSHIP BETWEEN WINDOW SIZE AND PAGE FAULT BEHAVIOR FOR NON-PAGING PAGES OF PROGRAMS 8, 9, AND 11

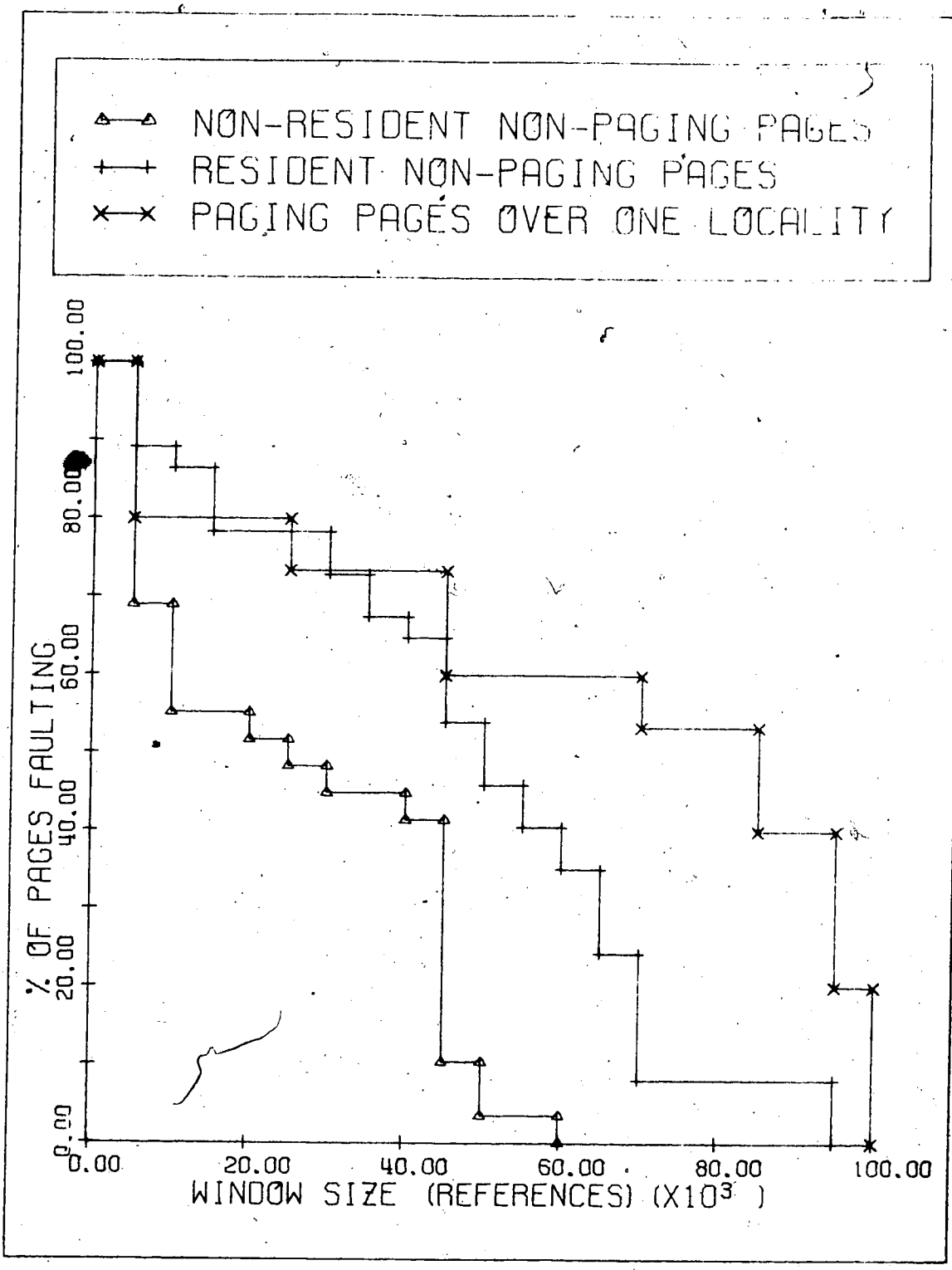


FIG. 28 : RELATIONSHIP BETWEEN WINDOW SIZE AND PAGE FAULT BEHAVIOR FOR THE THREE PAGE TYPES OF PROGRAMS 8, 9, AND 11

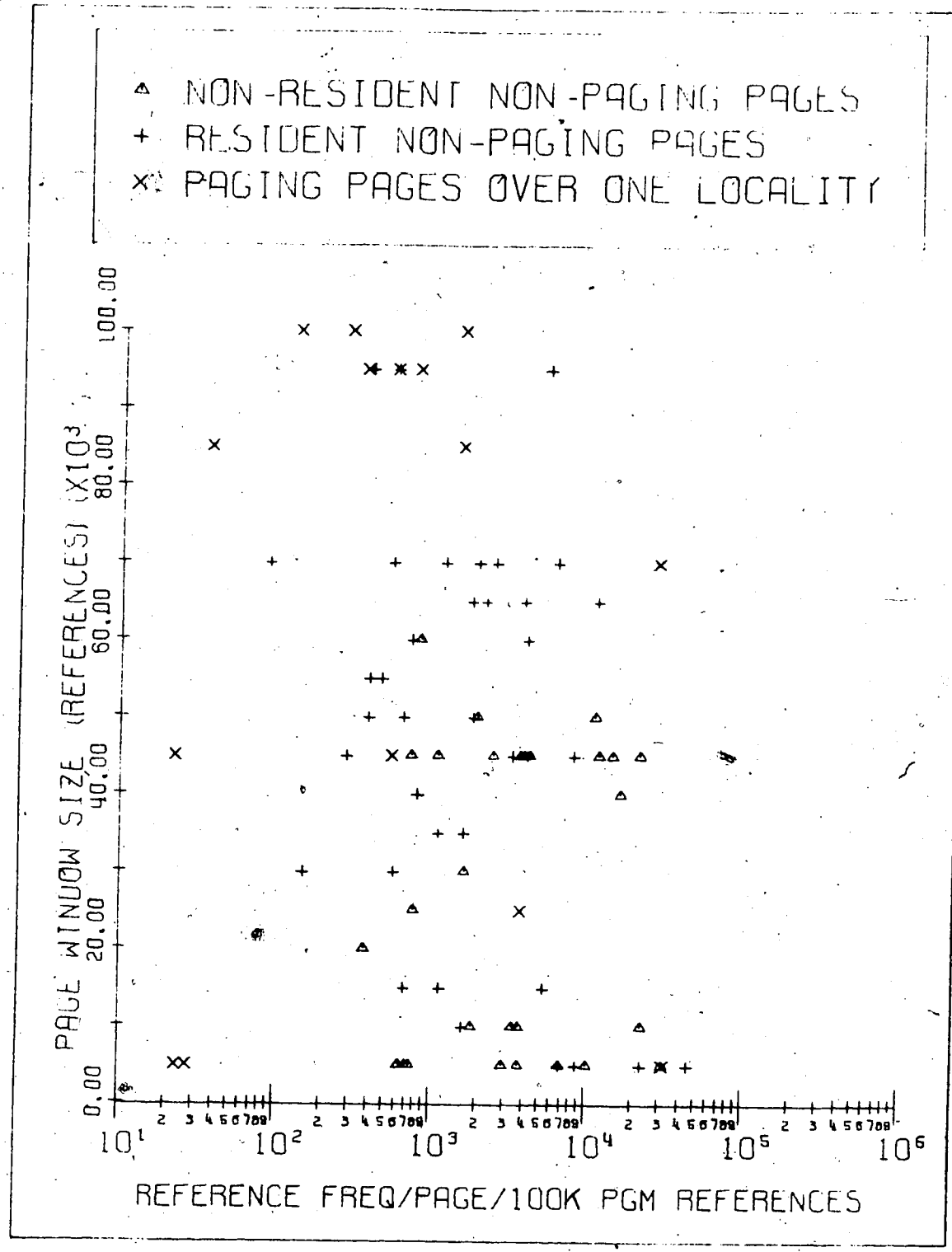


FIG. 29 : MINIMUM WINDOW SIZE FOR NO-FAULT PAGE BEHAVIOR AGAINST FREQUENCY OF REFERENCE/PAGE FOR PROGRAMS 8, 9, AND 11

VII: Conclusion

The assumptions and procedures for controlling the window size parameter when recognizing a working set were described. Included was a description of actual and postulated implementations of the parameter for computer operating systems. Two models of page residency which may be used to determine window sizes were also discussed.

The study of program behavior indicated that the minimum average missing page rate occurred at 83,000 references (given that the largest window size studied was 100,000 references). A more "stable" working set program behavior appeared for window sizes greater than 55,000 references. An exponential fit was applied to the missing page rates of all the programs. The average value of the coefficients A and B was detemined as $.23 \times 10^{-3}$ references and $-.31 \times 10^{-4}$ references, respectively. Distributions of these same parameters could possibly be used to quantatively classify input job streams.

Differences between programs appeared to be greater than any working set similarities induced by similar source code. The execution of different instruction types was not significantly related to the working set characteristics of

programs. The study indicated that neither source nor execution code appeared suitable for adaptive window size control purposes.

The concepts of the Working Set Model were extended to include the behavior of individual pages. The average value of the average working size of a page for the programs studied was .67 pages. Pages were initially divided into two types, "paging" and "non-paging". Given that the paging pages were "in-and-out" of memory an average of 2.8 times per 10^6 references, there was little distinction between the two page types according to their average rate of reference. The non-paging pages were further differentiated into "resident" and "non-resident" types. The non-resident types entered memory only once and tended to cluster in fixed localities of reference size. The savings in memory use by taking advantage of this characteristic proved to be insignificant. Although all three page types showed distinct characteristics in the window size desired for complete memory residency, there appeared to be little relationship between page type, reference activity, and window size desired. Therefore, adaptive windowing at the page level appeared to be either impractical or not possible.

There were several ideas suggested in this study for further research. They were divided into two groups; those relating to computer paging systems in general and those questions relating to the Working Set Model. The initial group of suggestions will be addressed first.

How much awareness of the system should the average programmer have. Given the intricacies of paging systems, efforts spent in improving program techniques for providing better memory utilization may prove beneficial. Should energies be spent in this direction or should memory inefficiencies be "written-off" as part of the cost of a paging installation? Obviously, these questions cannot be answered with a simple yes or no; but deserve more investigation, both from the system design as well as software production point of view.

Two theoretical research problems need further consideration. A model for program behavior is needed which is also faithful to real program internal structure and I/O characteristics. Secondly, a more rigorous investigation of program properties and their relationship to the types of instructions executed is required.

Two practical research problems involving paging

computers deserve further attention. There is a need for additional empirical study of the factors influencing the p and n variables in Prieve's model of page residency. Secondly many computer installations use pageable re-entrant system code available for use by all executing programs. What is the contribution of these pages to the total computer paging load?

A number of research ideas are suggested for the study of the Working Set properties. How rapidly does the working set size $S(t,T)$ actually change? An empirical study of this type will indicate the validity of $\overline{S}(T)$ as an estimator of program memory demands. How much variation is there in the total paging demand on a multiprogramming system? A study in this area will determine the theoretical validity of using either the fixed or variable window size. Does a change in the working set size indicate a change in the working set? Further research should be done to determine if this technique can be used for controlling the size of T. Exponential fits were applied to program missing page curves. Would conic fits prove to be more "correct"? Attempts are made in this study to classify programs according to their working set characteristics. More research should be done to determine if this classification approach is valid. Finally, the study of page working sets

indicated that non-paging pages tend to group in fixed localities of reference. Do those pages having the same locality also tend to group temporally during execution?

The determination of the value of T for a working set implementation is important not only for practical consideration but also for its implications to the model itself. The model appears to suggest that paging is a result of pages entering and leaving a "continuously" running program's working set. However, paging on current systems often appears to be a result of the stop-start nature of program execution in this type of environment. That is, the "loading" and "un-loading" of entire working sets are what primarily constitute a system paging rate, rather than paging from programs running continuously. How can this contradiction be explained?

Figure 30 is an idealized graph of real time against the average total pages allocated to a program. The upper line represents a program under a working set implementation. The lower line represents the same program's page requirements under a traditional LRU environment. The program begins execution at times t0 and t2, and completes its time slice at t1 and t3.

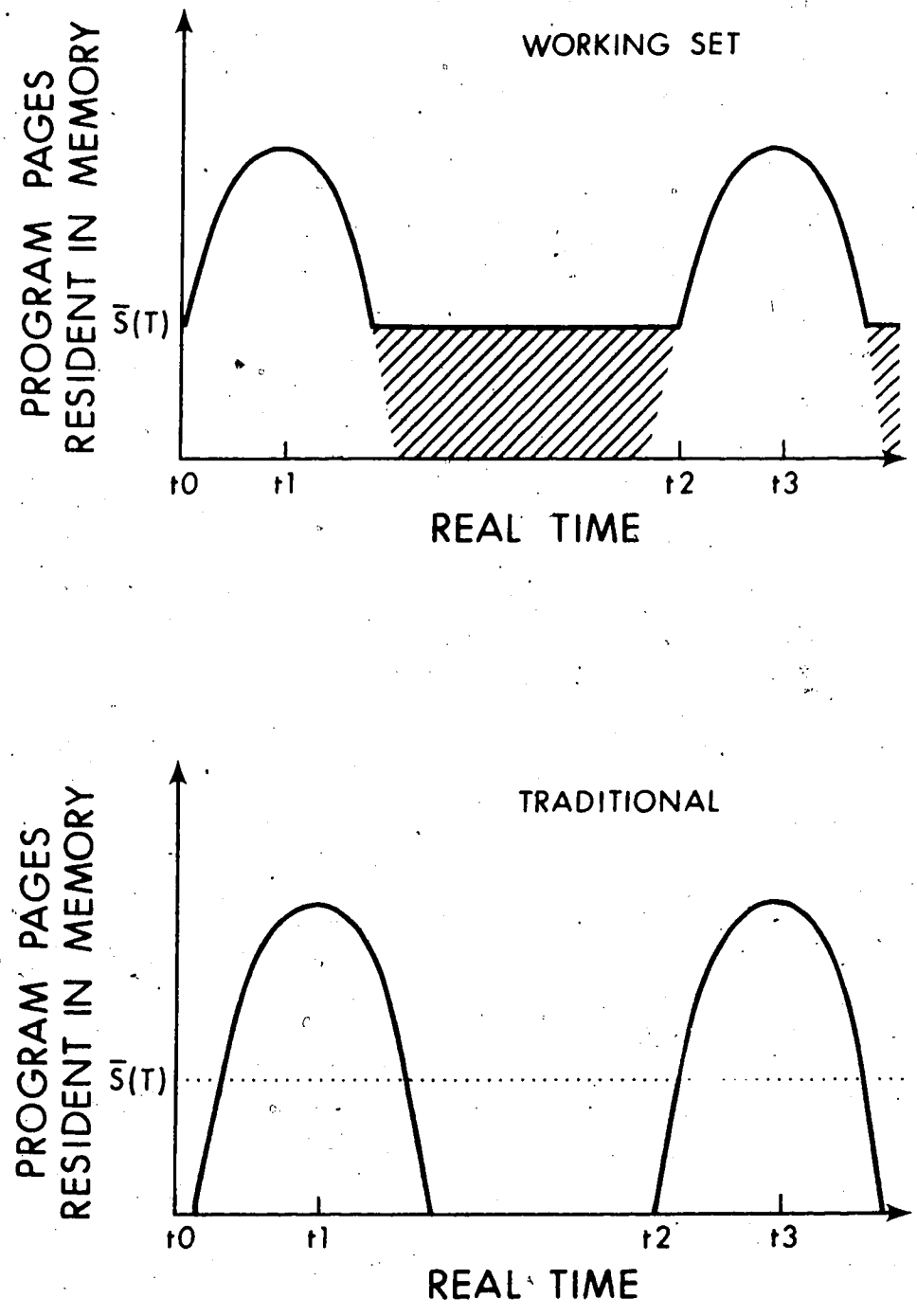


Fig. 30 :- Real Time Program Memory Allocation for Working Set and Traditional Memory Management Schemes

If the system uses a working set approach, the minimum number of pages a program will retain between executions is $\overline{S}(T)$. The humps in the upper curve indicate those pages not currently in $\overline{S}(T)$ but may be re-claimed. This is basically the same situation for the standard memory allocation scheme except that a program may have all its pages removed between executions. Hence, the lower line indicates the stop-start phenomenon where each program execution results in an entire working set to be first loaded and then unloaded. Although the working set approach uses the additional memory indicated by the shadowed regions, it does not require the $2*\overline{S}(T)$ extra pages to be moved for each program execution. The reserving of a program's $\overline{S}(T)$ working set between time slice executions is the working set principle. Since the amount of real memory is restricted, the multiprogramming level of a working set computer will therefore be dependent and indeed controlled by the size of T. The explanation of the contradiction emphasizes the basic difference between paging using a working set approach and when using traditional schemes for memory management.

Unfortunately, three aspects of system operation tend to interfere with the working set principle. When a program exhausts its quantum, it is usually demoted to the ready set, its pages being released for system use. When again

promoted to the running set, its entire working set must again be loaded. However, this can be a relatively "rare" event [R5], unless one or both of the following two aspects are prevalent.

If the system is heavily loaded, it may be necessary to deactivate tasks before their quantum has expired. This may be done in an attempt to maintain response time. The procedure eventually become self-defeating since the influence of the working set principle is dissipated and the initial memory gains made through deactivation become offset by the time consumed in increased program page "loads" and "unloads".

Finally, I/O blocking usually results in jobs being removed from the running set. Although I/O influence may be reduced by buffering, the induced stop-start program behavior cannot be otherwise controlled. Hence, a working set approach to memory management is not nearly as successful in an I/O bound environment as it would be in a more "balanced" system.

BIBLIOGRAPHY

[A1] Aho, A.V., Denning, P.J., Ullman, J.D., "Principles of Optimal Page Replacement", Journal of the ACM, vol. 18, no. 1, 1971, pp. 80–92.

[A2] Alexander, M.T., Time Sharing Supervisor Programs, The University of Michigan Computing Center, May 1969.

[A3] Arden, B.W., Galler, B.A., O'Brien, T.C., Westervelt, F.H., "Program and Addressing Structure in a Time Sharing Environment", Journal of the ACM, vol. 13, no. 1, 1966, pp. 1–16.

[B1] Batson, A., Ju, S., Wood, D., "Measurements of Segment Size", Second Symposium on Operating Systems Principles, Oct. 1969, pp. 25–29.

[B2] Belady, L.A., "A Study of Replacement Algorithms for a Virtual Storage Computer", IBM Systems Journal, vol. 5, no. 2, 1966, pp. 78–101.

[B3] Belady, L.A., Kuehner, C.J., "Dynamic Space Sharing in Computer Systems", Communications of the ACM, vol. 12, no. 5, 1969, pp. 282–288.

[B4] Bovet, D.P., "Memory Allocation in Computer Systems", Clearinghouse For Federal Scientific and Technical Information, AD 670499, June 1968.

[B5] Brawn, B.S., Gustavson, F.G., "Program Behavior in a Paging Environment", Proceedings of the AFIPS 1968 FJCC, vol. 33, pp. 1019–1032.

[B6] Bobrow, D.G., Burchfiel, J.D., Murphy, D.L., Tomlinson, R.S., "Tenex, a Paged Time Sharing System for the PDP–10", Communications of the ACM, vol. 15, no. 3, 1972, pp. 135–143.

[B7] Burroughs Corporation, "Master Control Program for the B5500".

[C1] Coffman E.G., Varian, L.C., "Further Experimental Data on the Behavior of Programs in a Paging Environment", Communications of the ACM, vol. 11, no. 7, 1968, pp. 471–474.

[C2] Comeau, L.W. "A Study of the Effects of User Program Optimization in a Paging System", First Symposium on Operating System Principles, Oct. 1967.

[C3] Corbato, F.J., Vyssotsky, V.A., "Introduction and Overview of the MULTICS System", Proceedings of the AFIPS 1965 FJCC, vol. 27, 1963, pp. 185-196.

[C4] Corbato, F.J., "A Paging Experiment with the Multics System", Rep. MAC-M-384, MIT Project MAC, Cambridge, Mass., 1968, AD 687552.

[D1] Davis, A.R., Personal communications, Nov. 1973-Feb. 1974.

[D2] Denning, P.J., "The Working Set Model for Program Behavior", Communications of the ACM, vol. 11, no. 5, 1968, pp. 323-333.

[D3] Denning, P.J., "Thrashing, Its Causes and Prevention", Proceedings of the AFIPS 1968 FJCC, vol. 33, pp. 915-922.

[D4] Denning, P.J., "Virtual Memory", Computing Surveys, vol. 2, no. 3, 1970, pp. 153-189.

[D5] Denning, P.J., "On Modeling Program Behavior", Proceedings of the AFIPS 1972 SJCC, vol. 40, 1972, pp. 937-943.

[D6] Denning, P.J., Schwartz, S.C., "Properties of the Working-Set Model", Communications of the ACM, vol. 15, no. 3, 1972, pp. 191-198.

[D7] Dennis, J.B., "Segmentation and the Design of Multiprogrammed Computer Systems", Journal of the ACM, vol. 12, no. 4, 1965, pp. 589-602.

[D8] Doherty, W. J., "Scheduling TSS/360 for Responsiveness", Proceedings of the AFIPS 1970 FJCC, vol. 37, 1970, pp. 97-111.

[F1] Fine, G.H., Jackson, C.W., McIsaac, P.V., "Dynamic Program Behavior under Paging", Proceedings 21st National Conferences of the ACM, 1966, pp. 223-228.

[F2] Fotheringham, J., "Dynamic Storage Allocation in the Atlas Computer", Communications of the ACM, vol. 4, no. 10, 1961, pp. 435-436.

[F3] Freibergs, I.F., "The Dynamic Behavior of Programs", Proceedings of the AFIPS 1968 FJCC, vol. 33, 1968, pp. 1163-1167.

[G1] Gary, M.R., Graham, R.L., Ullman, J.D., "Worst Case Analysis of Memory Allocation Algorithms", Proceedings of the 4th Annual Symposium on the Theory of Computing, May 1972,pp. 143-150.

[G2] Greenberg, M.L., "An Algorithm for Drum Management in Time-Sharing Systems," Third Symposium on Operating System Principles, Oct. 1971, pp. 141-148.

[H1] Hatfield D.J., Gerald, J., "Program Restructuring Techniques for Virtual Memory", IBM Systems Journal, vol. 10, no. 3, 1971, pp. 168-192.

[H2] Hatfield, D.J., "Experiments on Page Size, Program Access Patterns and Virtual Memory Performance", IBM Journal of Research and Development, vol. 16, no. 1, 1972, pp. 58-66.

[H2] Hatfield D.J., Gerald, J., "Program Restructuring Techniques for Virtual Memory", IBM Systems Journal, vol. 10, no. 3, 1971, pp. 168-192.

[I1] Iliffe, J.K., Basic Machine Principles, American Elsevien Publishing Co. Inc., N.Y., 1968, pp. 22-25.

[I2] Iliffe, J.K., Jodeit, J.G., "A Dynamic Storage Allocation Scheme", Computer Journal, vol. 5., pp. 200-209.

[I3] IBM System/360 Operating System MFT Guide GC 27-6939-10.

[I4] IBM System/360 Operating System MVT Guide GC 28-6720-4.

[I5] IBM System/360 Model 67 Functional Characteristics GA 27-2719-3.

[K1] Kernighan, B.W., "Optimal Segmentation Points for Programs", Second Symposium on Operating Systems Principle, Oct. 1968, pp. 47-53.

[K2] Kilburn, T., Payne, R.B., Howarth, D.J., "The Atlas Supervisor", Proceedings of the 1961 Eastern Joint Computer Conference, pp. 279-294.

[K3] Kilburn T., Edwards, D.B., Lanigan, M.J., Sumner, F.H., "One Level Storage System", IRE Transactions, EC-11, April 1962, pp. 223-235.

[K3] Knuth, D.E., The Art of Computer Programming, vol. 1, Addison Wesley, Reading, Mass., 1968, pp. 435-455.

[K7] Knuth, D.E., "An Empirical Study of FORTRAN Programs", Software Practice and Experience, vol. 1, no. 2, 1971, pp. 105-134.

[K6] Kuehner, C., Randell, B., "Demand Paging in Perspective", Proceedings of the AFIPS 1968 FJCC, vol. 33, pp. 1011-1018.

[L1] Lewis, P.A.W., Shedler, G.S., "Emperically Derived Micromodels for Sequences of Page Exceptions", IBM Journal of Research and Development, vol. 17, no. 2, 1973, pp. 86-100.

[L2] Lowe, T.C., "Analysis of Boolean Models for Tune-Shared Paged Environments", Communications of the ACM, vol. 12, no. 4, 1969, pp. 199-205.

[L3] Lowe, T.C., "Automatic Segmentation of Cyclic Program Structures Based on Connectivity and Procesion Timing", Communications of the ACM, vol. 13, no. 1, 1970, pp. 3-6.

[M1] Mattson, R.L., Gecsei, J., Slutz, D.R., Traiger, I.L., , "Evaluation Techniques for Storage Hierarchies", IBM Systems Journal, vol. 9, no. 2, 1970, pp. 78-117.

[M2] McKellar, A.C., Coffman, E.G., "Organizing Matrices and Matrix Operations for Paged Memory Systems", Communication of the ACM, vol. 12, no. 3, 1969, pp. 153-165.

[M3] Morris, J.B., "Demand Paging Through Utilization of Working Sets on Maniac II," Communications of the ACM, vol. 15, no. 10, 1972, pp. 867-872.

[M4] "*TALLY", Michigan Terminal System: Public File Descriptions, vol. 2, Jan. 1972, pp. 331.

[M5] Michigan Terminal System : MTS and Computing Services, The University of Alberta Computing Services Dept., Edmonton, Alta., vol. 1, Nov. 1971.

[M6] Michigan Terminal System : Public File Descriptions, The University of Alberta Computing Services Dept., Edmonton, Alta., vol. 2, Jan. 1972.

[O1] O'Neill, R.W., "Experiences Using a Timesharing Multiprogramming System with Dynamic Address Relocation Hardware", Proceedings of the AFIPS 1967 SJCC, vol. 30, pp. 611-621.

[O3] Oppenheimer, G., Weizer, N., "Resource Management for a Medium Scale Time Sharing Operating System", Communications of the ACM, vol. 11, no. 5, 1968, pp. 313-322.

[O2] Organick, E.I., The Multics System, The MIT Press, Cambridge, Mass., 1972.

[P1] Pankhurst, R.J., "Program Overlay Techniques", Communications of the ACM, vol. 11, no. 2, 1968, pp. 119-125.

[P2] Parmelee, R.P., Peterson, T.I., Tillman, C.C., Hatfield, D.J., "Virtual Storage and Machine Concepts", IBM Systems Journal, vol. 11, no. 2, 1972, pp. 99-130.

[P3] Peters, C.B., "Experiments in Automatic Paging", vol. 1, Informatics Inc., Nov. 1971, AD 734253.

[P4] Prieve, B.G., "Using Page Residency To Select the Working Set Parameter," Communications of the ACM, vol. 16, no. 10, pp. 619-620.

[R1] Ramamoorthy, C.V., "The Analytic Design of a Dynamic Look Ahead and Program Segmenting System for Multiprogrammed Computers", Proceedings of the 21st National Conference of the ACM, 1966, pp. 229-239.

[R2] Randell, B., Kuehner, C.J., "Dynamic Storage Allocation Schemes", Communications of the ACM, vol. 11, no. 5, 1968, pp. 297-305.

[R3] Randell, B., "A Note on Storage Fragmentation and Program Segmentation", Communications of the ACM, vol. 12, no. 7, 1969, pp. 365-369.

[R4] Rodriguez-Rosell, J., "Experimental Data on How Program Behavior Affects the Choice of Scheduler Parameters", Third Symposium on Operating System Principles, Oct. 1971, pp. 156-163.

[R5] Rodriguez-Rosell, J., "The Design, Implementation, and Evaluation of a Working Set Dispatcher", Communications of the ACM, vol. 16, no. 4, 1973, pp. 247-253.

[R6] Rosen, S., Programming Systems and Languages, McGraw-Hill Book Co., 1967.

[R6] Rosen, S., "Electronic Computers: A Historical Survey", Computing Surveys, vol. 1, no. 1, March 1969, pp. 7-36.

[R6] Rosin, R.F., "Supervisory and Monitor Systems", Computing Surveys, vol. 1, no. 1, March 1969, pp. 37-54.

[S1] Salton, G., Automatic Information Organization and Retrieval, McGraw-Hill Book Co., 1968, pp. 135-139.

[S2] Sayre, D., "Is Automatic 'Folding' of Programs Efficient Enough to Displace Manual?", Communications of the ACM, vol. 12, no. 12, 1969, pp. 656-660.

[S3] Shemer, J.E., Gupta, C., "On the Design of Bayesian Storage Allocation Algorithms for Paging and Segmentation", IEEE Transactions on Computers, July 1969, pp. 644-651.

[T1] Thorington, J.M., Irwin, J.D., "Adaptive Replacement Algorithms for Use in Paged Memory Computer Systems", Project Themis: Information Processing, Technical Report AV-T-18, July 1971.

[T2] Thorington, J.M., Irwin, J.D., "An Adaptive Replacement Algorithm for Paged Memory Computer Systems", IEEE Transactions on Computers, vol. C-21, no. 10, 1972, pp. 1053-1061.

[V2] Varian, L.C., Coffman, E.G., "An Empirical Study of the Behavior of Programs in a Paging Environment", First Symposium on Operating System Principles, Oct. 1967.

[V3] Ver Hoef, E., "Automatic program segmentation based on Boolean connectivity", Proceedings of the AFIPS 1971 SJCC, vol. 39, pp. 491-495.

[W1] Weizer, N., Oppenheimer, G., "Virtual Memory Management in a Paging Environment," Communication of the ACM, vol. 11, no. 5, May 1968, pp. 313-322.

[W2] Winder, R.O., "A Data Base for Computer Performance Evaluation", Computer, Mar. 1973, pp. 25-29.

APPENDIX 1: PROPERTIES OF THE EXTENDED WORKING SET MODEL OF PROGRAM BEHAVIOR

All of these proofs are analogous to those given by Denning in "Properties of the Working Set Model" [D6].

1. P10: $e(i) = \bar{S}(1,i) \ldots \leq \bar{S}(T,i) \leq \bar{S}(T+1,i) \leq 1$

   Property P10, states that the average working size for page i is non-decreasing and bounded above and below. It follows from:

   a) $W(t,T,i)$ is a subset $\leq W(t,T+1,i)$

   b) $S(t,T+1,i) \leq 1$

   c) the definition of $\bar{S}(T)$

2. P11: $\bar{S}(T+1,i) - \bar{S}(T,i) = \bar{M}(T,i)$

   Property P11 states that the "slope" of $\bar{S}(T,i)$ is the missing-page rate for page i, as indicated by the following:

$$W(t+1,T+1,i) = W(t,T,i) + \{r_{t+1}\}$$

$$S(t+1,T+1,i) = S(t,T,i) + \Delta_t(T,i)$$

$$\lim_{K\to\infty} \frac{1}{K} \sum_{t=1}^{K} S(t+1,T+1,i) = \lim_{K\to\infty} \frac{1}{K} \sum_{t=1}^{K} S(t,T,i) + \lim_{K\to\infty} \frac{1}{K} \sum_{t=1}^{K} \Delta_t(T,i)$$

   Therefore, $\bar{S}(T+1,i) = \bar{S}(T,i) + \bar{M}(T,i)$

   which is the same as P11.

3. P12: $0 \ldots \leq \bar{M}(T+1,i) \leq \bar{M}(T,i) \ldots \leq \bar{M}(0,i) = e(i)$

The upper and lower bounds are established by the definition of $\bar{M}(T,i)$. It remains to be shown that

$$\triangle_t(T+1,i) \leq \triangle_t(T,i). \qquad (1)$$

If (1) is true, then both sides can be summed t=0 to K-1, and divided by K. If the limit of K tending to infinity is taken, then the result by definition is $\bar{M}(T+1,i) \leq \bar{M}(T,i)$ for all T.

Therefore, when $\triangle_t(T+1,i) = 0$, then (1) is true. When $\triangle_t(T+1,i) = 1$, $r_{t+1} = i$ is not in $W(t,T+1,i)$. But $W(t,T,i) \leq W(t,T+1,i)$, therefore $\triangle_t(T,i) = 1$, as well. Thus, $\triangle_t(T+1,i) \leq \triangle_t(T,i)$ implying by definition that $\bar{M}(T+1,i) \leq \bar{M}(T,i)$.

4. P13: $\bar{M}(T,i) = e(i) - e(i)F(T,i) = \sum_{y>T} e(i)f(y,i)$

Property P13 states that $\bar{M}(T,i)$ can be regarded as the probability that the interreference distance $x_i > T$, the window size. Because of the definition of $\triangle_t(T,i)$ the sum

$$\frac{1}{K}\left\{\left[\sum_{t=0}^{K-1} \triangle_t(T,i)\right] - 1\right\}$$

can be thought of as the fractional number of

occurrences of the event $x_i > T$ for page i (The -1 term represents the removal of the first reference to page i since it does not really represent an interreference gap). Since, by definition

$$f(y,i) = F(y,i) - F(y-1,i)$$

then

$$e(i) - e(i)F(T,i) = \sum_{y>T} e(i)f(y,i)$$ which represents

the fractional number of occurrences of the event y>T. Thus letting K tend to infinity, we have

$$\sum_{y>T} e(i)f(y,i) = \lim_{K\to\infty} \frac{1}{K} \left[\sum_{t=0}^{K-1} \Delta(T,i)_t - 1 \right]$$

$$= \lim_{K\to\infty} \frac{1}{K} \left[\sum_{t=0}^{K-1} \Delta(T,i)_t \right]$$

$$= \bar{M}(T,i) \text{ by definition}$$

This implies that $\bar{M}(T,i)$ can be thought of as the probability that page i will have an interreference distance greater than T.

5. P14: $\bar{M}(T+1,i) - \bar{M}(T,i) = -e(i)f(T+1,i)$

The missing page rate for page i at time = T for window size has a "slope" for its curve equal to the negative of the interreference density for time = T+1.

$$\bar{M}(T+1,i) - \bar{M}(T,i) = \sum_{y>T+1}^{---} e(i)f(y,i) - \sum_{y>T}^{---} e(i)f(y,i)$$

$$= -e(i)f(T+1,i)$$

6. P15: $\bar{S}(T,i) = \sum_{z=0}^{T-1} \bar{M}(z,i) = \sum_{z=0}^{T-1} (e(i) - e(i)F(z,i))$

$$= \sum_{z=0}^{T-1} \sum_{y>z}^{---} e(i)f(y,i)$$

This property shows how to calculate $\bar{S}(T,i)$.

From P13 and the following result

$$\bar{S}(T,i) = \bar{M}(T-1,i) + \bar{S}(T-1,i)$$

the calculation of $\bar{S}(T,i)$ is as follows:

$$\bar{S}(T,i) = \sum_{z=0}^{T-1} \bar{M}(z,i)$$

$$= \sum_{z=0}^{T-1} (e(i) - e(i)F(z,i))$$

$$= \sum_{z=0}^{T-1} \sum_{y>z}^{---} e(i)f(y,i)$$

7. P16: $\frac{\bar{S}(T+1,i) + \bar{S}(T-1,i)}{2} \leq \bar{S}(T,i)$

This property shows that the $\bar{S}(T,i)$ curve is monotonically non-decreasing and non-positively accelerated. By P12, $\bar{M}(T+1,i) \leq \bar{M}(T,i)$. Therefore, using this result and P11 gives:

$$\bar{S}(T+1,i) - \bar{S}(T,i) \leq \bar{S}(T,i) - \bar{S}(T-1,i)$$

which proves P16.

8. P17: $\lim_{T\to\infty} \bar{S}(T,i) = 1$

This property shows that the working size of a page is bounded above by 1, if the page i is recurrent. As T tends to infinity, a recurrent page of a program is any page referenced more than once. Consider the following:

$$\bar{S}(T,i) = \sum_{z=0}^{T-1} \sum_{y>z} e(i) f(y,i)$$

Since e(i) is independent of summation, then

$$\bar{S}(T,i) = e(i) \sum_{z=0}^{T-1} \sum_{y>z} f(y,i)$$

$$= e(i) \sum_{z=0}^{T-1} z f(z,i)$$

$$\lim_{T\to\infty} \bar{S}(T,i) = e(i) \lim_{T\to\infty} \sum_{z=0}^{T-1} z f(z,i)$$

$$\text{But } \lim_{T\to\infty} \sum_{z=0}^{T-1} z f(z,i) = \bar{x}_i ,$$

where $\bar{x}_i$ is the expected interreference distance for page i (assuming stationarity). Therefore, the following holds

$$\lim_{T\to\infty} \bar{S}(T,i) = e(i)\bar{x}_i$$

But $e(i) = \frac{1}{\bar{x}_i}$

$$\lim_{T\to\infty} \bar{S}(T,i) = 1$$

9. P18: $\lim_{T\to\infty} \bar{M}(T,i) = 0$

By property P13, $\bar{M}(T,i) = e(i) - e(i)F(T,i)$. By definition, as T tends to infinity, then $F(T,i)$ tends to 1. Therefore, P18 is true.

10. P19: $\lim_{T\to\infty} \sum_{i=1}^{n} \bar{S}(T,i) = \lim_{T\to\infty} \bar{S}(T)$

This property states that the limit as T tends to infinity of the sum of the individual average working set sizes of the pages equals the limit of the working set size of the entire process.

$$\lim_{T\to\infty} \bar{S}(T) = \lim_{T\to\infty} \sum_{z=0}^{T-1} \sum_{y>z} f(y)$$

By definition, $f(y) = \sum_{i=1}^{n} e(i)f(y,i)$

$$\lim_{T\to\infty} \sum_{z=0}^{T-1} \sum_{y>z} \left(\sum_{i=1}^{n} e(i)f(y,i) \right)$$

$$= \lim_{T\to\infty} \sum_{i=1}^{n} \left(\sum_{z=0}^{T-1} \sum_{y>z} e(i) f(y,i) \right) = \lim_{T\to\infty} \sum_{i=1}^{n} \bar{S}(T,i)$$

Therefore, P19 is proven.

APPENDIX 2: ALGORITHMS USED FOR CALCULATING THE WORKING SET CURVES

I. Calculating the Working Set Curves for a Program

Define the following integer vectors with the indicated dimensions:

1. C(L+1), where L is the length, in references, of the largest window considered. Initially, all elements of C equal 0.
2. Time(n), where n is the number of different recurring pages in the reference string. For every i, Time(i) will contain the time of the most recent reference to page i. Initially, Time(i) equals -K for all i, so that the first reference to each page may be recognized.

Define the following additional variables:

3. S, represents the value of $\bar{S}(T)$.
4. M, represents the value of $\bar{M}(T)$.
5. $r_1, r_2, r_3, \ldots, r_K$, be the elements of the reference string of length K.
6. V, is a fractional value representing the ratio of the total number of references within the actual problem program code, to K, the total number of references in the reference string. The value of V

will equal 1.0 only if all the references in the string are within the problem code. This procedure allows the algorithm to calculate curves for only the problem program pages, yet retains the timing influence of references to "system" code.

7. G will contain the count of the number of references to problem program pages. Initially, G = 0.

The algorithm to calculate $\bar{S}(T)$ and $\bar{M}(T)$ for window sizes $1 \leq T \leq L$ for an entire program of n page follows:

```
t <-- 0;
while t<K
     begin t <-- t + 1;
           i <-- r_t ;
           if (i = problem program reference) then
               begin G <-- G + 1
                     j <-- t-Time(i);
                     if (j > K) then j <-- 1;
                     if (j > L) then j <-- L+1;
                     C(j) <-- C(j) + 1;
                     Time(i) <-- t;
               end;
     end;
V <-- G/K;
M <-- V;
S <-- 0;
T <-- 1;
while T≤L
     begin S <-- S + M;
           M <-- M - (C(T)/K);
           T <-- T + 1;
     end;
```

II. Calculating the Working Set Curves for an Individual Page

The definition of the vectors and variables are the same as for I. above except for the following:

1. Time is now a single variable which will contain the time of the most recent reference to the page being considered. Initially, Time equals -K, so that the first reference to page i can be recognized.
2. S, represents the value of $\bar{S}(T,i)$
3. M, represents the value of $\bar{M}(T,i)$
4. D, will contain the count of the number of references to the page being considered. Initially, D equals 0.
5. E, will replace the variable V. E takes the fractional value representing the ratio of D:K.

The algorithm for calculating the $\bar{S}(T,i)$ and $\bar{M}(T,i)$ values for window sizes $1 \leq T \leq L$ for a page p follows:

```
t  <--  0;
while t<K
    begin t  <--  t + 1;
          i  <--  r_t ;
          if (i = p) then
              begin D  <--  D + 1;
                    j  <--  t - Time;
                    if (j > K) then j  <--  1;
                    if (j > L) then j  <--  L+1;
                    C(j)  <--  C(j) + 1;
                    Time  <--  t;
              end;
    end;
E  <--  D/K;
M  <--  E;
S  <--  0;
T  <--  1;
while T≤L
    begin S  <--  S + M;
          M  <--  M - (C(T)/K);
          T  <--  T + 1;
    end;
```

APPENDIX 3: THE WINDOW SIZE OF MTS

The operating system used on the University of Alberta's IBM 360/67 computer is called the Michigan Terminal System or MTS. The multiprogramming supervisor used in the operating system is known as UMMPS. The operating system uses the concept of virtual memory for the problem program and non-resident system address space. The virtual memory is organized into 1024 word blocks called pages. Within the machine, there are 256 page frames of real memory of which approximately 150 are used for paging purposes. A typical load on the system is from thirty to forty terminal jobs, three batch tasks, plus approximately fourteen non-MTS jobs (of which only four are virtual and contribute very few pages to the total virtual memory). The total virtual memory runs typically between 1200-1400 pages.

Data was gathered on the characteristics of programs executing on the MTS system on two occasions; November 9/73, 10:44-14:22, and November 16/73, 13:36-17:06. The two runs indicated an average of 1209 and 1088 virtual pages on the system, respectively. For the remainder of the appendix, each result determined from the data runs will be given as two values; the first for Nov. 9, and the second value, for

Nov. 16. The data runs revealed an average of 31 and 37 "active" programs on the system. The virtual memory associated with these "active" programs is typically 800 pages, but for the data runs this amount averaged 991 and 927 pages. Although there are 150 real pages for paging use, the number of real pages allocated to the active programs usually averages 110 pages. Over the two data runs, this amount had an average value of 107 and 109 pages.

In comparison to programs executed under MTS, how typical are the fifteen program executions used in this study? The average program size determined from the MTS data is approximately 32 and 25 pages. However, this study is concerned only with those pages which were in the user segment (#5). Thus, the memory contributed by the I/O handlers and device support routines should not be considered. At the time of the data runs, MTS had a total of 356 virtual pages associated with the terminal device support routines. Also, the amount of memory allocated to each job for I/O purposes (buffers & handlers) is typically four pages. Since pageable system support code is not included in the total virtual memory count, the average size of a program executing under MTS can be determined by the following:

	November 9	November 16
Total virtual memory	= 1209	1088
- DSR code	= -356	-356
- I/O code buffers	= -4*31 = -124	-4*37 = -148
= Total user virtual memory	= 729 pages	584 pages

Therefore, average user virtual memory per program = 23.5 and 15.8 pages, respectively.

It is noted that both average program sizes are less than 28, the average number of pages used per program in the study. However, Friebergs [F3] found that 50% of all jobs in one university computing environment used less than six pages. If, for purposes of this study, the fifteen program runs are weighted to indicate this balance of small programs, then the average number of pages accessed per program can be reduced to approximately the respective average MTS program sizes. This balancing is accomplished by considering that 30% and 65% of the total programs executed during the MTS data runs are the same as program number 14 and those remaining are typical of the other fourteen programs studied. Then using the exponential curve representing the average missing page rate of each program, and the same weighting, an average missing page rate curve is produced for the MTS system. The value of the

coefficients for the curve for the two MTS data runs are A = $.178 \times 10^{-3}$ and $.104 \times 10^{-3}$ and B = $-.311 \times 10^{-4}$ and $-.302 \times 10^{-4}$. MTS typically pages at a rate of 70-90 pages per second. Because of a design characteristic of MTS operation, 2/3 of this paging consists of writes to the paging drum. This indicates an average missing page rate of 23-30 pages per second. The two data runs have also revealed that 8 and 5, respectively, are the average number of tasks on the operating system CPU queue that are ready for service. If the queue length is defined as the multiprogramming level, then the average paging rate per program has a range of 23-30/(CPU queue length) pages/second. Using this average paging rate range and the exponential curves for the average missing page rate just determined, t is found to have a range of 124,000-133,000 and 94,500-103,250 references to provide the average paging results per program. Since t is the window size, then MTS has an effective window size range of .095 to .133 seconds (assuming one reference takes an average of one microsecond).