MAC TR-127

AN EXPERIMENTAL ANALYSIS OF PROGRAM REFERENCE PATTERNS IN THE MULTICS VIRTUAL MEMORY

Bernard S. Greenberg

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2095 which was monitored by ONR Contract No. N00014-70-A-0362-0006

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

AN EXPERIMENTAL ANALYSIS OF PROGRAM REFERENCE PATTERNS IN THE MULTICS VIRTUAL MEMORY

by

Bernard Stewart Greenberg

Submitted to the Department of Electrical Engineering
on January 31, 1974 in partial fulfillment of the
requirements for the Degree of Master of Science.

ABSTRACT

This thesis reports the design, conducting, and results of an experiment intended to measure the paging rate of a virtual memory computer system as a function of paging memory size. This experiment, conducted on the Multics computer system at M.I.T., a large interactive computer utility serving an academic community, sought to predict paging rates for paging memory sizes larger than the existent memory at the time. A trace of all secondary memory references for two days was accumulated, and simulation techniques applicable to "stack" type paging algorithms (of which the least-recently-used discipline used by Multics is one) were applied to it.

A technique for interfacing such an experiment to an operative computer utility in such a way that adequate data can be gathered reliably and without degrading system performance is described. Issues of dynamic page deletion and creation are dealt with, apparently for the first reported time. The successful performance of this experiment asserts the viability of performing this type of measurement on this type of system. The results of the experiment are given, which suggest models of demand paging behavior.

THESIS SUPERVISOR: Jerome Howard Saltzer
TITLE: Associate Professor of Electrical Engineering

Acknowledgements

I wish to thank the M.I.T. Information Processing Center for the use of their machine and time-sharing service as live subjects for my experiments, and for the resources necessary to develop some of the necessary software.

I wish to thank Deborah Cohen for the typing of this thesis, and Muriel Webber for her superlative preparation of the diagrams herein, particularly the wondrous block-and-pointer diagram in Appendix A. Both of them have gone far beyond the call of duty in bringing this document to completion.

I wish to thank my fellow graduate students at Project MAC, particularly David Clark, Jerry Stern, Lee Scheffler, and Douglas Hunt, for all nature of help and inspiration along the way, and invaluable suggestions and insights.

I wish to thank Steven H. Webber of Honeywell Information Systems, Inc., for my apprenticeship in the skills of the Multics supervisor which were so necessary for this experiment.

I wish to thank Professor Michael D. Schroeder for reviewing many early versions of this thesis, and taking an active interest in its progress.

Finally, I wish to thank my thesis supervisor, Professor Jerome H. Saltzer, for the conception of this thesis, and the research leading up to it. From the very beginning, he has guided this work, in a very real sense an extension of his own, and with a keen sense of what was relevant and what was not, shaped the finished thesis. I thank him for his personal commitment of time and energy to this thesis, and helping me through many problematic areas within it. Without him, this thesis would not have been possible.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2095, and was monitored by ONR under Contract No. N00014-70-A-0362-0006.

Table of Contents

Table of Contents of Appendix A

List of Figures

Introduction

1. Brief Statement of the Problem

In this thesis, we describe and report the results of an experiment designed to predict the performance of automatically managed multilevel memory systems for a previously unexplored range of primary memory sizes.

2. Summary of Result

We have developed techniques for predicting memory system performance on an operative computer utility, utilizing an automatically managed multilevel virtual memory. Based upon established theoretical techniques, we have developed techniques to extract the necessary data from a computer utility functioning under a live load. In doing so, we considered problems of dynamic creation and deletion of pages which apparently have not been dealt with previously. The viability of these techniques was demonstrated by performing several measurements.

Using these techniques, we have found that, on the measured system, the rate of accesses to data outside of primary memory decreased drastically as primary memory size is increased above 2×10^8 bits (6 million 36-bit words, or 24 megabytes). We have found that the mean time between these accesses, as a function of primary memory size was best approximated by a function of at least the second order, and possibly exponential. Previous research on the system under consideration showed a linear function to hold for primary memory size up to 1.3×10^8 bits (4 million 36-bit words, or 16 megabytes) (S1). Although these results do not attempt to characterize Multics, we believe that they are reasonably representative of the observed class of user behavior.

3. Summary of the Work of this Thesis

By means of an experiment on the Multics computer system (B2), running on the Honeywell 645 at M.I.T., we have arrived at measurements of the predicted reference rates to secondary memory for hypothetical extensions of primary memory. These measurements were made on an actual user load, the M.I.T. community, and not any sort of benchmark or test load. From these measurements, models of program behavior in LRU*-managed storage hierarchies can be derived. We suggest here one such model.

The essential technique for deriving these predictions from such measurements is known in the literature (C1,C2) as the "extension problem". It is based upon the properties of a class of memory management algorithms known as "stack algorithms" (M1), which include LRU. Using these properties, we were able to simulate the operation of the LRU algorithm for larger primary memory sizes than the actual one present for the identical user load. The input to this simulation was a history of all references to data outside of primary memory, specifically, on disk, during the period of measurement. It is a property of the stack algorithms that one measurement and simulation can be used to predict secondary memory reference rates for all primary memory sizes.

The work reported in this thesis is significant because it is both the first measurement of this type on a paged, segmented, multiprogrammed computer system which has been reported, and an extension of our range of

*LRU, for Least Recently Used - a memory management policy whereby the least recently used data is moved to slower memory when space is needed in faster memory.

knowledge of the so-called "headway" function which we have described above. Previous measurements of this function (S1) involved other techniques, and only investigated it for primary memory sizes of up to 1.3 x 10^8 bits. Our measurements explored regions approaching 4 x 10^8 bits. Although there is no inherent limit on the range which could in principle be explored by our techniques, the limitation of our explorations is due only to the noteworthy fact that over a day's running of the experiment, no more than 4 x 10^8 bits of information were referenced more than once by the M.I.T. community.

The significance of the actual resulting measurement is twofold: First, it provides an example of typical behavior for the measured system. Second, it suggests more general models of program behavior.

4. Structure of this Thesis

Chapter 1 discusses the concepts of paging and virtual memory. We provide justification for the types of statistics and models we seek and describe how to use them in performance predictions. We discuss previous research in this area, and provide a more detailed statement of the novelty of this thesis.

Chapter 2 describes the experiment. We describe the relevant featrues of the so-called "stack" algorithms (M1), and the extension problem. We discuss the problems of adapting this type of experiment to the multilevel memory system of Multics. We describe the difficulties in performing this experiment on an operating computer utility, and the solutions we adopt.

Chapter 3 gives the results of the experiment. The results are presented graphically, and we suggest their interpretation. We analyze these results, and provide a detailed error analysis.

Chapter 4 is a summary of the work done. We suggest future directions for research, and pose some of the questions left unanswered by this thesis.

There are three appendices.

Appendix A is an extremely detailed description of the Multics paging control algorithm, as it was at the time of the experiment. We describe it on several levels, allowing comprehension by the reader on whichever one he chooses. This background is useful for full comprehension of certain design decisions in the planning of the experiment. It is also the first publication of this algorithm at this level of detail (Corbató (C4) provides a less detailed discussion).

Appendix B describes how the actual events of Multics memory management were mapped into the idealized events of theoretical interest to the experiment. We describe the modifications and the interface to the Multics supervisor necessary for this experiment. We assume that the reader has some comprehension of the previous appendix.

Appendix C is a graphical presentation of user load, idle time, and paging overhead on the Multics system on the days of the experiment. These figures were derived from routine metering performed by the administration of the M.I.T. Information Processing Center.

Chapter 1

1.1 Memory Performance Prediction as a Goal

As digital computer systems have increased in size and complexity since their inception almost twenty years ago, so have the memory architectures required to support increasingly advanced applications and systems. What is more, progress in memory technology has created a plethora of memory media, ranging over a wide gamut of costs, speeds, and properties. The desire for increased throughput, and in real-time systems, the desire for quick response, create a need for the fastest memory technology available. The fastest media, however, are almost always the most expensive on a cost-per-bit basis. Thus, for a given computer system to achieve or approach desired goals of memory access speed within a given economic constraint, it becomes useful for memory systems consisting of varying amounts of mixed memory technologies to be used in one installation.

Most computers of the past twenty years have used magnetic core as their main, or primary memory. That is to say, the processor was capable of fetching data and instructions only from core memory. Further memory demands were met by the use of tapes, disks, and other bulk media, whose contents could be transferred in or out of selected areas of primary memory by explicit program request. Most of the programs and operating systems designed for this type of architecture allocated these areas for input/output transfers in fixed, specific regions of primary memory. When programs could not fit in their entirety in primary memory, they were divided into independent pieces, or overlays, which were transferred in and out of primary memory essentially at their own discretion.

In the last few years, a strategy known as virtual memory has achieved popularity. With this scheme, programs are allowed, effectively, to reference data or instructions in primary memory or on any secondary storage device in an identical manner, creating the impression of a very large, or in some cases, conceptually infinite primary memory. References to secondary memory cause software intervention, signalled by specialized hardware, which results in selected code or data fragments being read into primary memory. Clearly, this implies replacement of some other code or data currently in primary memory, and in order to facilitate this task, such systems divide all primary and secondary storage into equal-sized areas, called blocks, or page frames. Information in the system is divided into pages, which may reside in various page frames at various times. This implementation of virtual memory is thus known as demand paging, as pages are read in on demand, i.e., when referenced. The selection of appropriate pages in primary memory for replacement is a critical issue, and is still a basis for much further study.

A page fault, as the software-assisted fetch of a page not in primary memory is called, represents lost time. The time required to access and transfer the copy of the page on secondary storage is time during which the requesting program may not run. The time that a processor must spend in page fault software, deciding on an appropriate page to replace, is a system overhead, which does not contribute to the progress of users' programs. Multiprogramming, a scheme almost universally used on medium and large scale systems, allows processors to serve one user's program while another's is suspended, say for a page fault. But even here, most systems limit the degree (number of simultaneously runnable users) of multiprogramming, and page faults can lead to a situation where a processor spends an undesirably large fraction of its time sitting idle, accomplishing no function at all. Furthermore, the primary memory space occupied by all of the faulting program is unusable by any program for the duration of the transfer. Thus, the minimization of page faults in a virtual memory system is extremely desirable. It is an important function of the page-replacement algorithm, as the procedure which selects pages for replacement at page-fault time is known, to attempt to minimize the number of page faults in the forseeable future. These decisions are usually made with information gleaned from observation of page usage in the immediate past, occasional knowledge of predicted page usage patterns, and some general models of program behavior.

Many page-replacement algorithms have thus been designed for virtual memory systems with the explicit objective of minimizing page faults. These algorithms are subject to mathematical analysis, which is not true of arbitrary user programs. Hence, by careful observation of the storage references made by a program or multiprogrammed collection of programs (although the latter clearly requires some further remarks) we can analyze its interaction with any given page-replacement algorithm running in any given size of primary memory, and ascertain which page faults would or would not have occured had primary memory been some other size. These techniques are not in general applicable to non-virtual memory systems, for many programs have no idea of how large a memory they are running in, or how to take advantage of it, and thus explicitly-requested data transfers are not affected by changing memory size in any interesting or easily analyzable way.

The ability to determine page fault rates (page faults per unit time) for different memory sizes is a powerful tool in both performance analysis and memory system engineering. Sekino (S2) has shown the explicit dependence of response time and throughput in multiprogrammed systems on the mean headway between page faults (MHBPF). This quantity describes the mean amount of useful work done by user programs between each two page faults. It is most conveniently measured in total references to the virtual memory. If the mean amount of system overhead associated with a page fault is known, as well as a proper characterization of system idle time, we may compute MHBPF from the mean real time between page faults (MTBPF) and the processor reference rate. Hence, predictive techniques to obtain page fault rates for contemplated memory sizes can be used to deduce the system throughput and response time figures which would result. Hence, if one can indeed predict these figures, the economic tradeoffs involved in acquiring improved memory system performance by increasing primary memory size may be evaluated more methodically.

The use of more than one type of secondary memory in a single system results in a situation where the average time to access a data item in any part of the storage system is a function of both the average access time to a data item in each unit and the probability of accessing that unit. In a demand paging system, the probability of accessing each unit is the sum of the probabilities of accessing each page stored on it. If one can associate these probabilities with given pages of such a system, one can create a composite memory system with an optimal average access time within any given cost constraint. Ramamoorthy and Chandy (R1) have given an algorithm, whereby such a system may be constructed out of any collection of memory types, whose speed and cost-per-bit characteristics are known. In any case, it is clear that one should keep the pages with the highest reference probability on the fastest storage devices. Although the identities of these pages may be determined by experimentation, observation, and program analysis, one can view these probabilities and/or identities as functions of time. Thus, one can devise algorithms which attempt to maintain pages with given ranges of next-reference probabilities on appropriate storage devices. It should be fairly apparent that this problem is identical to that of maintaining pages in primary memory with the intent of minimizing page faults. This will be discussed more later on. Thus, the design of an optimal multilevel storage system, as such configurations are known, can also be analyzed by the techniques of primary memory paging analysis. Again, the assumption of an appropriate model of program behavior, both in general and for the particular system at hand, is of crucial importance.

1.2 Program Reference Patterns and Models

Computer programs being among the most deterministic of all things, any characterization of the data reference patterns of any particular program may be obtained by the simulated running of that program and the observation of whatever characterizations are desired. However, system engineering requires characterizations of programs which are to be run, which, for the most part, have not yet been written. In a given computer system, running under a given operation system, most running programs have many features of their memory usage patterns in common. For instance, in an operating system where an Algol-60 or PL/I-like run-time stack is native to the environment, the pages containing the top of the stack will always have a higher next-reference probability then page representing lower regions. If the supervisor itself is paged, i.e., running in a virtual memory, the same as users' programs, the supervisor has its own reference patterns which will be present in any run of the system. The same is true of compilers, assemblers, system utilities, library routines, and other service programs. Code generated by the same compiler is likely to produce certain common features in its reference patters, particularly on a local level. Thus, there is great value in observing typical behavior of programs in a large computer system, and trying to formulate some model which is in some sense average or typical.

In a multiprogrammed system, this averaging is done for us in real time. An experimental observation of program behavior in a multiprogrammed computer system, made over some reasonable period of time, say a day, will produce a characterization of typical system behavior, if one indeed believes that such exists. This characterization takes into consideration all of the programs run in that day, and relies on the common features of programs discussed above to have any validity at all. The interval of a day is chosen as reasonable, for that is the cycle time of many forms of human interaction with a computer. People deal with an interactive computer system for several days, doing the same type of work at similar hours in the day.

The particular model of reference behavior that we seek describes next-reference probability of pages in a virtual memory system as a function of position in a certain dynamic ordering, known as a stack, imposed by the page-replacement algorithm. The class of algorithms amenable to this analysis are precisely those which would keep the top n pages of this ordering in an n-page primary memory, were it used to manage such. This will be discussed more fully in section 2.1. What is important here is that we can arrive at a function p(x), where p is the probability of reference to position x in this ordering. It is the object of that sub-class of these page-replacement algorithms which are actually useful for memory management to make this function monotonically decreasing. If the algorithm actually succeeds at this, it is clear that then pages which are most likely to be referenced will probably be in the n-page primary memory, and thus, the page-replacement algorithm has succeeded in minimizing references outside of the n-page primary memory, or page faults. In the case of multilevel memories, we can pick out whatever positions in the ordering are appropriate, by Ramamoorthy and Chandy's algorithm, and assign them to whatever storage unit is required. C. K. Chow (C3) has also given an algorithm where an optimal multilevel memory system within a cost constraint may be constructed directly from the function p(x).

1.3 The Experimental Determination of Predicted Headways

If we accept the function p(x) as a valid characterization of average and typical behavior in a multiprogrammed system, we may predict page fault headways for hypothetical memory extensions from it. Furthermore, the function p(x) may be measured experimentally. In this section, we show how to approximate and use p(x) in this way.

x is the position of a page in the algorithm-imposed ordering we have been discussing. Assume we have constructed the necessary tools to measure r(x), where r(x) is the number of times a page in position x of the ordering was referenced. Assuming pages which were never touched to be in position "infinity" of the ordering, then the relative frequency of touching a page in position x is

$$f(x) = \frac{r(x)}{\sum_{t=1}^{\infty} r(t)} \qquad (1)$$

Here, the numerator is the count of references to position x, and the denominator is the total number of references to all positions. If p(x) is indeed a valid characterization, f(x) should approximate p(x).

We have stated, that for the class of algorithms under consideration, the first k positions of this ordering at any time contain precisely those pages which would be in a primary memory of size k. Hence, references to pages in the first k positions of the ordering never cause a page fault in a k-page primary memory, and references to pages in any position beyond k always cause page faults. Hence, if a program makes H references to the virtual memory, the number of page faults it will take in the course of those references is identically the total number of references made to positions in the ordering beyond the primary memory size. Thus, the program, running in a k-page primary memory, will produce a mean headway

$$\mathrm{MHBPF}(k) = \frac{H}{\sum\limits_{t = k + 1}^{\infty} r(t)} \qquad (2)$$

This relation holds true for <u>any</u> primary memory size k. If we have an actual system, running in a primary memory of size n, we can predict the MHBPF which would result on this system were memory extended to size E, E being greater than n. We assume that we can measure MHBPF(n) on the existing system, and that a tool for measuring r(t), for $t>n$, is available. Then the same program which takes H virtual memory references will have a MHBPF in the E page memory of

$$\mathrm{MHBPF}(E) = \frac{H}{\sum\limits_{t = E + 1}^{\infty} r(t)} \qquad (3)$$

We now divide equation (3) by equation (2), obtaining

$$\frac{\mathrm{MHBPF}(E)}{\mathrm{MHBPF}(n)} = \frac{\sum\limits_{t = n+1}^{\infty} r(t)}{\sum\limits_{t = E+1}^{\infty} r(t)} \qquad (4)$$

Observe that this equation allows us to predict MHBPF from a measured MHBPF and measured reference counts, a fact which will be used later. We now rewrite equation (1) to read

$$r(t) = f(t) \sum_{u = 1}^{\infty} r(u) \qquad (5)$$

letting t be what was x and u be what was t. Substituting (5) in (4), replacing r(t), we obtain

$$\frac{\mathrm{MHBPF}(E)}{\mathrm{MHBPF}(n)} = \frac{\sum\limits_{t=n+1}^{\infty} f(t) \sum\limits_{u=1}^{\infty} r(u)}{\sum\limits_{t=E+1}^{\infty} f(t) \sum\limits_{u=1}^{\infty} r(u)} = \frac{\sum\limits_{u=1}^{\infty} r(u) \sum\limits_{t=n+1}^{\infty} f(t)}{\sum\limits_{u=1}^{\infty} r(u) \sum\limits_{t=+1}^{\infty} f(t)}$$

$$= \frac{\sum\limits_{t=n+1}^{\infty} f(t)}{\sum\limits_{t=E+1}^{\infty} f(t)} \tag{6}$$

Multiplying both sides by MHBPF(n), we obtain

$$\mathrm{MHBPF}(E) = \mathrm{MHBPF}(n) \frac{\sum\limits_{t=n+1}^{\infty} f(t)}{\sum\limits_{t=E+1}^{\infty} f(t)} \tag{7}$$

This equation states that mean headway between page faults which would result from a memory extension to E pages may be computed from the measured mean headway between page faults on the unextended memory, and a factor which is a function only of the program or programs being run and the memory sizes concerned. The work of our thesis is to compute this factor.

1.4 Previous Work in this Area

Since the advent of virtual memory computer systems, the function MHBPF(x) has been of great interest, being an easily identifiable characterization of memory system performance. Investigators have run many programs in simulation, obtaining this mean headway as a function of memory size experimentally. Almost all of these experiments have been done on machines which attempt to 'compress' a program into a smaller space than that in which it was intended to run. Such systems may typically attempt to fit five or ten programs, each running in a 32 k virtual memory into a core memory of 96 to 150 k. In such instances, the set of pages referenced by each program is small, as is the potential set which it can reference. These sets of pages are usually disjoint, as they represent disjoint virtual memories. Virtual memory in this case is simply a technique to force several programs into a primary memory too small to contain all of them.

Such work has been reported by Belady (B1), Belady and Kuehner (B3), and Fine et al. (F1), among others. A large amount of this work was done on an IBM M44/44X, a 7040 type machine at IBM Research Labs adapted to demand paging. Belady and Kuehner report an expected HBPF for single programs running on this system of the general form $e = a\ n^2$, n being primary memory size.

Brawn and Gustavson (B4) performed some measurements of typical computational programs running on the same M44/44X. These measurements were significant as they are apparently the first reported measurements of programs specifically written for a virtual memory. They observed the running time of programs, including page fault overhead, as a function of real memory size. No analytic models were suggested.

Some performance analysis done by Schwartz (S3) on a Burroughs model 6700 is also of interest here. In this system, all data available to a program is referenced as variable-size segments, brought into core on a demand basis. Program code and certain data segments are shared, and the amount of information potentially accessible to a program is extremely large. He reported headway functions of the form $e = \exp(a \cdot n)$, variables the same as above, for missing-segment exceptions as memory size was varied. (These were actual measurements performed on various memory configurations.)

The research which directly led to this thesis was done by Saltzer, and later by Saltzer, Webber, and Snyder(S1). Saltzer measured the MHBPF on the Multics system (B2) at M.I.T., with two different sizes of configured memory. He obtained the result $e = a \cdot n$, which has since been called the 'linear paging model'. Saltzer later reported the results of an experiment designed and conducted by Webber and Snyder, in which the reorderings of the list by which the Multics paging drum is maintained were observed. Using the techniques described in 1.2 above, MHBPF(n) was extrapolated to a memory size of 4000 pages (each Multics page is 1024 words by 36 bits), and was found to be still within experimental error of the linear paging model.

1.5 Novelty of the Work in this Thesis

The work performed in this thesis was originally conceived as an extension to Saltzer and Webber's experiment, elucidating the nature of MHBPF(n) for n greater than 4000 pages. The limitations of the linear model were sought, as was the nature of whatever model held beyond that range.

This series of experiments on the Multics system is unique for several reasons. The data accessible to any program in Multics is potentially the entire storage system, and all data accesses are made via the virtual memory mechanism. This is similar to the Burroughs scheme, but dissimilar to the paged 'compressing' type systems described above. Furthermore, sharing is an extremely important consideration in Multics, as all program code, including the supervisor, is shared. This thesis is also apparently the first reported attempt to deal with dynamically variable virtual memories, i.e., those whose size grows and shrinks on a second-to-second basis. The issues of dynamic page creation and destruction which result from this policy are systematically dealt with by our experiment.

The use of virtual memory seems to be gaining in popularity as large general-purpose information systems become more common. Increased interest in systematic protection schemes has resulted in many new designs for systems having segmented addressing features similar to those found in Multics. Demand paging has achieved considerably more popularity and widespread use than the Burroughs techniques as an implementation for segmentation, and has recently been added by IBM to their extremely popular System/370.

For these reasons, we feel that experiments made on a Multics-like system are relevant to data systems in the near future, and the reference patterns observed may have some features which are in some sense characteristic of programs running in segmented, paged, environments.

Chapter 2

2.1 Stack Algorithms and the Extension Problem

The substance of the experiment performed was to reconstruct the entire history of a day's LRU-maintenance of the Multics storage hierarchy, and attempt to predict page-fault headways for hypothetical memory configurations from this history.

The basic strategy of memory simulation used was that proposed by Mattson et al. (M1). This technique, known as stack simulation, relies on the fact that a large number of useful paging algorithms, including LRU, have the property that after any fixed number of addresses in an address trace have been processed by the algorithm, the pages which are left in primary memory are always a subset of what they would have been at the same point in the trace had primary memory been larger. This feature, known as the "inclusion property", thus defines the class of "stack algorithms". From this property, at any given point in the processing of an address trace an ordering can be constructed. The first page in this ordering would be that page which would then be in primary memory were it of single-page capacity, the second would be that page which would <u>also</u> be in primary memory were it of two-page size, the third that which would be added were memory of three-page size, and so forth. The history of the processing of an address trace can be viewed as a series of these orderings, which are known as "stacks", the single page corresponding to unit-size memory normally being considered the "top". As each new reference is processed, the algorithm causes the stack to be reordered, possibly corresponding to page motion for some size memory. The top n pages on the stack being the pages which would be in a memory of n-page capacity, any motion of a page into the top n-pages implies a physical reading of a page into primary memory. For a demand stack algorithm, this movement can occur only as the result of a page fault. Thus, we may infer the behavior of an n-page primary memory by observing the number of times that reference is made to position n+1 or beyond in such a stack. As we have defined this stack and the class of algorithms processing it as maintaining the first n pages in this stack in an n-page memory, no reference to position n or below can ever cause a page fault. Mattson's technique consists of taking a recorded or proposed address trace, running it through a program which constructs the sequence of stacks just described, and accumulates the total number of references to each position therein. When the processing of the trace begins, the stack is void, corresponding to an empty primary memory. At least until a given page is fetched into primary memory the first time, it will not have been in the stack at all, and its first fetch may be considered to have been made from position "infinity". As the trace progresses, and repeated references to pages are made, we accumulate counts for each position in the stack of how many times a page in that position was moved upward by the algorithm. It can be shown that for a demand stack algorithm, the only condition on which a page may move upward in the stack is that it is that page which has just been referenced. Simply, were this not the case, a page in position n would move into an n-1 page primary memory without having been referenced, and the algorithm would not be a demand paging algorithm. As the completion of the address trace, we can, for any n, sum the reference counts for positions n+1 to the total final length of the stack, plus the count for position "infinity", and this will be the number of page faults which would have transpired had that address trace been managed by the algorithm used in an n-page primary memory. Note that a single processing of the trace can be used to produce a result which can then be used to analyze any hypothetical memory size.

This technique allows us to ascertain the page fault count for the interval under consideration for any contemplated memory size. By simply dividing the total system headway during the actual trace by this page-fault count, we may thus ascertain the predicted mean time between page faults (MHBPF) for that storage system. Furthermore, we can plot the reference counts at each position, normalized with respect to the total number of reference counts, versus the position number, and obtain a graph which describes what we shall call access frequencies. With this, we can analyze the behavior of multilevel memory systems processing this trace, and obtain an optimal such system within cost constraints as described in Chapter 1. The shape of this graph also tells us much about the relative success of the particular algorithm in managing that particular address trace, without regard to any single memory configuration. We will consider the particular graph in the case of our results in greater detail in the next chapter, and in so doing further consider such graphs in general.

Our experiment sought to learn the shape and nature of this graph at positions corresponding to memory sizes of many thousands of pages. In order to record a reference to position n in a stack as described, there must clearly be n-1 items above it. This implies that at least n distinct items have been referenced by the time a reference to the n^{th} position occurs. It can be seen that the extent of the address trace required to produce meaningful statistics at the ten-thousandth page position would require a prodigious address trace. At this point advantage may be taken of another remarkable property of stack algorithms. It is possible to construct the portion of the stack from position n+1 to the end without a full address trace -- we use information extracted from a running algorithm managing an n-page primary memory at times when page faults occur. This is known as the "extension problem" (C1,C2). The technique is as follows: we maintain the stack (the "extension stack") for positions n+1 and beyond. When a page fault occurs, we know that the page faulted on cannot be in the first n positions of the stack -- if so, it would not have been faulted on. We locate the page in the extension stack; if not there, we may consider it as having been at position "infinity". The counter corresponding to the position from which the page was fetched is incremented. We remove the page in question from the extension stack: it is now in the top position of the real stack, which we are not maintaining. We now use whatever information is necessary, from that normally obtainable to the running algorithm, plus that we are maintaining, to reorder the extension stack according to the policy of the running algorithm. This reordering will usually include placing some page removed from primary memory by the running algorithm at some point in the extension stack. In the case where the replacement algorithm is LRU, the page removed from primary memory is placed on top of the extension stack, and all pages previously in the extension stack move down one location. Note that pages which were below the fetched page in the extension stack stay in place during the entire transaction.

The advantages of using a trace of a running algorithm in a large system measured over an extended period of time, as opposed to a trace obtained by simulation of a given program over a necessarily much shorter period of time are straightforward. We are interested in system performance on an hour-to-hour, not second-to-second, basis, and day-long measurements of a live system correspond to both the time scale and load mix of interest. As long as the accuracy of the measurement can be maintained, this day-long extension measurement is much more useful than the simulated running of a program.

As a demonstration of the power of this extension technique, we may consider the Multics system: 400,000 references to the virtual memory occur every second. Approximately 100 page faults occur each second. Recording 2 data items for each page fault, we have reduced the amount of data which must be recorded by a factor of two thousand.

It should be clear that the results of this experiment, although simulating hypothetical memory system performance, do not represent simulated results. The measurements made correspond to an uncontrolled user population during normal working days, using arbitrary programs under 6-way multiprogramming. The results thus show how a hypothetical memory system would have behaved under this real user load.

2.2 The Extension Problem and Multics

The Multics system has a physical memory consisting of 256K (1 K = 1024 36-bit words = 3.7 x 10^4 bits) to 384 K words (1.2 x 10^7 bits) of core, a 2000 to 4000 K word (1.5 x 10^8 bits) drum, and approximately 90,000 K words (3.3 x 10^9 bits) of disk, both moving and fixed head. The variabilities stated above are dependent upon the time of day and the user load, governed by administrative policy. The entire storage system is divided into 1024-word pages, and is managed by the demand paging mechanism (with the exception of several thousand words of non-pageable code and data, such as the code for the paging mechanism itself, which must be non-pageable in any case, and are thus not really of interest in memory performance prediction). The algorithm used to manage replacement of pages in the core memory is essentially LRU. The variation from LRU is explained in detail in Appendix A. Essentially, within the constraints of operating system overhead and the precision of measurement of recency of use provided by the hardware, it tries to implement LRU as closely as possible. Also, a non-demand prepage/postpurge policy was in effect during these measurements, which caused some pages to move in and out of core outside of the control of the LRU algorithm.

The 4000-page paging drum was at this time being used in a mode which attempted to overcome rotational latency by making multiple copies (S4), in this case two, and hence was of 2000 page capacity during these experiments. Since January, 1972, the drum has been used as part of a hierarchically managed storage system, as a buffer between core and the disk storage subsystem. In such systems, one attempts to keep pages with the highest access frequencies on the fastest devices, in order to minimize the system's mean access time. In an automatically managed system, the identity of those pages is constantly changing. As the stack access frequency graph discussed above can be used to associate access frequency with stack position, page replacement algorithms identical to those used to manage primary memory are frequently used to manage other devices in a hierarchical memory system to achieve this end. In the Multics system, another near-LRU algorithm is used to manage the drum, which is described as well in the Appendix. The drum management algorithm attempts to maintain copies of the top 2000 pages of the theoretical stack corresponding to the LRU algorithm on the drum. The model of program behavior implied by the LRU algorithm, and verified by the results of this experiment, implies that these pages are the most likely to be referenced, and at the time they are on the drum, thus have the highest access frequencies.

As currently implemented, a page which has been faulted on, and is not on the drum, is read into core from the disk. It will not be written to the drum until the core management algorithm decides to oust it from core. This implies that the pages corresponding to that portion of the LRU stack representing core are not completely a subset of those on the drum. Hence the drum will contain pages representing a 2000-page contiguous portion of the stack, whose topmost extreme is anywhere between the top of the stack and the size of core below it. Of the 256 to 384K, about 100K is not used for paging, leaving 150 to 280K for paging, thus this variability represents about 7 to 15% of the size of the drum.

The stack-reordering procedure of the LRU algorithm is one of the simplest possible: the referenced page moves to the top position of the stack, as is necessary in any demand stack algorithm. The pages between the top page and the old position of the referenced page all move down one position. Thus, to use the extension technique described above for the LRU algorithm, the only reordering information one need record is the identity of the page thrown out of position p (p being the size of the non-extended memory, in pages), which will then occupy position p+1, at each page fault, in addition to the identity of the page faulted on. The "pushed" page becomes the top of the extension stack, the page previously there becomes #2, etcetera, all the way down to the former position of the faulted-on page. This is what we have done with the Multics core-drum combination, considering it as a 2000+X page buffer, where X is some fraction of the size of core, itself at most fifteen percent of the drum, to account for the top-of-drum variability described. As positions p+1 and on, in our case, correspond to the disk subsystems, we need only record disk reads instead of page faults. (It is instructive to note that within the entire operation of a Multics system, not a single direct-access I/O transfer is done outside the paging mechanism, pre-paging being included in this consideration.) As disk reads are two to five per second, we have thus reduced our data-gathering chore by at least 95%. The experiment of Saltzer, Webber, and Snyder, which was similar in intent to this experiment, but more limited in scope, has already produced results (S1) for primary memory sizes up to the maximum size of the drum. For this reason, we did not consider it worthwhile to attempt to gather data for that portion of the stack corresponding to regions in the drum. Hence, the application of the extension technique to this core-drum combination was adequate. All else that was needed was the recording of information provided by the drum management algorithm as to the identity of pages thrown off the drum. It can be seen that these are the pages thrown out of the core-drum combination, given that no page is ever thrown out of core without having been written to the drum.

In order to determine the validity of this technique, the necessary programs were written and tested on a stand-along Multics machine. This machine had 131,072 words of core and a 256-page drum. Hence, most of the range of the regular Multics drum was in the extension region of this experiment. The mean headway curve resulting was very well approximated by a straight line, suggesting the linear paging model. This provided a good deal of confidence in both the technique and the software.

Hence, we see two types of motion between core-drum and disk. The reading of a page constitutes motion from disk into core-drum. The writing of a page, however, does not constitute outward motion. In general, writing is performed only when a copy of a page on disk is different from a drum or core copy. The outward motion corresponding to a read is really the claiming of the core or drum frame previously occupied by the page of interest. We call this phenomenom an "ousting".

Unfortunately, a problem arises with even this simple model. Certain pages of the storage system, 3 in all, corresponding to the system's top-level directory, are special-cased by the paging and drum-management algorithms such that they may never go on the drum. This is due to certain integrity issues involving the reliability of the drum and the extreme difficulty in reconstructing the contents of this directory. Hence, these pages are never written to the drum, and leave the "core" portion of the increasingly less theoretical LRU stack directly for the disk portion. (In this case, writing never takes place unless the concerned page has actually been modified (see Appendix A)). More unfortunately, these are among the most popular pages on the disk, as by the dictates of the LRU algorithm, they should have by all means been on the drum. Thus, we must check for pages being ousted directly from core to disk, and we thus have two varieties (core and drum) of ousting to be recorded. The interpretation of the data resulting from movement of these pages will be deferred until Chapter 3.

Thus, we need record upward stack movement into the core-drum combination, meaning disk reads, and downward movement, meaning oustings. Another event of interest is the creation and deletion of pages. In the current implementation of Multics, logical pages are created out of the void when a never before referenced page is referenced. By definition, all such pages contain zeros, and hence never involve disk reading. Furthermore, these page faults will occur regardless of what size primary memory is, and are thus not of interest in memory performance prediction. This last statement is somewhat subject to current design and user behavior. Were there a tremendous amount of fast, cheap primary memory, it is altogether possible that users would rarely delete programs or data, but simply rewrite or modify them, thus making page creation a much rarer event. We choose to ignore this possibility.

In the following discussion, "n" represents the size of "primary memory", in pages, in terms of the extension problem. In terms of the specific experiment on Multics, n is the size of the core-drum subsystem in pages. As was explained earlier, this is the size of the drum (2000 pages) plus a fraction of the size of core.

Page faults which cause creation of pages involve neither disk traffic, idle time, nor multiprogramming, and are thus not of interest in MTBPF calculations. Since all new pages are created in this way, we find them naturally falling past position n of the complete LRU stack, into the extension stack after sufficiently long disuse. Page deletion, on the other hand, can occur at any time in the life of a page. If a page is destroyed so soon after its creation that it has never passed position n in the stack, we are oblivious to its entire existence. If, however, it is destroyed at such time that it is beyond position n, its destruction must be accompanied by its excision from the extension stack. When a page is destroyed in core or on the drum, the next page to be faulted on replaces it without any page being pushed down the LRU stack. However, the position in the stack of the destroyed page is assumed by the page directly under it. The page fault following a page destruction creates only upward stack motion -- nothing is pushed down.

Consider, in our theoretical n-page primary memory system, a page in position n of the LRU stack. This page is now destroyed. A page in position n+m is now faulted on. In an actual memory system, this page will now be read into primary memory without any page being replaced, the destroyed page having created an empty page frame, but the newly faulted-on page will be at the top of the LRU stack. The formerly first to n-1st pages now become the second to nth pages in the stack. The n+1st (first page not in core) to n+m-1st pages retain their original stack position. (See figure 2.1). The n+m'th position in the new stack is in a situation akin to that of the nth position after the deletion of the page there: the page in the n+m+1st position cannot come up to fill the void --

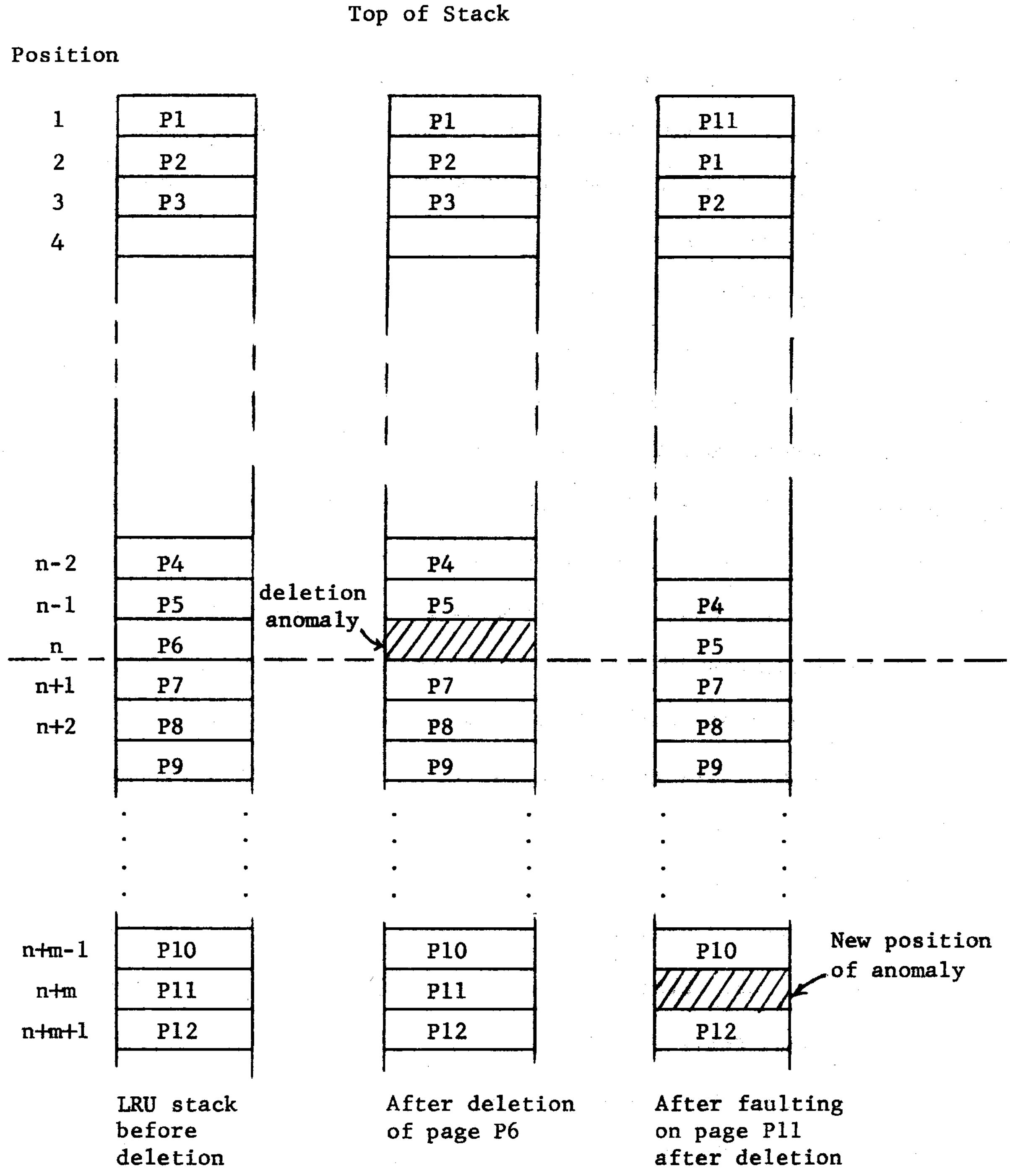


Figure 2.1 -- Behavior of anomalies resulting from deletion.

that would not be demand paging. Until some reference is made to a further position, say n+m+k, this will be the case. At the time the n+m+k reference is made, position n+m+k now becomes anomalous. Hence, we create an anomaly when a page is deleted, which propagates down the stack as any reference is made beyond it.

The strategy that we have chosen to deal with this, in the simulation, is simply to excise a page from the stack when it is deleted. Thus, any reference to a position beyond the excision will be tallied as a reference to position x instead of x+1. Note, however, that the position of the excision has then moved to x. All references in front of excisions are tallied correctly. The analysis of the inaccuracies resulting from this treatment is quite involved, and is covered in detail in section 3.4.3.

Thus, the data items which must be recorded in a trace are those representing 1) reading of pages into memory, by demand or prepaging, from disk, 2) claiming of pages by ousting pages from drum to disk or from core to disk, and 3) the deletion of pages from the storage system. Of these events, types 1 and 3 represent excision of a page from the extension stack, while type 2 represents the pushing of a page on to the top of the extension stack. Events (1) also cause the noting of the stack position of the page read, and the incrementing of a counter corresponding to that position. There are actually some other events which must be recorded in the case of the Multics system, but these are due to the particular implementation of the core and drum management algorithms, and are discussed in Appendix B.

The handling of page reads of pages which cannot be found in the stack, i.e., their first reference, requires some thought as to interpretation. Were this experiment run for a sufficiently long time, the appearance of such pages would cease. Pages which are created come down on the top of the stack, and any existent page which has not been referenced since the experiment began enters the core-drum extension-stack combination once and never leaves, until it is destroyed. These first references, as discussed, are counted in the "infinity" position of the stack. These fetches of pages not in the stack accounted for roughly a tenth (7881/74530) of all disk page fetches. These references do not affect the relative number of fetches to any two extension stack positions, as they would not be in a core-drum memory of any size until the first time they were referenced. Thus, when one considers disk accesses, one should consider these reads to be disk references in a core-drum system of any size. However, the longer the experiment runs, the fewer will these references become. Thus, since we are interested in steady-state behavior, we have chosen to consider these reads a start-up transient, and not count them in any calculation. They tell only of the length of the experiment, not of what is being measured.

2.3 Performing an Experiment on Multics

Having developed the theoretical bases of the extension problem, and adapted it to Multics, the next step was to proceed to construct the necessary software to develop the extension address trace, collect it, and perform the LRU stack simulation with it.

A privileged-access facility was set up in the Multics hardcore supervisor specifically for this experiment. When enabled, a trace of all of the events mentioned above was accumulated in a circular 1008-word buffer. Each trace item included the physical device address of some page being read, ousted from core-drum, or deleted, information as to which of these events is represented, and a flag indicating, for statistical purposes, whether or not it was one of the previously mentioned pages which are not allowed to go on the drum. Also recorded was information allowing the program which inspects this buffer to synchronize itself with it correctly. A program was developed which inspected this buffer regularly -- from the Multics standpoint, a privileged operation. This program assembled the buffer images into a continuous trace, which could be as long as necessary, suitable for further, repeated processing.

This strategy was decided upon because of the extensive time required to search an LRU stack for a given page, and the large amount of space required to store this stack. This ruled out the possibility of having a special-purpose module of the Multics supervisor perform the experiment in real time. The performance degradation necessitated by the time required to search and the space, which would have had to been non-pageable, to store the LRU stack would have been wholly unacceptable. Furthermore, the accumulating of the trace data for further processing allows many programs and versions of programs to be run on this data, increasing both its usefulness and the accuracy of the results obtained from it.

One disadvantage of the data collection strategy described is the possibility of the data collecting program losing synchronism with the circular trace buffer, i.e., data being overwritten by new data before it has been duly noted. This situation can come about when the data collecting program has made a decision about how often the buffer should be sampled, and an intense unexpected burst of activity causes the buffer to be written into significantly faster than before. The data-gathering program samples the buffer again, and notices that data has been lost, but anticipating further loss reschedules itself. Another way that data can be lost from buffer mis-synchronization is the data-gathering process falling behind in the multiprogramming queue due to Multics scheduling policies and heavy user load. The implementation of the data-gathering program tried to compensate for this by being written as a multi-process program, i.e., a program running in a coordinated way in many processes at once. Not only did this give it a scheduling advantage, but increased the reliability of the data-gathering operation as a whole.

Unfortunately, data losses of the types described were common, especially in initial, developmental runs of this software. The greatest losses would typically occur at midnight, when a large number of user programs scheduled to run then would, creating heavy paging activity referencing pages neither in drum nor core, and only one or two processes would be supporting the data-gathering operation. The extents and analysis of these losses are considered in the next chapter.

A danger of running a large complex data-gathering system in many processes is that of creating a great deal of activity which would bias the result of the measurements by measuring itself. The sharing features of the Multics system helped counterbalance this effect: all of the data bases and procedures of the data-gathering system were fully shared, having only one copy. Only the per-process work areas were not shared. The actual data-gathering system, in order to handle the control of multiple processes, possibly multiple terminals, and dynamic scheduling was, in fact, quite complex, requiring six separate procedures. The shared data bases and procedures totalled ten pages. Approximately two pages of work area per process were needed.

The data gathered was stored in data segments in the Multics virtual memory. The stack simulation was subsequently performed, using this data as the extension address trace, exactly as described above. The procedures which performed this reduction ran in an unrestricted Multics environment, and hence had practically no restriction on time or space. The LRU stack was represented as a list, in which each node represented a stack position. A push of a page onto the top of the stack required the allocation of a new node, and the redefinition of this node as the top of the stack. This node was then made to point to the former stack top. The excision of a page from the stack required locating the node corresponding to this page (each node contained a physical page address), the reallocation of this node, and the reconnecting of the list around it. For trace data representing disk reads, however, it was necessary to ascertain the position in the list of the relevant page. This required a search of the entire list. In order to reduce the work of discovering that a page was not in the list at all, a bit table was constructed, describing, for each possible physical disk address, whether or not it was in the list at all. This saved the necessity of walking the entire list.

The above list-maintenance algorithm spends a great deal of time searching the list to determine the position of pages in it. Several algorithms were considered to avoid the seemingly crude strategy of linear search, but most of these algorithms caused the list to grow increasingly disorganized, requiring periodic time consuming re-organizations, or required large amounts of data movement, a poor approach in a paged system. Because of the availability of a stand-alone machine which could easily provide the computer time necessary to perform this processing, the development of a better list-maintenance algorithm was not pursued further.

The result of the stack simulation was a table, describing for each position in the extension stack, how many times a page in that position had been referenced. The sum of all of these counts, plus those at position "infinity", represented the total number of all page fetches from disk during the period of the measurement. Although a graphical display of this information is of some interest, the calculation of MHBPF was the immediate objective. Thus, a table was created displaying, versus extension stack position, the total number of fetches observed divided by the sum of the counts for all of the positions further down the stack. For a given position N, the interpretation of this number, x, is as follows: had memory (core/drum) been extended N pages above its actual size, we would make one disk reference under that circumstance for every x references we make now. We thus refer to x as 'references per exception'. Note that we have <u>not</u> included the "infinity" fetches in the 'total reference' count in the actual results shown here, for the reasons discussed in section 2.2.

One may choose to interpret x as the "relative increase in mean headway", which is to say, the factor by which mean headway will increase over its current value if the extension of N pages is made to core-drum. For instance, if "references per exception" were 5 at N = 4000 pages, the interpretation would be: If we added another 4000 pages to the drum, we would fault to the disk one-fifth as often as we do now. This constitutes, in one sense, the raw data result of the experiment. Now,

$$\frac{\text{References observed}}{\text{References beyond position N}} \text{ X } \text{Mean time between measured references}$$

$$= \frac{\text{References observed}}{\text{References beyond position N}} \text{ X } \frac{\text{Time duration of experiment}}{\text{References observed}}$$

$$= \frac{\text{Time duration of experiment}}{\text{References byond position N}}$$

$$= \text{Mean time between references beyond position N}$$

$$= \text{Expected mean time between references were memory extended by N}$$

Multiplying this number by the measured system headway in virtual memory references during the experiment, and dividing by the time duration of the experiment, we obtain the expected mean headway between page faults were memory extended by N.

We have displayed both references per exception, versus memory extension size, and predicted inter-reference headway, as a function of memory size.

Note that in all of this discussion, mean time computed from an experiment taking many hours must be taken quite literally. The averaging effect over a day of usage varying between a heavy user load and solely the data-gathering program* produces a result which is really applicable to neither, but only a theoretical load somewhere in between. For this reason, we feel that 'references per exception' is a more useful interpretation of the results of this experiment than 'mean time between exceptions'. Attempting to tune a system to the theoretical point described by such measurements will not help the system when it needs the most help.

The reference counts and references per exception were subsequently displayed in printed tabular form, and the references per exception versus stack position plotted on a Stromberg-Carlson 4020 Microfilm Recorder. Some of these results are reproduced and discussed in detail in the next chapter.

*Some more precise descriptions of the exact user load during the experiment are provided in Appendix C.

The Results of The Experiments

3.1 The Conduction of the Experiments

The experiment was conducted about a dozen times, but early runs suffered heavily from [illegible]. Hence, we have chosen to display and analyze here the [illegible] which were most accurate in this regard.* These [illegible] traffic nature") 21" and "dtm 23" were [illegible] on May 7, 1973, from 9:45 AM to midnight, and May 14, 1973, [illegible] AM to midnight. In either case, the load on the [illegible] ranged from moderately light to average. For dtm 21, user load ranged between 20 and 40 users for most of the experiment, but ranged between 10 and 20 before 10:00 AM and after 6:00 PM. For dtm 23, load ranged at about 30 users, but ranged between 20 and 30 between 1:00 PM and [illegible] PM. The paging drum was being used in a 2000 page, non-[illegible] ([illegible]) capacity for all of both experiments. For dtm 21, two processors and [illegible] pageable pages of core were on line for the entire experiment. The same was on line for most of dtm 23, except for the period from [illegible] to [illegible] PM, when one processor and 170 pageable core pages were on line. User load, idle time, and system page fault overhead for both experimental periods are given accurately in Appendix C.

3.2 The Results of the Experiment

The form that we have chosen to display is that of an "exception ratio", or MHBPF(E)/MHBPF(n), where n is the adjusted 'primary' memory size (core-drum) as explained in section 2.2, and E is a hypothetical memory, both in pages. This exception ratio is the quantity expressed by equation 1.6. We plot this ratio versus primary memory extension in figure 3.1. We express the abscissa of our graph as 'memory extension', which is the hypothetical increase of core-drum instead of absolute memory size because of the variability of the size of core-drum as discussed in Chapter 2. The size of core-drum is not the sum of sizes of core and drum, because of duplications, created pages in core which have not been copied out to drum, and possibly even different configured sizes of core. The extension size to core-drum is meaningful however, because the data and results derived from the measured data represent the behavior of a hypothetical extension of the given size, oblivious to all of the above considerations. If a figure for the size of core-drum is needed, 2100 pages is reasonable. The shape of this graph suggests an exponential behavior. Thus, we next plot this ratio on a logarithmic vertical axis, to better view this behavior. This is figure 3.2. The plot almost traverses the graph diagonally, suggesting the straight line which would correspond to an exponential. We have drawn a straight-line approximation, which corresponds to

$$\mathrm{MHBPF}(E)/\mathrm{MHBPF}(n) = 3.42e^{(E-n)/(7.00 \times 10^{7} \text{ bits})} \qquad (1)$$

The surprising closeness of the dtm 21 and dtm 23 plots gives some confidence in this result. A similarity to the unpublished Burroughs re-

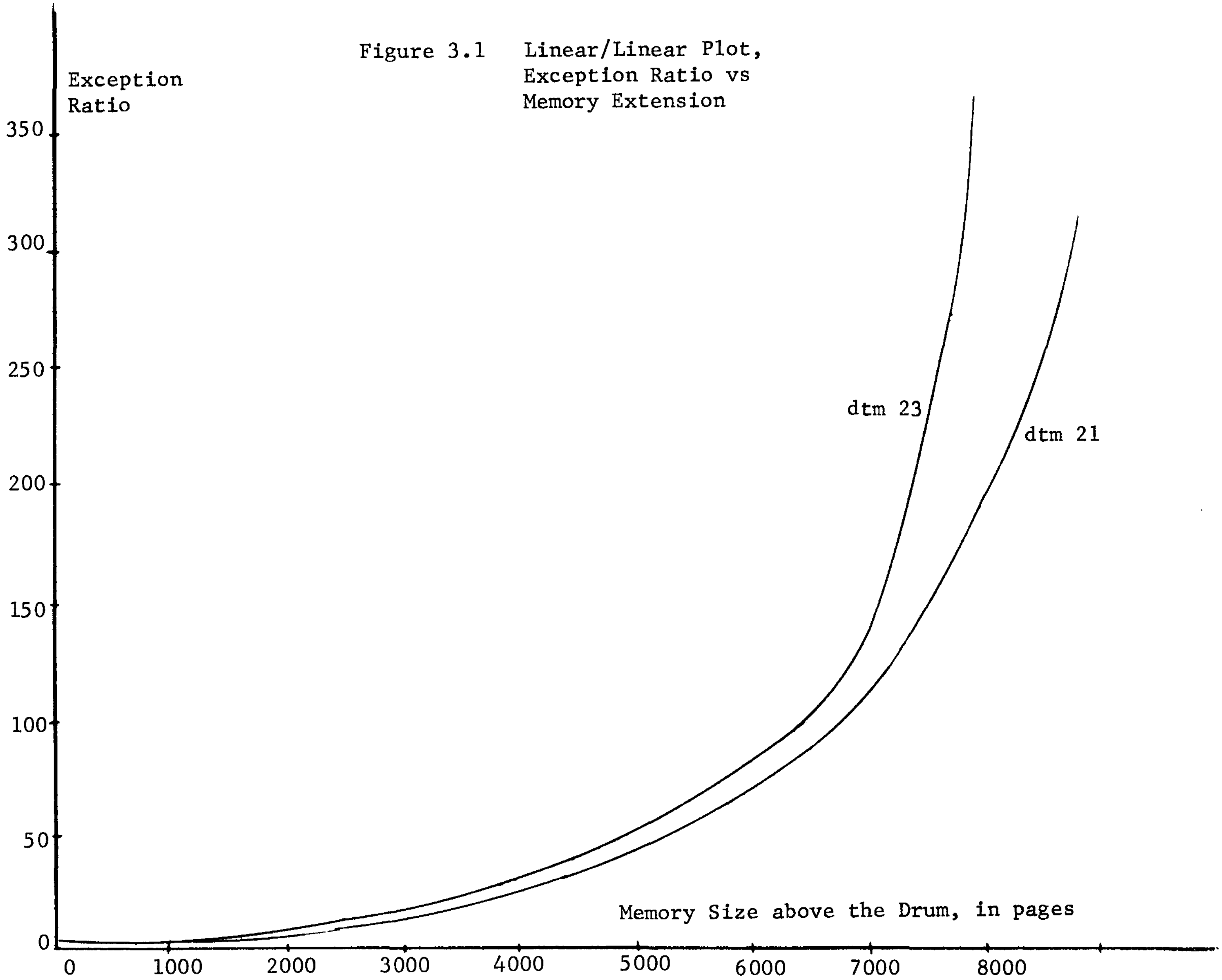
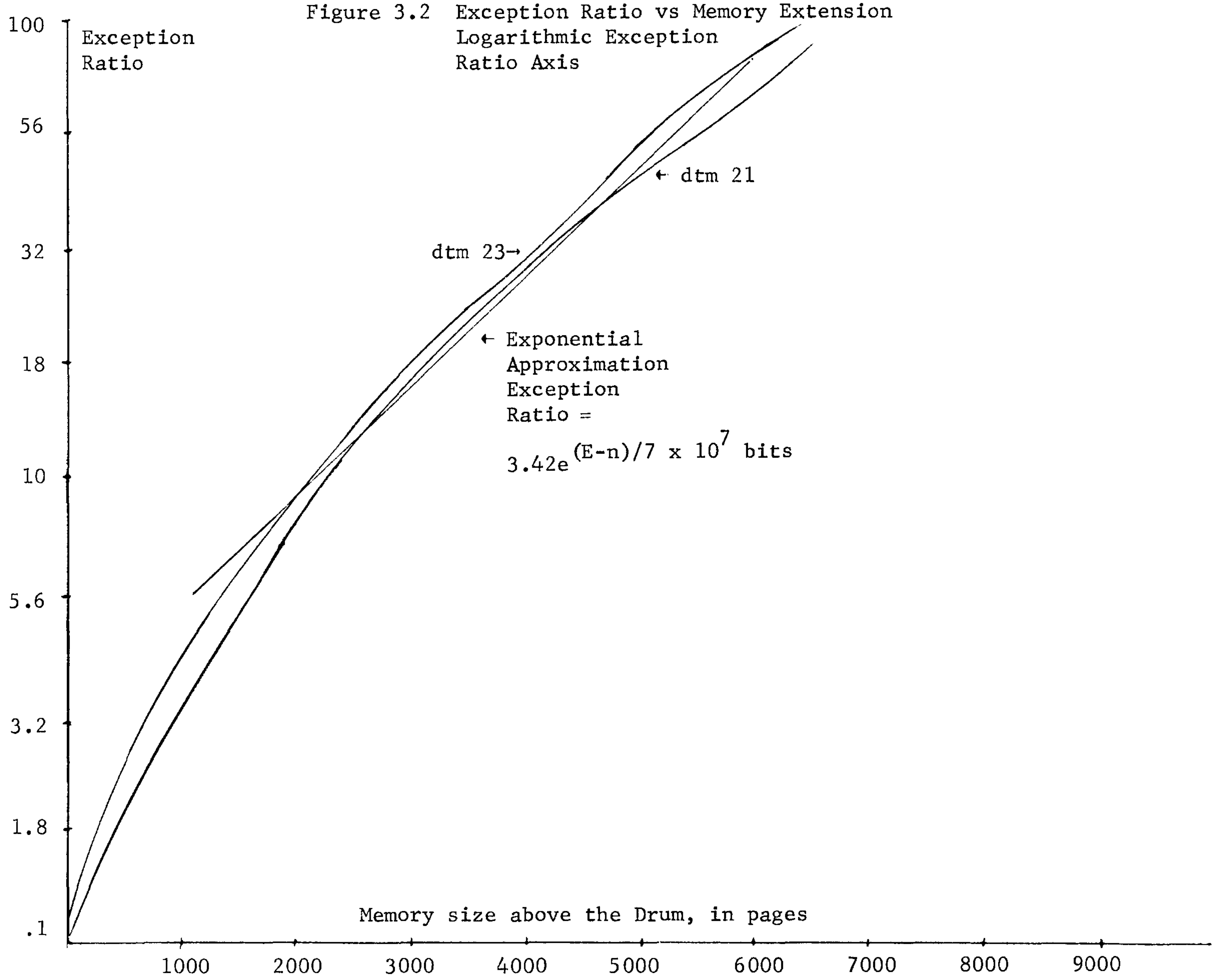


Figure 3.1 Linear/Linear Plot, Exception Ratio vs Memory Extension

Figure 3.2 Exception Ratio vs Memory Extension Logarithmic Exception Ratio Axis



sult (S3) mentioned in Chapter 1 may be noted, subject to the limitations discussed there. This similarity goes only as far as that Schwartz noticed an exponential mean-headway function for missing data in primary memory as memory size was increased.

It can be seen that the low regions, i.e., below 2000 pages above the drum, fall short of the exponential approximation suggested. Thus, we provide figure 3.3, in which we display the low region of figure 3.1. This plot shows a decidedly less than exponential behavior. From one viewpoint, this is comforting, as the experiment of Saltzer, Webber, and Snyder (S1) measured this function in this range, and obtained a linear function. However, our plot seems to grow considerably faster than linearly in this region. This can be seen as a noticeably different change in behavior.

An attempt to resolve these contradictions is provided by figure 3.4, in which both memory extension and exception ratio are plotted logarithmically. It is seen that the higher regions of this plot approach quadratic slope, and the increasing slope lends more confidence to the exponential suggested above. However, no uniform quadratic curve suggests itself.

The most that can be said is that MHBPF(K)/MHBPF(n) is a function of at least second order, as K-n exceeds 3000 pages.

We also provide figure 3.5, in which we plot MHBPF(K) directly as a function of memory extension, by multiplying the ordinate of figure 3.1 by the measured MHBPF on both measurements. Note that a given exception ratio does not correspond to the same MHBPF on both measurements, as different mean headways were observed. This is due to the variability of the system

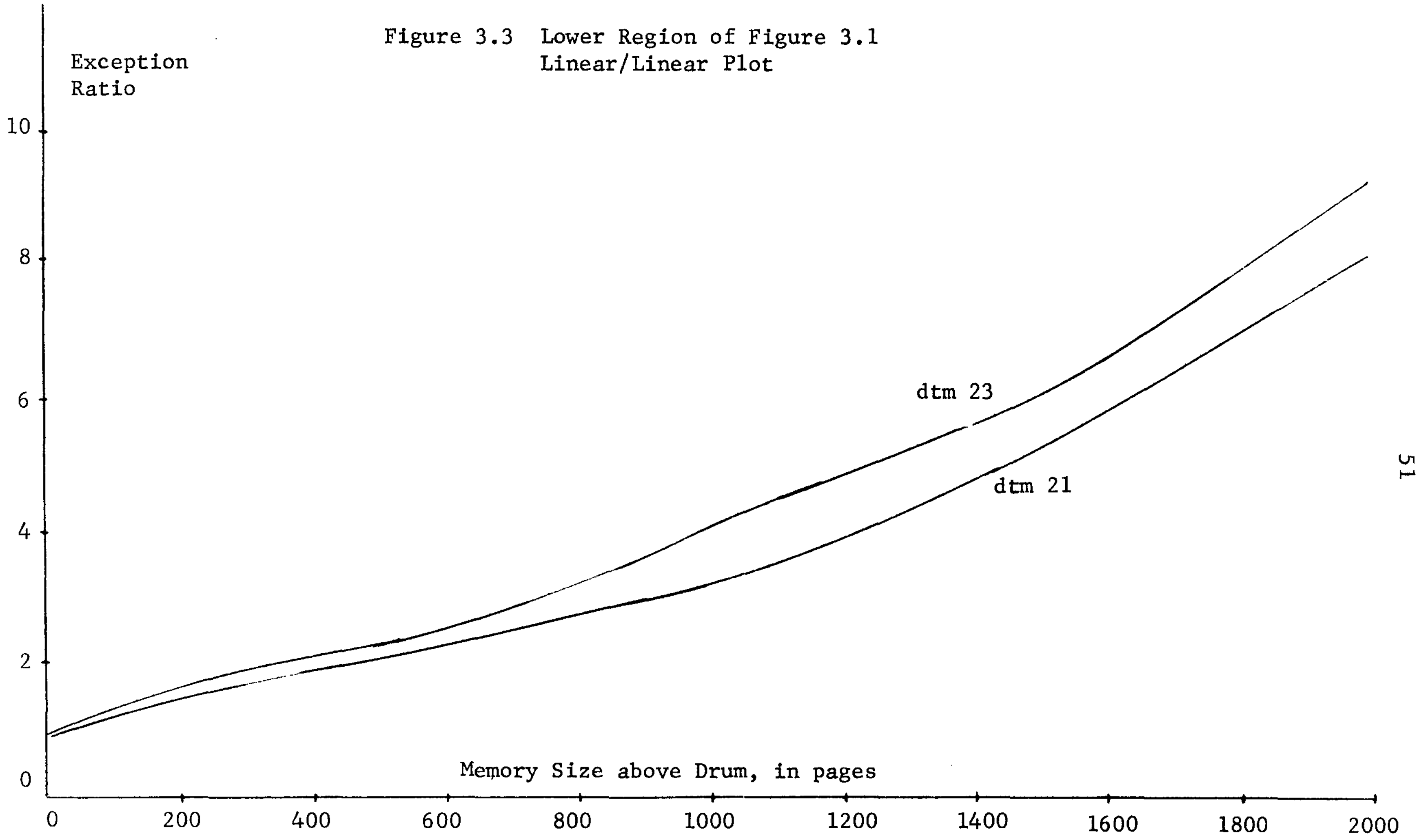


Figure 3.3 Lower Region of Figure 3.1
Linear/Linear Plot

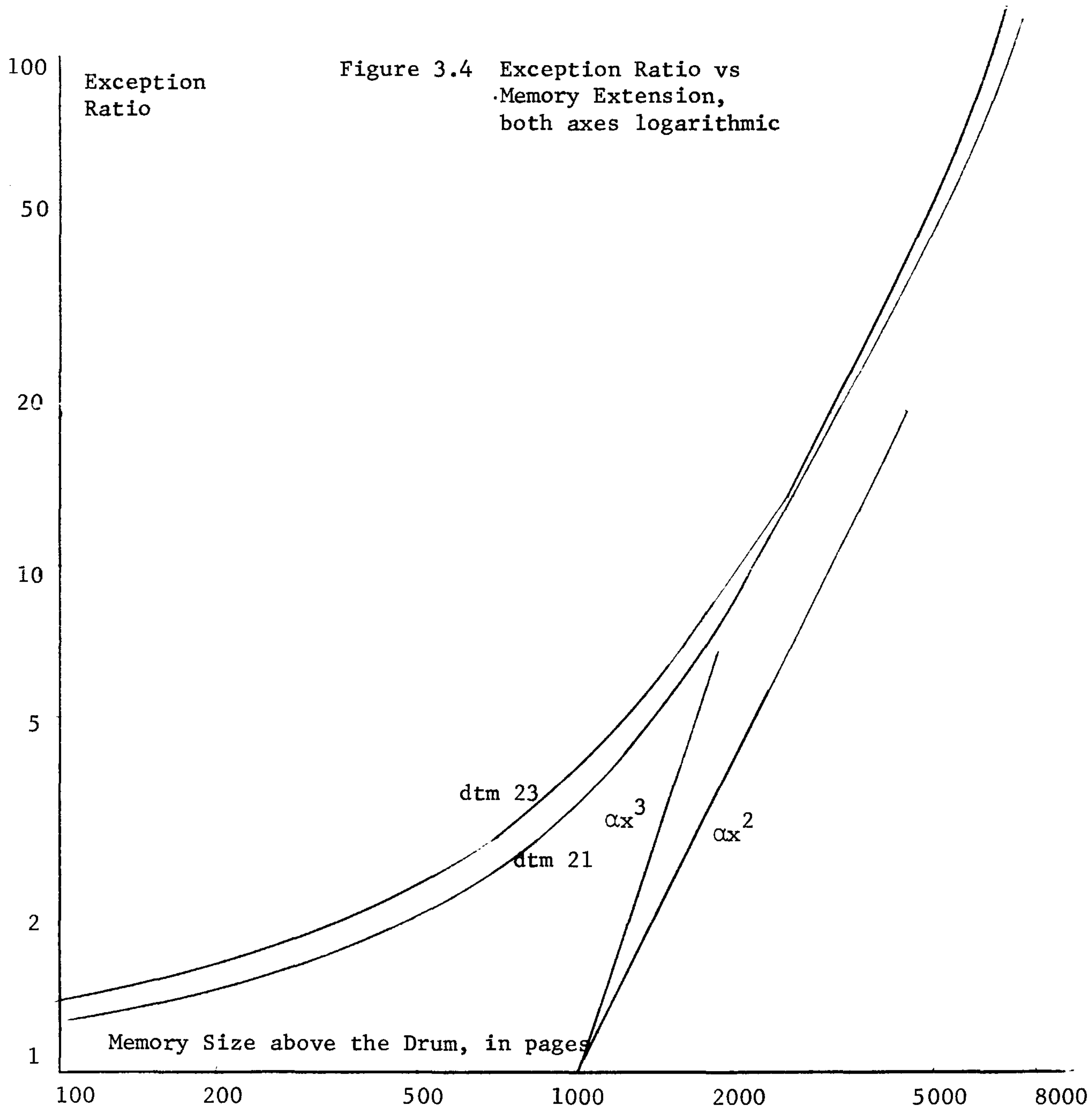


Figure 3.4 Exception Ratio vs Memory Extension, both axes logarithmic

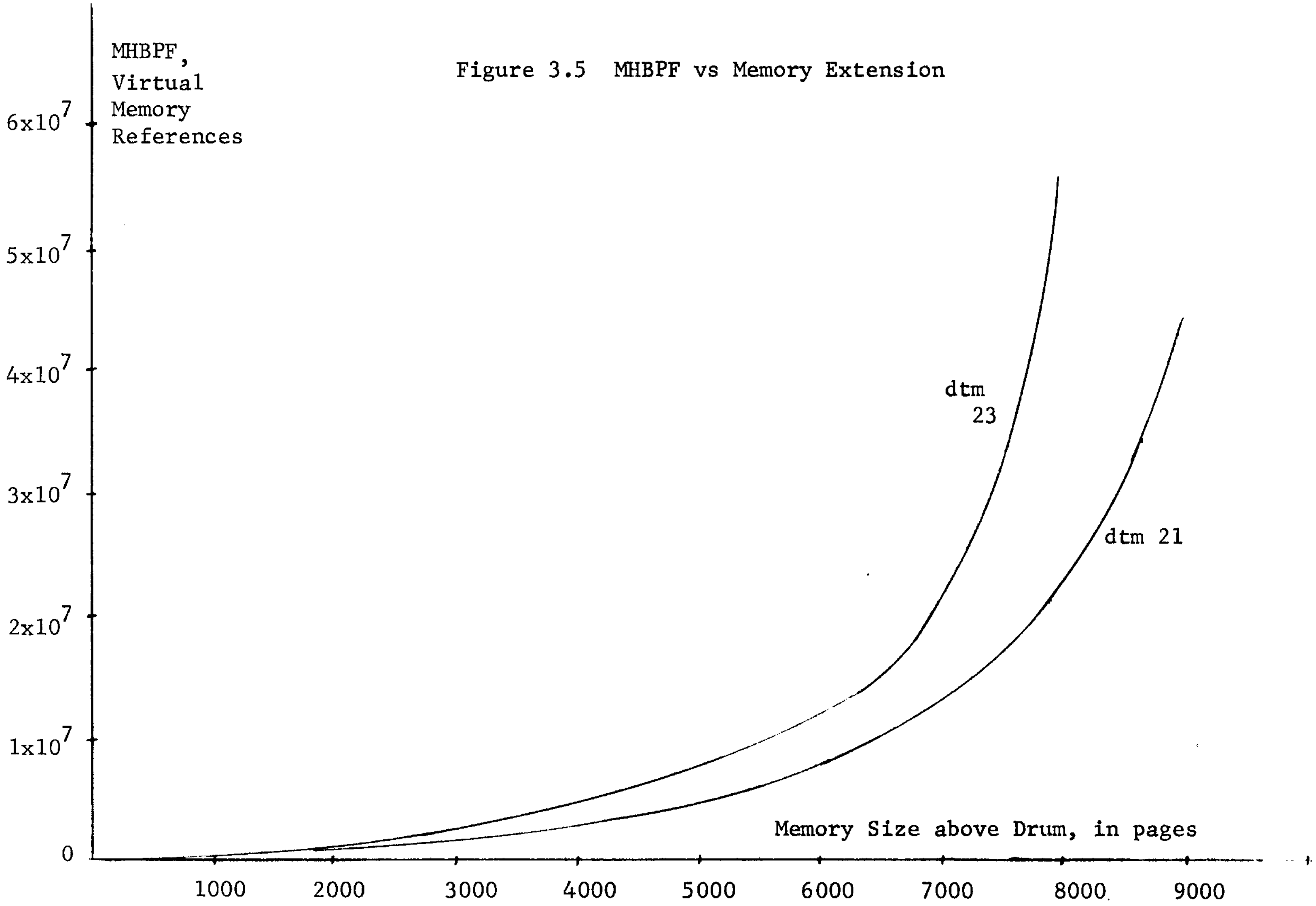


Figure 3.5 MHBPF vs Memory Extension

load on these days. The difference, however, is not very significant.

3.3 Reference Probability Models [illegible]

The characterisation of the reference [illegible] of a program or group of programs as an LRU stack depth reference probability distribution is considerably more [illegible] (MHBPF) as a system-independent characterisation of program behaviour. We proceed to derive p(x) from MHBPF(x) from equation 1.6.

Let K be MHBPF(n), for the [illegible] [illegible]. Let us represent the quantity [illegible] as S. Then equation 1.6 may be rewritten as

$$\mathrm{MHBPF}(x) = \frac{K\,S}{\sum_{t=x+1}^{n} p(t)} \qquad (2)$$

We write p(x) instead of f(x) in this derivation as we are assuming a sufficiently [illegible] [illegible] may be assumed to approximate p(x). Let [illegible] We then have

$$\mathrm{MHBPF}(x) = \frac{K\,S}{P(x)}$$

or, rewriting,

$$P(x) = \frac{K\,S}{\mathrm{MHBPF}(x)} \qquad (3)$$

We are interested in finding the probability of reference to position x, which must be beyond position n in the LRU [illegible]. If we are only considering references to positions beyond n, we define the probability of references to a position beyond n as [illegible]. Hence, S = 1, and

$$P(x) = \frac{K}{\mathrm{MHBPF}(x)} \qquad (4)$$

Now, by definition of P(x)

$$\frac{P(x+1) - P(x)}{1} = -p(x) \qquad (5)$$

[illegible]

$$p(x) = [illegible] \qquad (6)$$

Substituting (4) in (5),

$$[illegible] \qquad (7)$$

Hence, if [illegible]

$$[illegible] \qquad (8)$$

An 'exponential' paging model, as in equation 3.1, gives, from

MHBPF(x) = $ce^{\gamma(x-n)}$

$$p(x)_{exponential} = [illegible] \qquad (9)$$

Similarly, MHBPF(x) = cx^2 gives

$$p(x)_{quadratic} = \frac{2K}{x^3} \qquad (10)$$

and H(x) = bx^3 gives

$$p(x)_{cubic} = \frac{3K}{x^4} \qquad (11)$$

the polynomial headway function MHBPF(x) = αx^k giving in general

$$p_k(x) = \frac{kn^k}{x^{k+1}} \tag{12}$$

which subsumes (8), (10), and (11). The exponential model (9) is the only one of these probability distributions which is characterized by an independent parameter, γ. Letting $\lambda = 1/\gamma$, we rewrite (9) as

$$p(x) = \frac{1}{\lambda} e^{-(x-n)/\lambda} \tag{13}$$

λ has the dimensions of pages. It in some sense characterizes a 'radius of locality of reference' of the programs running. It is the mean fetch depth into the extension stack.

3.4 Accuracy of the Reported Results

The question of accuracy of the results of a supposedly deterministic simulation seems at first to be unnecessary. However, this simulation was based upon a measurement. Thus, the techniques used to interface this experiment to the Multics system (see Appendix B) became a source of inaccuracy. Furthermore, the behavior of anomalous pages (the so-called "global transparent paging device" pages) caused significant deviation from the assumed LRU model. The deletion of pages in LRU list created problems, as an inordinate amount of effort would have been required to handle these correctly (see Chapter 2).

Thus, we will consider three sources of inaccuracy: lost data, global transparent paging device pages, and list deletions.

3.4.1 The Effect of Lost Data

The loss of data was due to failure to retrieve it from the Multics supervisor before it was overwritten. This was a consequence of the circular buffer strategy chosen to solve the problem of real-time storage of this data. These strategies were discussed in detail in section 2.3.

The effect of these losses are twofold: some counters for stack positions were not incremented for lost data, and the ordering of the stack was affected by this lost data. We consider these problems separately.

3.4.1.1 Lost Counter Accuracy

In order to deal with either of these problems, we assume that lost data has no correlation to page reference patterns. We thus deduce that it shares the same distribution over stack position as the successfully accumulated data, and the shape of the resulting histogram is not severely affected by this loss. For the measurement "dtm 23", the most successful and accurate of thos made, 435 trace items were lost out of a total of about 200,000 successfully recorded items. This represents a total inaccuracy in counting of less than one quarter of a percent. For the slightly less accurate "dtm 21", 1200 items were lost at various times. Measured against the 150,000 items successfully collected here, this is still less than one percent.

3.4.1.2 Stack Shifting Inaccuracies

This type of inaccuracy, resulting from inaccurate reconstruction of the LRU extension stack, is considerably more subtle, and damaging in its effect. Failure to notice certain movement into and out of the extension stack causes the stack to stray progressively farther from a realistic reconstruction. Items remain in the simulated extension stack which were in fact removed by lost data, and items which should have been pushed on its top are not so pushed. The result of items not being removed correctly is twofold: first, the item will appear twice in the stack when it is pushed out legitimately onto the top of the extension stack in the future, and items further down the stack than the false appearance will have their stack position incorrectly recorded. The double appearance is only a problem because of the latter effect. The stack-managing algorithm of the simulation program used a bit table to record the known presence of every storage system page in the extension stack. Thus, the legitimate pushing (the second time) of such a page has no effect, and the later fetching of that page from the extension stack fetches the correct instance, and the bit table indicates the page as no longer being in the stack.

The result of items not being pushed because of failure to record their pushing is similar. Their absence at the top of the extension stack causes all items below them, which is the entire stack, to have their positions incorrectly recorded. These items appear one position higher than they should be, for each missing item. Thus, until the missing item is later requested from the stack, by virtue of a recorded fetch, all items which were on the stack before the failure to place the missing item will have their positions incorrectly tallied. Also, a later reference to the unpushed page will be recorded as a transient, not-in-stack page fetch, as discussed in section 2.2. The effect of not accounting this fetch to any stack position was already discussed.

Note that the effect of pushing and then fetching any single page has no effect on the extension stack orderings before the pushing and after the fetching. That is to say, if the pushing and fetching were not recorded at all, only the stack orderings between the two would be incorrect. Thus, if at any one time a data retrieval from the hardcore supervisor notices that X data items were lost, all such push-fetch pairs within the X lost data items have no stack-reordering inaccuracy associated with them, and only the lost-counter inaccuracy occurs. A reference-push pair, on the other hand, causes both types of inaccuracy. Although it seems evident, by locality of reference, that any string of contiguous lost data items must contain a large number of push-fetch pairs, i.e., a page recently pushed out of core-drum is one of the most likely to be fetched back in soon, a more careful mathematical analysis shows this to be false. Based upon parameters derived from the data accurately recorded in "dtm 23", 100 page fetches within a string of contiguous lost data items will statistically include only 4 fetches of pages pushed within the lost data items.

We must thus assume the worst case, that every lost data item was in fact a push or a fetch not properly matched within the lost data. Hence, for the 435 lost trace items in "dtm 23", the effect of losing this data could not have been worse than the pushing of 435 never-referenced-again pages on the top of the extension stack, or the excision of 435 random points from the stack. In the first case, the result is that later fetches are accounted to lower-numbered positions than they should have been, and in the second case, some later fetches are accounted to positions of higher number than they should have been. In either case, the total effect is that of an uncertainty of either +435 or -435 on the stack position axis of any derived graph. In reality, the lost data must contain an almost equal number of pushes and fetches. (By conservation of pages, the difference must be exactly the difference between pages created and destroyed in core-drum.) As a result, a typical misplaced page in the extension stack will suffer an average displacement of 435/2, due to the lost fetches. Hence, the total uncertainty in the stack position axis of any derived graph is not greater than plus or minus one-half the number of lost data items. For "dtm 23", this is 218 positions. Most of our graphs are plotted to a resolution of 500 stack postions. Compared to the 8000 or so positions of interest, this inaccuracy is not very significant.

Summarizing, the effect of lost trace data is seen both as lost accuracy in counting, and uncertainty in the stack-position axis of derived graphs. Both uncertainties are proportional to the amount of lost data.

3.4.2 Global Transparent Paging Device Inaccuracies

This type of inaccuracy results from the special handling of the system's top level directory pages, three in all. These pages are ousted from the core-drum combination prematurely, as dictated by a system reliability policy which attempted to insure the integrity of these pages by keeping them off of the drum. As a result, they were in fact used more often than the pages legitimately at the top levels of the extension stack, and thus, had no right to be in this stack at all. That is to say, they would have been on the drum at almost all times had they not been so special-cased.

The anomalous effect of these pages had been seen early in the work of this thesis. An experiment designed to discover the significance of this effect revealed that on some days, between one-half and one-fifth of all Multics disk traffic was a result of these special-cased pages. Thus, our experiment was modified to note in its trace data when such pages were being fetched or ousted from core-drum. The mechanism by which this information was transmitted was somewhat ad hoc, and is not shown in Appendix A.

The predominant inaccuracy caused by these pages is a distortion of the very low end of the f(x) and r(x) curves (see section 1.3). The stack-reordering inaccuracy created by these pages cannot be more than plus or minus three positions (as there are only three of these pages) at any point in time or stack, and is thus totally insignificant. As these pages rightfully belong on the drum, they are usually fetched very soon after they are ousted, and thus, never migrate very far down the stack.

Thus, many reference counts at low-numbered stack positions are attributable to these pages. If the core-drum combination were extended by any finite amount, and managed as currently done (i.e., at the time of the experiment), the anomalous references would still appear outside of core-drum. One should thus consider these references to be to stack position "infinity", meaning that they would be disk references no matter how far core-drum were extended, or simply "highly anomalous", and not considering them at all. The latter course, which we have chosen here, is equivalent to ignoring the effect of these pages on the extension stack ordering, and considering them to be outside the domain of the extension stack, that is, in core-drum. The effect of removing references to such pages on MHBPF(n) is easily calculated. Starting from equation 3.4, we multiply both sides by MHBPF(n), and obtain

$$\mathrm{MHBPF(E)} = \mathrm{MHBPF(n)} \frac{\sum_{t = n+1}^{\infty} r(t)}{\sum_{t = E+1}^{\infty} r(t)}$$

If E is greater than the deepest position in the extension stack to which any of the anomalous pages ever migrates, the only place in this equation where anomalous pages are counted is MHBPF(n). The effect of removing the anomalous fetches from this quantity is simply to scale it proportionately to the number of page fetches to be not considered. That is, if T page fetches (other than the "startup transient" fetches of section 2.2) were observed, A of them to the anomalous pages,

$$\frac{\mathrm{MHBPF(n)}_{\mathrm{adjusted}}}{\mathrm{MHBPF(n)}_{\mathrm{old}}} = \frac{\mathrm{MHBPF(E)}_{\mathrm{adjusted}}}{\mathrm{MHBPF(E)}_{\mathrm{old}}} = \frac{T - A}{T}$$

This ratio was observed to be between .92 and .96 for the measurements "dtm 23" and "dtm 21" displayed here.

Summarizing, the anomalous drum-abhorring pages create an inaccuracy of about 4 to 8 percent in the headways and headway ratios calculated from the measured data.

3.4.3 Inaccuracies Resulting from List Deletions

These inaccuracies result from a design decision to implement a simple, fast list-maintenance algorithm, as correct treatment of these deletions would require a fairly time-consuming technique. Hence, we proceed to analyze the extent of the inaccuracies resulting from this inaccurate treatment of deletions.

Recall from Chapter 2.2 that the deletion of pages in core-drum does not affect the ordering of the extension stack. Such a deletion implies that a fetch into core-drum will occur with no corresponding ousting. As this is in fact what happens, there is no inaccuracy involved with core-drum (or "out of list") deletions.

The deletion of a page from the extension stack creates a "moving anomaly", as discussed in section 2.2. All references to pages in positions in front of the anomaly (which occupies the position of the deleted page in the extension stack) are tallied correctly. The first reference to a page behind the anomaly is tallied incorrectly, because we chose not to record the anomaly. It is recorded as being one page closer to the top of the extension stack than it should have been. However, the anomaly now moves down to the position of that fetch. The situation is now the same as had the page just referenced been deleted. References in front of that position are tallied correctly, and exactly one reference behind it is tallied incorrectly, and the anomaly moves down.

We proceed to analyze the motion of such an anomaly down the extension stack. In the worst case, the page deleted was at the very top of the extension stack, and thus, the next reference is guaranteed to be tallied incorrectly. Probabilistically, this reference will be to that extension stack position which is the mean of the distribution f(x), the measured reference frequency distribution. There is a probability of q that this reference will be as far down the LRU stack that a fraction q of the weight of the distribution f(x) is below it. Now for the measurement "dtm 23", there were about 2000 in-list deletions per 50,000 in-list reads. This means that there was one deletion per 25 reads. In order to calculate the probability that a deletion anomaly is past a certain depth in the extension stack by 25 reads, we consider the experiment, tried 25 times, of encountering a read at least that far down the stack. The probability of success of one read being in that portion of the stack where a fraction q of the weight of f(x) is left is exactly q. The probability of a failure is (1-q). The probability of 25 failures is $(1-q)^{25}$. The probability of at least one success in 25 tries is one minus the probability of exactly 25 failures, or $(1-(1-q)^{25})$. For q = .1, i.e., the ninety percentile point of f(x), it is .93. For q = .05, it is .72. Hence, by the time the next deletion is recorded, it is quite likely that the anomaly generated by the previous deletion is quite far down the extension stack. Hence, for the upper portion of the extension stack, the effect of deletions do not cumulate. Hence, each deletion generated an inaccuracy of one stack position for each read behind it, but the corresponding anomaly moves sufficiently rapidly down the extension stack that the effect of later deletions are independent. Thus, for the upper portion of the stack, the result of these deletions is a total uncertainty of one stack position, a negligible amount.

The above reasoning correctly implies that the anomalies resulting from deletions accumulate at the lower reaches of the extension stack, and that fetches from these regions have a large cumulative error. In order to analyze this effect, we construct a queueing theoretical model of list deletions. A deletion anomaly causes the first fetch to a position behind it to be off by one position. Several deletion anomalies in the extension stack cause the first fetch behind them to be off by that many positions (the number of deletion anomalies). However, once this first fetch has occured, the next fetch from this position will be off by one less position, and so on, until all of the deletion anomalies have moved behind the position in question, and fetches from this position are tallied correctly. Thus, we may construct the following interpretation: The deletion anomalies in front of position p form a queue. Each fetch behind position p "services" one request. i.e., removes one item from the queue. The rate of arrivals to this queue, in the worst case (all deletions from the very top of the extension stack) is the rate of deletions. The rate of service is the rate of references to stack positions behind position p. The length of the queue is the number of outstanding anomalies in front of position p, which is the total error in stack position by which fetches from position p will be tallied. Assuming exponentially distributed arrivals and services, with respective means λ and μ, the average queue length at position p is known to be $L = 1/(1-\lambda/\mu)$ from queueing theory. λ/μ, the ratio of arrival rate to service rate, is the total number of deletions (assuming the worst case) divided by the number of references past position p, both immediately obtainable from the measured data. As there were 2000 total deletions*, queue length approaches infinity at the point in the extension stack where 2000 references were counted below that point. This point is at 3650 pages depth. At 500 positions

*In measurement "dtm 23".

before this, queue length is down to 4. At only 100 positions before it, queue length is 20. Hence, up to 3000 pages, the effect is negligible. At positions below that point where there are 2000 page fetches recorded below, the queue grows faster than it is serviced. At the time the experiment is stopped, the number of deletion anomalies remaining in queue in front of position p', where p' is beyond the 2000 reference point in the extension stack is the total number of deletions (in the worst case) minus the number of references beyond position p'. As both of these quantities presumably grow at a constant rate, the average error in stack position of a recorded reference to position p' is one-half this queue length. This allows us to reconstruct an approximation of the correct r(x) and f(x), and then all of the resulting curves, by recreating a better, more accurate r(x), r'(x) as

$$r'(x) = r(x) \text{ for } x < 2000 \text{ pages}$$

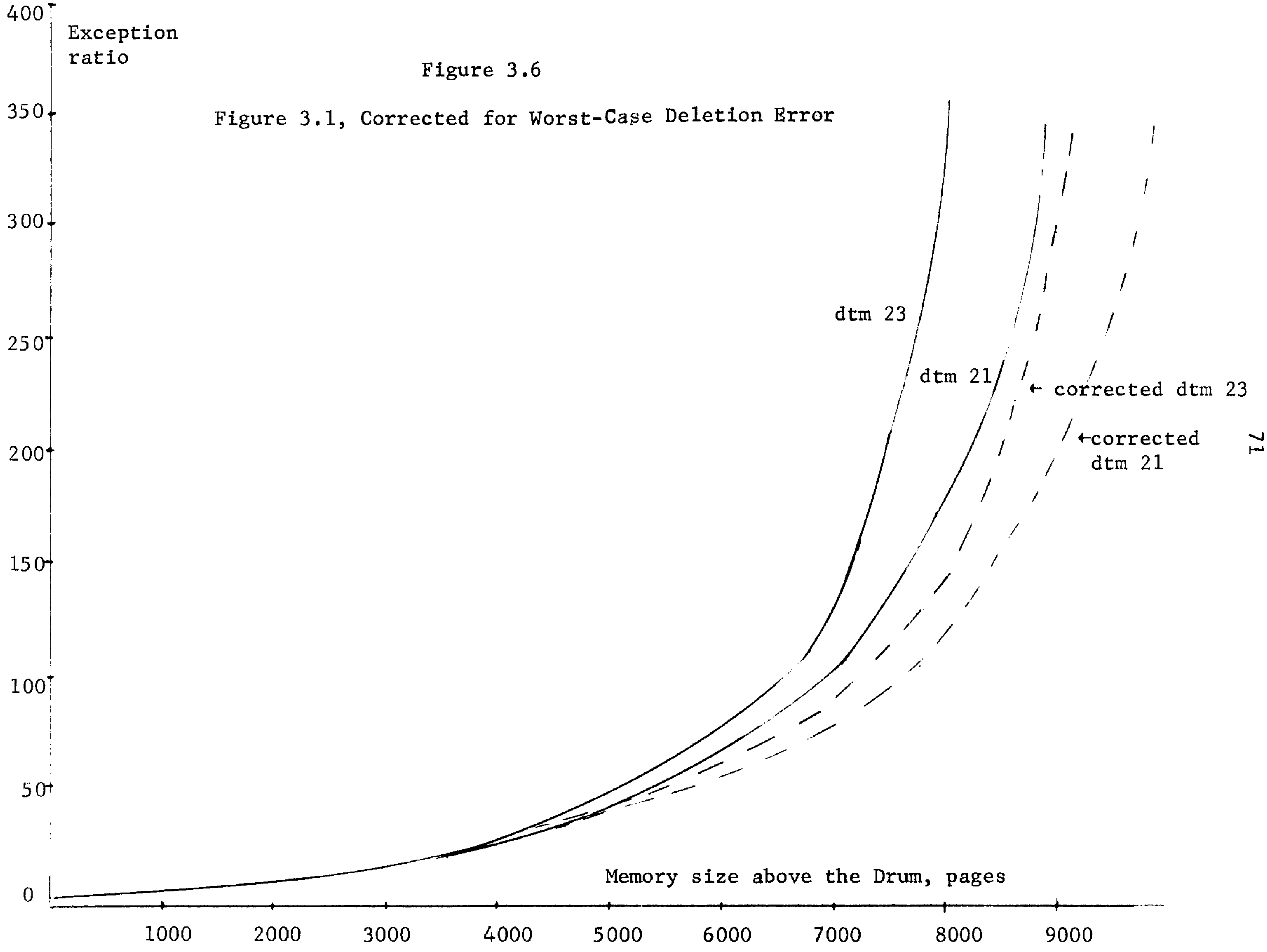
$$r'\left(x + \frac{2000-t(x)}{2}\right) = r(x) \text{ for} \geq 2000 \text{ pages}$$

$$\text{where } t(x) = \sum_{y=x}^{\infty} r(y)$$

This implies that at the very tail of the distribution, there is a stack position inaccuracy of 2000/2 = 1000 positions. At a stack depth of 5000 positions, there is an inaccuracy of 500 positions. This does not seriously affect the shape of the exception ratio and MHBPF curves in the region of interest as one can see from figure 3.6. We have re-plotted here figure 3.1 and corrected as above.

Figure 3.6

Figure 3.1, Corrected for Worst-Case Deletion Error



3.4.4 Other Inaccuracies

Another possible source of inaccuracy was the forced oustings of pages to disk directly from core, not due to global transparency to the paging device. A close inspection of "try_to_write_page" in Appendix A reveals that pages which should be ousted from core to the paging device (the drum) are occasionally ousted to disk, because there is no room on the paging device. This action avoids recursion in the process of finding a free core frame, as the latter process would otherwise possibly involve ousting pages from drum, which could require finding a free core frame. Although we do not have data on the frequency of this occurrence on the days of the experiment, we have observed Multics at other times, and the percentage of disk writes caused by such oustings is less than a tenth of a percent of all disk writes. It is true of Multics that the ratio of reads to writes remains fairly constant. Each read corresponds to one page fetch, and each fetch must be accompanied by an ousting at some time. Hence, 'forced' oustings must be a similarly small percentage of all oustings, and not a significant effect.

3.5 Correlation between dtm 21 and dtm 23

Observing figure 3.2, the correlation between the two plots is fairly remarkable. Within any reasonable accuracy for what is meant to be used in engineering approximations, these curves represent a measurement of the same quantity.

3.6 Our Result and the Linear Model Measurements

Our mean headway curve (figures 3.1, 3.3) shows distinct differences from the curves given in S1 for the mean headway function of the Multics system. We can attempt to rationalize these differences by understanding the nature of the user load during the two different experiments.

The measured mean headway between disk page faults, in terms of virtual memory references per disk fault, was between two and three times the figure measured in S1. What is more, the slope of the two curves differs, ours starting out at almost six times the slope of the curve S1. We attribute this to differing values of λ in equation 3.13, in terms of the model proposed in this experiment. More specifically, the 'tightness' of working sets was greater for our experiment, and the number of distinct users was fewer, causing even greater tightness of the system's "combined working set" at any time. The measurements given in S1 were made during a day of very heavy system usage, in August 1972. User load at this time consisted primarily of systems programmers engaged in program development, an activity which references vast extents of libraries, tools, and specialized procedure and data. These users were also operating without economic restriction, and thus had little incentive to minimize the resources used by their activities. Our experiments were conducted at a time when some of the Multics user load had shifted to the Honeywell 6180 Multics system, in a state of development at that time. All of the systems programmers had moved to the new machine at the time of our experiments, and the remaining user load was quite light, consisting of the M.I.T. academic community. The lightness of user load also implies a smaller number of distinct users.

The mean headway curves which can be extrapolated from our measurements may be viewed as a function of user load / system working set tightness, giving rise to the family of graphs of figure 3.7. From this, it can be seen that a linear region on a curve corresponding to a large system working set can correspond to a non-linear region on a graph corresponding to a smaller system working set, and the latter will rise faster than the former. If one draws the line C C' corresponding to the core-drum to disk boundary, both the differences on measured headway and slope differences can be more readily understood.

Another factor which gives rise to the family of graphs in figure 3.7 is the transient response of the experiment. As the length of the LRU extension stack grows, so does the observed value of λ. Especially when user load is light (smaller number of disk references per hour), it takes longer to develop curves of low λ than high λ, and this was the case in our measurements. Hence, it is possible that a more extensive measurement could have allowed a curve of higher apparent λ to result.

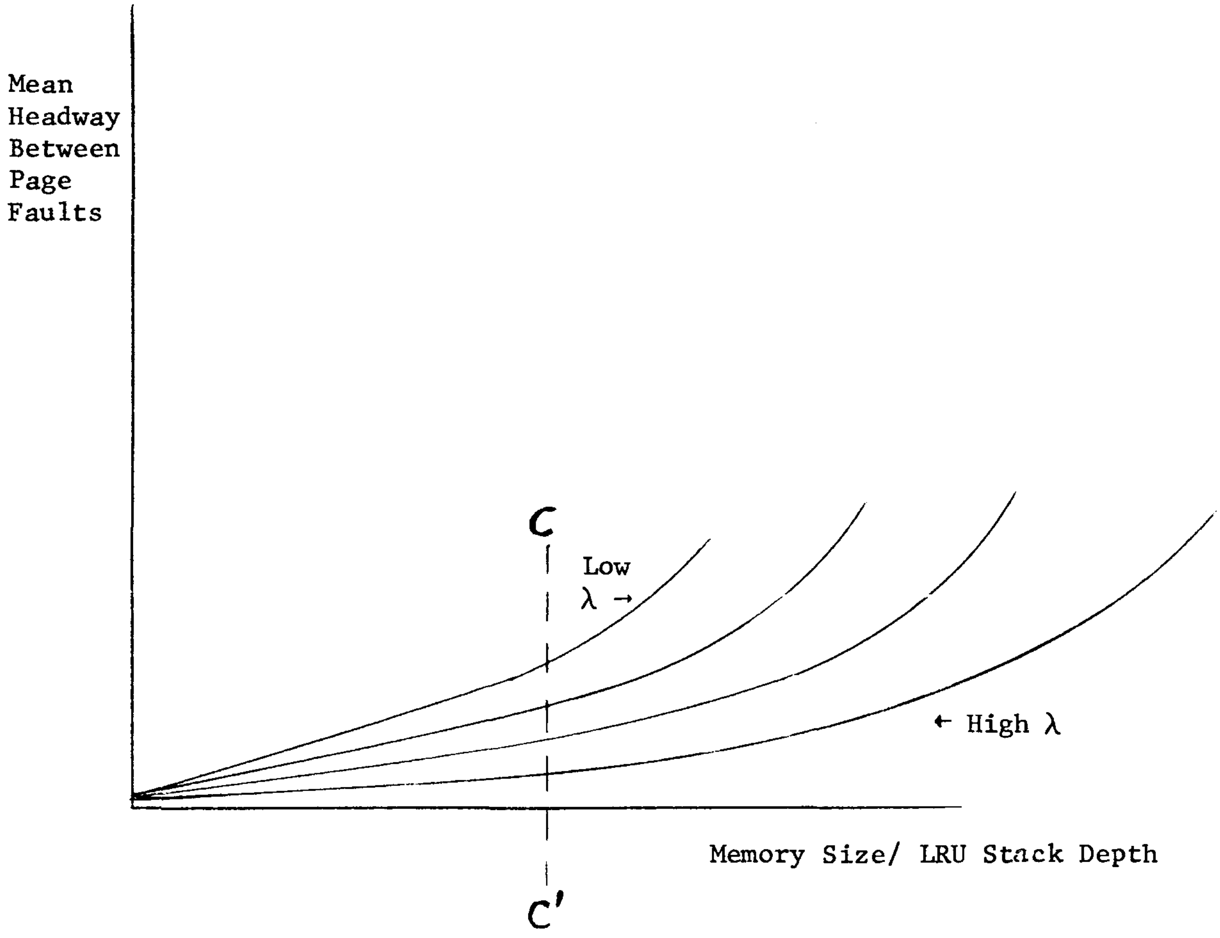


Figure 3.7 Family of Headway Curves of Differing λ

Chapter 4

Conclusions and Suggestions for Future Research

4.1 Conclusion

The most general and useful conslusion that can be drawn from this thesis is that increased primary memory size decreases page fault overhead in virtual memory systems very sharply as it grows, the decrease being much greater than proportional to the increase in memory size.

We hypothesize that the reference patterns observed, and the headway functions derived are characteristic of a large-scale computer utility being used by an academic community through interactive consoles. The data being referenced on Multics was accessed through a virtual memory mechanism: were it accessed via explicit disk requests on some other type of computer utility, we expect to see the same patterns and headway functions.

The most specific and concrete result which we have arrived at is a measurement of the mean headway function for Multics, showing how page fault overhead decreases as primary memory size approaches 4 x 10^8 bits.

4.2 The Paging Model Suggested

The mean headway function MHBPF(x), where x is primary memory size in pages, may be expressed as a polynomial in x,

$$\text{MHBPF}(x) = \alpha_0+\alpha_1 x+\alpha_2 x^2+\alpha_3 x^3 \ldots + \ldots \qquad (1)$$

Saltzer's measurements suggest that

$$\text{MHBPF}(x) = \alpha_1 x \qquad (2)$$

is an adequate characterization of the paging behavior of Multics in the range $0 \leq x \leq 1.3 \times 10^8$ bits. Our experiment shows, for the particular measurements we made, that the quadratic term in (1) becomes significant at x = approximately 1.0×10^8 bits. The trends indicated by figures 3.1 and 3.4 suggest that higher terms become significant as x is increased further. This would be consistent with the observation made abount arbitrary increase in primary memory size on a virtual memory system made above. Belady and Kuehner (B3) assert MHBPF(x) to approximate $\alpha \cdot x^k$ for 'real life programs'. In measurements made on IBM M44/44X and System 360/67 machines, they found k to take values 'in the vicinity of 2'. This model also can be described by the general representation of equation (1) above. These observations also suggest, as does figure 3.2, that

$$\text{MHBPF}(x) = (\alpha_0-\beta) + \beta e^{(\frac{\alpha}{\beta})x} \qquad (3)$$

is in some cases a valuable approximation. The constant term becomes insignificant for sufficiently large x, and we may write

$$\text{MHBPF}(x) = \beta e^{(\frac{\alpha}{\beta})x} \qquad (4)$$

a very simple model which is very appealing. As was shown in Chapter 3, this model corresponds to a reference probability distribution

$$p(x) = e^{-x/\lambda} \qquad (5)$$

where x is LRU stack depth and p(x) is the probability of reference to that position. This simple model of program behavior is particularly appealing, as it characterizes "program size" as a distribution. Denning (D1) has given the concept of 'working set' as a measure of program size, within a given time interval. Equation (5) is or a more specific class of program characterizations, expressing the 'size' of the program as a distribution. The parameter λ may be viewed as a 'radius of locality' of the programs running, expressing in some sense their 'tightness' or 'togetherness'. In this sense, λ is akin to the concept of working set.

4.3 Unanswered Questions and Future Directions

The most obvious extension of the work presented here is to extend upward the range of primary memory sizes for which the nature of MHBPF(x) is known. Although the techniques used in this thesis are completely extensible in this regard, it is not clear whether or not there is any value in such research beyond some point. If a certain amount of primary memory reduces secondary memory accesses to once an hour, for instance, the issue of secondary memory access time versus cost quickly takes precedence over primary memory cost versus secondary memory reference overhead. For instance, our measurements predict that another seven million words of core-drum would reduce disk references to once every two minutes. At this rate, the economic viability of a fast disk is a greater issue than the performance improvement resulting from more core-drum. For instance, a large (10^{12} bit) slow (1 sec access time) store might be quite acceptable as a backing store.

Another area of research is to fully understand the program behavior patterns which are responsible for models of program behavior such as Saltzer's linear model and the model proposed above. We understand the working set model because we know that program loops, subroutines, etc., cause repeated reference to certain data items, and this behavior is somewhat extensible to larger views of programs. We do not know what "causes" the linear model, or other such models in this sense. We can understand "distribution" type models by the same considerations of 'spatial locality' and 'temporal locality' (M2) on which the working set model and the LRU replacement algorithm are based, but we have no insight into the basis for any particular distribution in program behavior.

A very major unknown limitation on the work of this thesis is the applicability of its result. No attempt has been made to determine precisely what aspects of Multics user behavior were responsible for the effects noticed. It is not even clear how specific the results of this thesis are to the LRU algorithm.

An interesting direction of research would be to perform a similar experiment on some LRU-managed system which does not utilize virtual memory. A large data management system utilizing an LRU-managed buffer pool might be such a system. The common aspects of user behavior accessing a large on-line data base might show through here, as it is the referencing patterns, not referencing methods which are interesting.

Systems such as IBM's OS/VS2 and VM/370, involving paged virtual memories and multiple address spaces provide a fruitful ground for comparison. In these systems, features of sharing and data addressing are quite different than Multics, but the amount of data to be addressed and the primary memory usage strategy are not altogether dissimilar.

An important direction to be pursued is that of repeating this experiment on Multics, reliably, many times, and determine day-to-day and hour-to-hour variations in the behavior of MHBPF(n). The thrust of this thesis was to develop and apply the techniques stated herein, and others must use these tools to correlate MHBPF (n) to whatever factors appear as influential.

An interesting issue is to relate the parameters λ and α_i in all of these models of program behavior to other observable parameters of program behavior and system configuration. This would constitute a long step toward theoretical understanding of the behavior which underlies these models.

Sekino (S2) shows the significance of MHBPF(x) in system performance calculations, particularly throughput and response time. Although our results may be used in these calculations, we have not pursued this course here.

Appendix A

A Structured Program Description of Multics Page Control

This appendix describes the functioning of the fault and interrupt driven mechanisms within the Multics virtual memory management algorithm as it existed in May, 1973, at the time of the experiment. Only the paths within the so-called 'Page Control' subsystem relevant to this thesis have been shown. This excludes some fairly complex mechanisms relating to error handling and the allocation of page tables. Within the paths shown here, however, this results in only a few small omissions.

The aim of this appendix is to familiarize the reader with the internal operation of page control to whatever depth is necessary for comprehension of the rest of this thesis, particularly Chapter 2 and Appendix B. To this end, we have provided a description on several levels.

The most detailed description of page control given here is an approximately "structured" program, in which we have functionally modularized page control into 14 small routines. We have taken the liberty of creating a new language in which to write this program, which we explain within. We feel that this language conveys the general class of manipulation described herein with a maximum of clarity and succinctness.

We have liberally renamed objects, substituting names which we feel are more mnemonic than the actual names used in Multics. We have also made minor modifications to control flow, and subroutinized routines which were not originally subroutines where we felt that clarity would be aided. In any case, the algorithm as given is essentially identical to the actual assembler-code algorithm at the time of the experiment, with respect to state, sequencing, and side effects.

The plus sign (+) in the left-hand margin denotes references to routines explained in detail within.

A Brief Overview

Multics manages both core and drum (the latter known as the "paging device", or "pd") by approximations to the least-recently-used algorithm. Two lists, the core used list and the paging device used list are maintained for this purpose, the top of each list designating the least recently used page (which is the best choice for replacement), and the bottom of each list designating the most recently used page (which is the worst choice for replacement) on the respective devices. How these lists are maintained can best be learned by reading the program that we have provided. The core used list contains logical descriptions of core frames, including pointers to descriptions of logical pages and/or paging device records when such entities may be associated with the core frame. Similarly, the paging device used list contains logical descriptions of paging device records, including pointers to descriptions of logical pages and core frames, when such entities may be associated with the paging device record.

Multics tries to maintain copies of the most recently used P pages (where P is the size of the paging device, in records) of the storage system on the paging device. The most recently used C pages (where C is the size of core memory in page frames) are to be in core, as well. (It is assumed that C is less than P.)

Thus, pages being ousted from core may be written to the paging device, even if a good copy exists on disk. This fact should be kept strongly in mind when reading "try_to_write_page". Except for the case where the paging device has no copy, pages which were identical to pages in secondary storage are never written out. Pages of zeros are never written out, but their logical description is so modified that they are created in core when faulted on.

The processor hardware maintains usage information about a logical page in a hardware descriptor. Specifically, the occurence of usage and/or modification is noted in the descriptor.

A page fault is resolved by finding a page of core into which to bring the page, and bringing it in. Finding a page of core consists of reorganizing the core used list to reflect the latest usage information, and finding the least recently used page frame, and using it. Pages which have been marked as modified cannot be claimed in this way, but are written out. When the writing is complete, at some future time, the page will be in the same state as a page which has not been recently used or modified, and will be claimed in the handling of some future page fault. Note that this 'writing' consists of initiating the physical operation, but not waiting for it to complete. It is at this writing time that secondary stoage is allocated, and pages containing zeros are noted. It is at the time that zero pages are noted and that secondary storage is deallocated.

At the beginning of page fault handling, housekeeping is performed on the paging device, which consists of trying to insure that at least ten records are either free or in the process of being freed. This is done by removing as many of the least recently used pages on the paging device as necessary. When a page is so moved, it is checked (via software-maintained switches) to see if it is identical to a copy on disk. If so, it may simply be deallocated from the paging device. If not, a sequence known as a <u>read-write sequence</u> (rws) must be performed. This sequence consists of allocating a page of core to be used as a buffer, reading the page into it from the paging device, writing it to disk, and deallocating the paging device copy. The core buffer is then freed.

A page fault which occurs on a page for which a read-write sequence is in progress causes an event known as an _rws abort_ to occur. The freeing of the buffer page and the paging device page are inhibited, and the buffer page is used as the core copy of the page, and the fault is resolved.

An Explanation of the Language Used to Express this Description

The language which we have used to describe Page Control is a bastardization of PL/I, with new primitives for some basic operations (enqueue, masked procedures, etc.) and an Algol 68-like formalism for representing relationships among structured entities.

Underlined words are language keywords. Lower-case identifiers represent names of subroutines, functions, or labels. Identifiers beginning with an upper-case character represent references to cells, which will be described below. Statement syntax is essentially the same as PL/I, but ":=" is used for assignment, and "=" is used to test equality. There is no lexical nesting of procedure or begin blocks.

A program consists of begin blocks, entered from the outside world in some unspecified way, procedures and functions, and declarations. declare (dcl) declarations may appear anywhere, including outside of blocks, and are global in scope. They define the class and type of variables, and the types of Objects used by the program. local declarations appear within blocks, and define a local scope of variables, identical to that produced when a variable is used as a formal parameter in a procedure or function.

The point of this language is to associate cells with values. The domain of values is the space of Objects. Objects are unique. Two cells have equal values if and only if their values are the same Object.

There are three classes of Objects: primitive Objects, structured Objects, and set Objects. Within each class, there are different types of Objects. Objects have no names. Only primitive Objects can be referred to explicitly, i.e., other than by reference to a cell having the desired Object as a value, or a function returning the desired Object.

Primitive Objects can be of three types. The first is boolean. There are exactly two boolean Objects. One can be referred to explicitly as true, the other false. The second is arithmetic. There is a first-order infinity of these objects, which are actually the integers. They can be referrred to explicitly as 75, 16777216, -283, etc. The third is literal. They are simply arbitrary primitive Objects, whose only useful property is their uniqueness. They can be referred to explicitly as "foo", "bar", "no stuff", etc. They are not character strings in any sense, but simply unique primitive Objects of type literal.

Structured Objects consist of a finite number of cells. Any cell can have as a value only one type of Object (implied is one class, as well). These cells are called components of the Object. These cells do have names, and they are specified in a declaration which describes the concerned type of structured Object.

Set Objects consist of an ordered set of Objects of the same type and class. All references except enqueue and dequeue, however consider the set Object as unordered. One can add to or enqueue to a set Object, remove from or dequeue from it, ask if a given Object is a member of it, or cause a cell to be assigned successive values, each value being a different Object in the set Object, in no particular order.

Variables are the other type of cell. A variable can hold only one class and type of Object, just like the other type of cell, the structured Object component.

Assignment (performed by ":=" operation in do statements and assignment statements) consists of replacing the value of a cell with another value, i.e., changing the value of the cell. The Object which was the previous value is neither changed nor destroyed in any way.

Binding consists of saving the value of a variable when a procedure, function, or begin block is entered, and restoring it when it is exited. The latter operation is called unbinding. All assignments and bindings made between the time a variable is bound and the corresponding unbinding have a transparent effect when the block performing the binding is exited. A local declaration of a variable in a block causes such a binding to take place for that variable when the block is entered, and the corresponding unbinding. Binding also takes place for variables used as formal parameters to procedures and functions. In this case, after the old value is saved, the value of the corresponding formal argument is assigned to the variable. Hence, all calls may be seen as "call by value".

To refer to an Object, one can either refer to a cell containing it, or, if it is primitive, one can refer to it explicitly. To refer to a variable, simple state its name. To refer to a component of a structured Object, state its component name, an open parenthesis, a reference to the structured Object, and a close parenthesis.

An assignment is a reference to a cell, ":=", and a reference to an Object of the same type and class declared for that cell.

Variables need not be declared. The default class of any cell is structured, with a type the same as its name. The syntax for a structured Object type declaration is as follows:

$\left\{\begin{matrix}\text{declare}\\ \text{dcl}\end{matrix}\right\}$ structured Foo (compdcl-1, compdcl-2,...compdcl-n);

[ ] = optional { } = select one

The compdcls, or component declarations, are of the same syntax as variable declarations, except that the name is the name of the component, and the optional keyword variable is illegal.

The syntax for a variable or structured Object component declaration is as follows:

$\left\{\begin{matrix}\text{declare}\\ \text{dcl}\end{matrix}\right\}$ [variable] Foo [type][objtyp]

where objtyp is either boolean, literal, arithmetic, any structured Object type named in a structured Object type declaration, or set objtyp, where objtyp is, recursively enough, any possibility named in this sentence.

local declarations only name their variable, although they can declare its type as well.

do statements differ from PL/I in that any cell can be used to the left of the ":=", not necessarily variables. The particular form "do Foo: = range Bar" means that the value of Bar is a set object, and the do is to iterate over each Object therein, in no special order.

The special constructor function construct is used to create new structured objects. The syntax of a reference to it is

construct Foo (compname-1:object-1,compname-2:object-2...),

whose value is the new Object.

The unique Object "null" can be used as a value of any cell. It has all types and classes.

The predicate void takes as an argument a reference to a set Object, and returns true or false (boolean Objects), depending on whether or not it is empty. The operators "=" and "≠" may be used to test if two references are equal, i.e., refer to the same Object. An appropriate *boolean* Object is returned as a value. The operators "*or*", "*and*", and "*not*" operate on *boolean* Objects in the obvious way. The conventional arithmetic operators operate upon *arithmetic* Objects, returning an *arithmetic* Object with the expected value.

if statements have as their predicate a reference to *boolean* Object.

A *call* statement consists of the word *call* followed by either a procedure name and an optional argument list or a complex function reference and an argument list. An argument list is a parenthesized list of (possibly zero) references to Objects separated by commas. A complex function reference is a function reference to some outside-of-the-language function which will return as a value a *procedure*, which one depends on the arguments to the function, which will be called by the call statement, with the arguments to the call.

The evaluation of arguments in *or* and *and* is conditional, as in Lisp 1.5 (M3) and proceeds from left to right.

A Program to Find the Man Who Owns the Black House, and Have Him and His Father Switch Houses

```
declare structured Person
   (Father type Person,
    House);
declare structured House
   (Color literal,
    Owner type Person);
declare Son Person, House2 House;
declare Brooklyn set House;       /*assumed to be initialized*/


switch_houses:begin;
   do House := range Brooklyn;                    //search the set "Brook-
                                                 // lyn"
      if Color(House) = "black" then do;          //found him
         House2 := House(Father(Owner(House)));   //find the other house
         Son := Owner(House);                     //remember who is the
                                                 // son
         House(Son) := House2;                    //Son now owns House2
         Owner(House2) := Son;
         House(Father(Son)) := House;             //Father owns house
         Owner(House := Father(Son);
         return;
      end;
   end;
end;
```

A Top-Level Programmatic View of Page Control Activity

A page fault causes the following: (page_fault)

The paging device is housekept.

Transient conditions such as i/o in progress or an rws on the faulted page are noticed and handled.

A free page is claimed, and the faulted page is read or created into it.

If i/o was started, the page is waited for.

Finding a free page consists of the following: (find_core)

The core used list is searched for a good candidate.

Recently used pages are not good candidates. They are skipped, and re-judged as not-so-recently used for next time.

Pages which have been modified (stored into) cannot be claimed now. They are written out, and re-judged as not to have been modified.

A page which has not been modified, and has been used approximately less recently than any other page, is pre-empted from its core frame, and this core frame is the new free page frame.

Writing a page out consists of the following: (write_page)

The page's contents are checked, and if all zeros, the page is flagged as not needing to be read or written - No writing takes place, and disk and paging device space allocated to the page are freed.

The page is given a residence on disk, if it does not already have one.

The page is given a residence on the paging device, if it does not already have one, and one is available.

The page is written out to its residence on the paging device, if it has one, otherwise to disk. The completion of i/o is not waited for.

Housekeeping the paging device consists of the following: (get_free_pd_record)

An attempt is made to insure that there are ten paging device records free or being freed, which is done as follows:

The pd used list is searched for a good candidate to pre-empt.

The search is made starting at the least-recently used pd record.

Records which contain pages in core are recently used. They are rejudged as such and skipped.

Records containing pages identical to pages on disk are acceptable. The pages in them are pre-empted, and the record is now free.

Other records have to be written back to disk, which is done by performing a read-write sequence (rws) on them.

Performing a read-write sequence on a page consists of the following: (start_rws,rws_done)

A free page of core is obtained.

The page is read into it from the paging device.

When the read is completed, the page is written out to the disk.

When the write is completed, the page of core and the paging device record are freed.

A page fault on the page involved in the sequence at any point during it causes the sequence to be aborted at the next complete operation in the sequence, and the core page is used as the page's home in core.

A Top-Level Description of the Objects Used by Page Control

A Page Object is the logical description of some page of the storage system, as opposed to a page frame on some device.

A Descriptor Object, in actuality a "page table word", is the physical descriptor by which a processor accesses a page. It contains a core address, usage bits, and a bit which causes a fault when off.

A Coreadd Object describes a physical core block. It describes the status of this block, including, implicitly, its position in the core used list.

A PDrec Object describes a paging device record, or frame. It describes the status of this frame, including, implicitly, its position in the paging device used list.

A Devadd Object represents a physical disk or drum address, and its contents. Included in this object is an identification of the device on which this page frame resides.

An Io-status Object is a hardware-generated object, which describes an input-output operation which has completed.

An Io-program Object is a sequence of commands for the system i/o controller to give to an i/o device. It specifies the type of operation required, the record within the device concerned, and a core address concerned.

A *Trace-Datum* Object is a recorded datum of information about traffic between disk and core-drum, for the purpose of the thesis experiment.

```
/*   Description of the structured Object types used by Page Control.

       Recall that  the default type of a structured Object component is
       the same as its name. */

dcl structured Page                    // Represents a page of some segment of Multics, as opposed to
                                       // a page of core or some device.
           (Descriptor,                //  The hardware descriptor by which processors access the contents of Page.
           Devadd,                     //  The physical disk or pd address  from which Page should be
                                       // read or written to.  If On_pd is true, is a pd address. Otherwise,
                                       // it is a disk address.  A  Devadd of "null", however, represents a
                                       // page full of zeros.
           Coreadd,                    //  The core frame associated with this page. Valid only when
                                       // Addressable (Descriptor (Page)) is true.
           PDrec,                      //  If On_pd is true, this is the pd record used by Page.
           Event literal,              //  Some literal quantity unique for each page.  Used to identify the
                                       // occurence of events associated with this page in interprocess signaling.
           Io_in_progress boolean,     //  Truth indicates i/o in progress, or at least not known to have
                                       // completed, on Page.
           On_pd boolean,              //  Specifies that Page has an allocated PD record, namely PDrec (Page).
           Wired boolean,              //  Indicates that Page must always remain addressable.
           Gtpd boolean);              //  Indicates that Page is forbidden to go on the pd, for reliability reasons.


dcl structured Descriptor              //  Represents a page table word (ptw), the physical descriptor by
                                       // which processors access a page.
           (Phys_Coreadd arithmetic,   //  The physical core address occupied by the page to which this  Descriptor
                                       // belongs.  Valid if and only if Addressable is true.
           Addressable boolean,        //  Truth allows Phys_Coreadd to be used by the processor. Falsity
                                       // causes the procedure "page_fault" to be executed.
           Usage boolean,              //  Set by the hardware whenever this Descriptor is used,
                                       // or more accurately, fetched into the associative memory.
           Modified boolean);          //  Set by the hardware whenever a store-type operation is
                                       // performed using this descriptor, or an associative memory
                                       // copy thereof.

dcl structured Coreadd                 //  Represents a core page frame.
           (Page,                      //  "null" represents an unallocated page frame.  Otherwise, the
                                       // Page contained in this frame.  This is only for normal page-holding
                                       // use, not rws's.
           Phys_Coreadd arithmetic,    //  The physical core address represented by this frame.
           Next type Coreadd,          //  The next more recently used core frame.
           Previous type Coreadd,      //  The next least recently used core frame.
           Io_read_or_write literal,   //  If Io_in progress(Page(Coreadd)) is true, or Rws_in_frame,
                                       // tells which direction of i/o is being performed.
           Rws_in_frame boolean,       //  Signifies an rws in progress in this frame.
```

```
          PDrec);                          //  Used only if Rws_in_frame is true. Specifies PDrec having an rws.


dcl structured Devadd                      //  Represents  a physical device address.
          (Device literal,                 //  Identifies a secondary storage device.
          Phys_Devadd arithmetic);         //  Identifies a physical record number on some device.

dcl structured PDrec                       //  Represents a paging device (pd) record.
          (Page,                           //  If In_use is true, describes the Page on this record.
          Diskaddr type Devadd,            //  If In_use is true, describes the disk address occupied by our page.
          Devadd,                          //  The physical device address of this record.
          Coreadd,                         //  When Rws_in_progress is true, describes the core frame
                                           // being used as an rws buffer.
          Next_pd  type PDrec,             //  Describes the next more recently used pd record.
          Previous_pd type PDrec,          //  Describes the next least recently used pd record.
          Event literal,                   //  A unique literal associated with this pd record.  Used to identify the
                                           // latter in interprocess signaling.
          In_use boolean,                  //   Tells if this pd record is in use or free.
          Rws_in_progress boolean,         //  Signifies that an rws or rws abort is  in progress in this pd record.
          Incore boolean,                  //  Signifies that the page in this pd record is in core right now.
                                           // Used for maintaing the LRU ordering of the pd used list.
          Modified_from_disk boolean,      //  Truth indicates  that the pd copy of page   is different
                                           // than the disk copy.
          Abort_flag boolean,              //  Turned on to start an rws abort by some process faulting on an rws'ing
                                           // Page.
          Abort_complete boolean);         //  Signifies that post_page (q.v.) has aborted an rws, and a cleanup
                                           // by rws_abort (q.v.) is expected.

dcl structured Io_program                  //  A portion of a channel program.
          (Direction literal,              // Indicates read or write.
           Phys_Devadd arithmetic,         // Physical device address involved.
           Phys_Coreadd arithmetic,        // Physical core address involved.
           Next type Io_program);          // Next program in channel queue.


dcl structured Io_status                   //  Represents a completed I/o operation to an
                                           // I/o control routine.
          (Phys_Devadd arithmetic,         //  Identifies the  physical device address involved.
           Phys_Coreadd arithmetic,        //  Identifies the physical core address involved.
           Io_program,                     //  The Io_program object which initiated the operation
                                           // causing this status to be generated.
           Coreadd);                       //  The Coreadd object associated with Phys_Coreadd.  Although
                                           // not actually present here, the one-to-one mapping between
                                           // Coreadd's and valid Phys_Coreadd's lets us use this here for
                                           // clarity.

dcl structured Trace_datum                 //  An item of trace data for the experiment.
          (Devadd,                         //  The disk address concerned.
           Type literal);                  //  The direction and description of logical motion.
                                 //  "read" -- a page fault to disk or an rws abort
                                 //  "write" - an rws initiation
                                 //  "virtual" - an ousting from core to disk
//  "pd_virtual" - an ousting from   the pd to disk
//  "delete" - the deletion of a page.
```

The Global Variables Used by Page Control

dcl

Page_table_lock literal, A quantity used to insure that only one processor at a time is in page control. A process desiring to "hold" this lock loops continuously until it is unlocked, and then locks it.

CoreTop type Coreadd, The least recently used Coreadd Object.

Writes_outstanding arithmetic, The number of write operations started which have not yet been known to complete. Used as a heuristic to call post_any_io.

Rws_active_count arithmetic, The number of read-write sequences which have been initiated and not yet known to be completed.

Number_of_free_pd_records arithmetic, The number of paging device records free or in the process (rws) of being freed.

Top_of_pd_used_list type Pdrec, The least recently used PDrec Object.

Channel_Queue type Io_program, The executable queue of i/o programs for a disk or drum.

Experiment_active boolean, Tells if metering experiment is in progress.

Trace_queue set Trace_datum, The total of all trace data accumulated by the experiment.

Undocumented Routines Referenced in this Program

page_wait (literal)
Suspends the calling process until a call to notify is made with the identical literal. page_wait also unlocks the page_table lock once the traffic control data bases are locked.

notify (literal)
Causes any process which called page_wait with the identical literal to be resumed.

clear_associative_memory
Causes all processors to clear their associative memories. This routine does not return until all processors have indicated that they have done so. Used to force access turnoffs and modified bit turnoffs to take effect.

allocate_disk_record()
Returns an unallocated Devadd Object. Marks it as allocated.

relinquish_disk_space(Phys_Devadd)
Marks a Devadd Object as unallocated to allocate_disk_record.

start_io(Io_program)
Starts a channel executing an i/o program.

thread_to_top(Coreadd)
Changes core used list and value of CoreTop such that Coreadd is moved to the top of the Core used list (least recently used). CoreTop now = Coreadd.

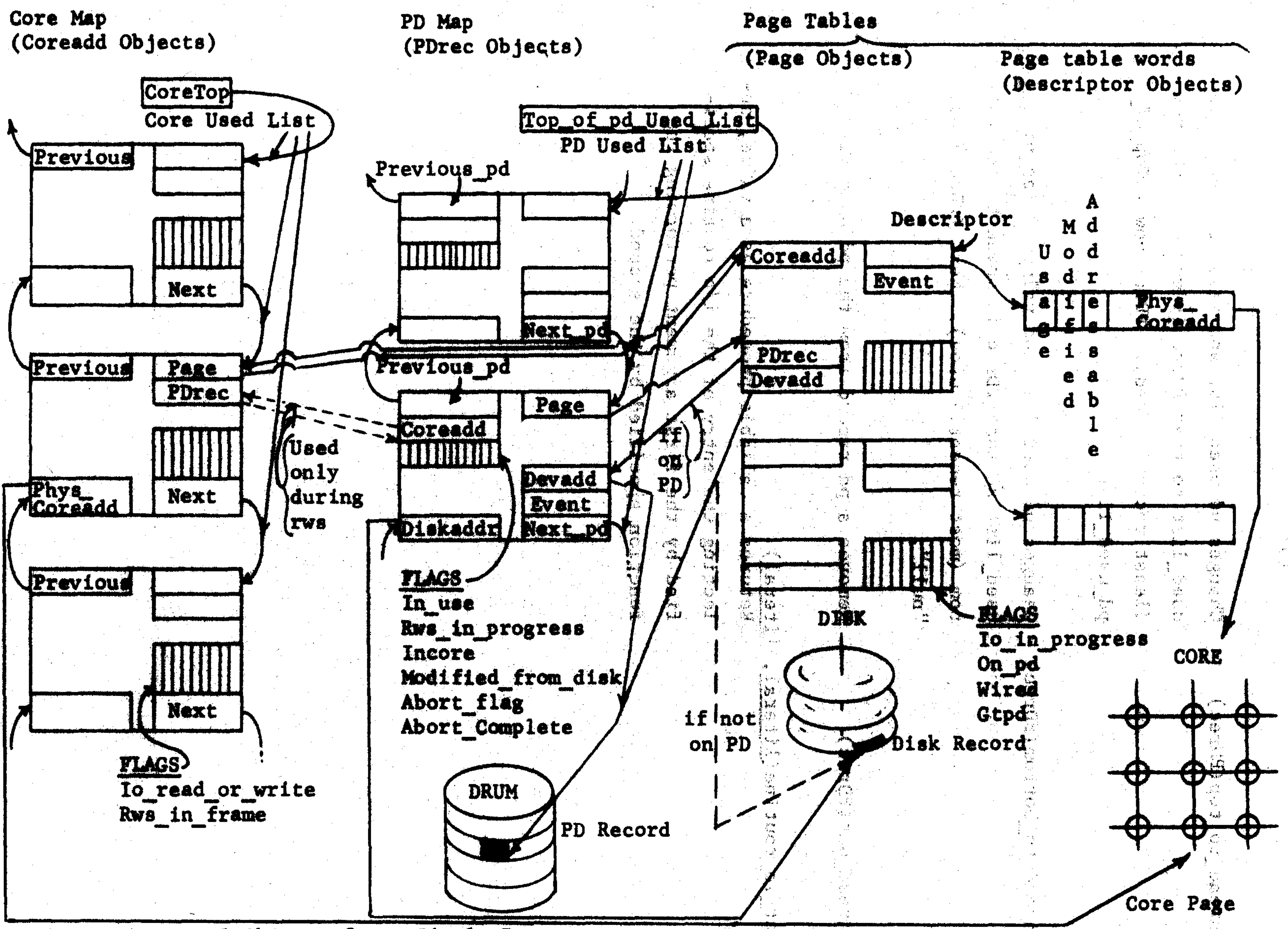
thread_to_bottom(Coreadd)
Changes core used list and value of CoreTop such that Coreadd is moved to the bottom of the core used list (most recently used). Next (Coreadd) now = CoreTop.

pd_thread_to_top(PDrec) Changes pd used list and value of Top_of_pd_used_list so that PDrec is moved to the top (least recently used). PDrec now = Top_of_pd_used_list.

pd_thread_to_bottom(PDrec) Changes pd used list and value of Top_pf_pd_used_list so that PDrec is moved to the bottom (most recently used). Next(PDrec) now = "nil".

pd_thread_out(PDrec) Removes a pd record from the pd used list.

select_io_routine(literal, literal) Returns a callable procedure as a value, selecting the entry of the I/O routine specified by the first argument, performing the function specified by the second.



The Page Control Objects for a Single Page

Figure A.1

page_fault

```
dcl Scu_data  Virtual_reference;
dcl structured Virtual_reference (Page, Segment);
page_fault: begin    masked;                               // This procedure is transferred to when the
                                                           // hardware determines that a reference has been
        local Page;                                        // made through a Descriptor whose Addressable bit
                                                           // is false. Return from this procedure causes
                                                           // a second attempt to make that reference.
                                                           // This procedure is entered in such a way
                                                           // that all external (i.e., i/o, etc) interruptions
                                                           // are disabled when it is entered. They are reenabled
                                                           // when it is exited.  This is because such
                                                           // interruptions might try to lock the page-table lock.

        set_lock Page_table_lock;                          // Prohibit access to page control by other
                                                           // faulting processors.
        Page := Page (Scu_data );                          // Determine from processor state at fault time which
                                                           // page was faulted on.
        do while Number_of_free_pd_records < 10;           // Housekeep the paging device - try to have some free
                                                           // pd records for the find_core calls which will
            call get_free_pd_record;                       // surely follow.
        end;

        if  Addressable (Descriptor(Page))                 // It is possible that we took a page fault while the
            then unlock Page_table_lock;                   // page table was locked, and the process holding the
                                                           // lock brought the page in. Exit if this is true.
            else if Io_in_progress (Page)                  //   It is possible that we took a page fault on a page
                                                           // which some other process has started bringing in.
                then call page_wait (Event(Page));         // Wait for it.
                                                           //  The system routine page_wait causes the
                                                           // suspension of the calling process until
                                                           // some other process calls the routine notify
                                                           // with the identical  Event with which page_wait
                                                                   // was called.  Page_wait also unlocks the
                                                           // page table lock once he has locked his data bases.

                else if On_pd (Page) and Rws_in_progress(PDrec(Page))      // If this page is on the paging
                    then do;                               // device, it is possible that a read-write
                                                           // sequence may be in progress for it.
                                                           // It must be aborted.
                            call rws_abort (Page);         // Abort the rws, or possibly clean up an
                                                           // already complete abort. Unless we are
                                                           // cleaning up, we will wait for it.
```

```
                                   If Rws_in_progress(PDrec(Page))    // If we have cleaned up, we are
                                                            // finished. If we started an abort or
                                                            // noticed one in progress, we must wait.
                                     then call page_wait (Event(PDrec(Page))); //   Wait for the abort to
                                                                                // Complete.
                              end;
                         else do;                           // Normal case - we must bring Page in.
                                   call read_page (Page);   //   Start read-in of Page. If page is empty
                                                            // (all zeros), or was on a fast device, we may be done.
                                   If io_in_progress(Page)  //   If real i/o was started, and not finished,
                                      then call page_wait(Event (Page));          // we must wait for post_page
                                                            // to post Page.
                                      else unlock Page_table_lock;    // Page was all zeros or on fast device.
                                                                      // All done.
                              end;
          return;                                           //   Restore machine state at time of fault. Retry
                                                            // faulting reference.
end page_fault;
```

read_page

```
     read_page: procedure(Page);                                  //   This procedure is responsible for

                                                                  // causing a faulted page to appear in core.
                                                                  // If I/o is necessary, it is initiated.  If not,
                                                                  // a page of zeros is created.
               local Coreadd;

               If On_pd (Page)                                    //   Although this check is made in page_fault,
                  then If Rws_in_progress (PDrec (Page))          // It is conceptually important that it be made here,
                       then return;                               // for read_page may be called by other system
                                                                  // functions.  If an rws is in progress on Page,
                                                                  // we cannot read it, and our caller must either
                                                                  // give up or initiate abort proceedings.
+              Coreadd := find_core();                            //  Allocate a page of core for the page.

               Page (Coreadd) := Page;                            //   Indicate that this page belongs here.
               If Devadd (Page) = "null"                          //   A null devadd indicates that a page is defined
                  then do;                                        // to  contain zeros.
                         Phys_Coreadd(Coreadd) -> (1024 words) := 00000.... ;// Make it all zeros.
+                        Coreadd(Page) := Coreadd;                //   Indicate that this frame belongs to Page.
                         call make_accessible (Page);             //   Reset fault bit in the Descriptor, allowing
                                                                  // processors to reference Page.
                       end;
                  else do;                                        //   Page has non-null devadd, must be read in.
                         Io_in_progress (Page) := true;           //   Indicate that Page has I/o in progress. It is
                         Io_read_or_write(Coreadd) := "read";     // important that this be done before the I/o is
                                                                  // initiated, so that the I/o routine can reset these
                                                                  // flags if a fast device finishes early.
+                        call device_read (Devadd(Page), Phys_Coreadd(Coreadd));     //Start the read.
                         If On_pd (Page)                          //   Let pd management know that this page
                            then do;
                                    Incore(PDrec(Page)) := true;  // has a page in core (recently used).
                         /*         call pd_thread_to_bottom(PDrec(Page)); // This line is not in the actual code.
                                                                  // Its absence constitutes a bug which was found
                                                                  // later.  It helps maintain the LRU ordering
                                                                  // of the PD used list. */
                                 end;
                       end;
     end read_page;
```

find_core

```
find_core: function();                                      //   This function implements the Multics core
                                                            // page replacement algorithm.  It is called to
                                                            // housekeep core and return one free Coreadd Object.
          dcl CoreTop Coreadd;                              //  Its basic data base is the core used list, which
                                                            // is the ordering of Coreadd  Objects defined by the
                                                            // sequence of Next components of Coreadd Objects.
                                                            // The variable CoreTop has as a value the least
                                                            // recently used Coreadd.  Next(Coretop),
                                                            // Next(Next(CoreTop)), and so on, are Coreadd
                                                            // Objects having seen increasingly recent use. The core
                                                            // used list is circuler, so Next (most recently used
                                                            // Coreadd) = CoreTop. The algorithm is due  to
                                                            // Corbatô.

          local Frame type Coreadd;
          local Page;

          local  io_skip_counter arithmetic,
                 Loop_counter arithmetic;

          io_skip_counter := 0;                             //   Initialize check for excessive looping.

          do Loop_counter := 0                              //   We search the used list as long as necessary.
            repeat Loop_counter + 1                         // The default is to crash Multics.
            while Loop_counter < 131072 ;                   // We will always return a free page, or die.

                    do while Writes_outstanding > 30;       //   If this routine has queued many writes, see if
                         call post_any_io;                  // some have completed since we started looping.
                                                            // (Remember that interrupts are masked).
                         io_skip_counter := 0;              //  Reinitialize I/o check.  We have just done what
                    end;                                    // this check could ask us to do;

                    if Page (CoreTop) = "null"              //   We check the supposedly least recently used page to
                       then do;                             // see if it is entirely unallocated (could happen by
                                                            // use for an rws, or a page deletion). If so, we take it.
                                  Frame := CoreTop;         //   This will be the page frame.
                                  CoreTop := Next (CoreTop);//   Mark this page as the most recently used, and the
                                                            // next least recently used the least.  Notice that
                                                            // this common operation is trivial only because of the
                                                            // circularity of the core used list.
                                   return Frame;            //   Return this core frame as useable.
                            end;

                    if io_in_progress (Page (CoreTop))      //   If, in the next line, we are going to skip this
```

```
                                        // frame because of  i/o, other than an rws going on,
                                        // meter the times that we have done so.
   then io_skip_counter := io_skip_counter + 1;

if io_in_progress (Page (CoreTop))      //   See if we must skip this frame due to i/o
      or Rws_in_frame (CoreTop)         // in progress there.
   then if  io_skip_counter <100        //   If we have skipped a large number of pages due to
           then do;                     // to i/o in them, see if any i/o has since completed.
                    call post_any_io;
                    io_skip_counter := 0; // Reset this high-water-mark.
                end;                    //   Repeat the loop, trying this last i/o-skipped
                                        // page again.
           else CoreTop := Next (CoreTop);        // Skip over this core frame, consider Next to
                                                  // be LRU.
   else if Wired (Page(CoreTop))                  //  Skip over pages never claimable.
           then CoreTop := Next (CoreTop);
           else if Usage (Descriptor (Page(CoreTop)))       // See if page has been recently used.
                   then do;
                              Usage(Descriptor(Page(CoreTop))) := false;
                                                            //  If so, reinitialize check for
                                                            //next time, and skip this
                                                            //page, making
                                                            // it the most recently used.  This
                                                            // Usage bit is set true by the
                                                            // hardware when the descriptor is used.
                              CoreTop := Next (CoreTop);    //   Skip over it.
                        end;
                    else do;                      // At this stage, the page at the top of the
                                                  // core used list has not been recently used.
                                                  // It is a prime target for replacement. We will
                                                  // see if it needs to be written out.  If no
                                                  // i/o is in progress when try_to_write_page
                                                  // returns, we can claim the page.
                              Page := Page(CoreTop);        // Consider frame's current resident.
                              Frame := CoreTop;             // And consider the frame.
                              CoreTop := Next (CoreTop);    // Make the page under consideration
                                                            // the most recently used. If we ultimate-
                                                            // ly claim it, this was the right move.
                                                            // If we do not, it will be due to recent
                                                            // use or i/o, and it is still good.
                              call try_to_write_page(Page); // See if page needs writing out.
                                                            // Initiate such i/o if so.
                              If not io_in_progress (Page)  // If try_to_write_page succeeded in
                                                            // totally writing out page (fast pd),
                                                            // this holds.  Otherwise, move on.
                                 then do;                   // Try to claim page for real.
                                         call make_nonaccessible (Page);
                                                            // Turn off access to Page. We do not
                                                            // exit this call until all processors
                                                            // have verified that they have flushed
                                                            // Descriptor (Page) from
                                                            // their associative
```

```
                                          // frame because of I/o, other than an rws going on,
                                          // meter the times that we have done so.
then io_skip_counter := io_skip_counter + 1;

If io_in_progress (Page (CoreTop))        //   See if we must skip this frame due to I/o
   or hws_in_frame (CoreTop)              // in progress there.
then if  io_skip_counter <100             //   If we have skipped a large number of pages due to
                                          // to I/o in them, see if any I/o has since completed.
   then do;
      call post_any_io;
      io_skip_counter := 0; // Reset this high-water-mark.
                                          //   Repeat the loop, trying this last I/o-skipped
                                          // page again.
      end;
else CoreTop := Next (CoreTop);           // Skip over this core frame, consider Next to
                                          // be LRU.
else if Wired (Page(CoreTop))             // Skip over pages never claimable.
then CoreTop := Next (CoreTop);
else if usage (Descriptor (Page(CoreTop)))   // See if page has been recently used.
   then do;
      usage(Descriptor(Page(CoreTop))) := false;
                                          // If so, substitutes check for
[illegible]
                                          // [illegible] used. This
[illegible]
      CoreTop := [illegible]
   end
else do;
[illegible]
      Page := Page(CoreTop);
      Frame := CoreTop;
      CoreTop := Next (CoreTop);          // [illegible] resident.
[illegible]
      call try_to_write_page(Page);       // See if page needs writing out.
      if not io_in_progress (Page)        // [illegible] if so.
[illegible]
      then do; [illegible]                // [illegible] page for real.
         call make_nonaccessible(Page);   // Turn off access to Page. We do not
                                          // exit this call until all processors
                                          // have verified that they have flushed
                                          // Descriptor (Page) from
                                          // their associative
```

wan_wait:

110

try_to_write_page

```
    try_to_write_page:procedure (Page);                         //   This procedure determines if a page has been
                                                                // modified, and thus needs to be written out. It also
                                                                // checks for the case where a page should be written to
                                                                // the paging device due to recency of use.

              dcl Top_of_pd_used_list PDrec;

              If On_pd (Page)                                   //   If the page already has a copy on the paging device,
                 then If Modified (Descriptor (Page))           // the decision to write is the same as whether or not
+                        then call write_page (Page,"modified","pd_ok");  // Page has been modifeled.
                         else;                                  //   Page not modified, already on pd, need not write.
                 else                                           //   Page is not on the paging device.  If it can go there,
                                                                // we will put it there.
                        If  Gtpd (Page)                         //   If pake is forbidden to go on pd,
                                  or in_use (Top_of_pd_used_list)          //  or there is no space for it,
                                                                // we cannot put it on the pd.
+                          then If Modified(Descriptor(Page))   //    Write it back to disk if modified,
                                   then call write_page(Page,"modified","no_pd");
                                   else;                        //    If not modified, exit.
                           else                                 //  Page will go on pd, whether modified or not.
                                  If Modified(Descriptor(Page)) //   If page is modified, indicate it and write it.
+                                    then call write_page (Page, "modified", "put_on_pd");
+                                    else call write_page (Page,"not_mod", "put_on_pd");       // Write even if not modified.
              return;
    end try_to_write_page;
```

write_page

```
write_page: procedure (Page, Modflag, PDflag);
                                                       // This procedure, which is called when it is
                                                       // determined that a page must be written out,
                                                       // does so.  It is defined in Multics that
                                                       // a page of zeros is never writeen out, but specially
                                                       // flagged.  We make that check here.
        declare Modflag  literal, PDflag literal;     //   Page is the page of interest here.  Modified
                                                       // is either "modified" or "not_mod", telling us
                                                       // whether or not to  turn on Modified_from_disk
                                                       // (PDrec(Page)).  PDflag, if "put_on_pd", tells
                                                       // us to allocate a new pd record for  Page.

+       if page_is_zero (Page) then return;           //   If page contained zeros, no write need be done,
                                                       // and we return.
        Modified (Descriptor (Page)) := false;         //   We have noted modification to Page.  Any
                                                       // modification after this point (actually the next
                                                       // call will be noted by find_core once the i/o that
                                                       // we will start has finished.
        call clear_associative_memory;                 //   Let other processors who modify this page note
                                                       // that we have turned off 'modified'.
        te_in_progress(Page) := true;                  //   Assert that core frame used by Page is not claimable.
        te_read_or_write(Coreadd(Page)) := "write";    //   Let post_page know what kind of i/o took place.
        if Devadd(Page) = "null"                       //   If page has no secondary storage home, i.e., was

           then Devadd (Page) := allocate_disk_record(); // previously all zeros, give it one.
        if PDflag = "put_on_pd"                        //   If page is to be put on pd, due to recency of use,
+          then call allocate_pd(Page);                // allocate a pd record.  This will change Devadd of
                                                       // Page, and save the current Devadd in Diskaddr (PDrec
                                                       // (Page)).
        if On_pd (Page)                                //   If Page is, at this stage, on the pd,
           then do;                                    // then update the status of its pd record.
                if Modflag = "modified"                //   If Page has been modified while in core,
                   then Modified_from_disk (PDrec(Page)) := true; // we must indicate that the pd copy that we
                                                       // are about to start writing is different from the
                                                       // disk copy.
                call pd_thread_to_bottom (PDrec(Page)); //   Indicate that this pd record has seen  recent use.
              end;
+       call device_write(Devadd(Page), Phys_Coreadd(Coreadd(Page)));        //   Start the actual write.
        return;
end write_page;
```

allocate_pd

```
allocate_pd: procedure(Page);
                                   // [illegible]
                                   // [illegible]
                                   // [illegible] is the current use of the pd used
                                   // [illegible] is used.
    local Force,
    [illegible] [illegible]_list Force,
        [illegible]_records [illegible];
    Force := [illegible]_list;
                                   // [illegible] here.
    [illegible] := [illegible];
    [illegible] := [illegible];
                                   // [illegible]
    [illegible] := [illegible];
    [illegible]
    [illegible]
                                   // [illegible] is being used.
    [illegible] := [illegible];
                                   // [illegible]
    [illegible] := [illegible];
    [illegible] := [illegible];
    [illegible] := [illegible];
                                   // [illegible] will keep it up to date.
    [illegible]
end allocate_pd;
```

page_is_zero

```
page_is_zero: function (Page);                            //   This procedure is called  to determine whether
                                                          // or not  a page frame contains all zeros.  If it
                                                          // does, any disk or paging device space allocated to
                                                          // the page  is relinquished.
          If (Phys_Coreadd (Coreadd (Page))) -> (1024 words) = 000000....       //   See if page frame physically
                                                          // contains zeros.
            then do;
                      call make_nonaccessible (Page);     //   Turn off the addressability of the page.  We will
                                                          // check again if page is all zeros since the page has
                                                          // not been addressable.  We do this instead of simply
                                                          // turning off access and checking because the vast
                                                          // majority of pages checked for zero are not zero, and
                                                          // turning off the access would cause another processor
                                                          // attempting to reference this page to fault, and loop on
                                                          // the page table lock, which we have locker.

                       If (Phys_Coreadd(Coreadd(Page))) -> (1024 words) = 00000... // See if still zero.
                          then do;                        //   It is zero.  Relinquish secondary storage.
                                   Modified (Descriptor(Page)) := false;
                                                          //  Indicate that we are aware of page having been
                                                          // modified (to zeros). We need not clear the associative
                                                          // memories, because, you will recall, access is off.
                                   If On_pd(Page)         //   Relinquish the pd record, if there is one.
                                       then do;
                                            Devadd (Page) := Diskaddr (PDrec(Page)); // Page's nearest home is disk.
                                            In_use (PDrec(Page)) := false;       // The pd record may be used.
                                            On_pd (Page) := false;               // Page no longer on pd.
                                            Number_of_free_pd_records     // Increment free record count.
                                              := Number_of_free_pd_records + 1;
                                            call pd_thread_to_top (PDrec(Page));     // Suggest this
                                                                  // pd record for immediate claiming
                                                                  // by allocate_pd.
                                        end;
                                   call meter_disk (Devadd(Page), "delete");     //   Inform the disk I/O experiment
                                                                  // of a page deletion. (Actually, this
                                                                  // call is in relinquish_disk_space).
                                   call relinquish_disk_space (Devadd (Page));   //   Free the disk space allocated to Page
                                   Devadd (Page) := "null";       //   Indicate that Page must be all
                                                                  // zeros.  read_page inteprets this
                                                                  // null devadd as an indication that
                                                                  // a page of zeros is expected.
                                   If (we were called from the page fault        //   Page can be considered to have
                sequence as opposed to post-  // been written.  This non-local goto
                purge or other activity)                     // saves find_core the effort of
                                                             // turning off access, which is already
                then goto not_mod in find_core;              // off, and annoying the entire system
                                                             // with yet another global associative
                                                             // memory clear.

            return true;                                     // [illegible] that page was zero.
        end;
    else call make_accessible (Page);                        // [illegible] was modified since first
                                                             // [illegible] [illegible].
                                        // Page is not same, return this fact.
      end;
      return false;
end page_is_same;
```

get_free_pd_record

```
get_free_pd_record: procedure;                          //   This procedure is called to increase the
                                                        // number of free pd records.  It maintains the
                                                        // LRU discipline on the pd used list.  It
                                                        // returns when  it has freed one, or has started
                                                        // a large number of read-write sequences.  Since
                                                        // starting a read-write sequence indicates that one
                                                        // more record will be available,  the free count
                                                        // is incremented when this is done.  (see start_rws).

    declare Top_of_pd_used_list type PDrec;
    local Save_pd_first type PDrec;                     //  Used to control loop.
    declare Number_of_free_pd_records arithmetic;

    local Rws_ctr arithmetic, PDrec;
    local First_time boolean;

    First_time := true;                                 //  Initialize variables to check when
    Save_pd_first := Top_of_pd_used_list;               // do loop has passed over all pd records.

    Rws_ctr := 0;                                       //   Initialize the count of initiated rws's.
    do forever;                                         //   Repeat this loop as long as necessary.
       do PDrec := Top_of_pd_used_list                  //   Loop over the pd used list, starting at the
           repeat Next_pd (PDrec)                       // most claimable end.
           while (First_time or (PDrec ¬= Save_pd_first));

           First_time := false;

           If  in_use (PDrec)                           //   We can only free those records not already free.
             then if incore (PDrec)                     //   If the page on this pd record is in core,  it is
                  then call pd_thread_to_bottom (PDrec); // surely among the most recently used on the pd.
                  else if modified_from_disk(PDrec)       //  By the fact that we have gotten to this
                                                          // pd record starting at the top, it should be
                                                          // queued.
                    then do;                              //   If pd copy is different than disk copy,
                           call start_rws (PDrec);  // we must rws.
                           Rws_ctr := Rws_ctr + 1;  //   Maintain a count of the number of rws's
                                                      // that we have started.
                           If Rws_ctr > 30            //  If we have started a large number of Rws's,
                             then return;           // surely we have made enough pd records free
                                                    // that we will not be called again.
                                                    // (see page_fault).
                         end;
```

```
else do                       //   Disk copy same as ed copy.  Can oust the
                              //   page from [illegible] core/memory.
        Number_of_free_[illegible]
        [illegible]
```

post_page

```
post_page:procedure(Coreadd);                                  //  This procedure is responsible for  changing the
                                                               // state of page control data bases when the  completion
                                                               // of an i/o operation is observed. It is invoked
                                                               // from individual device control routines.

          local Page;

          if Rws_in_frame (Coreadd)                            // If there was an rws here, go process the completion
                                                               // or aborting.
             then call rws_done (Coreadd);

             else do;                                          //   This was a normal page read or write.
                    Page := Page(Coreadd);                     //   Identify the page involved in this operation.
                    Io_in_progress (Page) := false;            //   Turn of i/o flag.
                    if Io_read_or_write (Coreadd) = "read"     //   See if read or write completed.
                       then do;
                               Coreadd(Page) := Coreadd;       //   Page will be made accessible in this frame.
                               call make_accessible (Page);    //   Insert  physical core address, turn on access.
                            end;

                       else do;                                //   Handle a write completion.
                               Writes_outstanding := Writes_outstanding  - 1;   //  Maintain heuristic for find_core.
                               call thread_to_top (Coreadd);            //  Make this core frame the most likely
                                                               //  candidate for claiming.  The usual reason that a
                                                               // write was started is that it was a good candidate for
                                                               // claiming in the first place.  If Page has been used,
                                                               // (this includes modified) since the Usage bit was turned
                                                               // off, find_core will not claim this page now. Otherwise,
                                                               // it will  be the very next page claimed.
                        end;
                    call notify (Event(Page));                 // Cause any process waiting for the completion of this
                                                               // i/o operation to resume.
          return;
end post_page;
```

start_rws

```
start_rws:procedure (PDrec);                                   //  This procedure initiates the moving of a
                                                               // modified page from the paging device to the disk.
        local Coreadd;
        declare Rws_active_count arithmetic;                   //  This counter is a heuristic for limiting rws activity.

        Coreadd := find_core();                                //  Get a page of core for the rws buffer.
        Rws_in_frame (Coreadd) := true;                        //  Mark this page frame as unclaimable. Flag
                                                               // also lets post_page know what to do.
        Rws_in_progress (PDrec) := true;                       // Mark this pd record as having an rws in progress.
        io_read_or_write (Coreadd) := "read";                  // Indicate the direction of i/o for post_page.

        Coreadd (PDrec) := Coreadd;                            //  Set up this relation, so that rws_abort can
                                                               // find Coreadd.
        PDrec(Coreadd) := PDrec;                               //  Set up this relation, so that rws_done can find PDrec.
        call pd_thread_out (PDrec);                            //  Thread PDrec out of used list, so it can't be claimed.
        call device_read(Devadd(PDrec), Phys_Coreadd(Coreadd));     //  Start and possibly finish the read.
        call meter_disk (Diskaddr(PDrec), "write");            //  Meter motion to the disk.
        Rws_active_count := Rws_active_count + 1;              // Maintain Heuristic.
        Number_of_free_pd_records := Number_of_free_pd_records + 1; // Indicate that this pd record
                                                               // is in the process of being freed.
        Modified_from_disk(PDrec) := false;                    //  Indicate that this pd record will be same as disk
                                                               // copy.

        do while Rws_active_count > 30;                        //  If there is a large amount of rws activity going on,
                                                               // wait for some of it to subside.
            call post_any_io;                                  //  See what has completed.
        end;
        return;
end start_rws;
```

rws_abort

```
rws_abort: procedure(Page);                                 //   This procedure is invoked when a page fault
                                                            // is taken on a page which has an rws in progress.
          local PDrec;                                      //   There are three such cases. 1) No abort has
                                                            // been initiated. We initiate one, and wait
                                                            // for notification from rws_done. 2) Another
                                                            // process has initiated one.  We wait for it.
                                                            // 3) We have been notified by rws_done, and must
                                                            // "clean up" the abort.
          PDrec := PDrec (Page);                            //  The PD record is of intense interest here.
          If Abort_flag (PDrec)                             //  If this is so, case 2 or 3 above.
             then If Abort_Complete (PDrec)                 //  If this is so, case 3. We clean up, and the
                     then do;                               // rws and the rws abort are over.
                              Abort_flag(PDrec),            //  No more rws abort, or rws.
                              Abort_Complete(PDrec} Rws in progress(PDrec) := false;
                              Coreadd(Page) := Coreadd(PDrec);        // Use rws buffer as a home for Page.
                              call make_accessible (Page);            // Insert physical address into
                                                            // Descriptor, and turn on access.
                              call meter_disk(Diskaddr(PDrec),"read");
                                                            //  This rws abort represents negation of
                                                            // logical movement to the disk, and hence we
                                                            // report it to the experiment as a fetch.
                              call thread_to_bottom (Coreadd(Page));  //  Make core page appear recently used.
                              call pd_thread_to_bottom (PDrec);       //  Make pd page appear recently used.
                              Incore (PDrec),in_use (PDrec) := true;  //  Update status of pd record.
                              Rws_active_count := Rws_active_count -1;          //  Maintain this heuristic.
                              Rws_in_frame (Coreadd(PDrec)) := false; // Housekeep, housekeep.
                              Number_of_free_pd_records
                                 := Number_of_free_pd_records -1;     // Reverse decision made
                                                                      // by start_rws.
                         end;                               //  Return to page_fault with Rws_in_progress off.
                   else;                                    //  Case 2. Abort already started. page_fault will
                                                            // wait for it.
              else Abort_flag (PDrec) := true;              //  Case 1. Abort the rws. page_fault will wait for it.
         return;                                            //  Return to page_fault to either continue
end rws_abort;                                              // and restart the fault, or wait.
```

rws_done

```
rws_done:procedure(Coreadd);                         //   This procedure handles the completion of
                                                     // I/o done on behalf of read-write sequences.
                                                     // Aborts are noticed here, as well.

      declare Rws_active_count arithmetic, Writes_outstanding arithmetic;
      local PDrec;
      PDrec := PDrec(Coreadd);                       //  Identify the concerned pd record from the
                                                     // field specially reserved for this purpose.
      If Abort_flag (PDrec)                          // If an abort was requested
       then do;                                      // abort the rws.
        If Io_read_or_write (Coreadd) = "read"       //  If a read was aborted, we will not write,
          then Modified_from_disk (PDrec) := true;   // and must re-indicate that record differs
                                                     // from disk.
          else Writes_outstanding := Writes_outstanding - 1;  //  Otherwise, maintain write count.

        Abort_Complete (PDrec) := true;              //  Indicate that we have aborted the rws. We
                                                     // cannot make the page addressable because
                                                     // (point of fact!) we have no way to locate
                                                     // Page(PDrec) in the actual implementation.
                                                     // This is due to not having enough space to
                                                     // save the required pointers.
       call notify (Event (PDrec));                  //  We now cause all processes who faulted on
                                                     // this page during the rws to resume. They
                                                     // will all re-take the page faults which made them
                                                     // first see this rws, and the first one to lock
                                                     // the page table lock will make the page
                                                     // addressable. The others will find that
                                                     // there is no more page fault (see page_fault)
                                                     // and simply return.
      end;
     else do;                                        //  A normal rws read or write completion.
            If Io_read_or_write(Coreadd) = "read"    //  If a read finished, Start the write.
              then do;
                   Io_read_or_write(Coreadd) := "write";  // Indicate I/o direction for next time.
                   call device_write(Diskaddr(PDrec),Phys_Coreadd(Coreadd));  // Start the write.
                 end;
              else do;                               //  The write, and hence the rws, has finished, successfully
                                                     // (i.e., without an abort.)
                   Rws_active_count := Rws_active_count - 1;       // Maintain the rws activity heuristic.
                   Writes_outstanding := Writes_outstanding - 1;   // Maintain find_core's heuristic.
                   Rws_in_frame (Coreadd) := false;                // Turn off rws indicator.
                   Rws_in_progress (PDrec) := false;               // Turn off rws indicator for pd record.
                   In_use (PDrec) := false;                        // This record is now free.
```

```
                               call pd_thread_to_top (PDrec);                   // This record is now claimable.
                               Devadd (Page(PDrec)) := Diskaddr(PDrec);         // The page that was on this pd record
                                                                                // is now on disk.
                               On_pd(Page(PDrec)) := false;           // Let all know.
                               Page (Coreadd) := "null";              // The core block used as an rws buffer is now
                                                                      // immediately claimable.
                               call thread_to_top (Coreadd);          // Make it best candidate for claiming.
                            end;
                   end;
          return;
end rws_done;
```

auxiliaries

```
                /* Although some of these short routines might better be expressed in
                   line, they are conceptually modules in their own right, and
                   may be called from other points in the system.*/

          /* Small auxiliary routines */

device_read:procedure (Devadd, Phys_Coreadd);                // Called to initiate a read -
                                                             // selects correct i/o routine.

          declare Phys_Coreadd arithmetic;

          if Device (Devadd) ¬= "drum"                       // meter disk reads
             then call meter_disk (Devadd, "read");          // meter it
          call (select_io_routine_entry(Device(Devadd), "read"))
                    (Phys_Devadd (Devadd), Phys_Coreadd);    // call right routine.
end device_read;

device_write: procedure (Devadd, Phys_Coreadd);              // Called to write a page.

          Writes_outstanding := Writes_outstanding +1;
          call (select_io_routine_entry (Device(Devadd), "write"))
                    (Phys_Devadd (Devadd), Phys_Coreadd);
end device_write;

make_accessible: procedure(Page);                            // Called to make a page accessible.

          Phys_Coreadd (Descriptor(Page)) := Phys_Coreadd(Coreadd(Page));       // Fill in physical address.
          Addressable (Descriptor (Page)) := true;                              // make page addressable.
end make_accessible;

make_nonaccessible:procedure(Page);                          // Called to make a page non-accessible.
          Addressable (Descriptor (Page)) := false;          // make page non-addressable.
          call clear_associative_memory;                     //   flush descriptor from associative memories.
end make_nonaccessible;

meter_disk:procedure (Devadd,Type);
                                                             // Principal procedure of LRU metering experiment.
          declare Type literal, Experiment_active boolean;

          if not Experiment_active then return;              // cannot accumulate data if buffer not wired.
          Trace_datum:=construct Trace_datum(Devadd:Devadd, Type:Type);
          enqueue (Trace_datum,Trace_queue);
end meter_disk;

post_any_io:procedure;                            // This routine is called in any situation where page
                                                  // control discovers some I/o bottleneck.
                                                  // It polls I/o routines for complete status. They will
                                                  // call post_page if any status arrives.
          declare io_devices set literal;

          local Device literal;

          do Device := range io_devices;                    // loop over all io devices.
               call (select_io_routine_entry (Device,"post"))
                              ();                 // make an appropriate call.
          end;
end post_any_io;
```

fixed_head_control

/* A Typical Paging I/O Control Routine */

```
/* This routine is the i/o control routine for the fixed-head disk. There exist routines
   almost identical to it for the moving-head disk and drum. The routine select_io_routine_entry
   (not given here) is used to select the appropriate entry of   the appropriate routine
   given the device identifier and the function to be performed. */


declare Phys_Devadd arithmetic, Phys_Coreadd arithmetic;

fixed_head_read:procedure (Phys_Devadd, Phys_Coreadd);               // Read entry.
          call fixed_head_start (Phys_Devadd,Phys_Coreadd,"read");    // Call common queueing routine.
   end;


fixed_head_write:procedure (Phys_Devadd, Phys_Coreadd);              // Write entry.
          call fixed_head_start (Phys_Devadd, Phys_Coreadd, "write"); // Call common queueing routine.
   end;

fixed_head_start:procedure (Phys_Devadd,Phys_Coreadd,Direction);     // Common routine to queue fixed-head
                                                                     // disk requests.
          declare Direction literal;
          call fixed_head_post;                                      // See if any operations
                                                                     // have completed.
          enqueue (construct io_program
                              (Phys_Devadd:Phys_Devadd,              // Construct a channel program
                               Phys_Coreadd:Phys_Coreadd,            // and enqueue it.
                               Direction:Direction,
                               Next:"null"),Fixed_Head_Channel_Queue);
          if (fixed head disk is not busy) then call start_io(Fixed_Head_Channel_Queue);
                                                                     // There is now work for the
                                                                     // fixed head disk. Start it if
                                                                     // it is idle.
 end;


fixed_head_post:procedure;                                  // Used to post completed operations.
          do io_status := range (any complete i/o status);  //Look at all new status.
               remove io_status from (set of  complete status);      // Take it out of hardware queue.
               call post_page (Coreadd (io_status));                 // Inform page control.  See the
                                                            // declaration of io_status.
               remove io_program (io_status) from Fixed_Head_Channel_Queue;
          end;
  end;
```

Appendix B

Implementation of the Hardcore Meters

In this appendix, we relate the exact identities of the measured events of Multics page control with which this experiment was concerned. This is necessary both to provide validity for what we have done, and to help others design similar techniques for other systems. It is assumed that the first appendix has been at least partially understood, perhaps with the overview sections fully understood.

We also discuss here the techniques used in implementing the Multics Supervisor interface for this experiment.

As should be clear from Chapter 2, we are interested in metering movement of pages in and out of the composite entity of core-drum. This "movement" in fact consists of copy creation and copy destruction. Movement "into" core drum consists of the creation of a page copy in core-drum where there previously was none, and movement "out of" core drum consists of the destruction of core or drum copies of a page, such that there is no copy in core-drum. We speak of this creation and deletion as movement because it is represented as movement of pages in an LRU stack.

We will now analyze the different types of motion in and out of core-drum. Pages come into core-drum either from the outside, i.e., disk, or by being created in core. Pages entering from disk can only do so as the result of a page fault to disk or a pre-paging from disk, so a call to "meter_disk" (see Appendix A) was installed in the i/o dispatching routine to record all reads from disk. Pages created in core never involve input/output. For the most part, these are pages which were never touched before, and would thus cause a page fault no matter how large core-drum were. These page faults, however, involve neither multiprogramming, i/o, nor idle time, and are thus of slightly less interest in performance predictions than more general page faults. We chose to ignore them. There is one other type of inward motion, which will be motivated in our discussion of outward motion.

Outward motion consists of oustings from core-drum. This consists of oustings from drum (which, as can be verified from "get_free_pd_record" in the last appendix can only happen if there is no copy in core) or from core. Oustings from core are only oustings from core-drum if page control (specifically, "try_to_write_page") decides that it should not be written to the drum because of either lack of space there or the concerned page is one of the special-cased "gtpd" pages. We will first consider the oustings from drum. The ousting of a page which is different than its disk copy, if one was ever made, is accomplished by the initiation of a read-write sequence (rws). These rws initiations were thus metered as outward movement. The ousting of a page which is identical to a disk copy is done by simply claiming the drum frame (see "get_free_pd_record"), and this event was likewise noted. Oustings from core interest us whenever they are not oustings to the drum. (We define an ousting "from core to the drum" to be an ousting from core when a copy of the concerned page is on drum. Note that this implies an ordering of the hierarchical memory system.) These oustings from core normally happen only for the special "Global transparent paging device (gtpd)" pages of the root directory, whose treatment was already fully covered, and in bad cases of page faults or rws initiations, when there are no free drum frames available.

This case was also covered. As an interesting consequence of this definition of an ousting not to the drum, we observed a large number (approximately 2400 per hour) of oustings of pages which were all zeros, and were all zeros when brought into core (the conjunction of these statements essentially implies that these pages had no copies on either drum or disk).* Some special experiments designed to discover the source of this peculiar traffic were essentially fruitless. The data reduction programs described in section 2.3 were modified to ignore these anomalous oustings.

One consequence of metering read-write sequence initiation as outward motion is that the aborting, or reversal due to a page fault, of a read-write sequence must be metered as inward motion. This was done (see "rws_abort").

One remaining event which had to be metered was that of page destruction. The event we chose to represent this destruction was the handing back of the disk frame, if one existed, to the free disk pool, of any disk frame at all. This happens in two cases. First, explicit page destruction via the deletion of segments of the virtual memory requested by supervisor call, or their explicitly requested truncation causes this to happen. Secondly, as we have described, find_core deallocates both disk and drum frames when a page containing all zeros (a void page) is found with its used bit off. As described in section 2.2, we are interested only in the destruction of pages which are not in core-drum. The destruction of any such non-void page will always involve the deallocation of a disk frame, and thus will be properly metered. The destruction of void pages is not

*Even though this constituted about one quarter of all core-drum oustings, they bear absolutely no significance to the experiment.

a significant event, as they do not occupy a place in either the LRU or LRU extension stacks of section 2.1. The discovery of a newly void page by find_core also causes such an event to be recorded in the trace data as a page deletion. However, this page cannot be in the extension stack, because it was found by find_core because it was, in fact, in core. The data reduction programs were aware of these out-of-list deletions, and duly ignored them. The destruction of pages in core-drum which were never ousted is handled and ignored by this same mechanism.

Interface Details

The Multics hardcore interface for this experiment was designed to be a semi-permanent part of the Multics system, and thus have as little effect as possible on it when not it use. Thus, a page of the virtual memory was allocated for the circular buffer described in section 2.3 and its auxiliary data. When the experiment was enabled, via highly privileged supervisor primitive, this page was given a dedicated page frame and withdrawn from the pool of pageable core. This was necessary to insure that page control, when storing data in this buffer, would not take a page fault. Page control was also made to check a switch (the "enabled/disabled" switch) as to whether or not this had been done before attempting to reference the buffer. Another highly privileged supervisor primitive freed the page frame given to this buffer, resetting this switch before doing so.

The copying of data out of this buffer, via privileged supervisor entry point, ostensibly requires simply copying its contents into a user-specified area. However, it was an aim of the interface design to insure that the buffer would not change while the information was being copied. This could happen by either the processor not doing the copying taking a page fault, or the processor doing the copying taking a page fault referencing the user's area. Hence, to insure that no page control activity took place while this data was being copied, the data-gathering primitive had to lock the "page table lock" while doing this copying. This, in essence, prevents page faults from being processed, and cannot be done until any other process has unlocked this lock. The effect of this lock is to insure that only one processor is in page control at a time. When one has the page table lock locked, one must not take a page fault, or infinite looping will result when that processor tries to lock the page table lock to process it. What is more, the page fault handler is not recursive. Thus, it was necessary to "wire" a dedicated page of the virtual memory (allocate a dedicated core page frame and withdraw the latter from the pool of pageable core) to copy the wired buffer into. Both pages being wired (the buffer and the temporary copy page) ensured that no page fault would take place during the copy. The contents of the copy page could then be copied to the user-specified area after the page table lock had been unlocked.

A further difficulty arose because the segment containing the temporary copy page is a one-per-system segment, and thus could not be used by two processes simultaneously. Thus, a lock had to be used to exclude such use of this segment. This lock would be locked by any process wanting to gather data before it wired the temporary copy page, and unlocked after it had been unwired. A process or finding the lock locked would be multiprogrammed, and the associated process notified when the lock was unlocked.

The code which copies the wired buffer into the temporarily wired temporary copy page is entered only when the latter has been wired. However, it is possible that the former may not be wired, specifically, if the experiment has not been enabled. If this is the case, a fatal page fault with the page table lock locked would result. To avoid this, the enabled/disabled switch must be checked by this code, but it cannot check this switch until the page table lock is actually locked. Only when it is locked can no page possibly be made unreferenceable, as no other process can be in page control. The enabled/disabled switch is turned to disabled

BEFORE the buffer is unwired, and the buffer can only be made unreferenceable AFTER the page table lock is locked AFTER it has been wired. Thus, the sequence of events in a call to gather is as follows:

1. Attempt to lock the copy page lock; multiprogram and retry if failure.
2. Wire the temporary copy page.
3. Attempt to lock the page table lock; loop until successful.
4. Inspect the enabled/disabled switch; copy the wired buffer into the copy page if enabled, else copy zeros.
5. Unlock the page table lock.
6. Copy the temporary copy page out to the user-specified area.
7. Zero the temporary copy page, and unwire it.
8. Unlock the copy page lock; notify any waiting processes.

The step of zeroing the copy page is done so that this page will be immediately claimable to find_core. This is done as both a friendly gesture and an attempt to keep this page off of the drum and out of the disk traffic visible to the experiment. The page frame is always void when unwired (returned to the pool of pageable core).

The sequence for enabling the experiment is as follows:

1. Wire the buffer page.
2. Set the enabled/disabled switch to enabled.

The sequence for disabling the experiment is as follows:

1. Set the enabled/disabled switch to disabled.
2. Unwire the buffer page.

The only remaining question of locking is that of the buffer becoming unwired as page control is placing data in it. This cannot happen. Any page control operation sequence other than those just described can be summarized as:

1. Attempt to lock the page table lock; loop until successful.
2. Do all nature of page control.
3. Conditionally, unlock the page table lock and end this sequence.
4. Check the enabled/disabled switch; add data to the buffer if and only if it is enabled.
5. Go back to step 2.

The making unreferenceable of pages by find_core falls under step 2 above. During the checking of the switch and the placing of data in the buffer, this making unreferenceable cannot happen on this processor. The page table lock excludes any other processor, and there is no problem.

Appendix C

System Performance Graphs during Experiments

We present here graphs of user load, cpu utilization, and paging overhead as functions of time of day on the days of the two experiments, "dtm 21" and "dtm 23". This data was condensed from a graphical presentation of these parameters routinely prepared by the M.I.T. Information Processing Center. It is given here to provide a feeling for the relative user load during the experiment, and to allow a rough approximation to toal system headway during the experiment to be computed. This may be computed by multiplying the time of the experiment (roughly 14 hours, or 50,000 seconds) by the fraction of the system which was not idle time or paging overhead (quite roughly, 40%), obtaining 20,000 seconds, and multiplying by the system memory reference rate (400,000 references per second), obtaining 8×10^9 virtual memory references.

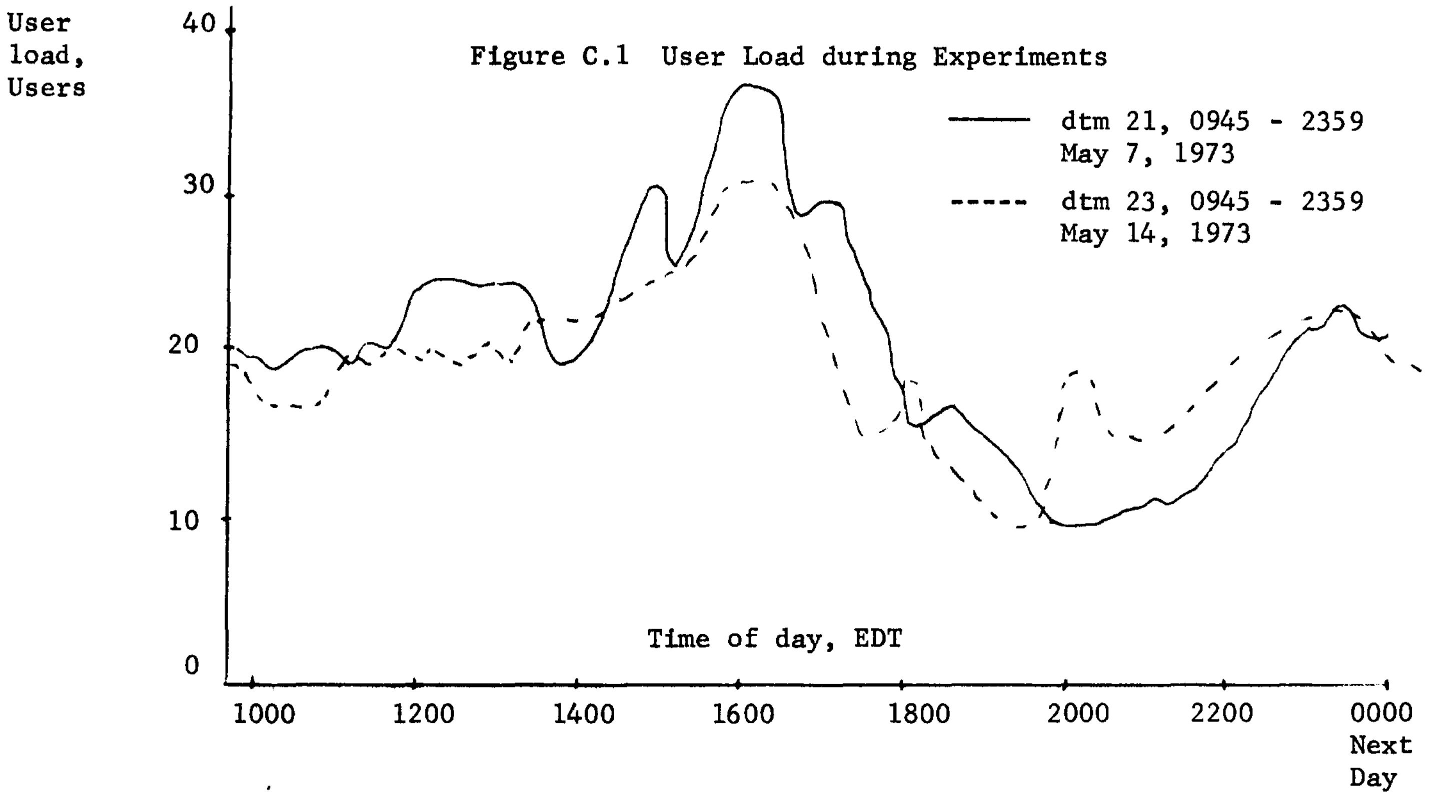


Figure C.1 User Load during Experiments

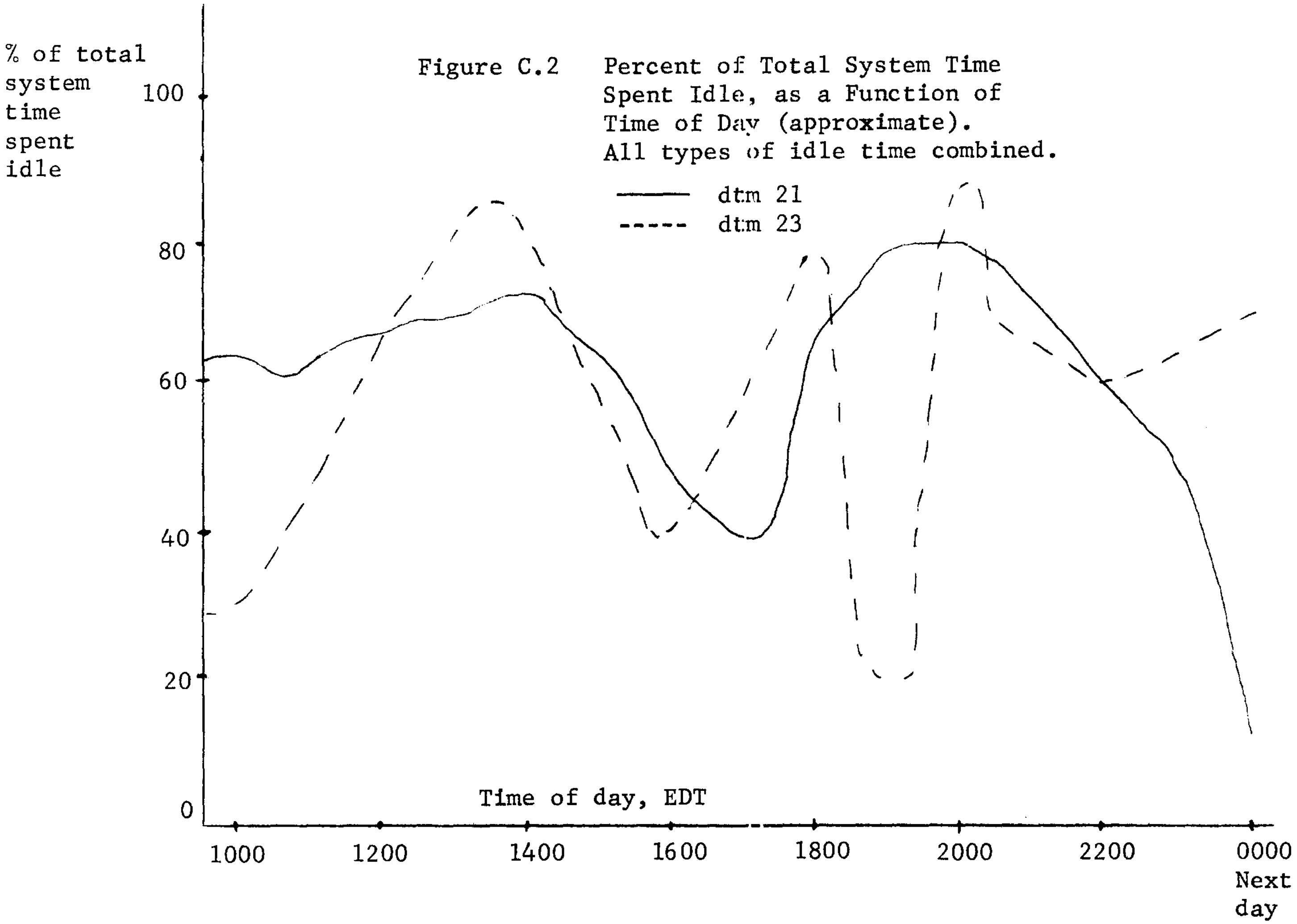
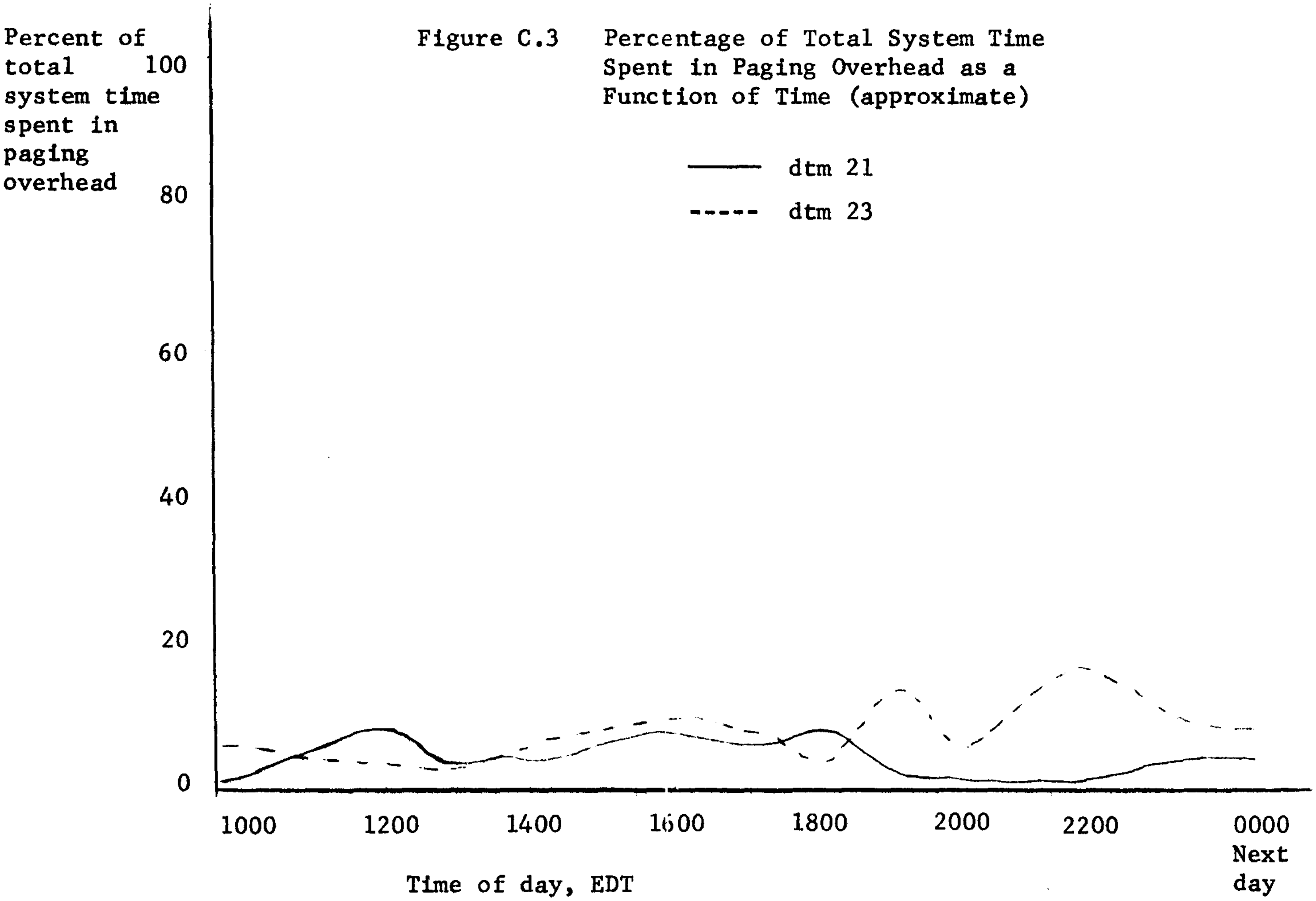


Figure C.2 Percent of Total System Time Spent Idle, as a Function of Time of Dav (approximate). All types of idle time combined.

Figure C.3 Percentage of Total System Time Spent in Paging Overhead as a Function of Time (approximate)



Bibliography

B1 Belady, L.A., "A Study of Replacement Algorithms for a Virtual-Storage Computer", IBM Systems Journal 5, 2 (1966), pp. 78-101.

B2 Bensoussan, A., Clingen, C.T., and Daley, R.C., "The Multics Virtual Memory: Concepts and Design", Communications of the ACM 15, 5 (May, 1972), pp. 308-318.

B3 Belady, L.A., and Kuehner, C.J., "Dynamic Space-Sharing in Computer Systems", Communications of the ACM 12, 5 (May, 1969), pp. 282-288.

B4 Brawn, Barbara S., and Gustavson, Frances G., "Program Behavior in a Paging Environment", AFIPS Conference Proceedings 33, 2 (1968 FJCC), AFIPS Press, Montvale, N.J., pp. 1019-1032.

C1 Coffman, E.G., and Jones, N.D., "Priority Paging Algorithms and the Extension Problem:, Proc. Switching and Automata Theory Symposium, Oct. 1971. IEEE Computer Society, Northridge, Cal., pp. 177-180.

C2 Coffman, E.G., and Randell, B., "Performance Predictions for Extended Paged Memories", Acta Informatica 1, (1971), pp. 1-13.

C3 Chow, C.K., "On Optimization of Memory Hierarchies", IBM Research Report RC 4015, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., Sept. 1972.

C4 Corbató, F.J., "A Paging Experiment with the Multics System" in Ingard, In Honor of P.M. Morse, M.I.T. Press, Cambridge, Mass., (1969), pp. 217-228.

D1 Denning, Peter J., "The Working Set Model for Program Behavior", Communications of the ACM 11, 5 (May 1968), pp. 323-333.

F1 Fine, Gerald H., et al., "Dynamic Program Behavior under Paging", ACM Proceedings of the 21st National Conference, P-66, Thompson Books, Washington, D.C., (1966), pp. 223-228.

M1 Mattson, R.L., et al., "Evaluation Techniques for Storage Hierarchies", IBM Systems Journal 9, 2 (1970), pp. 78-117.

M2 Madnick, S.E., "Storage Hierarchy Systems", Ph.D. Thesis, M.I.T. Dept. of Electrical Engineering, April, 1973.

M3 McCarthy, John, et al., Lisp 1.5 Programmer's Manual, M.I.T. Press, Cambridge, Mass., 1965.

R1 Ramamoorthy, C.V., and Chandy, K.M., "Optimization of Memory Hierarchies in Multiprogrammed Systems", Journal of the ACM 17, 3 (July 1970), pp. 426-445.

S1 Saltzer, Jerome H., "A Simple Linear Model of Demand Paging Performance", Accepted for publication in the Communications of the ACM.

S2 Sekino, [illegible] Computer Systems", [illegible] Engineering, [illegible]

S3 Schroeder, [illegible] Monitor to Evaluate the Performance of [illegible] period of [illegible]

S4 [illegible]

CS-TR Scanning Project
Document Control Form

Date : 3 / 7 / 96

Report # LCS-TR-127

Each of the following should be identified by a checkmark:
Originating Department:

□ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) □ Technical Memo (TM)
□ Other:

Document Information

Number of pages: 140 (146-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
□ Single-sided or
☒ Double-sided

Intended to be printed as :
□ Single-sided or
☒ Double-sided

Print type:
□ Typewriter □ Offset Press □ Laser Print
□ InkJet Printer ☒ Unknown □ Other:

Check each if included with document:

☒ DOD Form □ Funding Agent Form ☒ Cover Page
□ Spine □ Printers Notes □ Photo negatives
□ Other:

Page Data:

Blank Pages(by page number):

Photographs/Tonal Material (by page number):

Other (note description/page number):

Description : Page Number:

IMAGE MAP: (1-140) UN#'ED TITLE PAGE, 2-140
(141-146) SCANCONTROL, COVER, DOD, TRGT'S (3)

Scanning Agent Signoff:

Date Received: 3 / 7 / 96 Date Scanned: 4 / 22 / 96 Date Returned: 4 / 22 / 96

Scanning Agent Signature: Michael W. Cook

BIBLIOGRAPHIC DATA SHEET	1. Report No. MAC TR- 127	2.	3. Recipient's Accession No.

4. Title and Subtitle

An Experimental Analysis of Program Reference Patterns in the Multics Virtual Memory

5. Report Date: Issued May 1974

6.

7. Author(s)

Bernard S. Greenberg

8. Performing Organization Rept. No. MAC TR- 127

9. Performing Organization Name and Address

PROJECT MAC; MASSACHUSETTS INSTITUTE OF TECHNOLOGY:
545 Technology Square, Cambridge, Massachusetts 02139

10. Project/Task/Work Unit No.

11. Contract/Grant No.

N00014-70-A-0362-0006

12. Sponsoring Organization Name and Address

Office of Naval Research
Department of the Navy
Information Systems Program
Arlington, Va 22217

13. Type of Report & Period Covered: Interim Scientific Report

14.

15. Supplementary Notes

S.M. Thesis, M.I.T., Department of Electrical Engineering, January 31, 1974

16. Abstracts : This thesis reports the design, conducting, and results of an experiment intended to measure the paginf rate of a virtual memory computer system as a function of paging memory size. This experiment, conducted on the Multics computer system at M.I.T., a large interactive computer utility serving an academic community, sought to predict paging rates for paging memory sizes larger than the existent memory at the time. A trace of all secondary memory references for two days was accumulated, and simulation techniques applicable to "stack"type paging algorithms (of which the least-recently-used discipline used by Multics is one) were applied to it.

A technique for interfacing such an experiment to an operative computer utility in such a way that adequate data can be gathered reliably and without degrading system performance is described. Issues of dynamic page deletion and creation are dealt with, apparently for the first reported time. The successful performance of this experiment asserts the viability of performing this type of measurement on this type of system. The results of the experiment are given, which suggest models of demand paging behavior.

17. Key Words and Document Analysis. 17a. Descriptors

Virtual memory

Demand paging

Computer System Performance evaluation

Computer System Performance prediction

Stack algorithms

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement Unlimited Distribution Write Project MAC Publications	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 141
	20. Security Class (This Page UNCLASSIFIED	22. Price

THIS FORM MAY BE REPRODUCED