VIRTUAL STORAGE MANAGEMENT IN THE ABSENCE OF REFERENCE BITS

by

Özalp Babaoğlu

Memorandum No. UCB/ERL M81/92

17 November 1981

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Virtual Storage Management in the Absence of Reference Bits

Virtual Storage Management in the Absence of Reference Bits

Özalp Babaoğlu

Ph.D.
in Computer Science

Electrical Engineering
and Computer Sciences

[illegible]
Domenico Ferrari
Chairman of Committee

ABSTRACT

Virtual storage management requires a policy to replace data in primary storage with incoming data from secondary storage. To be effective, such a policy must select for replacement data that is not going to be needed in the near future. Having no knowledge about future demands for data, the replacement policy must anticipate them based on past demands.

In most implementations of virtual storage systems, past demand for data is recorded in a *reference bit* associated with that data. These bits can then be examined and/or altered by the replacement policy. This thesis extends the virtual storage concept to within each storage hierarchy level. The analysis of such hierarchical replacement policies confirm their suitability for managing storage hierarchies that lack reference bits.

Our preliminary studies are concerned with results that can be used in the evaluation of virtual storage systems in general. This includes the development of a program that is capable of synthesizing certain referencing behaviors in a virtual storage.

Then, a class of *hybrid* replacement policies that employ different algorithms for the management of data in two logical partitions of primary storage is introduced and analyzed. It is shown that under certain conditions these hybrid policies incur little additional cost and perform as if reference bits were available. Trace-driven simulations are conducted to validate the findings of the analytic studies. These indicate that the conditions under which the hybrid policies exhibit good performance are rarely satisfied in an actual system.

As alternatives, the Clock and Sampled Working Set replacement policies are developed for this environment and shown to perform more robustly with respect to most variations encountered in a typical system.

Based on this work the global Clock algorithm is adopted as the page replacement policy in a virtual storage extension made to the UNIX operating system. The system runs on the VAX-11/780 computer, which lacks reference bits.

Formal models based on inventory control theory are finally developed to optimize certain policy parameters adopted in the implementation.

TABLE OF CONTENTS

ACKNOWLEGMENTS

I would like to express my gratitude to Professor Domenico Ferrari for being my advisor, teacher and, most important of all, friend during the last four years. Through his encouragement and guidance has this work come to being.

I am also indebted to the following persons and organizations:

Professor Alan J. Smith for the careful reading of this thesis and invaluable suggestions throughout.

Professor Sheldon M. Ross for inspiring me through his deep knowledge of applied probability models and being a member of my thesis committee.

The Italian National Research Council for supporting me through a fellowship during my stay in Italy in 1980. Professors Enrico Magenes and Giuseppe Serazzi of the Numerical Analysis Institute at Pavia for being gracious hosts in an intellectually and culturally stimulating environment.

The National Science Foundation for support as research assistant under Grants No. MCS 7824618 and MCS 7807291.

My colleagues Edwin Lau, Makoto Kobayashi, Jehan-Francois Paris, Juan Porcar, Newton Faller, Luis Cabrera and Frank Olken of the PROGRES (PROGram REStructuring) group at Berkeley for their friendship that went much beyond academics. I am especially indebted to my friend Juan Porcar for acting as the "Berkeley Connection" during my several unforeseen periods of absence within the course of preparing this thesis.

Bill Joy for teaching me the internals of UNIX and playing an important role in realizing the material of Chapter 6.

Finally, my parents and brother for their love and support since the beginning.

CHAPTER 1

VIRTUAL STORES

1.1. Introduction

Since its introduction with the Atlas computer [Fot61a, Kil62a], the virtual memory concept has been extensively studied and implemented in various forms [Den70a]. The primary motivation in the design of these systems is to achieve a cost-speed tradeoff across a hierarchy of storage levels. Here, we constrain our studies to the classical two-level hierarchy where the first level is fast-but-expensive *main memory* (also called *primary*, *physical* or *real memory*), and the second level is a slow-but-cheap *secondary memory* (also called *backing store*). Informally, a virtual memory system tries to create a memory hierarchy (the *virtual memory*) that has the speed characteristics of the first level but the cost and size characteristics of the second level and is transparent to user programs. Thus, from the user's point of view, the system appears to support a one-level address space. This greatly simplifies the programming task as then users can write programs as if they were to be run in a very large one-level memory.

1.2. Virtual Memory Management

The dynamic relocation capabilities of the host system determine the form of the virtual memory mechanism that can be implemented on it. These capabilities allow the *logical* addresses that are generated by the program to be mapped into *physical* addresses at the time of execution. In a purely *segmented* virtual memory, the logical address space of the program

is divided into smaller logical segments of arbitrary size [Den70a]. A segment is any logical unit of the program with an associated name. In this scheme, an item is referenced by naming the segment containing it along with its location within the segment. The mapping of logical addresses to physical addresses is done through the segment table for the program. Segmentation is a natural choice for managing logical address spaces but results in inefficient use of main memory due to fragmentation. Alternatively, the entire logical address space of the program can be divided into equal-sized blocks called *pages*. Similarly, the main memory consists of *page frames* that are the same size as pages. The resulting virtual memory technique, called *paging*, maps the logical address space of a program to the main memory through the page table. Each logical address implicitly names a page and an offset within the page. Paging, then, solves the fragmentation problem of segmentation (except in the very last page of the program) but is unsuitable for managing the logical address space since it treats the entire program as one segment. The two techniques can be combined in what is called *paged segmentation* to employ paging within the several segments of the program.

The virtual memory system we will deal with is a paging system. Since the logical address space of a program can be much larger than the size of main memory, there must be provisions for executing programs that are partially loaded. A reference by the program to a page not currently in main memory causes a *page fault*. These events initiate the transfer of information between the levels of the memory hierarchy.

As with all finite-capacity, shared-resource management problems, the implementation of paging involves various policies. Specifically, the policies relevant to the management of the main memory resource in a paging environment are:

(i) The *fetch policy*: Determines when and how much information to transfer from secondary storage to main memory.

(ii) The *replacement policy*: Selects the page(s) to be removed from main memory so that their page frame(s) can be used to hold incoming information.

(iii) The *placement policy*: Determines where in main memory to place the incoming information. In the paging systems we will consider, there are usually no alternatives for this decision since the page frame for the incoming page is that just selected by the replacement policy.

The class of paging schemes we consider employ *demand fetch* policies. In a demand fetch policy, the page containing the requested information is brought into main memory at the time of the page fault and not earlier. The alternative of *prepaging* [Smi78a, Lau79a], whereby information is transferred to main memory before it is needed, will be discussed in Chapter 6.

For the effective operation of a computer system with virtual memory, it is desirable to minimize the rate at which programs reference missing information since the speed ratio of the memory levels is large. Consequently, the performance measure we will use to judge the effectiveness of various algorithms is based on the page fault rate. The page fault rate observed in a system is heavily dependent on the nature of the programs that are executing and, to a lesser extent, on the particular page replacement policy employed [Bel66a, Fra74a, Smi76a]. We comment on each of these factors below.

1.2.1. Program Behavior

If virtual memory systems ever come close to achieving their goal of having a two-level hierarchy exhibit the speed of the faster level, it is only because programs do not generate page reference strings that are random. The success of virtual memories relies entirely on the property of reference strings that is known as *locality of reference* [Spi72a, Den72a]. Informally, this property states that the pages referenced by a program in a short interval of time constitute a small subset of its pages, called *spatial locality*, and that this set of pages varies slowly in time; the latter aspect of the property is called *temporal locality*.

It is certainly possible to improve the performance of a virtual memory system by altering the programs that run on it so that they generate well-behaved page reference strings. This technique is called *program restructuring* and has been studied extensively [Hat71a, Fer74a, Fer76a, Lau79a]. Our work will not extend to this method of performance enhancement -- we will assume that the programs to be executed on the system are unalterable.

1.2.2. Page Replacement Policies

The world of replacement policies or algorithms (we use the two terms interchangeably) can be partitioned into two classes: unrealizable and realizable algorithms.

1.2.2.1. Unrealizable Algorithms

Unrealizable algorithms, also called look-ahead algorithms, are those algorithms that require knowledge of future elements of the reference string and cannot therefore be implemented in real time. The MIN algorithm [Bel66a] for fixed partitions and its variable partition counterpart, the VMIN algorithm [Pri76a] are examples of unrealizable algorithms. Upon a page fault, the MIN replacement algorithm removes from main memory the page that will not be referenced for the longest period of time into the future; the VMIN algorithm replaces a page that will not be referenced sooner than τ virtual time units, where τ is the parameter of the algorithm. The Generalized Optimum policy (GOPT) [Den78a] extends the VMIN algorithm to variable size segments rather than fixed size pages, whereas the DMIN algorithm [Bud81a] minimizes the product of the memory space occupied by the program and the *real* time delays encountered due to faults. Although unrealizable in practice, these algorithms define the theoretical minimum number of page faults necessary for a given reference string in a fixed and variable memory partition, respectively. Thus, they serve as useful benchmarks or lower bounds.

1.2.2.2. Realizable Algorithms

The Least Recently Used (LRU) [Mat70a] and the Working Set (WS) [Den68a] algorithms are realizable counterparts of MIN and VMIN respectively. These algorithms can be implemented in real time since LRU operates by replacing the page that has not been referenced for the longest period of time in the past, while WS retains in memory exactly those pages that have been referenced in the preceding τ time units.

Other realizable algorithms include First-In-First-Out (FIFO) which removes from main memory the oldest page, and Random (RAND) which removes a page selected at random over all the pages in main memory [Cof73a, Kin71a]. Note that while MIN, LRU, FIFO and RAND operate in a fixed partition of memory, VMIN and WS result in a variable partition whose mean size depends on the setting of the algorithm parameter τ.

Although realizable, the pure LRU and WS algorithms are rarely implemented in practice due to their high implementation cost in hardware and/or software. The LRU algorithm requires an ordering to be maintained of all the pages according to their recency of usage. Since this list must be updated at each memory reference, it must be implemented in hardware in order to keep overhead at acceptable levels. Oliver [Oli74a] describes such an implementation on the CDC Star computer system. The WS algorithm, on the other hand, requires recording the time of the last reference to each page, and updating the working set after each reference. Again, any efficient implementation must rely on a great deal of hardware support.

The majority of the practical implementations that constitute the class of realizable algorithms are approximations of pure LRU or of pure WS. Their exact form is often dictated by the type of support provided in the hardware. Examples of these algorithms include the CLOCK [Cor68a, Eas79a], the Sampled Working Set (SWS) [Fog74a, Pri74a] and the Page Fault Frequency (PFF) algorithms [Chu76a]. The hardware support that all of these algorithms base their decisions on is a reference bit associated with each page frame. A reference to a page results in the corresponding reference bit being turned on. This bit, in turn, can be examined and reset by the replacement algorithm.

In this dissertation, we extend the virtual memory concept to within each memory hierarchy level. Each level is thought of as containing a two-level hierarchy within it. This hierarchy is not a physical one as in the primary-secondary memory case, but rather an artificial one arising from the employment of two different replacement algorithms. Given two replacement algorithms, one of which has good performance but high implementation cost and the other poor performance but low implementation cost, we propose schemes that result in an overall algorithm having the performance characteristics of the first and the cost characteristics of the second. The utility of these hierarchical paging strategies in a hierarchical storage system lacking page reference bits is obvious.

1.3. Evaluation Techniques

In our studies, performance evaluation of a virtual memory system consists of obtaining expressions or numerical values for the page fault rate generated by the execution of a program as a function of the amount of main memory allocated to it. To accomplish this, we resort to analytic methods based on stochastic models of various system components (including programs) and trace-driven simulations. To keep the complexity of the studies within reasonable limits, we study the execution of each program in isolation (i.e., *uniprogramming* environment) although we comment on the implications of their interactions in a multiprogramming system.

1.3.1. Stochastic Models

Digital computer systems along with the programs they execute are finite state machines. As such, their operation is entirely deterministic (given deterministic programs). By choosing to observe them at suitable fixed-length time intervals, we can also describe the operation of the common input/output devices as transitions between a finite number of states. The state space of the composite system (cpu, programs and input/output devices), while finite, is extremely large. Consequently, any analytic study of the system having this deterministic view becomes intractable even for the simplest of systems.

To model computer systems analytically, one often constructs a stochastic model of one or more of the components. For example, the operation of a disk device can be assumed to result in a random delay corresponding to the service request that is drawn from a given probability distribution. Similarly, the execution of a program can be modeled as a sequence of processor activity intervals and input/output activity intervals, where the lengths of the intervals are random variables with appropriate distributions. In both of these examples, we were able to model the particular system component in a very simple way that is able to hide the many internal states that are of no interest to us.

In this dissertation, we make extensive use of stochastic models for the analysis of system performance. Certain results from elementary probability theory and general stochastic process theory (with renewal and Markov processes as special cases) will be used without proof. References [Fel68a, Ros72a, Ros70a] can be consulted for an in depth treatment of that material.

1.3.2. Trace-Driven Simulation

Rather than building a stochastic model for an event, we can use data that was collected from an instance of the process that generated the events. The collection of such data, called a *trace*, can be input to a simulator representing the remaining portions of the system. In our studies, the results obtained using stochastic models are validated through trace-driven simulations, where the trace data represents the execution of a program by the sequence of memory addresses it generated.

1.4. Summary and Concluding Remarks

The next two chapters are concerned with the development of the appropriate tools to be used in virtual storage performance evaluation studies. In the next chapter, the problem of generating memory reference strings that are to be used instead of real program address traces with a generator based on the Least-Recently-Used Stack Model (LRUSM) of program behavior is considered. A method to transform the stack distance probability mass function that drives the generator is proposed which results in memory reference strings that are a fraction of the original string length while preserving most of its essential performance characteristics. The reduced string can be processed in the same way as the original string for virtual memory studies that deal with memory sizes greater than k, the parameter of the transformation.

In Chapter 3 we apply the results of the previous chapter to the problem of constructing synthetic programs that can be used for performance studies of systems which support virtual memory. Due to its significant effect on the performance of such systems, the usual characterization of program behavior is extended to include the memory referencing pattern. The results of Chapter 2 are applied to this problem and we outline how to construct a synthetic program that is able to reproduce a given lifetime curve. The statistical and practical limitations of this method are also discussed. Results obtained from an actual implementation of the proposed program indicate that it is able to conform to a given lifetime function as it executes in varying amounts of memory.

The central theme of Chapter 4 is the development and analysis of page replacement algorithms in a virtual memory environment where hardware

collection of page reference information is lacking. This study was motivated by the existence of such an environment in the VAX-11/780 computer system. We introduce a class of page replacement algorithms for the VAX that partition the main memory into two logically disjoint regions and employ different policies for their management. Software is used to collect page reference information. The memory-management scheme implemented by VMS, the vendor-supplied operating system for the hardware, is shown to be a member of this broader class of *hybrid* policies for which we derive expressions for the relevant performance indices based on the Independent Reference Model (IRM) of program behavior.

Due to the complexity of the expressions derived in Chapter 4, the cost incurred by using software to detect references to pages in order to overcome the deficiencies in the hardware cannot be dealt with analytically. Furthermore, the computational complexity of the results which have been obtained makes the analysis of program models that are realistic in size prohibitive. Chapter 5 extends our analysis of the hybrid policies by performing trace-driven simulation studies. From these, we find the conclusions derived from unrealistically small analytic models to be also applicable to the real programs we experimented with. Furthermore, the hybrid policies are found to lack robustness with respect to our performance measure in a multiprogramming environment. As alternatives, we considered the *clock* and the *sampled working set* policies in our simulations.

Based on its relative performance and ease of implementation, we chose the global clock policy as the page replacement algorithm for the virtual memory extensions to the UNIX operating system for the VAX-11/780 computer system. Chapter 6 describes this effort, which involved converting a swap-based system to one employing paging. Measurement results comparing the two live systems under identical workloads are presented.

The problem of optimal free memory pool size is considered in Chapter 7. We are able to formalize the problem by using results from inventory control theory. In our model, the free memory pool appears as a stock room containing a certain commodity with a stochastic demand process. Optimum policies for the model are those that minimize the long run operating costs for holding the inventory at a certain level, ordering additional items, and loosing orders due to depleted inventory. By a mapping of these costs and actions to the free memory pool management problem, we can obtain simple policies that have been shown to be optimal in the sense described above. Traces of memory demand from the system described in Chapter 6 are studied to test the assumptions made about the demand process. Requests for memory are seen to exhibit serial correlation contrary to the model assumption. Although not incorporated into the model at this point, such properties of the demand process can be exploited by using forecasting techniques.

Chapter 8 concludes the thesis by summarizing the major findings and indicating avenues for future research.

1.5. References

[Bel66a] L. A. Belady, "A Study of Replacement Algorithms for a Virtual Storage Computer," *IBM Syst. J.* 5 pp. 78-101 (1966).

[Bud81a] R. L. Budzinski, E. S. Davidson, W. Mayeda, and H. S. Stone, "DMIN: An Algorithm for Computing the Optimal Dynamic Allocation in a Virtual Memory Computer," *IEEE Trans. Software Eng.* SE-7(1) pp. 113-121 (January 1981).

[Chu76a] W. W. Chu and H. Opderbeck, "Program Behavior and the Page Fault Frequency Replacement Algorithm," *Computer* 9 pp. 29-38 (November 1976).

[Cof73a] E. G. Coffman and P.J. Denning, *Operating Systems Theory*, Prentice-Hall, Enlewood Cliff, New Jersey (1973).

[Cor68a] F. J. Corbato, "A Paging Experiment with the Multics System," Project MAC Memo MAC-M-384 Mass. Inst. of Tech. (July 1968). Published in In Honor of P. M. Morse ed. Ingard MIT Press 1969, pp. 217-228

[Den68a] P. J. Denning, "The Working Set Model of Program Behavior," *Comm. ACM* 11(5) pp. 323-333 (May 1968).

[Den70a] Peter J. Denning, "Virtual Memory," *Comptng. Surveys* 2(3) pp. 153-189 (September 1970).

[Den72a] Peter J. Denning and Jeffrey R. Spirn, "Some Thoughts About Locality in Program Behavior," *Proc. Symp. on Computer Communications Networks and Teletraffic,* pp. 101-112 , New York, N. Y.(1972).

[Den78a] Peter J. Denning and Donald R. Slutz, "Generalized Working Sets for Segment Reference Strings," *Comm. ACM* 21(9)(September 1978).

[Eas79a] M. Easton and P. A. Franaszek, "Use Bit Scanning in Replacement Decisions," *IEEE Trans. Comptrs.* C-28 pp. 133-141 (February 1979).

[Fel68a] W. Feller, *Introduction to Probability and its Applications*, Wiley (1968). 3rd. ed., vol. 1

[Fer74a] Domenico Ferrari, "Improving Program Locality by Critical Working Sets," *Comm. ACM* 17(11) pp. 614-620 (November 1974).

[Fer76a] Domenico Ferrari, "The Improvement of Program Behavior," *Computer* 9(11) pp. 39-47 (November 1976).

[Fog74a] M. H. Fogel, "The VMOS Paging Algorithm - A Practical Implementation of a Working Set Model," *Operating Systs. Rev.* 8 pp. 8-17 (January 1974).

[Fot61a] John Fotheringham, "Dynamic Storage Allocation in the Atlas Computer Including an Automatic Use of Backing Store," *Comm. ACM* 4(10) pp. 435-438 (October 1961).

[Fra74a] P. A. Franaszek and T. J. Wagner, "Some Distribution Free Aspects of Paging Algorithm Performance," *J. ACM* 21 pp. 31-39 (January 1974).

[Hat71a] D. J. Hatfield and J. Gerald, "Program Restructuring for Virtual Memory," *IBM Syst. J.* 10(3) pp. 168-193 (1971).

[Kil62a] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner, "One Level Storage Systems," *IRE Trans. Electronic Computers* EC-11 pp. 223-235 (April 1962).

[Kin71a] W. F. King, "Analysis of Demand Paging Algorithms," *Proc. IFIPS Congress*, pp. TA-3-155 - TA-3-159 , Ljubljana, Yugoslavia(1971).

[Lau79a] E. Lau, "Performance Improvement of Virtual Memory Systems by Restructuring and Prefetching," Ph.D. Th., Univ. California Berkeley, California (1979).

[Mat70a] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Syst. J.* 9 pp. 78-117

(1970).

[Oli74a] N. A. Oliver, "Experimental Data on Page Replacement Algorithm," *Proc. NCC*, pp. 179-184 (1974).

[Pri74a] B. G. Prieve, "A Page Partition Replacement Algorithm," Ph.D. Th., Univ. of California Berkeley, California (1974).

[Pri76a] B. G. Prieve and R. S. Fabry, "VMIN- An Optimal Variable Space Replacement Algorithm," *Comm. ACM* 19 pp. 295-297 (May 1976).

[Ros70a] S. M. Ross, *Applied Probability Models with Optimization Applications*, Holden-Day, San Francisco (1970).

[Ros72a] S. M. Ross, *Introduction to Probability Models*, Academic Press, New York, New York (1972).

[Smi78a] Alan Jay Smith, "Sequential Program Prefetching in Memory Hierarchies," *Computer* 11(12) pp. 7-21 (December 1978).

[Smi76a] A. J. Smith, "Analysis of the Optimal Look Ahead Demand Paging Algorithms," *SIAM J. Comptng.* 5 pp. 743-757 (December 1976).

[Spi72a] J. Spirn and P. J. Denning, "Experiments with Program Locality," *Proc. Fall Joint Comptr. Conf.*, pp. 611-622 (1972).

CHAPTER 2

EFFICIENT GENERATION OF MEMORY REFERENCE STRINGS

2.1. Introduction

Trace-driven simulation is a frequently used method for studying the performance of various aspects of storage hierarchies, ranging from cache buffers to file systems [Bel66a, Lau79a, Mat70a, Smi81a]. Although we limit our discussion to the classical primary memory-secondary memory level of the hierarchy in a paging environment, the ideas and results we will present are naturally applicable to the others levels. The trace data used in these studies consists of a record of all the memory accesses (data and instructions) generated during an interval in the execution of a program. One of the drawbacks of this approach is that the selected programs have to be interpretively executed in order to obtain the desired address traces, unless the system being used has appropriate tracing facilities. Furthermore, simulation studies dealing with realistically long trace data (at least a few million references) are very costly both in space and time. To reduce these costs, trace-driven simulation studies can work with a *reduced* version of the original trace. A scheme to obtain such a reduced trace was first proposed by Coffman and Randell [Cof71a] for studies employing the class of *stack policies* [Mat70a]. The method was recently applied to *working set* environments by Alanko et al [Ala80a]. In both schemes, the reduced trace is constructed by recording the events corresponding to the *entry* and *exit* of pages from the set of memory resident pages as the original trace is processed by the particular management policy with a given control parameter. Due to the

inclusion property of stack policies [Mat70a] and of the working set policy [Fra78a] with respect to their parameters, studies employing values of policy parameters greater than those used for the reduction can be carried out on the reduced trace with exact results.

Smith [Smi77a] has studied two other trace reduction methods that are approximate in nature. The first of his methods, the *Stack Deletion Method* with parameter k, removes from the original trace data all references to pages that are elements of the set of the k *most recently used* pages. In other words, this scheme is identical to the one proposed by Coffman and Randell for the *Least Recently Used* (LRU) policy except for the fact that the *exit* events corresponding to pages leaving the top k elements of the stack are omitted in the reduced trace. The second method studied by Smith, the *Snapshot* or *Reference Set* method [Pri74a, Lau79a] with sample interval T, removes from the original trace data all references that are re-references to pages within a given sample interval of length T. The claim that these compression techniques preserve the essential performance characteristics of the original reference string has been verified empirically when they are processed by a wide variety of paging algorithms [Smi77a].

An alternative approach to reducing the space cost of such studies is to use a model of program behavior such as the *Independent Reference Model* (IRM) [Aho71a, Bas76a, Spi72a] or the *Least-Recently-Used Stack Model* (LRUSM) [Rau77a, Spi76a] in conjunction with a random number source, thus obtaining a generative model [Spi77a]. Since most such models require a small fixed number (usually proportional to the number of pages contained in the program) of parameters to identify them, memory reference strings of arbitrary length can be generated one reference at a time with essentially no space cost. Note that, from the operational viewpoint of the simulator, the trace-driven and the generator-driven methods are identical.

In addition to their ability to generate reference strings of arbitrary length, probabilistic models of program memory reference behavior are compact, often analytically tractable, and can be modified to represent diverse behaviors through an appropriate perturbation of their parameters [Spi76a]. The goal of this chapter, however, is not to advocate the global substitution of actual address traces with probabilistic models. We are simply interested in an efficient method for generating through the LRUSM memory reference strings that can be used in studies where the predictive capabilities of this particular model are deemed to be of sufficient accuracy. Further discussion of the nature of these studies, along with a formal description of the LRUSM, follows in the next section. Section 2.3 defines the transformation method and investigates its implications on the string length, the steady-state fault rate, and the mean memory occupancy. Section 2.4 presents a comparison of our proposed transformation method with the Stack Deletion Method. Finally, in section 2.5 we discuss a novel application of the developed method to the design of synthetic programs that are to exercise virtual memory systems.

2.2. The LRU Stack Model

2.2.1. The Descriptive Model

The model of program behavior that our study will be based upon is the LRUSM. In this model, the n pages that constitute the program's virtual address space are envisioned to be ordered in a stack according to their

recency of usage. The page referenced at time t, r_t, is the one currently occupying stack position d_t. We assume that the page references occur at equidistant time points and that the interval between references is taken to be the unit of time. The sequence of stack positions $\{d_t\}$ is called the *distance string*. The LRUSM assumes the d_t's to be independent and identically distributed samples from the population $(1,2,3,\ldots,n)$ with the stationary probabilities $\Pr\{d_t=j\}=\delta_j$. The probability mass function (pmf) $D=(\delta_1,\delta_2,\ldots,\delta_n)$, where $\sum_{i=1}^{n}\delta_i=1$ and $\delta_i>0$ for all i, uniquely defines the LRUSM. An instance of the LRUSM corresponding to a particular program execution is realized upon providing point estimates for the n parameters of D.[1] Each generated reference causes the stack to be updated by placing the referenced page, r_t, at the top (stack position 1) and all pages in stack positions 1 through d_t-1 to be shifted down one position so as to preserve the LRU ordering [Mat70a]. The set of pages occupying stack positions below d_t remain unaffected. In terms of this description of the LRUSM, the Stack Deletion Method of trace data compression with parameter k can be characterized as removing from a given trace all references that are to stack positions less than or equal to k.

Properties of the LRUSM based on analytic and experimental studies have been extensively reported in the literature [Lew71a, Lew73a, Rau77a, Smi76a, Spi76a]. Although it is able to model the steady-state page fault rate and the mean memory occupancy characteristics of real programs under a variety of paging algorithms with reasonable accuracy [Raf76a, Smi76a, Spi72a] (prediction errors in one study [Spi72a] averaged about 10% for the mean working set size and about 30% for the page

[1]Actually, only $(n-1)$ of these parameters are independent.

fault rate), the LRUSM has several known shortcomings at a more microscopic level of program behavior. In particular, the assumptions of independent and identically distributed stack distances in the LRUSM have been formally shown to be inadequate for particular address traces [Lew73a, Lew71a]. The inter-fault times, which are independent, identically and geometrically distributed with a constant parameter as a consequence of this lack of serial correlation in the distance string for the LRUSM, have been empirically observed to be correlated and to have highly skewed distributions with long tails [Lew73a, Lew71a]. Finally, the property of the LRUSM (independent of the stack distance pmf) that results in a reference string where each page of the program is accessed with the same frequency (in the limit as the length of the string tends to infinity) is not representative of real programs [Bas76a, Lau79a, Cof73a]. In light of these shortcomings, the use of the LRUSM as a *micromodel* in a two-level model of program behavior is more appropriate [Den80a]. As an example, the second level, or *macromodel*, could represent phase transitions and consist of a semi-Markov chain [Kah76a].

2.2.2. The Generative Model

Constructing a generative model based on the LRUSM simply involves generating the distance string $\{d_i\}$ as independent samples from the population $(1,2,3,\ldots,n)$ according to $\mathbf{D}$ and obtaining the reference string $\{r_i\}$ through the required stack manipulations. The stack distance pmf $\mathbf{D}$ is said to *drive* the generator. The d_i's can be obtained by transforming a sequence of uniform pseudo random numbers through a technique such as the *aliasing* scheme of Walker [Wal77a].

One possible approach to reducing the time cost of a generator-driven simulator is the following. The generator is allowed to run unaltered as described above. The resulting reference string is processed by the Stack Deletion Method of Smith with parameter k thus resulting in a compressed version of the generated string. This compressed string is then used as the actual input to the simulation study. What we seek is a method for combining the generation and compression functions whereby we obtain the compressed string directly as the output of the generation process.

2.2.3. Model Use

In light of the deficiencies listed in section 2.2.1, the use of the LRUSM in a particular study has to be justified, no matter how efficient the generation process. These accuracy and validity concerns are universal to all modeling efforts and must be resolved prior to model use. Typically, selected predictions of an instance of the model are compared to measured results in order to determine their accuracy. This step should then be repeated for several other instances of the model to establish a domain of validity. Model use can then proceed and obtain further predictions from yet different model instances with attention paid to keeping the model within the above domain [Spi77a].

As shown by the published results regarding the accuracy of the LRUSM as a predictor of the page fault rate and mean memory occupancy [Raf76a, Smi76a, Spi72a], its use is natural for studies interested in these measures where real program trace data is unavailable. It is also conceivable to use LRUSM generated traces to develop and test simulators that will eventually run using real trace data in an effort to minimize the cost of program development. Perhaps the most common application of model generated traces, however, is found in those studies interested in the relative rather than absolute performances of various management schemes and/or algorithms and data structures used to process trace data (as an example, refer to the study by Olken [Olk80a], which compares three different data structures for obtaining LRU fault rate statistics for extremely large disk caches, where the file access string is generated through the LRUSM). Finally, for certain measures of interest in a given environment, the LRUSM is analytically intractable. In such cases one naturally resorts to simulation based on the LRUSM. The analysis by Rau [Rau79a] of the effective bandwidth of an interleaved memory system, where the program memory module reference behavior is modeled through the LRUSM, is a case in point.

2.3. The Transformation Method

In this section we will develop a method (called the *transformation method*) for modifying the stack distance pmf of an LRUSM, and will formally show that the resulting string preserves the steady-state page fault rate of the original string while being only a fraction of its length. In section 2.3.2, the implications of this transformation on the mean memory occupancy statistic are investigated.

2.3.1. Fault Rate Characteristics

Our strategy in developing the method will be as follows. We will review some of the basic properties of the LRUSM and eventually derive an expression for a confidence interval of the steady-state page fault rate. We will repeat these steps also for an incompletely specified LRUSM obtained from

the original model through a transformation of the stack distance pmf. The requirement that a statistic based on this new LRUSM result in a confidence interval of the same width at the same confidence level as the original LRUSM will then be used to completely define the transformation.

Let $\mathbf{D}_1 = (\delta_1, \delta_2, \ldots, \delta_n)$ be the driving stack distance pmf for the original generator, hereafter referred to as GEN1. For $\mathbf{D}_1$, we define the cumulative probabilities as $\Delta_j = \sum_{i=1}^{j} \delta_i$.

The page faults due to the reference string generated by GEN1 in a fixed memory partition of m pages managed by the LRU policy constitute a discrete time renewal process. In particular, the inter-fault time distribution is geometric with parameter $\lambda_m = \sum_{i=m+1}^{n} \delta_i = (1-\Delta_m)$ [Spi76a].

Let the random variable $N_1(t_1;m)$ denote the number of faults generated until some arbitrary time t_1 in a fixed size memory of m pages. By renewal theory [Ros70a], the steady-state fault rate is given by

$$\lim_{t_1 \to \infty} \frac{N_1(t_1;m)}{t_1} = (1-\Delta_m) = \lambda_m . \qquad (2.3.1.1)$$

Equation (2.3.1.1) holds with probability 1.

For $t_1 < \infty$, however, applying Chebyshev's inequality for the random variable $N_1(t_1;m)/t_1$ we have

$$\Pr\left\{ \left| \frac{N_1(t_1;m)}{t_1} - \lambda_m \right| < \varepsilon \right\} > 1 - \frac{\sigma^2(t_1)}{\varepsilon^2} \qquad (2.3.1.2)$$

for any $\varepsilon > 0$ and where $\sigma^2(t_1) = Var(N_1(t_1;m)/t_1)$.

For fixed t_1 and m, the random variable $N_1(t_1;m)$ has a binomial distribution with parameters $(1-\Delta_m)$ and t_1 because it simply counts the number of successes in t_1 Bernoulli trials with a constant probability of success. Thus, the variance $\sigma^2(t_1)$ is given by

$$\sigma^2(t_1) = Var(N_1(t_1;m)/t_1) = (\Delta_m(1-\Delta_m))/t_1. \qquad (2.3.1.3)$$

Note that $\left[\frac{N_1(t_1;m)}{t_1} - \varepsilon,\ \frac{N_1(t_1;m)}{t_1} + \varepsilon\right]$ is a level $(1 - \frac{\sigma^2(t_1)}{\varepsilon^2})$ confidence interval of length 2ε for λ_m.

Now consider a second generator, called GEN2, that is driven by the following transformed stack distance pmf:

$$\mathbf{D}_2 = (\delta'_1,\delta'_2,\ldots,\delta'_k,R\delta_{k+1},R\delta_{k+2},\ldots,R\delta_n)$$

where k is the fixed parameter of the transformation and R is an arbitrary positive constant.

Note that the transformation of δ_i $(1 \le i \le k)$ is left unspecified; the only constraint it has to satisfy is

$$\sum_{i=1}^{k}\delta'_i = 1-R(1-\Delta_k) \qquad (2.3.1.4)$$

so that $\mathbf{D}_2$ is indeed a pmf.

As before, let the random variable $N_2(t_2;m)$ denote the number of faults resulting from GEN2 by some arbitrary time t_2 under the same circumstances as with GEN1 (i.e., fixed memory partition of m pages managed by the LRU policy).

Note that, for GEN2 with $k<m$, the statistic $N_2(t_2;m)/Rt_2$ is an unbiased

estimate of λ_m. Furthermore,

$$Var(N_2(t_2;m)/Rt_2) = (1-\Delta_m)(1-R(1-\Delta_m))/Rt_2. \qquad (2.3.1.5)$$

since $N_2(t_2;m)$ is binomially distributed with parameters $R(1-\Delta_m)$ and t_2. As for GEN1, the level of a confidence interval of length 2ε for λ_m in terms of this new statistic is given by

$$\Pr\left\{\left|\frac{N_2(t_2;m)}{Rt_2} - \lambda_m\right| < \varepsilon\right\} > 1 - \frac{Var(N_2(t_2;m)/Rt_2)}{\varepsilon^2}. \qquad (2.3.1.6)$$

The motivation for the transformation method becomes evident if we make the following observation: for some given t_1, there exist values of R and t_2 with $t_2<t_1$ such that $N_2(t_2;m)/Rt_2$ is as good a statistic for λ_m as $N_1(t_1;m)/t_1$. We formalize this statement in the following proposition:

Proposition : For some given memory size m, time t_1, and transformation parameter $k<m$, the smallest possible time t_2 for which the statistic $N_2(t_2;m)/Rt_2$ achieves the same confidence as $N_1(t_1;m)/t_1$ for the steady-state fault rate λ_m is obtained by a $\mathbf{D}_2$ where $R=R^* = 1/(1-\Delta_k)$.

Proof : Equating the right hand sides of equations (2.3.1.2) and (2.3.1.6) corresponding to the confidence levels due to GEN1 and GEN2 respectively for a given ε, and substituting equations (2.3.1.3) and (2.3.1.5) for the two variances, we obtain

$$1-\frac{\Delta_m(1-\Delta_m)}{\varepsilon^2 t_1} = 1-\frac{(1-\Delta_m)(1-R(1-\Delta_m))}{\varepsilon^2 Rt_2}. \qquad (2.3.1.7)$$

Simplifying and solving for t_2 yields

$$t_2 = \frac{t_1(1-R(1-\Delta_m))}{R\Delta_m}. \qquad (2.3.1.8)$$

Thus, the minimization of t_2 as a function of R can be formulated as the following optimization problem:

$$\text{minimize } t_2 = t_1(1-R(1-\Delta_m))/R\Delta_m$$
$$\text{subject to } 0 \le R \le \frac{1}{1-\Delta_k}$$

where the constraints simply ensure that D_2 is a valid pmf. Note that for given t_1 and m, t_2 is a strictly decreasing function of positive R. Therefore, the minimum t_2 value is that imposed by the upper bound on R. That is, the optimum value of R is given by $R = R^* = \frac{1}{1-\Delta_k}$

■

Evaluating t_2 at $R = R^*$ yields

$$t_2^* = (\frac{\Delta_m - \Delta_k}{\Delta_m})t_1.$$

Note that

$$t_2^* \le \frac{t_1}{R^*},$$

with the equality holding only when $m=n$. This observation leads us to the conclusion that the length of the string generated by GEN2 need only be at most $1/R^*$th the length of that due to GEN1 to achieve the same confidence for λ_m for all memory sizes greater than k.

Recalling the form of D_2 and equation (2.3.1.4), the stack distance pmf transformation implied by the optimal value of R is of the form

$$D_2 = (0,0,\ldots,0,R^*\delta_{k+1},R^*\delta_{k+2},\ldots,R^*\delta_n).$$

Given D_2, the operation of GEN2 is identical to that of GEN1 except that now the d_i's are generated as independent random variables with the frequencies specified by D_2.

We make the following observations about the transformation method:

(1) The transformation preserves the long-run relative occurrences of stack depths greater than k, i.e., $\delta_i/\delta_j = \delta'_i/\delta'_j$ for all i and $j>k$.

(2) GEN2 produces no references to stack depths less than or equal to k, while it references depths greater than k with increased probabilities. This result shows the analogy between the above scheme of generating memory references and the Stack Deletion Method of compressing existing memory reference trace data.

(3) The optimal value of R, R^*, may be interpreted as the expected number of references until the first reference to a stack depth greater than k. This confirms our earlier observation that GEN2 suppresses the references to the top k pages of the stack.

2.3.2. Mean Memory Occupancy Characteristics

In the previous section, we have shown that the proposed method preserves the page fault rate characteristics of the original string in an environment managed by a particular fixed partition policy (namely LRU). Due to its strong interaction with the process scheduling mechanism and significant impact on overall system performance, mean memory occupancy in a variable partition environment is another important property associated with a reference string. As an example of a variable partition policy, we will consider the behavior of the output of GEN2 when processed by the working set algorithm [Den68a].

Let $\omega(\tau)$ denote the steady-state working set size with parameter (window size) τ. Recalling the definitions of the LRUSM and of the working set policy, the steady-state working set size distribution can be expressed through the recursive relationship

$$\Pr\{\omega(\tau)=i\} = \Delta_i \Pr\{\omega(\tau-1)=i\} + (1-\Delta_{i-1})\Pr\{\omega(\tau-1)=i-1\} \quad (2.3.2.1)$$

where $\Pr\{\omega(1)=1\} = 1$ [Spi77a].

As applied to the LRUSM that drives GEN2, equation (2.3.2.1) becomes

$$\Pr\{\omega(\tau')=i\} = \Delta'_i \Pr\{\omega(\tau'-1)=i\} + (1-\Delta'_{i-1})\Pr\{\omega(\tau'-1)=i-1\} \quad (2.3.2.2)$$

where the transformed cumulative stack distance probabilities are given by

$$\begin{aligned}\Delta'_i &= \sum_{j=1}^{i}\delta'_j = \sum_{j=k+1}^{i} R^*\delta_j \\ &= R^*\Big(\sum_{j=1}^{i}\delta_j - \sum_{j=1}^{k}\delta_j\Big) \\ &= R^*(\Delta_i-\Delta_k) = \frac{\Delta_i-\Delta_k}{1-\Delta_k}\end{aligned}$$

and we replace the original window size by

$$\tau' = \tau/R^* = (1-\Delta_k)\tau\,;$$

note that τ is scaled down by a factor of R^* since each reference generated by GEN2 advances the clock by R^* ticks rather than by one.[2] If the δ_i's are

[2]As we have defined it, R^*, being the mean of a random variable, is a real number. Conceptually, there is no reason why the clock of the simulator cannot be advanced by this nonintegral amount. However, for studies that require R^* to be integer, we can define $R^* = \lfloor 1/(1-\Delta_k) \rfloor$ (i.e., we can truncate it to an integer), in which case the probability measure $1-R^*(1-\Delta_k)$ as given by

nonzero for all $i>k$, equation (2.3.2.2), which is valid only for $i>k$, has the closed form solution [Spi77a]

$$\Pr\{\omega(\tau')=i\} = \begin{cases} (1-\Delta'_{k+1})(1-\Delta'_{k+2})\ldots(1-\Delta'_{i-1}) \displaystyle\sum_{j=k+1}^{i}\left[\frac{\Delta_j'^{\,\tau'-1}}{\prod_{\substack{l=k+1\\ l\neq j}}^{i}(\Delta'_j-\Delta'_l)}\right] & ,\ k<i\le n \\ 0 & ,\ i\le k. \end{cases}$$

Having the distribution of $\omega(\tau)$ at hand, the mean, $\overline{\omega}(\tau)$, can be obtained trivially.

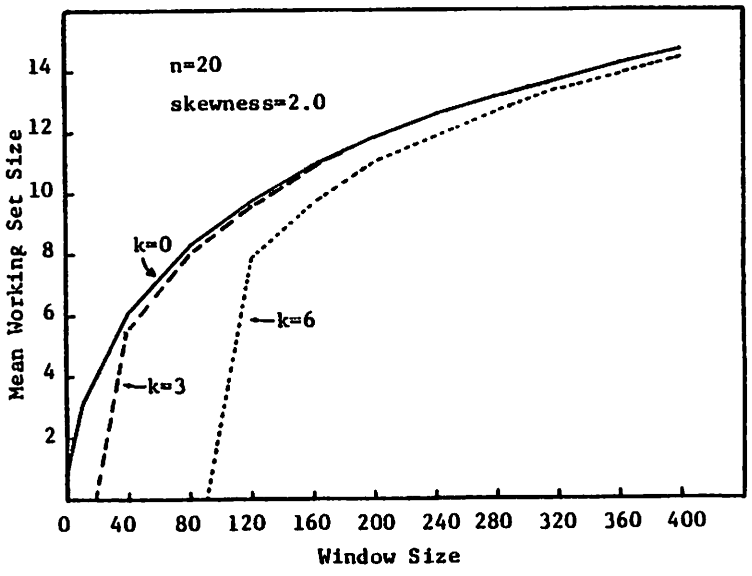


Figure 2.3.2.1 Mean working set size of sample program vs. window size for various values of the transformation parameter k.

equation (2.3.1.4) will have to be assigned (in an arbitrary manner) to the first k stack positions.

Ideally, one would like to show analytically that, for small k and large τ (note that these are the normal operating conditions for the method), the first moment of the above distribution approaches the mean working set size of the original string. However, due to the complexity of the expressions involved, only numerical results have been obtained. Figure 2.3.2.1 presents the mean working set size of a sample program whose LRUSM parameters were obtained through Zipf's Law with skewness 2.0[3] [Knu73a]. The figure shows that the transformation preserves with good approximation the first moment of the working set size distribution even for large k, particularly when the window size τ is large. Recall that setting $k=0$ results in the null transformation and reduces GEN2 to GEN1. For the example at hand, the relatively large errors encountered for small values of τ are simply due to the scaling that is performed on the window size (which happens to be $\tau'=\tau/15.15$ when $k=6$).

In an effort to identify the region of validity with respect to k, we plot the percentage error of $\bar{\omega}(\tau)$ and the reduction ratio (t_2^*/t_1) as functions of k in Figure 2.3.2.2. For the particular value of τ being used, the error in $\bar{\omega}(\tau)$ is very close to zero for all $k \le 7$. These values of k are such that the following inequality is satisfied:

$$\tau' = (1-\Delta_k)\tau \ge n\,, \qquad (2.3.2.4)$$

where n is the number of pages in the program. This condition parallels the one requiring memory sizes greater than k to be used in the study of fault rate statistics under LRU management. Figure 2.3.2.2 also shows that a substantial reduction ratio in string length is obtained for this set of transformation parameter values.

[3]According to Zipf's Law with skewness 2.0, the stack distance pmf is $\delta_i = c/i^{2.0}$, where c is a normalizing constant.

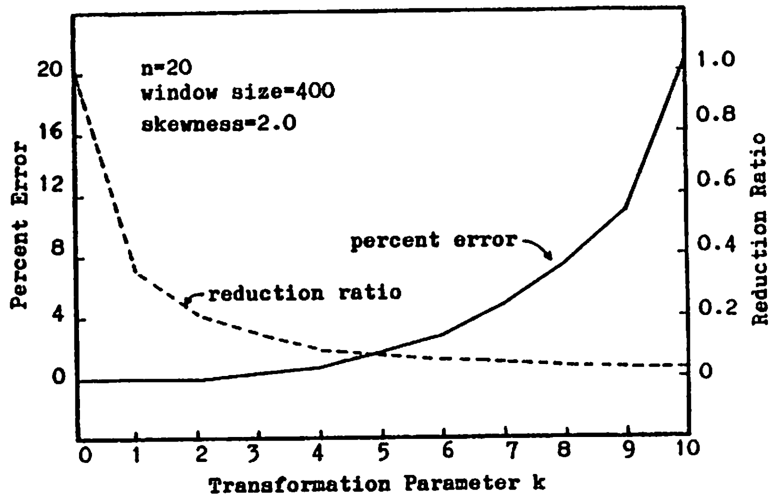


Figure 2.3.2.2 Percentage error and reduction ratio for sample program as a function of the transformation parameter.

The above results for the mean working set size are directly applicable to the steady-state fault rate observed under working-set memory management, since one can express the fault rate as the first difference[4] of the mean working set size [Spi77a]

$$\lambda(\tau) = \bar{\omega}(\tau) - \bar{\omega}(\tau-1).$$

Because GEN2 consistently underestimates the mean working set size, its predictions at τ and $\tau-1$ can be expressed as $\bar{\omega}(\tau)-e(\tau)$ and $\bar{\omega}(\tau-1)-e(\tau-1)$ respectively, where $e(\tau)$ and $e(\tau-1)$ are the corresponding error terms and

[4]This is the discrete time analog of the first derivative

$\omega(\tau)$ and $\omega(\tau-1)$ are the true mean working set values. We observe that for similar window sizes, the errors in the mean working set predictions of GEN2 are also similar, i.e., $e(\tau)\approx e(\tau-1)$, so that these error terms approximately cancel each other out in the fault rate estimate $\lambda(\tau)$. In other words, although the curves in Figure 2.3.2.1 display large absolute errors in the mean working set size for certain values of τ and k, their *slopes* at any given value of the window size are very close.

2.4. Comparison with the Stack Deletion Method

In the previous section we have seen that the Stack Deletion Method and the proposed transformation method are very similar to each other. We note, however, that the page names associated with the references in the strings resulting from the compression of the GEN1 output by the Stack Deletion Method and in that produced by GEN2 are *not* the same. This can be explained by observing that, while the transformation method generates no references to stack depths less than or equal to k, the Stack Deletion Method processes *all* of the references in the original string (in this case, that produced by GEN1), thus causing stack updates at each reference. In other words, the two methods produce results that have similar *distance* strings (the d_i's) but different *reference* strings (the r_i's).

Parameter k	WATFIV	FFT	APL
2	4.54	5.08	5.26
3	7.04	11.64	9.55
4	9.80	24.80	13.15
5	13.00	33.75	17.85
8	25.38	73.34	43.48
12	38.17	178.22	88.50

Table 2.4.1[5] Ratio of GEN1 string length to GEN2 string length (ℓ_1/ℓ_2')

[5]Generated from data presented by Lau [Lau79].

The reduction in string length due to both methods is given by $1/(1-\Delta_k)$. Table 2.4.1 shows the length reductions obtained when we apply the transformation method with various values of the parameter k to the LRUSM of three sample programs. More data about the traces from which the LRUSM for the three programs were obtained are presented by Smith [Smi76b, Smi77a].

2.5. Applications

The application of the transformation method to generator-driven simulation studies within the framework discussed in section 2.2.3 is immediate. The simulators being driven by GEN2 process the references just as before, but increment the clock by the quantity R^* instead of by 1 at each reference. Choosing a value of k to use as the transformation parameter involves a tradeoff between the simulation speed-up desired and the range of validity of the results. Working-set management studies should be restricted to the set of window size values that satisfy inequality (2.3.2.4), while LRU replacement studies are applicable only for memory sizes greater than k.

Although we have emphasized generator-driven simulation studies as the main application area, the method can also be used in the construction of synthetic program design for virtual-memory environments. Recall that a synthetic program [Buc69a] is a parameterized piece of code that consumes controlled amounts of system resources. In a virtual-memory environment, programs consume resources (CPU cycles, disk I/O bandwidth, etc.) not only explicitly, but also implicitly (CPU cycles, main memory, and paging I/O bandwidth) because of the automatic memory management functions. One of the simplest ways to characterize the behavior of a program in a virtual memory environment is to specify its *lifetime* function [Bel69a]. This function gives the mean number of instructions executed by the program between consecutive page faults when it is allocated m page frames of memory[6]. Suppose we extend the classical synthetic program requirements by specifying a lifetime function it has to conform to. We proceed by determining the parameters of the LRUSM corresponding to the given lifetime function [7]. The synthetic program is simply an implementation of the reference string generation algorithm that is based on the above-constructed LRUSM. However, the execution of this synthetic program (observed at the memory reference level) results in a string of the form $q_1,q_2,\ldots,q_Q,r_1,q_1,q_2,\ldots,q_Q,r_2,\ldots$ The Q references (all to a small set of pages containing the code and data for the synthetic program) between each of the desired references (the r_i's) are due to random number generation, stack updating and other functions that the program has to perform. To be able to run the synthetic program in real time, these Q references are clearly undesirable and will be termed *overhead references*. Suppose that we apply our transformation method with parameter k to the LRUSM built into the program, with k being the smallest integer for which $R^* \geq Q$. Now, the Q overhead references appear to be part of the desired reference string, and the synthetic program reproduces the paging behavior specified by the input

[6] In an extended definition of the lifetime function, the memory allocated to the program can vary over time and we let m represent its mean value. In our case however, we are concerned with LRU replacement with a fixed partition of m pages.

[7] Under LRU management, there exists a one-to-one correspondence between an LRUSM and a nondecreasing lifetime function.

lifetime function within the applicability of the transformation method with parameter k in *real time*. Further details of the topic along with an actual implementation are discussed in the next chapter.

2.6. Conclusions

We have presented a method for the efficient generation of memory reference strings based on the LRU stack model of program behavior. The claim that the shorter output of the modified generator preserves the page fault rate characteristics of the original string (i.e., that produced by the unmodified generator) when processed by the LRU policy was shown analytically for memory sizes greater than k. The range of applicability under the working set memory-management policy is not as sharply defined. However, some necessary conditions that need to be satisfied for the fault rate and mean memory occupancy results to be valid were also presented.

The method provides a generator that is extremely economical both in space and in time and can be used, whenever the LRUSM has been judged an adequate model of the characteristics that are of interest to the particular study, as a source of memory references for any simulator that relies on trace data as input. Another interesting application of the method to the construction of synthetic programs for virtual-memory environments has also been briefly discussed.

2.7. References

[Aho71a] A. V. Aho, P. J. Denning, and J. D. Ullman, "Principles of Optimal Page Replacement," *J. ACM* 18 pp. 80-93 (January 1971).

[Ala80a] T. O. Alanko, I. J. Haikala, and P. H. Kutvonen, "Methodology and Empirical Results of Program Behaviour Measurements," *Performance Eval. Rev.* 9(2) pp. 55-66 (Summer 1980). Proceedings of Performance 80

[Bas76a] F. Baskett and A. Rafii, "The A_0 Inversion Model of Program Paging Behavior," Stanford U. Comp. Sci. Dept., Report STAN-CS-76-579 (October 1976).

[Bel66a] L. A. Belady, "A Study of Replacement Algorithms for a Virtual Storage Computer," *IBM Syst. J.* 5 pp. 78-101 (1966).

[Bel69a] L. A. Belady and C. J. Kuehner, "Dynamic Space Sharing in Computer Systems," *Comm. ACM* 12(5) pp. 282-288 (May 1969).

[Buc69a] W. Buchholz, "A Synthetic Job for Measuring System Performance," *IBM Syst. J.* 8 pp. 309-318 (1969).

[Cof71a] E. G. Coffman and B. Randell, "Performance Prediction for Extended Paged Memories," *Acta Informatica* 1(1) pp. 1-13 (1971).

[Cof73a] E. G. Coffman and P.J. Denning, *Operating Systems Theory,* Prentice-Hall, Enlewood Cliff, New Jersey (1973).

[Den68a] P. J. Denning, "The Working Set Model of Program Behavior," *Comm. ACM* 11(5) pp. 323-333 (May 1968).

[Den80a] Peter J. Denning, "Working Sets Past and Present," *IEEE Trans. Software Eng.* SE-6(1) pp. 64-84 (January 1980).

[Fra78a] Mark A. Franklin, G. Scott Graham, and R. K. Gupta, "Anomalies With Variable Partition Paging Algorithms," *Comm. ACM* 21(3) pp. 232-236 (March 1978).

[Kah76a] Kevin Kahn, "Program Behavior and Load Dependent System Performance," Ph.D. Th., Purdue U. Computer Science Department (May 1976).

[Knu73a] D. E. Knuth, *The Art of Computer Programming - Sorting and Searching,* Addison-Wesley, Mass. (1973). vol. 3

[Lau79a] E. Lau, "Performance Improvement of Virtual Memory Systems by Restructuring and Prefetching," Ph.D. Th., Univ. California Berkeley, California (1979).

[Lew71a] P. A. W. Lewis and P. C. Yue, "Statistical Analysis of Program Reference Patterns in a Paging Environment," *Proc. IEEE Int. Comptr. Soc. Conf.,* pp. 133-134 , Boston, Mass.(September 1971).

[Lew73a] P. A. W. Lewis and G. S. Shedler, "Empirically Derived Micro Models for Sequences of Page Exceptions," *IBM J. Res. Develop.* 17(2) pp. 86-100 (March 1973).

[Mat70a] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Syst. J.* 9 pp. 78-117 (1970).

[Olk80a] F. Olken, "Efficient Methods for Calculating the Success Function of Fixed Space Replacement Policies," M.Sc. Report, Univ. of California Berkeley, California (1980).

[Pri74a] B. G. Prieve, "A Page Partition Replacement Algorithm," Ph.D. Th., Univ. of California Berkeley, California (1974).

[Raf76a] A. Rafii, "Empirical and Analytical Studies of Program Reference Behavior," Ph.D. Th., SLAC Report 197, Stanford California (1976).

[Rau77a] B. R. Rau, "Properties and Applications of the Least-Recently-Used Stack Model," Stanford U. Digital Syst. Lab. Report No. 139 (May 1977).

[Rau79a] B. R. Rau, "Program Behavior and the Performance of Interleaved Memories," *IEEE Trans. Comptrs.* C-28 pp. 191-199 (March 1979).

[Ros70a] S. M. Ross, *Applied Probability Models with Optimization Applications*, Holden-Day, San Francisco (1970).

[Smi76a] A. J. Smith, "Analysis of the Optimal Look Ahead Demand Paging Algorithms," *SIAM J. Comptng.* 5 pp. 743-757 (December 1976).

[Smi76b] A. J. Smith, "A Modified Working Set Paging Algorithm," *IEEE Trans. Comptrs.* C-25 pp. 907-914 (September 1976).

[Smi77a] A. J. Smith, "Two Simple Methods for the Efficient Analysis of Memory Address Trace Data," *IEEE Trans. Software Eng.* SE-3 pp. 94-101 (January 1977).

[Smi81a] A. J. Smith, "Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms," *IEEE Trans. Software Eng.* SE-7(4) pp. 403-417 (July 1981).

[Spi77a] Jeffrey Spirn, *Program Behavior: Models and Measurements*, Elsevier North-Holland, New York (1977).

[Spi72a] J. Spirn and P. J. Denning, "Experiments with Program Locality," *Proc. Fall Joint Comptr. Conf.*, pp. 611-622 (1972).

[Spi76a] J. Spirn, "Distance String Models for Program Behavior," *Computer* 9 pp. 14-20 (November 1976).

[Wal77a] A. J. Walker, "An Efficient Method for Generating Discrete Random Variables With General Distributions," *ACM Trans. Math. Software*

3(3) pp. 253-256 (September 1977).

CHAPTER 3

CONSTRUCTING SYNTHETIC PROGRAMS FOR VIRTUAL MEMORY

3.1. Introduction

The concepts of synthetic program and virtual memory (throughout this thesis, we consider a *paged* implementation of virtual memory) have been with us for many years [Buc69a, Den70a]. As introduced by Buchholz, a synthetic program is a highly parametric program that is able to mimic a wide range of behaviors as measured by the amount of system resources consumed, for instance, CPU cycles and I/O bandwidth. While performing no useful task, the behavior of a synthetic program can be tailored to match that of any one of the actual programs that constitute a given system's workload.

Workload models are important constituents of system models such as queueing networks or simulators, but can also be used, when in executable form, to drive actual systems [Fer72a]. In particular, synthetic workloads consisting of synthetic programs may be used in empirical performance studies (see [Sre74a] for an example). Performance evaluation studies of the tuning, upgrading and competitive system procurement type that use executable program models such as synthetic programs need to have access to the actual system. In return, they provide greater credibility than analytic model or simulation-based studies since they employ the real system (hardware, operating system, compiler-generated code, etc.). Compared to benchmarking, the use of synthetic programs has the additional advantages of transportability and flexibility. For an extended discussion of these and

related topics refer to chapters 5 and 6 of [Fer78a].

One of the most important factors affecting the performance of a virtual-memory system is the manner in which programs running on it access their address spaces. Thus, synthetic programs that are constructed to be used for performance evaluation studies of such systems must be able to reproduce this behavior in a controllable manner. Compared to other aspects of program behavior (such as CPU time required, I/O activity generated, etc.), reproducing the memory referencing pattern is a far more difficult task. Fortunately, the problem has received wide attention and there exist models of varying complexity and accuracy for program memory referencing behavior [Lew73a, Spi72a, Spi76a, Spi77a, Raf76a]. To be effective in performance studies of virtual-memory systems, a synthetic program must incorporate one of these models. That way, its memory referencing behavior can be varied in a controllable manner by modifying the model's parameters.

In a recent study comparing our paging subsystem for UNIX[1], and VMS[2] (the two operating systems that exist for the VAX-11/780 computer system), Kashtan used three synthetic programs that had different memory referencing patterns [Kas80a]. While these programs, which advance through their address spaces (i) sequentially, (ii) in uniformly distributed random increments, and (iii) in random increments normally distributed about the current page with a given standard deviation, may adequately model programs for a particular application, namely, image understanding (in Kashtan's case), they lack the ability to capture the behavior of a broader class of programs that exhibit varying degrees of spatial and temporal

[1] UNIX is a Trademark of Bell Laboratories.

[2] VMS and VAX are Trademarks of Digital Equipment Corporation.

locality [Spi72a]. Furthermore, Kashtan's approach does not deal in a satisfactory manner with the problem of *overhead references* resulting from that amount of computation the synthetic program has to perform in order to generate the next desired reference according to the three patterns described above.

In the following sections of this chapter, we discuss the suitability of the Least-Recently-Used Stack Model (LRUSM) of program behavior to be incorporated in a synthetic program and propose certain modifications to the approach described above that allows us to deal effectively with the "overhead references" problem. The statistical and practical limitations of our approach are discussed in the light of a prototype implementation.

3.2. The Model

Consider the *page reference string* $r_1, r_2, \ldots, r_{i-1}, r_i, r_{i+1}, \ldots$ that is observed as a result of the execution of a given program. Each member r_i of the string is an element of the set of page names, $\{1,2,\ldots,n\}$, where n is the size of the program's address space. Requiring that an instance of our model corresponding to the given program generate a page reference string identical to the observed string would be overdemanding and for most practical purposes, useless. Instead, we will be satisfied if the model is able to reproduce certain *functions* which are defined over the string. Namely, we will be interested in reproducing $\Theta(A,\alpha)$ and $M(A,\alpha)$, which correspond to the mean time interval between consecutive page faults and the mean memory occupancy, respectively, that result when the string is processed by the page replacement algorithm A with parameter α. The choice of these two measures follows directly from the observation that two of the more important resources in a virtual-memory system are paging I/O bandwidth and physical

memory.

Amongst the variety of models that have been proposed as predictors of program memory referencing behavior, the one we will incorporate in our synthetic program is the LRUSM [Spi76a, Cof73a]. Properties of the LRUSM, which is able to capture certain aspects of the "locality of referencing" behavior, have been analytically and experimentally investigated [Rau77a, Smi76a, Spi77a, Raf76a]. As stated in the previous chapter, the LRUSM has been observed to be a reasonable predictor of the two measures that have been defined over the page reference string; the mean inter-fault time and the mean memory occupancy [Spi76a, Raf76a].

We recall the notation introduced in chapter 2, where the d_i's are the independent and identically distributed stack distances having the common probability mass function (pmf) $\mathbf{D}=(\delta_1,\delta_2,\ldots,\delta_n)$ and cumulative stack distance probabilities $\Delta_i = \sum_{j=1}^{i}\delta_j$. Note that, since we are interested in long-run statistics, the initial stack contents are immaterial.

3.2.1. Estimating the Model's Parameters

An instance of the LRUSM, corresponding to a given program, is realized by estimating the n parameters of the model. Although the stack distance probabilities can be measured directly from a given page reference string, we will choose to estimate them from the curve of mean inter-fault time vs. mean memory occupancy. This so-called *lifetime* curve [Bel69a] is one of the more natural ways of specifying referencing behavior in addition to being one of the measures we are directly interested in.

Figure 3.2.1.1 displays the lifetime curve of an n-page program which references its address space in a uniformly distributed random manner as

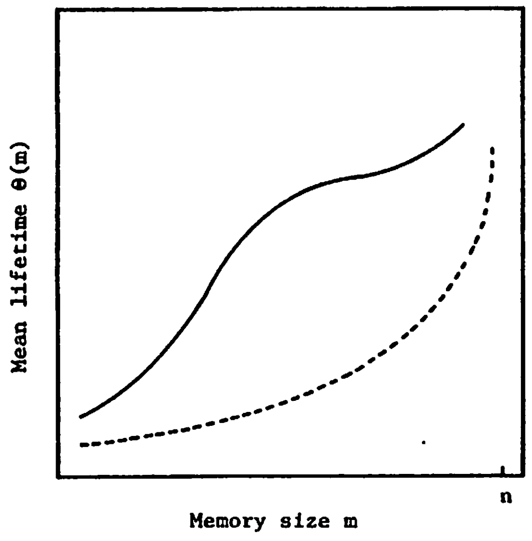


Figure 3.2.1.1 Lifetime curves of two programs with varying amounts of locality.

indicated by the dashed curve of the form $\Theta(m)=c\cdot n/(n-m)$. The solid curve, on the other hand, represents a program which exhibits a more *local* behavior. Unless otherwise noted, we let A=LRU and drop it from our notation of $\Theta()$ and $M()$.

Returning to the LRUSM, in a memory of m page frames that is managed by the LRU policy, the events corresponding to page faults constitute a discrete-time renewal process with a geometric inter-event time distribution [Ros70a]. The parameter of the inter-fault distribution is simply given by $p = \Pr\{page\ fault\} = 1-\Delta_m$ and the mean inter-fault time by $1/p$. Recalling the definition of the lifetime curve, for the memory size m, the quantity

$\Theta(m)$ equals $1/p$ if the lifetime curve is obtained under an LRU replacement policy (or some approximation of it). Thus, in general, we obtain the LRUSM parameters through the set of equations[3]

$$\Delta_m = 1 - 1/\Theta(m)\,, \qquad 1\le m\le n \qquad (3.2.1.1)$$

where we define $\Theta(n)$ to be ∞.

3.2.2. Generation of Reference Strings

Given an instance of the LRUSM where the model parameters, $\mathbf{D}=(\delta_1,\delta_2,\ldots,\delta_n)$, have been obtained from a given program as outlined in the preceding section, we construct a *generator* of page reference strings based on the LRUSM by sampling the stack distances, d_i's, from a population having distribution $\mathbf{D}$ and then transforming them into page names through the stack updating mechanism. Formally, the steps that need to be performed are outlined below:

GEN1 (Generator of page reference strings based on LRUSM).

G0: (Initialization) Initialize the stack with an arbitrary content.

G1: (Random number generation). Generate a pseudo-random number uniformly distributed in the interval (0,1).

G2: (Transformation). Transform the uniform random variable to another integer random variable, d_i, that is distributed according to $\mathbf{D}$.

G3: (The actual reference). The page r_i, currently occupying stack position d_i, becomes the next reference.

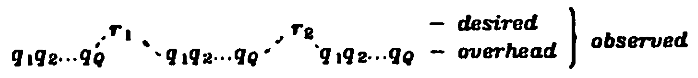
G4: (Stack updating). Update the stack by placing the referenced page at the top and pushing pages in positions 1 through d_i-1 down one posi-

[3] We assume that $\Theta(m)$ is expressed in "memory references per inter-fault interval".

tion.

G5: (Loop). Go to G1.

Within the limitations due to statistical convergence (to be discussed later) and those of the LRUSM, the reference string, $r_1, r_2, r_3, \ldots$, generated in the above manner will in fact have as its lifetime curve the curve from which the model parameters were derived. The incorporation of the above generator into a synthetic program would then appear to be a solution to the problem at hand. However, if the synthetic program is to generate the next page to be referenced *on the fly* during execution, the amount of computation that is required for the task has to be dealt with effectively. The following sketch illustrates the problem:



The reference string $r_1r_2r_3 \ldots$ represents the *desired* string in that it captures the behavioral properties of the original program from which the instance of the LRUSM was derived. The generation of each one of the desired page references requires carrying out the steps outlined in G1-G4, which in turn result in the memory references, $q_1q_2 \ldots q_{Q-1}q_Q$, as a by-product[4]. From the point of view of the memory management mechanisms of the system on which the synthetic program is run, the observed memory reference string is the combined string having the form

$$q_1q_2 \ldots q_Q r_1 q_1 q_2 \ldots q_Q r_2 q_1 q_2 \ldots q_Q r_3 q_1 q_2 \ldots \ .$$

[4] Due to the probabilistic nature of the generation process, the number of memory references to be issued to generate one desired reference is a random variable, of which we consider the mean, Q.

where the desired string has been *diluted* by a factor of Q due to the *overhead references*.

Since each of the Q overhead references is to a small set of s pages that contain the instructions and the data necessary to carry out steps G1-G4 of the generation algorithm, for memory sizes greater than s we know that page faults can only occur at references $r_1r_2r_3 \ldots$ resulting in an observed lifetime $\Theta'(m) = Q \cdot \Theta(m)$, where $\Theta(m)$ is the lifetime due to the string $r_1r_2r_3 \ldots$ alone.

3.2.3. The Transformation Method

Consider a reference string generator GEN2, that is identical to GEN1 except that it is driven by the transformed stack distance pmf $\mathbf{D}^* = (0,0,\ldots,0,R\delta_{k+1},R\delta_{k+2},\ldots,R\delta_n)$, where k is the parameter of the transformation and $R = 1/(1-\Delta_k)$. In the previous chapter we have observed that, for memory sizes greater than k, the expected number of page faults due to a string of length L generated by GEN1 (the original generator) is identical to the expected number of page faults due to a string of length $\lfloor L/R \rceil$ generated by GEN2 (the transformed generator). Furthermore, for a reference string generated by GEN2, if virtual time is advanced by R units rather than by 1 unit between consecutive references, the transformation also preserves (within the limits discussed in section 2.2.3) the fault rate and the mean memory occupancy statistics with respect to the desired string produced by GEN1. Here, we will show how the method can be put to use in order to deal with the overhead references.

Since the Q overhead references cause no page faults, their effect on the desired string is simply the *stretching* of the time scale by a factor of Q. If, by anticipating this effect in advance, we transform the stack distance pmf that is used to drive the generator with a parameter k such that k is the smallest integer for which $R = 1/(1-\Delta_k) \geq Q$, in the observed string the Q overhead references will serve only to advance virtual time by Q units between the r_i's, just as required by the transformation method. In other words, rather than using the model parameters that are derived from the program directly, we use the transformed parameters, $\mathbf{D}^*$, in conjunction with the generator, thus generating a reference string with increased probability of faults such that, when *diluted* by the Q overhead references, the observed statistics are nearly those of the original program.

Yet another view of the solution is the following. In terms of the cumulative stack distance probabilities, the transformation can be expressed as

$$\Delta_m^* = \begin{cases} 0 & , m \leq k \\ R(\Delta_m - \Delta_k) & , m > k \end{cases}$$

from the definitions of Δ_m and $\mathbf{D}^*$.

Now, inverting equation (3.2.1.1) and substituting for Δ_m^* the above expression, we obtain

$$\begin{aligned} \theta^*(m) &= \frac{1}{1-\Delta_m^*} = \frac{1}{1-R(\Delta_m - \Delta_k)} \\ &= \frac{1}{1-\frac{\Delta_m - \Delta_k}{1-\Delta_k}} \\ &= \frac{1-\Delta_k}{1-\Delta m} = \frac{1}{R} \cdot \theta(m) \ , \quad m > k. \end{aligned}$$

Thus, the transformation method corresponds to using, in deriving the LRUSM parameters, the modified lifetime function

$$\theta^*(m) = \begin{cases} 1 & , m \leq k \\ \frac{1}{R}\theta(m) & , m > k \end{cases}$$

rather than the original lifetime function $\theta(m)$. Now, when we combine the effects of the transformation method and the Q overhead references on the lifetime function (see section 3.2.2), we realize that the observed lifetime function, $\theta'(m)$, is in fact identical to the original for all memory sizes greater than k. That is,

$$\theta'(m) = Q \cdot \theta^*(m) = Q(\frac{1}{R}\theta(m)) \approx \theta(m), \quad m > k$$

since we have chosen k such that $R \approx Q$. The validity of the approach is clearly restricted to memory sizes greater than k, the transformation parameter.

3.3. Statistical Considerations

It was mentioned above that, for a given value of the mean memory occupancy, one of the measures we are interested in, $\theta(m)$, is the mean of a random variable. Thus, in reporting point estimates for it based on our measurements, we need to be concerned with the length of the reference string from which the estimate is obtained (equivalently, the duration of the synthetic program's execution). In statistical terms, we are interested in the minimum sample size (page fault count) that is required for the true mean, $\theta(m)$, to be contained within a confidence interval about the observed mean, $\hat{\theta}(m)$. From our previous discussion, in a memory of m page frames managed by the LRU replacement policy, the set of random variables $X_1X_2X_3 \ldots X_N$ corresponding to inter-page fault times are independent and identically distributed random variables having a geometric distribution with mean $\mathbf{E}[X_i] = 1/(1-\Delta_m)$ and variance $Var(X_i) = \sigma^2 = \Delta_m/(1-\Delta_m)^2$. Using the fact that, for large samples (by the central limit theorem), the distribution of the sample estimate, $\hat{\Theta}(m)$, about the true mean, $\Theta(m)$, is approximately normal, an approximate confidence interval with confidence level $(1-\alpha)$ is constructed as

$$\hat{\Theta}(m) \pm \sigma\zeta(1-\tfrac{1}{2}\alpha)/\sqrt{N}, \tag{3.3.1}$$

where $\zeta(\alpha)$ is the αth quantile of the standard normal distribution. Thus, the minimum sample size required for the true mean to lie within an interval of length L about the observed mean with probability $1-\alpha$ is given by the inequality

$$N \geq (2\sigma\zeta(1-\tfrac{1}{2}\alpha)/L)^2. \tag{3.3.2}$$

To relate the above sample size (page fault count) to the reference string length, we make use of a result from renewal theory known as Wald's equation [Ros70a]. Let S be an integer-valued random variable which corresponds to the minimum page reference string length for which we observe N page faults. In other words,

$$S = \min\{s : I_1 + I_2 + \ldots + I_s = N\}$$

where

$$I_i = \begin{cases} 1, & \text{if the ith reference is a fault} \\ 0, & \text{otherwise.} \end{cases}$$

Since the event $\{S=s\}$ is independent of $I_{s+1}, I_{s+2}, \ldots$, S may be viewed as a *stopping time* for the sequence $I_1, I_2, I_3, \ldots, I_s$. Then, by Wald's equation, we have the following result:

$$\mathbf{E}\left[\sum_{i=1}^{S} I_i\right] = \mathbf{E}[S]\cdot\mathbf{E}[I_i].$$

As applied to our particular example, the expected minimum string length required to contain N page faults is given by

$$\mathbf{E}[S] = \mathbf{E}\left[\sum_{i=1}^{S} I_i\right] / \mathbf{E}[I_i] = N/(1-\Delta_m)$$

where N satisfies inequality (3.3.2).

3.4. Synthetic Program Overview

In this section, we sketch the principle components of a program that fulfills the requirements discussed in the preceding sections. The kernel of the program consists of an implementation of the generation algorithm outlined in section 3.2.2. However, before the generation can proceed, the input to the program (a point-by-point or analytic representation of the lifetime function) is used to derive the LRUSM parameters which are then transformed as described in section 3.2.3 (note that these two steps can be combined and the transformed parameters derived directly from the input). Apart from the data structures that are local to the generator's implementation, the program declares another single dimensional array of a fundamental data type (such as an integer) which constitutes the program's virtual address space. Given the output of the generator (a page name), the program accesses an element of this array that is known to be contained in the desired page [6]. Here, we assume that the host system page size and the

[6] The choice of a *read* or a *write* access to the desired page is a nontrivial one, in that dirty pages imply a write-back to secondary storage before reuse, thus affecting overall system performance. This decision can be randomized by basing it on some function of a pseudo-random number that is independent of the one used to generate the stack distances. Assumptions such as independent reads and writes where 90% of all references are reads and 10% are writes seem

fundamental data type storage size are known, so that the mapping of the array elements into pages can easily be deduced.

Although the implementation of the generation algorithm is a very straightforward task, it requires subtle coding and intelligent use of data structures in order to keep Q, the number of overhead references, to a minimum. The smaller the value of Q, the smaller the transformation parameter k, and thus the greater the range of validity of the synthetic program. Some of these issues are discussed in the next section.

3.5. Practical Considerations

As presented in section 3.2.2, the reference generation algorithm has two expensive operations that contribute substantially to the overhead references. These are (i) the generation of the stack distances with frequency given by $\mathbf{D}^*$ and (ii) mapping the stack distance to the page name and updating the stack to preserve LRU order. We will examine possible solutions to these two issues before presenting results from a sample implementation of the synthetic program.

3.5.1. Efficient Generation of Stack Distances

In the LRUSM, stack distances are independent random variables with a common general distribution. In a digital computer, there exist methods for the efficient generation of pseudo-random numbers that are uniformly distributed over some interval [Knu69a]. Given such a uniform random integer X, we are interested in transforming it to another discrete random variable Y that has an arbitrary distribution over this same interval. The classical

appropriate.

method of transforming Y through the cumulative probability function of X has the drawback that it requires a search amongst the range of Y (this can be accomplished at best in $O(\log n)$ steps, where n is the distinct number of values that Y can assume) and consequently is costly. Given two independent uniform random variables X and U, where X is uniform over the range of Y and U is uniform over (0,1), we state without proof that it is possible to construct an array of integers $A(X)$ and an array of probabilities $F(X)$ such that the random variable defined as

$$Y = \begin{cases} X & \text{if } U \le F(X) \\ A(X) & \text{if } U > F(X) \end{cases}$$

has a general distribution over the range of X. The algorithm for determining the *alias* vector $A(X)$ and the *cutoff* probabilities $F(X)$ from the desired distribution of Y is given in [Wal77a]. Since the construction of the $A(X)$ and $F(X)$ arrays can precede the string generation phase, we have reduced the cost of the stack distance generation operation to constant time (independent of n). Note that, since we generate a sequence of stack distances, at each reference we need only generate *one* pseudo-random number, X_i, over the set of possible stack depths and use the pair $(X_i, X_{i-1}/n)$ as the (X,U) pair as required by the algorithm. This is possible since the set of random numbers $\ldots X_{i-2}, X_{i-1}, X_i, X_{i+1}, \ldots$ produced by the generator are pair-wise independent and X_{i-1}/n is uniform over (0,1).

3.5.2. Stack Manipulation

Given a stack distance generated as described above, the next step in the generation algorithm requires that the page occupying that stack position be referenced and the stack be updated to preserve the LRU ordering.

The two operations that we have to perform on the stack data structure are an *index* operation to map the stack distance into a page name and an *update* operation to move the referenced page to the top. Consider an array data structure representation of the stack. The index operation can be performed in constant time, whereas the update operation requires $O(n)$ time, where n is the program size in pages. A linked list representation of the stack on the other hand requires $O(n)$ time for the index operation and constant time for the update. Since we are concerned with the maximum of the cost of the two operations, these two data structures are equivalent for our purposes. Consider a simple variation of the linked list implementation of the stack where we divide the n elements into $\lceil\sqrt{n}\rceil$ piles, each containing $\lfloor\sqrt{n}\rfloor$ elements (except for the last pile) with secondary pointers to the head of each pile. Now, both of the operations of interest can be performed in $O(\sqrt{n})$ time. Such a two-level data structure was first suggested by Franta and Maly [Fra77a] as an efficient representation of the event queue in simulations. Olken [Olk80a] has studied the dual of our problem (the computation of LRU hit ratios) and has compared the linked list representation with a binary tree[6] representation of the stack. Although the binary tree representation requires $O(\log n)$ time for the two operations we are concerned with, due to the complexity of the implementation, the simple two-level data structure described above has smaller actual cost for moderate sizes of the stack.

[6] Actually, the tree structure studied was an AVL tree rather than a perfect binary tree.

3.5.3. Sample Implementation

In an effort to gain insight into the magnitude of Q for a real implementation, the generator algorithm of section 3.2.2 incorporating the above enhancements was coded in the C programming language and run on a VAX-11/780 computer system under the Virtual Unix operating system (see chapter 6 for further details of this system). To minimize the number of input parameters, the LRUSM parameters were derived directly from Zipf's Law with a skewness of 2.0 [Knu73a], i.e., by the equation $\delta_i = c/i^{2.0}$, where c is a normalizing constant. In Figure 3.5.3.2 we plot the number of overhead references, Q, and the reduction ratio as a function of k, the transformation parameter. The reduction ratio curve is the plot of the expression $1/(1-\Delta_k)$, whereas the overhead curve was derived experimentally by timing the execution of 100,000 generation operations and then converting the mean time between desired references of this sample to the number of memory references through the constant of proportionality

$$\gamma = 1 \ \mu\text{second/memory reference.}$$

Note that, actually, this number γ is the mean of a random variable that is dependent on the instruction mix, cache hit ratio, and other architectural features. The figure indicates a nonconstant relationship between k and Q. This is due to the shifting of the stack distance pmf towards larger depths for increasing k, thus resulting in deeper index and update operations on the stack. Due to this unfortunate dependency of Q on k, we resort to graphical methods for the solution of $Q(k) \approx R(k)$[7]. For the sample program with 500 pages to which the figure refers, the value of k which satisfies this condition

[7] Since both $Q(k)$ and $R(k)$ are discrete functions, the equality $Q(k)=R(k)$ will rarely be satisfied for an integer value of k. In the case this equality is not satisfied, we choose the smallest k for which $R(k) \geq Q(k)$.

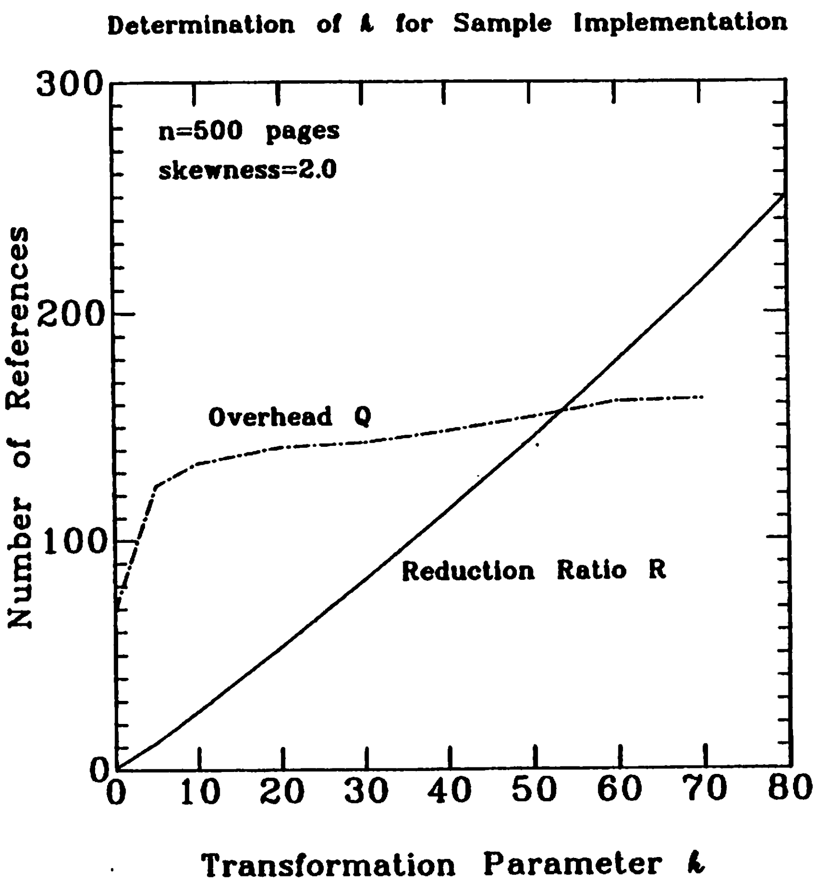


Figure 3.5.3.2 Variation of the overhead and of the reduction ratio as a function of the transformation parameter.

k	R			
	WATFIV	APL	FFT	SAMPLE
2	4.54	5.26	5.08	4.15
3	7.04	9.35	11.64	6.84
4	9.80	13.15	24.80	9.62
5	13.00	17.85	33.75	12.47
6	16.39	25.64	47.55	15.38
8	25.38	43.48	73.34	21.40
10	33.44	66.67	84.29	27.70
12	38.17	88.50	178.22	34.28
14	46.30	107.30	205.59	41.17
16	55.25	125.63	213.14	48.39
18	58.62	154.68	243.26	55.98
20	63.98	190.84	368.05	63.92
24	70.72	287.36	369.89	81.08
28	80.26	389.11	370.08	100.15

Table 3.5.3.1 String reduction ratios for three LRUSM and the sample implementation with 100-page address space.

is seen to be about 54 pages (less than 11% of the address space of the program).

To relate these values of Q to the transformation parameters that would have to be used when modeling real programs (as opposed to the hypothetical program we have modeled through Zipf's Law), we report in Table 3.5.3.1 the string reduction ratios (R values) for various values of k of three LRUSMs whose parameters were derived from three real programs called WATFIV, APL, and FFT [Lau79a] along with those of the sample implementation scaled down to a 100-page address space to make its size comparable with the others. The three LRUSM, WATFIV, APL, and FFT, consist of 96, 114, and 82 pages, respectively. More data about the traces from which the three LRUSMs were constructed can be found in [Smi76b, Smi77a]. Compared to these models of real programs, our contrived LRUSM has reduction ratios that are very similar to those of the WATFIV model over the displayed range of k. Note that, amongst the three real program models, WATFIV exhibits the least local behavior (i.e., has the least skewed stack distance pmf towards the top as can be observed from Table 3.5.3.1) and consequently requires the largest k for a given Q such that $R \approx Q$. Based on this observation, we claim that the k value reported for the sample implementation is a rather pessimistic one.

The above-constructed synthetic program was run with k=54 in varying amounts of memory under the Virtual Unix operating system. This was to confirm that the program in fact conformed to the desired lifetime function under real operating conditions. The desired and observed lifetimes for the program are shown in Figure 3.5.3.3. The observed lifetime points were obtained by timing the execution of the program and recording the number of page faults generated and the mean memory allocated to the program during the execution. As can be seen, the program indeed generated lifetimes very close to the desired lifetimes over a wide range of mean memory occupancies. The differences in the two curves can be attributed to several causes:

(i) The page replacement algorithm under which the program was run is *not* an implementation of the pure global LRU policy. The actual replacement policy employed by the Virtual Unix operating system is the global clock policy which is known to be only an approximation of the pure LRU policy (further details of the Virtual Unix memory management mechanisms will be discussed in chapter 6).

(ii) The amount of memory allocated to the program varied during its execution and we have only reported the mean. Since the desired lifetime curve is concave up, the line representing the linear combination of any two points on the curve will always fall above the curve.

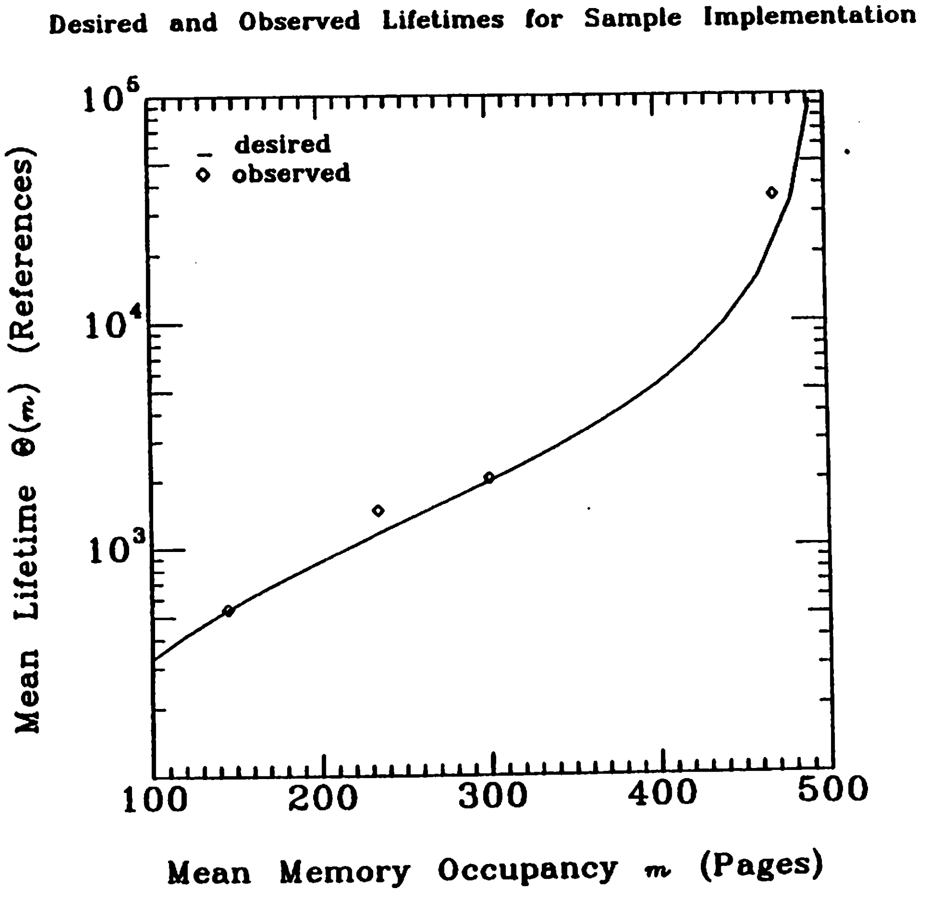


Figure 3.5.3.3 Execution of sample implementation showing desired and observed values for the lifetime.

(iii) For the points corresponding to large amounts of mean memory occupancy, the number of page faults generated was very small (less than 100), resulting in a very large confidence interval at a reasonable confidence level (see section 3.3).

3.6. Conclusions

We have discussed the construction of a synthetic program based on the LRUSM that is suitable for performance studies of virtual memory systems. A modification of the LRUSM that transforms the model parameters was shown to allow the generation of memory references *on the fly* as the synthetic program is run, at the cost of restricting the range of validity of the results to memory sizes greater than k, the parameter of the transformation. The study of an actual implementation of the generator has demonstrated that, for models of programs which exhibit realistic amounts of locality, this limitation of the transformation method is not unreasonable. The incorporation of the total CPU time requirement into the synthetic program can be effected simply by varying the length of the string generated (within the statistical convergence limits that have been discussed in section 3.3). As for the inclusion of explicit I/O requirements, this can be achieved simply by interspersing the I/O requests amongst the references generated (at the cost of further increasing Q and thus k).

3.7. References

[Bel69a] L. A. Belady and C. J. Kuehner, "Dynamic Space Sharing in Computer Systems," *Comm. ACM* 12(5) pp. 282-288 (May 1969).

[Buc69a] W. Buchholz, "A Synthetic Job for Measuring System Performance," *IBM Syst. J.* 8 pp. 309-318 (1969).

[Cof73a] E. G. Coffman and P.J. Denning, *Operating Systems Theory*, Prentice-Hall, Enlewood Cliff, New Jersey (1973).

[Den70a] Peter J. Denning, "Virtual Memory," *Comptng. Surveys* 2(3) pp. 153-189 (September 1970).

[Fer72a] Domenico Ferrari, "Workload Characterization and Selection in Computer Performance Measurement," *Computer* 5(4)(1972).

[Fer78a] D. Ferrari, *Computer Systems Performance Evaluation*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).

[Fra77a] W. R. Franta and K. Maly, "An Efficient Data Structure for the Simulation Event Set," *Comm. ACM* 20(8) pp. 598-602 (August 1977).

[Kas80a] D. L. Kashtan, "UNIX and VMS: Some Performance Comparisons," SRI International Internal Report (1980).

[Knu69a] D. E. Knuth, *The Art of Computer Programming - Semi Numerical Algorithms*, Addison-Wesley, Mass. (1969). vol. 2

[Knu73a] D. E. Knuth, *The Art of Computer Programming - Sorting and Searching*, Addison-Wesley, Mass. (1973). vol. 3

[Lau79a] E. Lau, "Performance Improvement of Virtual Memory Systems by Restructuring and Prefetching," Ph.D. Th., Univ. California Berkeley, California (1979).

[Lew73a] P. A. W. Lewis and G. S. Shedler, "Empirically Derived Micro Models for Sequences of Page Exceptions," *IBM J. Res. Develop.* 17(2) pp. 86-100 (March 1973).

[Olk80a] F. Olken, "Efficient Methods for Calculating the Success Function of Fixed Space Replacement Policies," M.Sc. Report, Univ. of California Berkeley, California (1980).

[Raf76a] A. Rafii, "Empirical and Analytical Studies of Program Reference Behavior," Ph.D. Th., SLAC Report 197, Stanford California (1976).

[Rau77a] B. R. Rau, "Properties and Applications of the Least-Recently-Used Stack Model," Stanford U. Digital Syst. Lab. Report No. 139 (May 1977).

[Ros70a] S. M. Ross, *Applied Probability Models with Optimization Applications*, Holden-Day, San Francisco (1970).

[Smi76a] A. J. Smith, "Analysis of the Optimal Look Ahead Demand Paging Algorithms," *SIAM J. Comptng.* 5 pp. 743-757 (December 1976).

[Smi76b] A. J. Smith, "A Modified Working Set Paging Algorithm," *IEEE Trans. Comptrs.* C-25 pp. 907-914 (September 1976).

[Smi77a] A. J. Smith, "Two Simple Methods for the Efficient Analysis of Memory Address Trace Data," *IEEE Trans. Software Eng.* SE-3 pp. 94-101 (January 1977).

[Spi77a] Jeffrey Spirn, *Program Behavior: Models and Measurements*, Elsevier North-Holland, New York (1977).

[Spi72a] J. Spirn and P. J. Denning, "Experiments with Program Locality," *Proc. Fall Joint Comptr. Conf.*, pp. 611-622 (1972).

[Spi76a] J. Spirn, "Distance String Models for Program Behavior," *Computer* 9 pp. 14-20 (November 1976).

[Sre74a] K. Sreenivasan and A. J. Kleinman, "On Construction of a Representative Synthetic Workload," *Comm. ACM* 17(3)(March 1974).

[Wal77a] A. J. Walker, "An Efficient Method for Generating Discrete Random Variables With General Distributions," *ACM Trans. Math. Software* 3(3) pp. 253-256 (September 1977).

CHAPTER 4

HYBRID PAGE REPLACEMENT POLICIES - ANALYTIC STUDIES

4.1. Introduction

Although it is well known that the LRU and WS page replacement algorithms have performances (as measured by the number of page faults generated for a given mean memory occupancy) superior to both FIFO and RAND [Kin71a, Raf76a], they are rarely implemented in practice due to their high cost in hardware and/or software. The author is aware of only one machine, the CDC Star-100 computer system [Oli74a], which implements LRU page replacement in hardware. However, hardware LRU management of cache memories is more common, as in the IBM 370/168 [Lip68a]. The situation is quite similar for the implementation of the WS policy [Mor72a].

The vast majority of actual implementations of replacement algorithms can be considered to be approximations of the pure LRU and pure WS algorithms. Their exact form is often dictated by the type of support provided in the host memory-management hardware. Examples of these approximations include the clock [Cor68a, Eas79a], sampled working set (SWS) [Fog74a, Pri74a], and page fault frequency (PFF) [Chu76a] replacement algorithms. The single common hardware feature that all of these algorithms base their decisions on is a *reference bit* associated with each page frame in main memory. A reference to a page causes the hardware to turn on the corresponding bit, which is then examined and reset by the replacement algorithm.

In this chapter, we address the problem of making page replacement decisions in the absence of reference bits. Note that, if given no information about page references between page faults, the two reasonable choices for a replacement algorithm are FIFO and RAND. In the following sections, we introduce a class of *hybrid* replacement policies that achieve performances close to those of LRU and WS while having implementation costs comparable to those of FIFO and RAND. The next section introduces the program model on which our analysis will be based.

4.2. The Independent Reference Model

The mathematical analysis of a replacement algorithm requires a model of the programs on which the policy operates. For our purposes, an execution of a program consisting of n pages labeled $\{1,2,\ldots,n\}$ results in a page reference string, $r_1,r_2,r_3,\ldots,r_{t-1},r_t,r_{t+1},\ldots$, where $r_t=i$ if page i is referenced at time instant t (memory references are assumed to occur at equidistant time points, and we define their distance to be the unit of time). We will assume a particularly simple stochastic structure for the reference string, known as the Independent Reference Model (IRM) [Aho71a]. As the name implies, the string $\{r_i;i=1,2,\ldots\}$ is assumed to be a sequence of independent, identically distributed random variables from the population $\{1,2,\ldots,n\}$ where $\Pr(r_t=i)=\beta_i$ for all t and $\sum_{i=1}^{n}\beta_i=1$.

Modeling an actual program with the IRM involves obtaining simple point estimates for the model parameters (β_i's) from an actual reference string generated by the program. Rafii has proposed a different method for obtaining estimates for the β_i's which he called the A_0 *Inversion Model* [Bas76a].

The resulting model, although structurally identical to the IRM, has much better predictive capabilities for real programs.

4.3. Hybrid Policies

In a demand-paged virtual memory system, referencing a page that is *invalid* -- not in main memory -- causes a trap, which is known as a *page fault*. We note that, even in the absence of reference bits, this address translation mechanism can be put to use to detect references to pages that are already in memory. All that is required is that we be able to distinguish these faults from normal page faults and refrain from initiating the I/O operation. This special state of a page will be called the *reclaimable* state and will be identified by one additional bit in each page table entry. Since this method of detecting references to pages comes at a cost (to be discussed later), we are interested in replacement algorithms that collect reference information only for a subset of the pages that a program has in memory. More formally, we have partitioned the set of pages in memory into two disjoint classes {*valid*} and {*reclaimable*}, such that

$$\{\text{memory}\} = \{\text{valid}\} \cup \{\text{reclaimable}\} \quad \textit{and}, \quad \{\text{valid}\} \cap \{\text{reclaimable}\} = \phi.$$

To keep the cost of generating spurious faults to the reclaimable pages at reasonably low levels, we would like

$$|\{\text{reclaimable}\}| \ll |\{\text{valid}\}|.$$

We make these statements more precise in the following sections. For reasons which will become clear below, we shall refer to the set {valid} as **top**, denoted **T**, and the set {reclaimable} as **bottom**, denoted **B**. Having partitioned the program's pages in memory into these two classes, we consider

various policies for top-to-bottom and memory-to-secondary storage replacements. Because we assume a single program to be executing for our investigation, the analyses presented in the forthcoming sections consider *local* management policies. Extensions of these policies to employ *global* replacement schemes required for multiprogramming environments are discussed in section 4.6 and in the next chapter.

The two reasonable choices for the management of **T** are FIFO and RAND, since for those pages we have no reference information. On the other hand, we can employ either the LRU or the WS algorithm for the management of **B** since the necessary information can be gathered at the times of these artificial page faults. This results in four possible combinations that make up the hybrid class to be studied: using the obvious notation, these hybrid algorithms are denoted by $H_{FIFO\text{-}LRU}$, $H_{RAND\text{-}LRU}$, $H_{FIFO\text{-}WS}$, and $H_{RAND\text{-}WS}$.

4.3.1. Fixed Partition Hybrids

Employment of the LRU policy for the management of **B** results in hybrid algorithms that operate in a fixed size memory partition. However, in a multiprogramming environment, the use of a common bottom amongst all the active processes results in a variable size partition for each even though the tops are strictly local. We comment about such extensions in section 4.6 and restrict our study here to uniprogramming environments. For the following analysis, assume that **T** consists of k pages (i.e., $|\mathbf{T}| = k$), where k is the parameter of the policy, whereas the fixed partition size is m pages (i.e., $|\{\text{memory}\}| = m$, $k \le m \le n$).

4.3.1.1. The FIFO-LRU Hybrid Policy

Given a page reference, r_t, at time t, the operation of the $H_{FIFO-LRU}(k)$ policy is as follows:

H1: If $r_t \in$ T, no control state change takes place. This is because this type of reference is transparent to our mechanism.

H2: (Reclaim) If $r_t \in$ B, then $T \leftarrow T + r_t - i$, where i is the FIFO page in T, and $B \leftarrow B - r_t + i$. Note that, in these expressions, "+" and "-" denote set membership operations.

H3: (Page fault) If $r_t \notin \{\text{memory}\}$ then $T \leftarrow T + r_t - i$, where page i is as in H2, and $B \leftarrow B + i - j$ where page j is the one that has been (approximately) least recently used amongst all pages.

We cannot state that page j is exactly the LRU page because the ordering amongst the top is by time of entry and not recency of use. Consequently, there may be pages in memory that have been referenced earlier than page j if, for example, page j was referenced just prior to its departure from T. A more appropriate name for the replacement policy employed in the bottom is *Least Recently Reclaimed*. In section 4.4 we present numerical results that suggest that, under a wide range of circumstances, the page replaced by these fixed partition hybrid policies from the bottom is very close to being the LRU page.

If we envision the control state associated with the algorithm to constitute a stack, the $H_{FIFO-LRU}$ policy can be regarded as a modification to the pure LRU policy where references to the top k positions of the LRU stack cause no control state change [Mat70a]. Note that, for the degenerate case $k=1$, the page replaced from memory to secondary storage by the $H_{FIFO-LRU}$ policy is exactly the same page that would be replaced by the pure LRU policy. Furthermore, when $k=m$, the $H_{FIFO-LRU}$ policy degenerates into the pure FIFO policy.

The performance index that we will use to compare different policies is the ***steady-state fault rate***. For a replacement algorithm A, the steady-state fault rate is defined as:

$$F(A) = \lim_{t \to \infty} [\Pr(r_t \notin \{\text{memory}\})].$$

In other words, $F(A)$ is the limiting probability with which a reference to a page causes a page fault.[1]

We are now in a position to derive an expression for $F(H_{FIFO-LRU}(k))$, based on the IRM. Note that the analysis technique, including the notation to be used, is similar to that used in [Cof73a]. Let $\mathbf{s}=[j_1, j_2, \ldots, j_k, j_{k+1}, \ldots, j_m]$ be an m-tuple (without repetitions) corresponding to the memory control state of the policy. The first k entries of s contain the page names that constitute T, whereas the remaining $m-k$ entries contain the names of the elements of B. Define the Markov chain $\{X_t, t=0,1,\ldots\}$ so that $X_t = \mathbf{s}$ if the memory control state at time t is given by s. Let $\mathbf{Q}=\{\mathbf{s}\}$ denote the state space of this Markov chain. From the description of s, one may conclude that Q consists of the set of all permutations of m elements chosen from n items. Therefore,

$$|\mathbf{Q}| = \binom{n}{m} m! = n!/(n-m)!.$$

The one-step transition probabilities denoted by

$$p(\mathbf{s}, \mathbf{s}') = \Pr(X_t = \mathbf{s}' \mid X_{t-1} = \mathbf{s}), \quad t \geq 1$$

can be determined easily based on the IRM parameters and on the

[1] For the class of hybrid policies, this limit always exists under the IRM.

algorithm's description. Specifically, for the $H_{FIFO-LRU}$,

$$p(s,s') = \begin{cases} \sum_{i=1}^{k}\beta_{j_i} & \text{, if } s'=s \\ \beta_{j_l} & \text{, if } s'=[j_l,j_1,j_2,\ldots,j_{l-1},j_{l+1},\ldots,j_m],\ k<l\le m \\ \beta_j & \text{, if } s'=[j,j_1,j_2,\ldots,j_{m-2},j_{m-1}],\ j\notin s \\ 0 & \text{, otherwise} \end{cases}$$

The three nonzero cases correspond to the H1, H2, and H3 events of the algorithm's description respectively. The above-defined chain is clearly homogeneous, aperiodic, and positive recurrent [Ros70a]. It can be easily shown that for $k \le n-2$, the chain is also irreducible. Thus, the limiting state occupancy probabilities, π, exist and satisfy

$$\pi = \pi \cdot \mathbf{P}, \qquad (4.3.1.1.1)$$

where $\mathbf{P}=[p(s,s')]$ is the one-step transition matrix. The limiting state occupancy probabilities, π, which are the eigenvalues of $\mathbf{P}$, have also to satisfy the normalization condition $\sum_{s\in Q}\pi_s = 1$. For a particular state $s=[j_1,j_2,j_3,\ldots,j_m]$, the matrix equation (4.3.1.1.1) can be written as

$$\pi_s = \pi_s\sum_{i=1}^{k}\beta_{j_i}+\beta_{j_1}[\sum_{j\notin s}\pi_{u_j}+\sum_{k<l\le m}\pi_{v_l}] \qquad (4.3.1.1.2)$$

where $u_j=[j_2,j_3,\ldots,j_m,j]$ and $v_l=[j_2,j_3,\ldots,j_l,j_1,j_{l+1},\ldots,j_m]$. Note that states u_j and v_l have been constructed so that a reference to page j_1 causes the algorithm to make a transition to state s.

Lemma 4.3.1.1.1: For the $H_{FIFO-LRU}$ policy with parameter k, the equilibrium probability of state $s=[j_1,j_2,j_3,\ldots,j_m]$ is given by

$$\pi_s = \frac{\prod_{i=1}^{m}\beta_{j_i}}{G(k)\prod_{i=2}^{m-k+1}D_i(s)}$$

where

$$G(k) = \frac{(n-m)!}{(n-k)!}\sum_{s\in Q}\prod_{i=1}^{k}\beta_{j_i} \quad \text{and} \quad D_i(s) = 1-\sum_{l=1}^{m-i+1}\beta_{j_l}.$$

Proof: It suffices to show that the proposed solution satisfies equation (4.3.1.1.2). First, note that, from its definition, the $D_i(\cdot)$ function for the states u_j and v_l can be written as

$$D_i(u_j) = 1-\sum_{l=2}^{m-i+2}\beta_{j_l} = D_{i-1}(s)+\beta_{j_1}, \quad i>1 \qquad (4.3.1.1.3)$$

and

$$D_i(v_l) = \begin{cases} D_i(s) & ,\ i\le m-l+1 \\ D_{i-1}(s)+\beta_{j_1} & ,\ i>m-l+1 \end{cases} \qquad (4.3.1.1.4)$$

In terms of $D_i(s)$, equation (4.3.1.1.2) becomes

$$\pi_s = \pi_s(1-D_{m-k+1}(s))+\beta_{j_1}\left[\sum_{j\notin s}\pi_{u_j}+\sum_{k<l\le m}\pi_{v_l}\right].$$

Substituting the proposed solution into the above equation, we have

$$\pi_s D_{m-k+1}(s) = \beta_{j_1}\left[\sum_{j\notin s}\frac{\beta_j\prod_{i=2}^{m}\beta_{j_i}}{G(k)\prod_{i=2}^{m-k+1}D_i(u_j)}+\sum_{k<l\le m}\frac{\prod_{i=1}^{m}\beta_{j_i}}{G(k)\prod_{i=2}^{m-k+1}D_i(v_l)}\right].$$

Using equations (4.3.1.1.3) and (4.3.1.1.4) for $D_i(u_j)$ and $D_i(v_l)$, respectively, and simplifying, we obtain

$$\pi_s \frac{G(k) D_{m-k+1}(s)}{\prod_{i=1}^{m} \beta_{j_i}} = \frac{D_i(s)}{\prod_{i=1}^{m-k} (D_i(s)+\beta_{j_1})} +$$

$$\sum_{k<l\le m} \frac{\beta_{j_1}}{\prod_{i=2}^{m-l+1} D_i(s) \prod_{i=m-l+1}^{m-k} (D_i(s)+\beta_{j_1})} . \qquad (4.3.1.1.5)$$

where we have used the fact that $\sum_{j\notin s} \beta_j = D_1(s)$.

Putting the right hand side terms over a common denominator and applying the transformations $y=m-k$ and $z=l-k$ to the indices results in

$$\pi_s \frac{G(k) \prod_{i=2}^{m-k+1} D_i(s)}{\prod_{i=1}^{m} \beta_{j_i}} =$$

$$\frac{\prod_{i=1}^{y} D_i(s) + \beta_{j_1} \cdot \sum_{1\le z\le y} \left[\prod_{i=y-z+2}^{y} D_i(s) \cdot \prod_{i=1}^{y-z} (D_i(s)+\beta_{j_1}) \right]}{\prod_{i=1}^{y} (D_i(s)+\beta_{j_1})} . \qquad (4.3.1.1.6)$$

Through the identity

$$\prod_{i=1}^{a} (Z_i+\lambda) = \prod_{i=1}^{a} Z_i + \lambda \cdot \sum_{1\le b\le a} \left[\prod_{i=1}^{a-b} (Z_i+\lambda) \cdot \prod_{i=a-b+2}^{a} Z_i \right].$$

the right hand side of equation (4.3.1.1.6) reduces to unity and finally we find that

$$\pi_s = \frac{\prod_{i=1}^{m} \beta_{j_i}}{G(k) \prod_{i=2}^{m-k+1} D_i(s)} = \pi_s$$

as claimed.

■

That the normalizing condition $\sum_{s\in Q} \pi_s = 1$ is satisfied can be shown by an aggregation argument where first π_s is summed over the $(n-k)!/(n-m)!$

states that have the same **T** (the first k elements of s) and then these aggregates are summed to cover the entire state space Q. The second sum can be shown to be the normalization condition for the equilibrium probabilities of the memory control states for a k-page memory managed by the pure FIFO policy.

Having the above lemma at hand, the following theorem can easily be proved:

Theorem 4.3.1.1.1: The steady-state fault rate generated by the $H_{FIFO-LRU}$ policy with parameter k and operating in a memory of m page frames is given by

$$F(H_{FIFO-LRU}(k)) = \sum_{s\in Q} G^{-1}(k) D_1^2(s) \frac{\prod_{i=1}^{m} \beta_{j_i}}{\prod_{i=1}^{m-k+1} D_i(s)}$$

where $G(k)$ and $D_i(s)$ are as defined above.

Proof: Note that, given the current memory state s, the probability of a page fault is simply $1-\sum_{i=1}^{m} \beta_{j_i}$ or $D_1(s)$. Thus, conditioning on the state s, we can write

$$\Pr(\text{page fault}) = \sum_{s\in Q} \Pr(\text{page fault} \mid X_t = s) \cdot \Pr(X_t = s).$$

In steady state, we can replace the above probabilities with their limiting values and obtain

$$F(H_{FIFO-LRU}(k)) = \sum_{s\in Q} (1-\sum_{i=1}^{m} \beta_{j_i}) \cdot \pi_s$$

$$= \sum_{s\in Q}\left[G^{-1}(k)D_1^2(s)\,\frac{\prod_{i=1}^{m}\beta_{j_i}}{\prod_{i=1}^{m-k+1}D_i(s)}\right] \qquad (4.3.1.1.7)$$

as desired.

■

Corollary 4.3.1.1.1: The steady-state fault rate for the pure LRU policy is given by

$$F(LRU) = \sum_{s\in Q}\left[D_1^2(s)\frac{\prod_{i=1}^{m}\beta_{j_i}}{\prod_{i=1}^{m}D_i(s)}\right].$$

Proof: For $k=1$, the $H_{FIFO-LRU}$ policy replaces the same page that the pure LRU policy replaces. Thus, $F(H_{FIFO-LRU}(1))=F(LRU)$. Substituting $k=1$ into equation (4.3.1.1.7) we immediately have

$$F(LRU) = \sum_{s\in Q}\left[D_1^2(s)\frac{\prod_{i=1}^{m}\beta_{j_i}}{\prod_{i=1}^{m}D_i(s)}\right]$$

since $G(1)=\frac{(n-m)!}{(n-1)!}\sum_{s\in Q}\beta_{j_1} = 1.$

■

Corollary 4.3.1.1.2: The steady-state fault rate for the pure FIFO policy is given by

$$F(FIFO) = \sum_{s\in Q}\left[G^{-1}(m)D_1(s)\prod_{i=1}^{m}\beta_{j_i}\right].$$

Proof: For $k=m$ (i.e., $B=\phi$), the $H_{FIFO-LRU}$ policy degenerates into the pure FIFO policy. Therefore, $F(H_{FIFO-LRU}(m))=F(FIFO)$. The result follows trivially upon substituting $k=m$ into equation (4.3.1.1.7).

■

4.3.1.2. The RAND-LRU Hybrid Policy

In this section, we consider the simple variant of the $H_{FIFO-LRU}$ policy where the page to be moved from T to B at the time of a replacement is selected at random, uniformly over the pages that currently constitute T. More precisely, the algorithm is identical to the $H_{FIFO-LRU}$ policy except that the top-to-bottom replacement is performed according to the RAND policy.

We proceed with the analysis after a formulation identical to that in the previous section. For this policy, however, the one-step transition probabilities are

$$p(s,s') = \begin{cases} \sum_{i=1}^{k}\beta_{j_i}\,, & \text{if } s'=s \\ \beta_j/k\,, & \text{if } s'=[j\ j_1 j_2\cdots j_{h-1}j_{h+1}\cdots j_k j_h j_{k+1}\cdots j_{m-1}] \\ & \text{where } j\not\in s \text{ and } 1\le h\le k \\ \beta_l/k\,, & \text{if } s'=[j_l j_1 j_2\cdots j_{h-1}j_{h+1}\cdots j_k j_h j_{k+1}\cdots j_{l-1}j_{l+1}\cdots j_m] \\ & \text{where } 1\le h\le k \text{ and } k<l\le m \\ 0\,, & \text{otherwise.} \end{cases}$$

Note that all pages in the top are eligible for replacement with the same probability $\frac{1}{k}$. It can easily be demonstrated that the resulting Markov chain is ergodic (for all k), and therefore π exists and is a solution to

$$\pi = \pi\cdot\mathbf{P}, \qquad (4.3.1.2.1)$$

where again $\mathbf{P}=[p(s,s')]$ is the one-step transition probability matrix. For state $s=[j_1,j_2,j_3,\ldots,j_m]$, equation (4.3.1.2.1) can be written as:

$$\pi_s = \pi_s \sum_{i=1}^{k} \beta_{j_i} + \frac{\beta_{j_1}}{k} \left[\sum_{j \notin s} \pi_{u_j} + \sum_{h=1}^{k} \sum_{k<l \leq m} \pi_{v_l} \right] \qquad (4.3.1.2.2)$$

where $\mathbf{u}_j = [j_2, j_3, \ldots, j_h, j_{k+1}, \ldots, j_k, j_{k+2}, \ldots, j_m, j]$ and $\mathbf{v}_l = [j_2, j_3, \ldots, j_h, j_{k+1}, j_{h+1}, \ldots, j_k, j_{k+2}, \ldots, j_l, j_1, j_{l+1}, \ldots, j_m]$ again, constructed to result in state $\mathbf{s}$ upon the referencing of page j_1.

Lemma 4.3.1.2.1: For the $H_{RAND\text{-}LRU}$ policy with parameter k, the equilibrium probability of $\mathbf{s} = [j_1, j_2, j_3, \ldots, j_m]$ is given by

$$\pi_s = \frac{\prod_{i=1}^{m} \beta_{j_i}}{G(k) \prod_{i=2}^{m-k+1} D_i(\mathbf{s})}$$

where $G(k)$ and $D_i(\mathbf{s})$ are as defined in Lemma (4.3.1.1.1).

Proof: To verify that the proposed solution (which, by the way, is identical to the solution of the $H_{FIFO\text{-}LRU}$ equilibrium equation) satisfies equation (4.3.1.2.2), we proceed as before and derive expressions for $D_i(\mathbf{u}_j)$ and $D_i(\mathbf{v}_l)$ in terms of $D_i(\mathbf{s})$. Using the definitions, we obtain

$$D_i(\mathbf{u}_j) = \begin{cases} D_{i-1}(\mathbf{s}) + \beta_{j_1} & , \ 1 < i \leq m-k+1 \\ D_i(\mathbf{s}) + \beta_{j_1} - \beta_{j_{k+1}} & , \ m-k+1 < i \leq m-h+1 \\ D_{i-1}(\mathbf{s}) + \beta_{j_1} & , \ i > m-h+1 \end{cases} \qquad (4.3.1.2.3)$$

and

$$D_i(\mathbf{v}_l) = \begin{cases} D_i(\mathbf{s}) & , \ i \leq m-l+1 \\ D_{i-1}(\mathbf{s}) + \beta_{j_1} & , \ m-l+1 < i \leq m-k+1 \\ D_i(\mathbf{s}) + \beta_{j_1} - \beta_{j_{k+1}} & , \ m-k+1 < i < -m-h+1 \\ D_{i-1}(\mathbf{s}) + \beta_{j_1} & , \ i > m-h+1 \ . \end{cases} \qquad (4.3.1.2.4)$$

Substituting the proposed solution into equation (4.3.1.2.2) yields

$$\pi_s \frac{G(k) D_{m-k+1}(\mathbf{s})}{\prod_{i=1}^{m} \beta_{j_i}} = \frac{1}{k} \sum_{h=1}^{k} \left[\sum_{j \notin s} \frac{\beta_j}{\prod_{i=2}^{m-k+1} D_i(\mathbf{u}_j)} + \sum_{k<l \leq m} \frac{\beta_{j_1}}{\prod_{i=2}^{m-k+1} D_i(\mathbf{v}_l)} \right] .$$

We proceed by using equations (4.3.1.2.3) and (4.3.1.2.4) for $D_i(\mathbf{u}_j)$ and $D_i(\mathbf{v}_l)$ in the above equation:

$$\pi_s \frac{G(k) D_{m-k+1}(\mathbf{s})}{\prod_{i=1}^{m} \beta_{j_i}} = \frac{1}{k} \sum_{h=1}^{k} \left[\sum_{j \notin s} \frac{\beta_j}{\prod_{i=2}^{m-k+1} (D_{i-1}(\mathbf{s}) + \beta_{j_1})} + \right.$$

$$\left. + \sum_{k<l \leq m} \frac{\beta_{j_1}}{\prod_{i=2}^{m-l+1} D_i(\mathbf{s}) \prod_{i=m-l+2}^{m-k+1} (D_{i-1}(\mathbf{s}) + \beta_{j_1})} \right]$$

$$= \frac{1}{k} k \sum_{j \notin s} \frac{\beta_j}{\prod_{i=1}^{m-k} (D_i(\mathbf{s}) + \beta_{j_1})} + \sum_{k<l \leq m} \frac{\beta_{j_1}}{\prod_{i=2}^{m-l+1} D_i(\mathbf{s}) \prod (D_i(\mathbf{s}) + \beta_{j_1})}$$

Notice that the above equation is identical to equation (4.3.1.1.5), which was obtained during the proof of Lemma 4.3.1.1.1. Thus, we conclude that the proposed π_s indeed satisfies equation (4.3.1.2.2).

■

Theorem 4.3.1.2.1: The steady-state fault rate for the $H_{RAND\text{-}LRU}$ policy with parameter k is equal to that of the $H_{FIFO\text{-}LRU}$ policy, i.e.,

$$F(H_{RAND\text{-}LRU}(k)) = F(H_{FIFO\text{-}LRU}(k))$$

Proof: The proof trivially follows from lemma (4.3.1.2.1) by conditioning on the states of the Markov chain.

■

Corollary 4.3.1.2.1: For programs whose behavior is perfectly represented by the IRM, the FIFO and RAND policies result in identical steady-state fault rates. That is,

$$F(FIFO) = F(RAND) \quad \text{under IRM.}$$

Proof: The proof trivially follows from Theorem (4.3.1.2.1) upon observing that, for $k=m$, the $H_{RAND-LRU}$ policy degenerates into the pure RAND policy and the $H_{FIFO-LRU}$ policy degenerates into the pure FIFO policy.

∎

Note that the above result has been obtained through a different method by Gelenbe [Gel73a].

4.3.2. Variable Partition Hybrids

In this section we consider hybrid policies that use WS management for B, thus resulting in variable size partitions. Note, however, that the partition size can never become less than k pages (the size of T), where k is a static parameter of the policy. Recall that the pure WS algorithm with parameter τ retains a page in memory only if it has been referenced at least once during the previous τ time units [Den68a]. In our case, since we have no information about references to a page during its membership in T, we must somehow estimate the last time it was referenced when it leaves T. We will consider two different estimates for this information in our analysis.

4.3.2.1. The FIFO-WS Hybrid Policy

Given the page reference r_t at instant t, define the following terms:

(i) An *event* is said to occur if $r_t \notin$ T.

(ii) A *page fault* is said to occur if $r_t \notin$ {memory}.

Obviously, all page faults are also events (since {memory}=T$\cup$B). Furthermore, with each page $i \in$ B, associate a time t_i that is an estimate of the time of last reference to that page. We can now describe the operation of the $H_{FIFO-WS}$ policy with parameters k and τ.

W1: (Null case) If $r_t \in$ T, no action is taken (actually, no action *can* be taken).

W2: (Event or fault) If $r_t \notin$ T, then T$\leftarrow$T$+r_t-i$, where page i is the FIFO page in T. As before, "+" and "-" denote set operations. We provide our estimate of the time of last reference to page i as the current time: $t_i \leftarrow t$. Note that this is actually one instant when we know page i cannot have been referenced since an event occurred and page i was in T. We comment on this choice of last reference estimates in the following sections. We update B by removing all pages with last reference time estimates earlier than the policy parameter τ: B$\leftarrow$B$+i-$J, where J$=\{j \in$ B$; t_j \le t-\tau\}$.

For the variable partition hybrid policies, we are interested in obtaining expressions for not only the steady-state fault rate, $F(A)$, but also the mean memory occupancy, $M(A)$, which is the expected number of pages that are in memory in steady state.

We proceed with the analysis after the following definitions and preliminaries. Let I_u be a random variable such that

$$I_i = \begin{cases} 1, & \text{if page i is in memory} \\ 0, & \text{otherwise} \end{cases}.$$

Let the random variable ψ_i denote the number of events that have occurred since page i last left T. As before, r_t represents the page name corresponding to the current reference. Note that, if T is examined in isolation, references that cause events will always correspond to page faults in a memory of

k page frames that is managed by the pure FIFO policy. Consider a Markov chain formulation of the FIFO policy completely analogous to our formulation in section 4.3.1.1 [Kin71a, Cof73a]. Let $\mathbf{Q}$ be the state space consisting of the states $s=[j_1,j_2,j_3,\ldots,j_k]$, representing all combinations of n items taken k at a time (note that, for the FIFO and RAND analysis, knowledge of the ordering amongst the combinations is not necessary). For a given page i, partition the state space into two disjoint sets $\mathbf{R}_i$ and $\mathbf{P}_i$ such that $\mathbf{R}_i$ contains all the states that include page i and $\mathbf{P}_i$ all other states. In other words, let $\mathbf{R}_i \subset \mathbf{Q}$ be such that $s \in \mathbf{R}_i$ iff $i \in s$. Note that $\mathbf{R}_i \cup \mathbf{P}_i = \mathbf{Q}$ and $\mathbf{R}_i \cap \mathbf{P}_i = \phi$. Let π_i denote the steady-state probability that page i is an element of $\mathbf{T}$. Then, from our pure FIFO analysis, the steady-state occupancy probabilities for the states of a memory of k page frames are known. The desired expression is obtained simply by summing the limiting state occupancy probabilities over all those states that contain page i. That is,

$$\pi_i = \Pr(i \in \mathbf{T}) = \sum_{s \in \mathbf{R}_i} \pi_s = G^{-1}(k) \sum_{s \in \mathbf{R}_i} \prod_{j \in s} \beta_j .$$

Let p denote the steady-state probability of an event given that page i is not in the top. That is,

$$p = \lim_{t \to \infty} \Pr(r_t \notin \mathbf{T} \mid i \notin \mathbf{T}).$$

Again from our Markov chain analysis of the pure FIFO policy, we obtain

$$p = \lim_{t \to \infty} \frac{\Pr(r_t \notin \mathbf{T}, i \notin \mathbf{T})}{\Pr(i \notin \mathbf{T})} = \frac{\sum_{s \in \mathbf{P}_i} \left[(1-\sum_{j \in s} \beta_j) \prod_{j \in s} \beta_j \right]}{G(k)\,(1-\pi_i)} .$$

Similarly, let q denote the steady-state probability of an event given that page i is in the top. Through an argument analogous to the one used in the derivation of p:

$$q = \lim_{t \to \infty} \frac{\Pr(r_t \notin \mathbf{T}, i \in \mathbf{T})}{\Pr(i \in \mathbf{T})} = \frac{\sum_{s \in \mathbf{R}_i} \left[(1-\sum_{j \in s} \beta_j) \prod_{j \in s} \beta_j \right]}{G(k)\,\pi_i} .$$

Let Γ_i be the steady-state probability that page i is referenced given that an event has occurred and page i is not in $\mathbf{T}$. An expression for Γ_i is derived as follows:

$$\Gamma_i = \lim_{t \to \infty} \Pr(r_t = i \mid r_t \notin \mathbf{T}, i \notin \mathbf{T}).$$

but

$$\begin{aligned}
\Pr(r_t = i \mid r_t \notin \mathbf{T}, i \notin \mathbf{T}) &= \frac{\Pr(r_t = i, r_t \notin \mathbf{T}, i \notin \mathbf{T})}{\Pr(r_t \notin \mathbf{T}, i \notin \mathbf{T})} \\
&= \frac{\Pr(r_t = i, i \notin \mathbf{T})}{\Pr(r_t \notin \mathbf{T}, i \notin \mathbf{T})} \\
&= \frac{\Pr(r_t = i \mid i \notin \mathbf{T}) \cdot \Pr(i \notin \mathbf{T})}{\Pr(r_t \notin \mathbf{T} \mid i \notin \mathbf{T}) \cdot \Pr(i \notin \mathbf{T})} \\
&= \frac{\beta_i}{\Pr(r_t \notin \mathbf{T} \mid i \notin \mathbf{T})} \qquad (4.3.2.1.1)
\end{aligned}$$

since, by the IRM assumption, pages are referenced independently of their position in memory. Now consider $\Pr(r_t \notin \mathbf{T} \mid i \notin \mathbf{T})$. This can be obtained by conditioning on the states of our Markov chain formulation as

$$\begin{aligned}
\lim_{t \to \infty} \Pr(r_t \notin \mathbf{T} \mid i \notin \mathbf{T}) &= \sum_{s \in \mathbf{Q}} \Pr(r_t \notin \mathbf{T} \mid i \notin \mathbf{T}, s) \cdot \Pr(s \mid i \notin \mathbf{T}) \\
&= \sum_{s \in \mathbf{P}_i} D_i(s) \frac{\pi_s}{(1-\pi_i)} . \qquad (4.3.2.1.2)
\end{aligned}$$

Finally, substituting equation (4.3.2.1.2) into (4.3.2.1.1) and invoking the definition of Γ_i:

$$\Gamma_i = (1-\pi_i)\beta_i / \sum_{s \in \mathbf{P}_i} D_i(s)\pi_s = \frac{\beta_i \sum_{s \in \mathbf{P}_i} \prod_{j \in s} \beta_j}{\sum_{s \in \mathbf{P}_i} \left[(1-\sum_{j \in s} \beta_j) \prod_{j \in s} \beta_j \right]} .$$

Given the above background, we can prove the following:

Lemma 4.3.2.1.1: The steady-state probability that page i is in a memory that is managed by the $H_{FIFO-vs}$ policy with parameters k and τ is given by

$$Pr(I_i=1)=\pi_i+(1-\pi_i)\cdot\sum_{j=1}^{\tau}\left[\Gamma_i(1-\Gamma_i)^{j-1}\cdot\sum_{l=j}^{\tau}\binom{l-1}{l-j}p^j(1-p)^{l-j}\right] \quad (4.3.2.1.3)$$

where

$$\pi_i = Pr(i\in\mathbf{T}) = \sum_{\mathbf{s}\in\mathbf{R}_i}\pi_{\mathbf{s}}$$

$$\Gamma_i = \lim_{t\to\infty}Pr(r_t=i\,|\,i\notin\mathbf{T}),$$

and

$$p = \lim_{t\to\infty}Pr(r_t\notin\mathbf{T}\,|\,i\notin\mathbf{T}).$$

Proof: Conditioning on the position of page i in memory, we obtain

$$\begin{aligned}Pr(I_i=1) &= Pr(I_i=1\,|\,i\in\mathbf{T})\cdot Pr(i\in\mathbf{T})+Pr(I_i=1\,|\,i\notin\mathbf{T})\cdot Pr(i\notin\mathbf{T})\\ &= 1\cdot\pi_i+(1-\pi_i)\cdot Pr(I_i=1\,|\,i\notin\mathbf{T}) \quad (4.3.2.1.4)\end{aligned}$$

since pages in **T** are always in memory. What we have to do is derive an expression for the probability that a page is still in memory given that it is not in **T**. We proceed by conditioning on the number of events that have occurred since page i last left **T**:

$$Pr(I_i=1\,|\,i\notin\mathbf{T}) = \sum_{j=1}^{\infty}Pr(I_i=1\,|\,i\notin\mathbf{T},\psi_i=j)\cdot Pr(\psi_i=j\,|\,i\notin\mathbf{T})\,. \quad (4.3.2.1.5)$$

Now, based on the IRM assumption, given that page i is not in **T**, referencing it at the instant of an event constitute independent Bernoulli trials [Fel68a] with probability of success given by Γ_i. Therefore, the number of events that have occurred since page i last left the top has a geometric distribution with parameter Γ_i. That is,

$$Pr(\psi_i=j\,|\,i\notin\mathbf{T}) = \Gamma_i(1-\Gamma_i)^{j-1}\,,\quad j\geq 1. \quad (4.3.2.1.6)$$

Returning to equation (4.3.2.1.5), the probability that page i is in memory given that it is not in the top and j events have occurred since it left the top can be expressed as

$$Pr(I_i=1\,|\,i\notin\mathbf{T},\psi_i=j) = Pr(T_1+T_2+\ldots+T_j\leq\tau)$$

where the T_i's are the interevent times given that page i is not in the top and τ is the policy parameter. Recall that with probability p a reference causes an event given that $i\notin\mathbf{T}$. Thus, with respect to events, memory references constitute independent Bernoulli trials with probability of success p under the given condition. Hence, the number of references between events has a geometric distribution with parameter p. That is,

$$Pr(T_i=x) = p(1-p)^{x-1}\,,\quad x\geq 1.$$

Finally, the random variable $T_1+T_2+\ldots+T_j$, which is the sum of j independent geometric random variables, has a negative binomial distribution with parameters p and j [Fel68a]. In other words,

$$Pr(T_1+T_2+\ldots+T_j\leq x) = \sum_{l=j}^{x}\binom{l-1}{l-j}p^j(1-p)^{l-j}\,,\quad x\geq j. \quad (4.3.2.1.7)$$

Substituting equations (4.3.2.1.6) and (4.3.2.1.7) into equation (4.3.2.1.5), we obtain

$$Pr(I_i=1\,|\,i\notin\mathbf{T}) = \sum_{i=1}^{\tau}\left[\Gamma_i(1-\Gamma_i)^{j-1}\sum_{l=j}^{\tau}\binom{l-1}{l-j}p^j(1-p)^{l-j}\right] \quad (4.3.2.1.8)$$

where the first summation terminates at τ since $Pr(T_1+T_2+\ldots+T_j\leq\tau)=0$ for all $j>\tau$ due to the fact that $T_i\geq 1$. Combining equations (4.3.2.1.8) and (4.3.2.1.4), the desired result is obtained.

■

Theorem 4.3.2.1.1: The steady-state fault rate for the $H_{FIFO-WS}$ policy with parameters k and τ is given by

$$F(H_{FIFO-WS}) = 1-\sum_{i=1}^{n}\beta_i\left[\pi_i+(1-\pi_i)\cdot\sum_{j=1}^{\tau}\left[\Gamma_i(1-\Gamma_i)^{j-1}\sum_{l=j}^{\tau}\binom{l-1}{l-j}p^j(1-p)^{l-j}\right]\right].$$

where π_i, Γ_i, and p are as defined in Lemma 4.3.2.1.1.

Proof: In steady state, the probability of page i causing a fault is simply the steady-state probability that it is not in memory and it is referenced. Thus, conditioning on the page,

$$F(H_{FIFO-WS}) = \sum_{i=1}^{n}(1-\Pr(I_i=1))\cdot\beta_i.$$

The result then follows trivially upon the substitution of equation (4.3.2.1.3) into the above expression.

■

Theorem 4.3.2.1.2: The mean memory occupancy due to the $H_{FIFO-WS}$ policy with parameters k and τ is given by

$$M(H_{FIFO-WS}) = \sum_{i=1}^{n}\left[\pi_i+(1-\pi_i)\cdot\sum_{j=1}^{\tau}\left[\Gamma_i(1-\Gamma_i)^{j-1}\cdot\sum_{l=j}^{\tau}\binom{l-1}{l-j}p^j(1-p)^{l-j}\right]\right]$$

where π_i, Γ_i, and p are as defined in Lemma 4.3.2.1.1.

Proof: Conditioning on the page,

$$M(H_{FIFO-WS}) = \sum_{i=1}^{n}\Pr(I_i=1).$$

The result follows immediately from the substitution of equation (4.3.2.1.3) into the above expression.

■

Corollary 4.3.2.1.1: For $k=1$,

$$F(H_{FIFO-WS})\le F(WS) \text{ and}$$
$$M(H_{FIFO-WS})\ge M(WS).$$

Proof: (Informal) From the algorithm's description, note that the $H_{FIFO-WS}$ policy with parameters $k=1$ and τ operates identically to the pure WS policy with the same parameter τ except that repeated references to the same page cause no memory state change. Therefore, we conclude that the memory states generated by the $H_{FIFO-WS}$ policy with $k=1$ are supersets of those generated by the WS policy. This obviously results in an increased mean memory occupancy and a reduced steady-state fault rate.

■

4.3.2.2. The RAND-WS Hybrid Policy

Here we consider a variation of the $H_{FIFO-WS}$ policy that operates exactly as described in the previous section except that, at the times of events, the page to leave the top is selected at random uniformly over the pages currently in **T**.

Theorem 4.3.2.2.1: The steady-state probability that page i is in a memory managed by the $H_{RAND-WS}$ policy with parameters k and τ is identical to that of the $H_{FIFO-WS}$ policy with the same parameters. That is,

$$\Pr(I_i=1) = \pi_i+(1-\pi_i)\cdot\sum_{j=1}^{\tau}\left[\Gamma_i(1-\Gamma_i)^{j-1}\cdot\sum_{l=j}^{\tau}\binom{l-1}{l-j}p^j(1-p)^{l-j}\right].$$

where π_i, Γ_i, and p are exactly as defined in Lemma 4.3.2.1.1.

Proof: We proceed in a manner similar to the $H_{FIFO-VS}$ analysis. Conditioning on the location of the page in memory,

$$\Pr(I_i=1) = \Pr(I_i=1 \mid i \in \mathbf{T}) \cdot \Pr(i \in \mathbf{T}) + \Pr(I_i=1 \mid i \notin \mathbf{T}) \cdot \Pr(i \notin \mathbf{T}).$$

From the Markov chain analysis of the RAND policy based on IRM, we know that the equilibrium probabilities for the states are the same as those for the FIFO policy (this is the $m=k$ special case of Lemma 4.3.1.2.1). Furthermore, by Corollary 4.3.1.2.1, the steady-state fault rates for the RAND and FIFO policies are identical. Therefore,

$$\Pr(I_i=1) = \pi_i + (1-\pi_i) \cdot \Pr(I_i=1 \mid i \notin \mathbf{T})$$

as before.

Consider $\Pr(I_i=1 \mid i \notin \mathbf{T})$. From the algorithm's description, it is clear that the probability of a page remaining in memory outside the top is independent of the time it spent in the top and depends only on the interevent time distribution and the probability of the page being referenced given that it is not in the top. Since both of these properties are derived from the Markov chain analysis of the RAND policy for the top, the expressions must be identical to those of the FIFO case. Thus, the proof proceeds just as that of Lemma 4.3.2.1.1.

■

Corollary 4.3.2.2.1: The steady-state fault rates and the mean memory occupancies for the $H_{RAND-VS}$ policy are identical to those of the $H_{FIFO-VS}$ policy with the same parameters.

Proof: Trivially follows since the expressions for $\Pr(I_i=1)$ are the same for both policies.

■

4.3.2.3. Simple Variations

As we noted in section 4.3.2.1, the proposed estimate for the time of last reference to a page as it leaves **T** is clearly optimistic for the $H_{FIFO-VS}$ policy. We have no way of knowing the exact time of last reference to the page. For the $H_{RAND-VS}$ policy, however, it is much more difficult to make a statement about the accuracy of this proposed estimate -- the page may, in fact, have been last referenced one time instant before it was selected to leave **T**. The relative merits of these two cases are discussed in the next chapter.

As an alternative, we can mark a page as having been last referenced at the instant that it *entered* the top rather than when it left it. This results in a rather pessimistic estimate since the probability that the page remains unreferenced between the time of entry and exit from the top is very small. Since pages cannot be removed from memory while they reside in the top, regardless of how long they have been there, under this variation the $H_{FIFO-VS}$ and $H_{RAND-VS}$ policies will have to be modified to remove from memory a page that has been in the top longer than the window τ as soon as it exits the top. To analyze this variant of the $H_{FIFO-VS}$ policy, define the random variable ∇_i to be amount of time page i remains in the top. For the $H_{FIFO-VS}$ policy, page i remains in the top for exactly k interevent times (due to the FIFO nature of the top) which are independently and identically distributed as geometric random variables with parameter q. Hence the random variable ∇_i has a negative binomial distribution with parameters q and k. Proceeding as in the proof of lemma (4.3.2.1.1), an expression for $\Pr(I_i=1 \mid i \notin \mathbf{T})$ can be derived by conditioning first on the time spent in the top:

$$\Pr(I_i=1|i\notin T) = \sum_{x=1}^{\infty} \Pr(I_i=1|i\notin T, \nabla_i=x)\cdot \Pr(\nabla_i=x)$$

$$= \sum_{x=k}^{\tau} \Pr(I_i=1|i\notin T, \nabla_i=x)\cdot \binom{x-1}{x-k} q^k (1-q)^{x-1}$$

$$= \sum_{x=k}^{\tau}\left[\binom{x-1}{x-k} q^k (1-q)^{x-k} \sum_{j=1}^{\tau-x}\left[\Gamma_i (1-\Gamma_i)^{j-1}\cdot \sum_{l=j}^{\tau-x}\binom{l-j}{l-1} p^j (1-p)^{l-j}\right]\right].$$

The desired fault rate and mean memory occupancy can now be obtained by substituting the above expression in equation (4.3.2.1.4). The analysis of the $H_{RAND-VS}$ policy under this variation can be carried out in a manner completely analogous to the above derivation. Note that, under this modification of the estimate of the time of last reference to a page,

$$F(H_{FIFO-VS}) \neq F(H_{RAND-VS})$$

and

$$M(H_{FIFO-VS}) \neq M(H_{RAND-VS})$$

except for the degenerate case $k=1$.

Since both of the proposed estimates for the time of last reference to a page are incorrect, perhaps a more reasonable estimate is the arithmetic average of the times of entry to and exit from the top. We comment further on these alternatives in the next chapter in light of our trace-driven simulation results.

4.4. Numerical Results

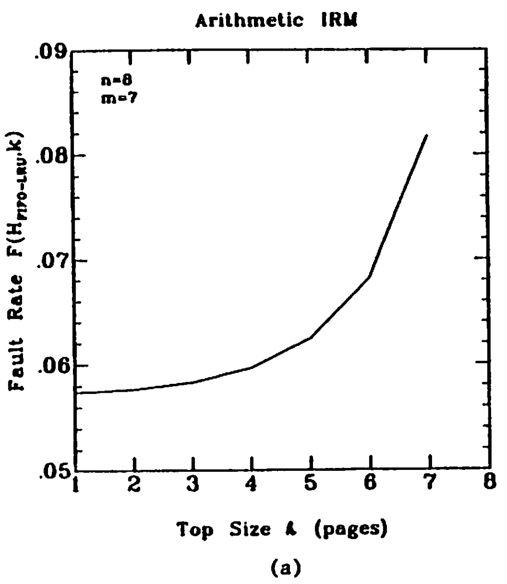
Given the closed-form expression for $F(H_{FIFO-LRU}(k))$ (recall that this is same expression for $F(H_{RAND-LRU}(k))$ in equation (4.3.1.1.6), we are interested in its functional dependency on the policy parameter k for various values of m and of the IRM parameters. However, due to the complexity of the expression, its form between the two end points $k=1$ and $k=m$ (which correspond to pure LRU and FIFO/RAND, respectively) is difficult to study analytically. We know that, for the IRM, the fault rate due to pure LRU is always less than or equal to the FIFO/RAND fault rate for all m and model parameters [Kin71a]. The shape of $F(H_{FIFO-LRU}(k))$ between these two points can be of three types:

(i) Straight line, meaning that $F(H_{FIFO-LRU}(k))$ is a linear combination of the LRU and FIFO/RAND fault rates. However, examination of the expression rules out this possibility since it cannot be written as the desired linear combination.

(ii) Concave down. We would be disappointed if this were the case since this result would contradict our desire of achieving fault rates close to LRU at costs comparable to FIFO/RAND. In other words, to keep the fault rate close to the LRU value, we would have to operate the policy with a small k resulting in a large $|\mathbf{B}|$ and thus a large cost incurred due to the reclaim events taking place from the bottom.

(iii) Concave up (convex). We would be happy since now we could operate the policy with a large parameter and still retain the low LRU fault rate as well as reducing $|\mathbf{B}|$ and thus the number of reclaim events.

To resolve this question, we resort to obtaining numerical values for the expression for various instances of the program model. To minimize the number of parameters involved, the two instances of the IRM we consider are generated through the equations $\beta_i=ci$ and $\beta_i=c^i$, which are called the *arithmetic* and the *geometric* model respectively. In both cases, the constant c is chosen such that $\sum_{i=1}^{n}\beta_i=1$. In Figure 4.4.1, $F(H_{FIFO-LRU}(k))$ is plotted as a function of k for the fixed values $n=8$ and $m=7$. Note the strong convexity of



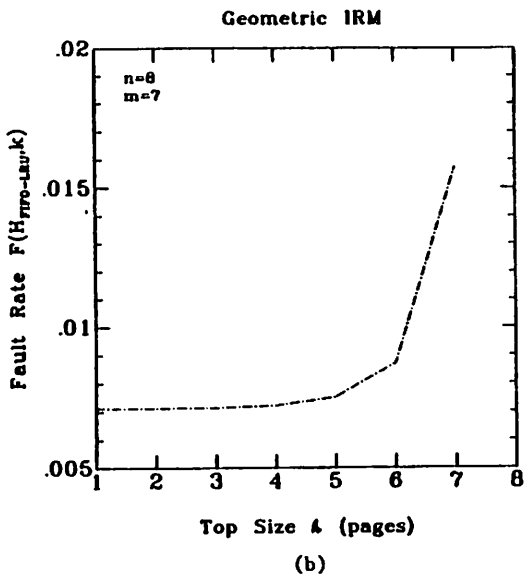


Figure 4.4.1 $H_{FIFO\text{-}LRU}$ fault rates for two sample programs as a function of the policy parameter k.

the two curves; particularly the one corresponding to the geometric IRM, where fault rates are achieved that are practically identical to the pure LRU fault rate even for top sizes of 5 pages (equivalently, bottom size of 2 pages). Note that, although not indicated in Figure 4.4.1, for the case where $\beta_i = 1/n$ (uniform distribution of the IRM parameters), *all* demand algorithms result in the same fault rate of $(n-m)/n$. For this case, $F(H_{FIFO\text{-}LRU}(k))$ is obviously a constant, i.e., it does not depend on k. Outside of this degenerate case, the strict convexity of $F(H_{FIFO\text{-}LRU}(k))$ as a function k for all m and IRM parameter values remains a conjecture. Next, compare the performances of the hybrid policies as a function of the mean memory occupancy for a fixed value of the parameter k and two instances of the IRM (again, the arithmetic and geometric models). In Figure 4.4.2 we have also included the pure FIFO (or RAND), LRU, WS, and A_0 results for comparison (informally, the A_0 policy replaces the page with the smallest probability of reference and has been shown to be the optimal policy for the IRM [Aho71a]). Note that, for the two hybrids, $H_{FIFO\text{-}LRU}$ and $H_{FIFO\text{-}WS}$ with parameters $k=4$, the fault rates rapidly approach those of LRU and WS, respectively, for memory sizes greater than the parameter value. For these cases, the relative fault rates due to $H_{FIFO\text{-}LRU}$ and $H_{FIFO\text{-}WS}$ for equal k are of the same magnitude as those of the pure LRU and WS.

The above results have to be interpreted with caution for two reasons.

(i) They are based on a model of program behavior that is known to lack many of the properties of real programs,

(ii) The numerical results presented are for unrealistically small values of the program size, n, and of the memory size, m, due to the factorial growth of the complexity of the expressions involved.

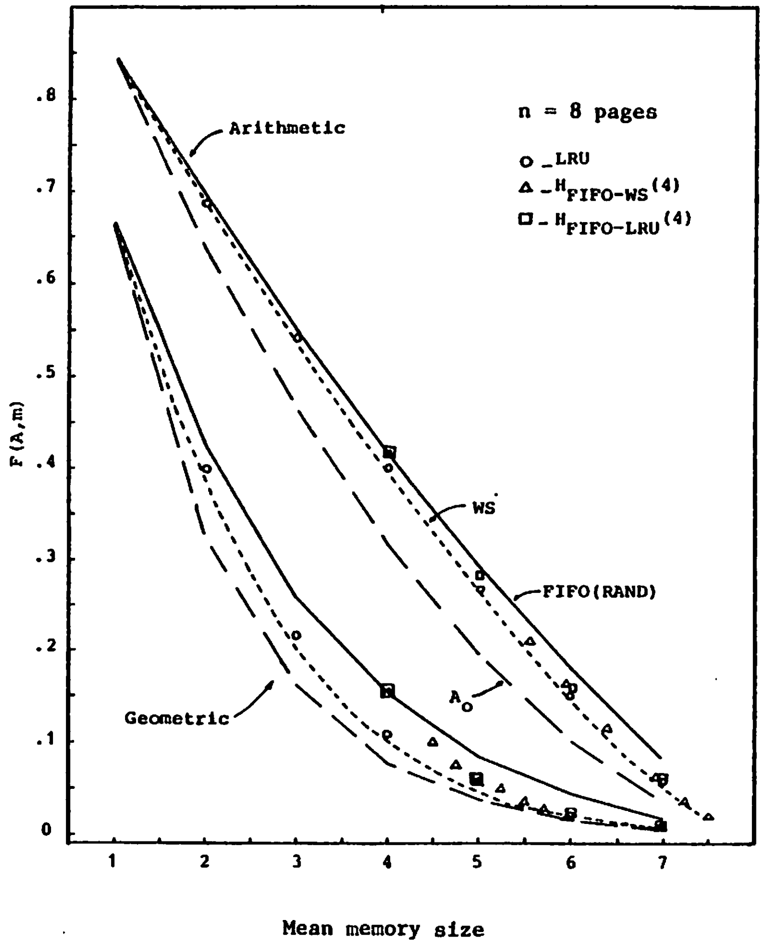


Figure 4.4.2 Fault rates for two sample programs under various replacement policies as a function of the memory size m.

This second limitation can be overcome partially through a numerical technique proposed by Fagin and Price [Fag76a]. However, both of these concerns will be addressed in the next chapter where we present results derived from trace-driven simulations.

4.5. Cost Considerations

Up to this point, the only performance measure we have been concerned with has been the steady-state fault rate as a function of the mean memory occupancy. While appropriate for policies such as pure LRU and WS, for the hybrid algorithms the cost of collecting page reference information through faulting on reclaimable pages must be considered. For example, the $H_{FIFO\text{-}LRU}$ policy achieves its minimum fault rate value for $k=1$. However, with this setting of the parameter, we force a reclaim fault at each reference that is not a repeat reference to the same page. Thus, what we seek is a value for the hybrid policy parameter, k, for which there are few reclaims while the fault rate is sufficiently close to that of the LRU or WS fault rate. More precisely, we are interested in the value of k that minimizes a cost function which is a weighted sum of the steady-state fault and reclaim rates. For a particular implementation, let α represent the ratio of the mean time delays encountered by a program due to a page fault and a page reclaim. Define the cost function $C(\cdot)$ to be

$$C(H_{FIFO\text{-}LRU}(k),\alpha) = [F(FIFO(k)) - F(H_{FIFO\text{-}LRU}(k))] + \alpha \cdot F(H_{FIFO\text{-}LRU}(k))$$

where the term in the square braces corresponds to the steady-state reclaim rate (i.e., the probability of a reference to a page in B). Analogous equations for the other hybrid policies can be written as well. Note that, while appropriate as a *responsiveness* measure from the point of view of a program, the cost function $C(\cdot)$ is inappropriate for system *throughput* considerations since the major part of the delay due to a page fault results from the I/O operation which can be overlapped with other CPU activity. On the other hand, a reclaim operation is performed by the CPU and cannot be overlapped.

Due to the complexity of the expressions for the steady-state fault and reclaim rates, an analytic minimization of the cost function defined with respect to k cannot be carried out. Empirical studies of this cost have been performed based on trace-driven simulations and are reported in the next chapter.

4.6. Conclusions

We have introduced a class of hybrid algorithms that are suitable for page replacement decisions in a virtual memory environment that lacks hardware reference bits. Expressions for the steady-state fault rates generated by these policies have been derived based on the Independent Reference Model of program behavior. Numerical results suggest that these algorithms are capable of achieving fault rates close to those of the pure LRU and WS policies while incurring costs comparable to those of the FIFO and RAND policies. For example, for the $H_{FIFO-LRU}$ policy applied to an eight-page program that is an instance of the IRM with geometric model parameters operating in a memory of 7 pages, we have observed that with a reclaimable set containing as few as 2 pages, fault rates are achieved that are practically the same as that produced by the pure LRU policy.

In a multiprogramming environment, the fixed partition hybrids can be extended in a natural way to operate as what may be considered to be FIFO-

Global LRU (GLRU) and RAND-Global LRU hybrids, thus resulting in variable partitions for the individual programs. This can be accomplished simply by maintaining a single fixed size bottom that contains the reclaimable pages of all the programs that are currently being multiprogrammed (the tops for each of the programs, however, are still maintained separately). In such an environment, the total number of page frames allocated to a process at any given point in time will be the size of its top plus a random variable that represents the number of pages belonging to the given process amongst the common bottom. Note that, under this extension, the study of the performance of individual programs is severely complicated due to their interactions with the other programs running concurrently.

Although not studied analytically, we note that this extension adequately models the memory management policy employed in the VMS operating system for the VAX-11/780 computer system [DEC78a]. The implications of this extension on the selection of the optimal policy parameter will be commented on in the next chapter based on simulation studies of real programs.

4.7. References

[Aho71a] A. V. Aho, P. J. Denning, and J. D. Ullman, "Principles of Optimal Page Replacement," *J. ACM* 18 pp. 80-93 (January 1971).

[Bas76a] F. Baskett and A. Rafii, "The A_0 Inversion Model of Program Paging Behavior," Stanford U. Comp. Sci. Dept., Report STAN-CS-76-579 (October 1976).

[Chu76a] W. W. Chu and H. Opderbeck, "Program Behavior and the Page Fault Frequency Replacement Algorithm," *Computer* 9 pp. 29-38 (November 1976).

[Cof73a] E. G. Coffman and P.J. Denning, *Operating Systems Theory*, Prentice-Hall, Enlewood Cliff, 'New Jersey (1973).

[Cor68a] F. J. Corbato, "A Paging Experiment with the Multics System," Project MAC Memo MAC-M-384 Mass. Inst. of Tech. (July 1968). Published in In Honor of P. M. Morse ed. Ingard MIT Press 1969, pp. 217-228

[DEC78a] DEC, *VAX-11/780 Software Handbook*, Digital Press (1978).

[Den68a] P. J. Denning, "The Working Set Model of Program Behavior," *Comm. ACM* 11(5) pp. 323-333 (May 1968).

[Eas79a] M. Easton and P. A. Franaszek, "Use Bit Scanning in Replacement Decisions," *IEEE Trans. Comptrs.* C-28 pp. 133-141 (February 1979).

[Fag76a] R. Fagin and T. G. Price, "Efficient Calculation of Expected Miss Ratios in the Independent Reference Model," IBM Research Report RJ-1849 (October 1976).

[Fel68a] W. Feller, *Introduction to Probability and its Applications*, Wiley (1968). 3rd. ed., vol. 1

[Fog74a] M. H. Fogel, "The VMOS Paging Algorithm - A Practical Implementation of a Working Set Model," *Operating Systs. Rev.* 8 pp. 8-17 (January 1974).

[Gel73a] E. Gelenbe, "A Unified Approach to the Evaluation of a Class of Replacement Algorithms," *IEEE Trans. Comptrs.* C-22 pp. 611-618 (June 1973).

[Kin71a] W. F. King, "Analysis of Demand Paging Algorithms," *Proc. IFIPS Congress*, pp. TA-3-155 - TA-3-159 , Ljubljana, Yugoslavia(1971).

[Lip68a] J. S. Liptay, "The Cache," *IBM Syst. J.* 7 pp. 15-21 (1968).

[Mat70a] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Syst. J.* 9 pp. 78-117 (1970).

[Mor72a] J. P. Morris, "Demand Paging Through Utilization of Working Sets on the MANIAC II," *Comm . ACM* 15 pp. 867-872 (October 1972).

[Oli74a] N. A. Oliver, "Experimental Data on Page Replacement Algorithm," *Proc. NCC*, pp. 179-184 (1974).

[Pri74a] B. G. Prieve, "A Page Partition Replacement Algorithm," Ph.D. Th., Univ. of California Berkeley, California (1974).

[Raf76a] A. Rafii, "Empirical and Analytical Studies of Program Reference Behavior," Ph.D. Th., SLAC Report 197, Stanford California (1976).

[Ros70a] S. M. Ross, *Applied Probability Models with Optimization Applications*, Holden-Day, San Francisco (1970).

CHAPTER 5

HYBRID PAGE REPLACEMENT POLICIES - EMPIRICAL STUDIES

5.1. Introduction

In the previous chapter we presented results about the performance of the hybrid policies based on analytic methods. The utility of these results is questionable for two reasons:

(i) They are based on an over-simplified model of program behavior— the IRM. This model specifically excludes the possibility of locality, known to be exhibited by real programs and to be a fundamental factor of virtual storage performance.

(ii) The numerical examples presented were obtained from unrealistically small programs and memory sizes due to the combinatorial explosion of the expressions for larger values of these sizes.

To resolve these issues in a clear-cut manner, we now turn to the study of these hybrid policies (and several others) using trace-driven simulation.

As introduced in Chapter 2, trace-driven simulation mimics the operation of a system as it would behave in response to input data that is recorded in the trace. For our purposes, the trace data consists of an address record for each memory access (both data and instruction) generated by a program during an interval of execution. Since the data originated from the execution of a real program, conclusions based on studies using this data do not require assumptions about the underlying model for the program. Furthermore, results for a range of operating conditions and memory sizes can usu-

ally be obtained with equal ease.

Our simulators simply implement the various page replacement algorithms as they would function in a uniprogramming environment; the memory references generated by the executing program are read from the address trace.

5.2. The Trace Data

The simulation studies described in this chapter are based on the trace data obtained from three programs. They were traced while running on an IBM 360/91 system at the Stanford Linear Accelerator Center. These programs represent a range of applications and behaviors and are referred to by the following names:

WATFIV The execution of the WATFIV compiler compiling a small program. The compiler size is 96 pages of 1024 bytes each; the trace length is 1048662 references.

APL A plotting program running under an APL interpreter. The interpreter size is 114 pages of 512 bytes each; the trace length is 2870921 references.

FFT An implementation of the Fast Fourier Transform. The program size is 82 pages of 512 bytes each; the trace length is 2954787 references.

More data about the source and nature of these programs can be found in [Smi76a].

Some of the difficulties encountered in using memory address trace data are discussed in Chapter 2. Often, such data is voluminous since tracing one second of program execution time can easily produce over one million references. Therefore, multiple simulation runs, representing different operating environments, over this data can be very expensive.

The compressed versions of the three address trace data, generated by Lau [Lau79a], were used in all of our simulation studies. The reduction method used by Lau is a combination of the Stack Deletion and Snapshot Methods as discussed in section 2.1. The method has two parameters, k and T, and retains references to the LRU stack positions greater than k in addition to the initial references to pages within each sample interval of length T, regardless of their position in the stack.

5.3. Fixed-Size Partition Empirical Results

5.3.1. Hybrid Policies

One of the main conclusions of the previous chapter was that the $H_{FIFO-LRU}$ or $H_{RAND-LRU}$ policy can achieve fault rates almost identical to those of the pure LRU fault rate even when the bottom consists of just a few pages. This conclusion was based on the numerical evaluation of the analytic expression obtained for the fault rate for two rather small programs operating in a small memory. Figure 5.3.1.1 shows that the same conclusions are also valid for the three programs available to us. All three results display strong convexity with a well-defined knee at approximately $k=40$ pages. Note that this represents a bottom size of 10 pages or, equivalently, 20 percent of total memory. We also note that the $H_{FIFO-LRU}$ and $H_{RAND-LRU}$ policies result in different fault rates for these programs. This confirms the fact that these programs do not satisfy the IRM assumptions. No general conclusion about the relative performances of the $H_{FIFO-LRU}$ and $H_{RAND-LRU}$ policies, however, can be derived from these results as each is uniformly superior in one instance while both perform about the same in the remaining instance.

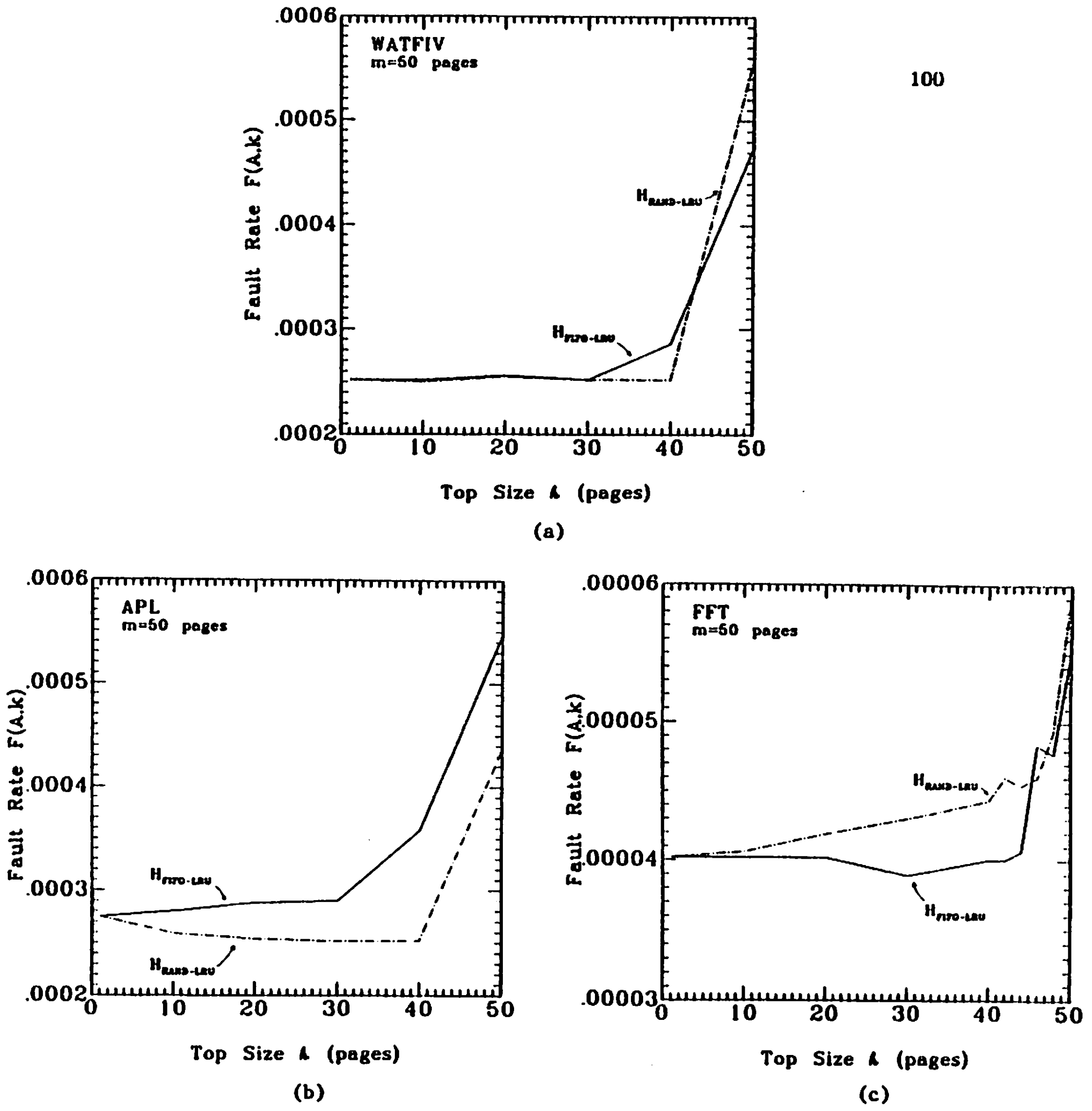


Figure 5.3.1.1 $H_{FIFO-LRU}$ and $H_{RAND-LRU}$ fault rates as a function of the policy parameter k for (a) WATFIV, (b) APL and (c) FFT.

The operation of these hybrid policies requires setting the policy parameter k to some value. If the cost of referencing a page that is in the bottom were negligible, then setting $k=1$ would almost always produce optimum performance with respect to the page fault rate (there are few points in the graphs of Figure 5.3.1.1 where the fault rate actually drops as k is increased beyond 1). However, since a finite cost is incurred each time a page in the bottom is referenced, the selection of the policy parameter should be guided by the desire to keep the fault rate close to the pure LRU value (i.e., corresponding to $k=1$) while minimizing the size of the bottom. This minimizes the rate of reclaims. Intuitively, the policy should be operated with the parameter set to a value close to the knee that occurs in all three fault rate graphs. Formally, we define a performance measure, $C(\cdot)$, that is the weighted sum of the fault rate and the reclaim rate for a given value of the policy parameter. For a page replacement algorithm A and a ratio of page fault service time to page reclaim service time given by α let

$$C(A,m,k,\alpha) = f(A,m-k)+\alpha\cdot F(A,m,k)$$

where m is the memory size, k is the policy parameter (or the top size) and $f(\cdot)$ is the reclaim rate. We will comment on the suitability of this measure for system throughput considerations in section 5.4.2.

Figure 5.3.1.2 displays this measure under the $H_{FIFO-LRU}$ and $H_{RAND-LRU}$ policies for the three programs. All evaluations of this weighted sum in this chapter have been carried out for $\alpha=100$. This is done for brevity of presentation, since all of our conclusions are also applicable to results for $\alpha=10$ and $\alpha=1000$ (i.e., spanning a range of three orders of magnitude). Furthermore, measurements from an actual implementation to be described in the next chapter are in agreement with this choice of α. With respect to this new

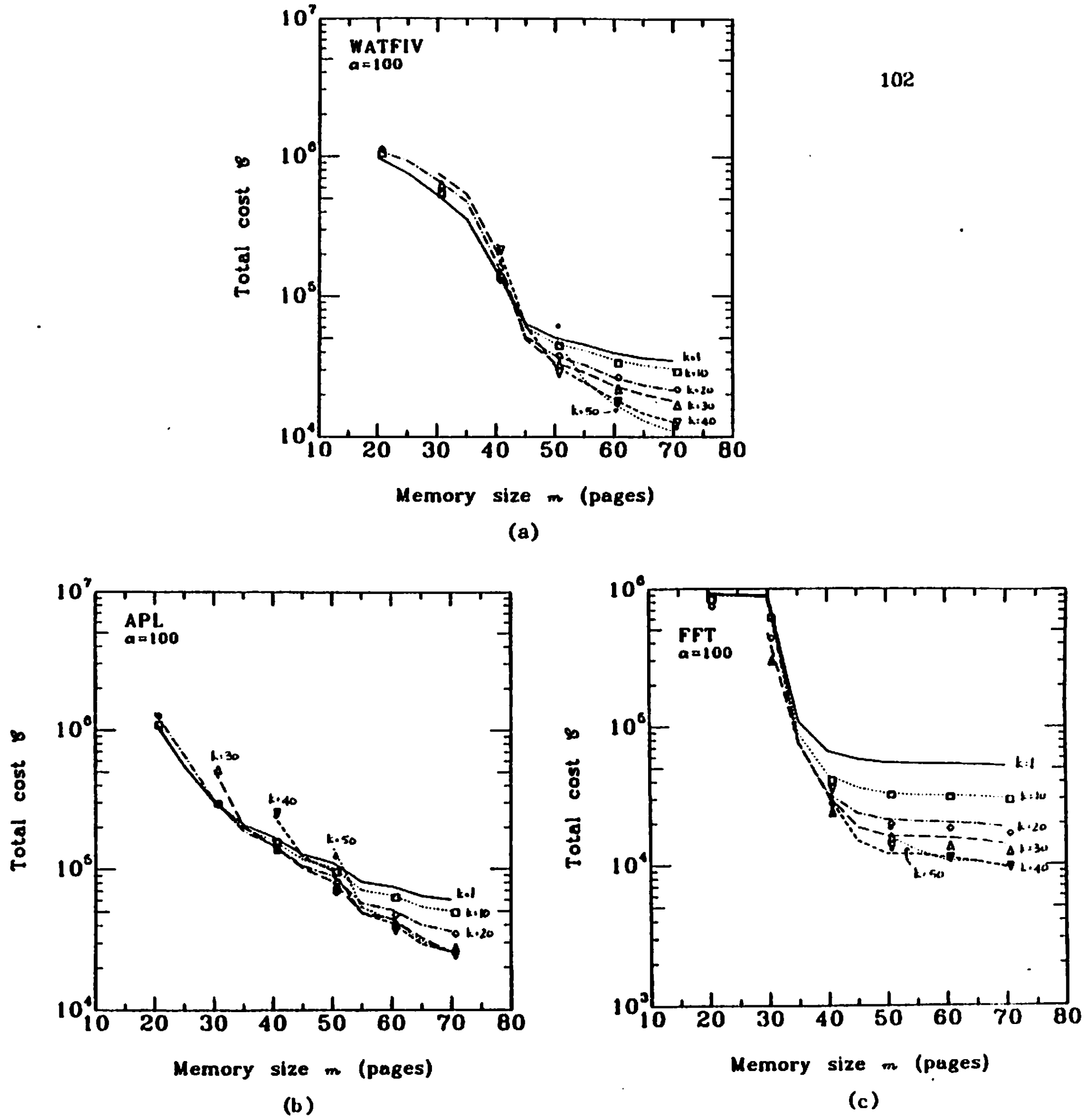


Figure 5.3.1.2 The weighted sum measure observed under $H_{FIFO-LRU}$ (lines) and $H_{RAND-LRU}$ (points) for various policy parameter as a function of the memory size for (a) WATFIV, (b) APL and (c) FFT. Note that the two policies produce identical results when $k=1$.

measure, the $H_{FIFO\text{-}LRU}$ and $H_{RAND\text{-}LRU}$ policies perform similarly. Because $H_{FIFO\text{-}LRU}$ is easier to implement than the $H_{RAND\text{-}LRU}$ policy[1] and has equal performance, it will be the only representative of the fixed-partition hybrids to be investigated in the sequel.

Given a particular program, α and the memory size m, we have seen that there usually exists a value of the policy parameter k that minimizes the $C()$ function. In an actual system, few, if any, of these values remain constant over time. As was mentioned in the previous chapter, an implementation of the $H_{FIFO\text{-}LRU}$ policy in a multiprogramming environment results in a bottom that is globally shared by all active programs. While the tops for all of these programs are local and have a fixed size, the number of pages in this global bottom associated with a particular program (i.e., the value of m for this program) varies in a complicated manner based on the activities of the other programs. Each program in the system has a different memory access behavior and would consequently require a different value of k for optimal operation even if m and α were fixed. These variations are illustrated in Figure 5.3.1.3, where the weighted sum measure is shown as a function of the policy parameter for varying memory sizes and different programs. The value of k that minimizes this function is clearly seen to be sensitive to the amount of memory available to a given program as well as being sensitive to the program itself for a fixed m. This undesirable property of the $H_{FIFO\text{-}LRU}$ policy will be further commented on in the next section.

[1] Implementing Random replacement for the top requires the generation of pseudo-random integer in the range [1...k] while implementing FIFO replacement requires simply the maintenance of a linked list of the pages in top.

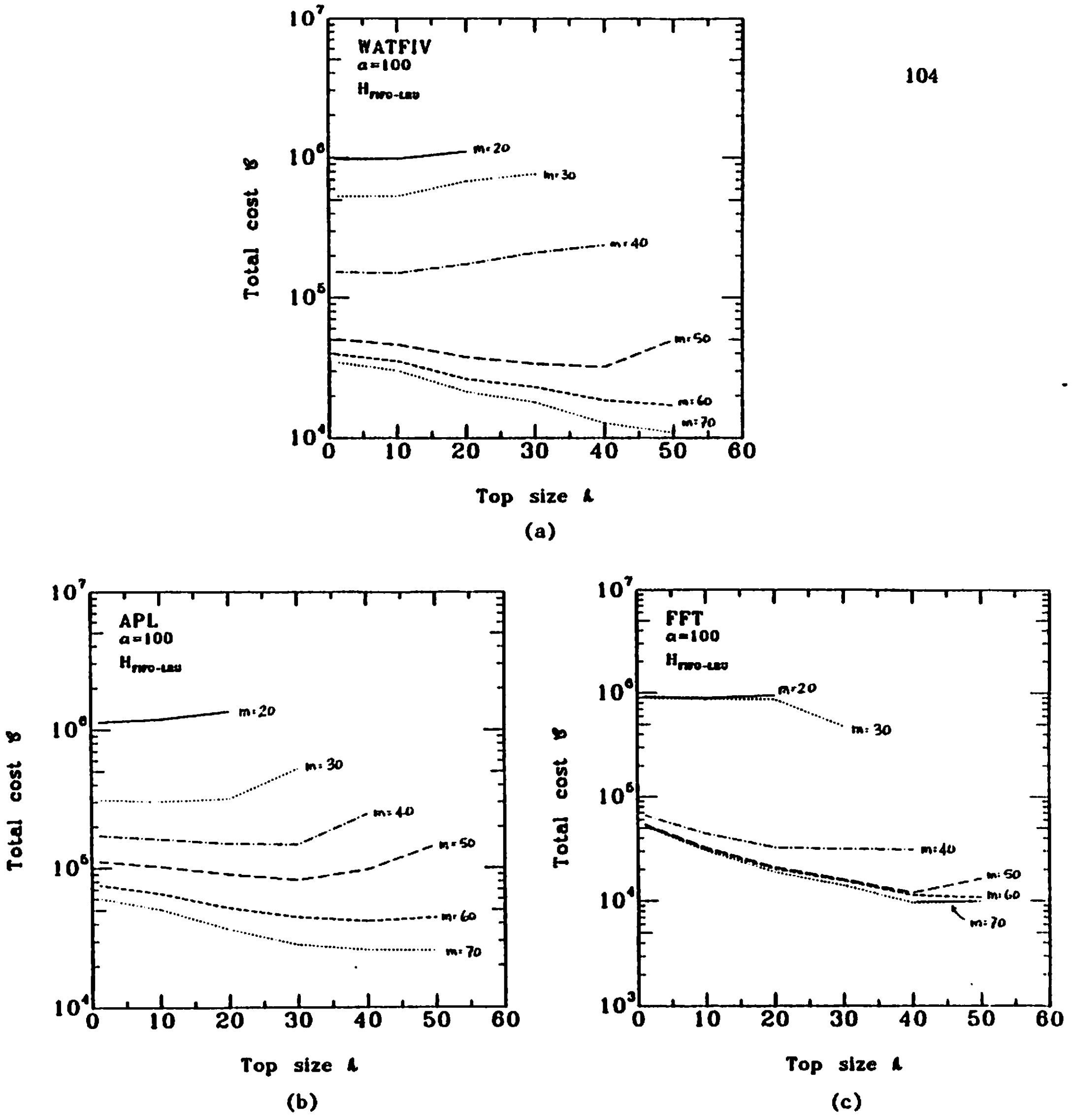
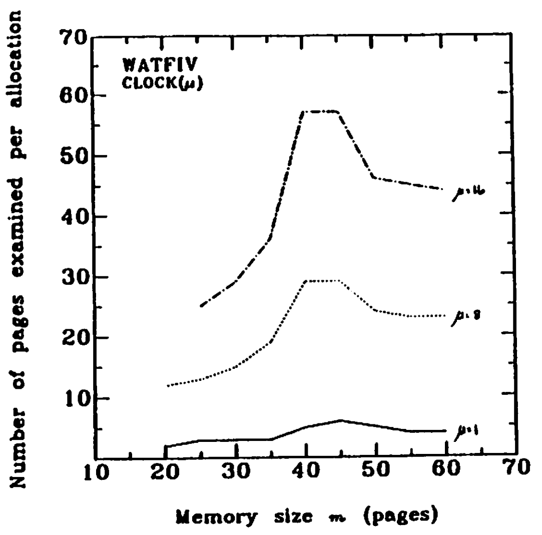


Figure 5.3.1.3 The weighted sum observed under $H_{FIFO\text{-}LRU}$ policy for varying amounts of memory as a function of the policy parameter and programs (a) WATFIV, (b) APL and (c) FFT.

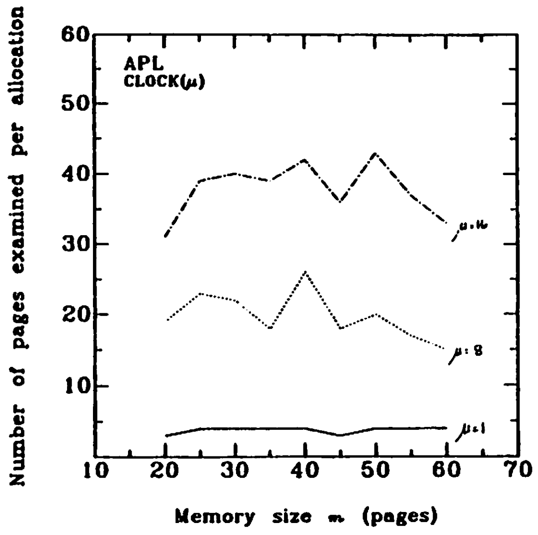
5.3.2. The Clock Page Replacement Algorithm

As an alternative to the hybrid fixed-size partition policies, we investigate the performance of the Clock page replacement algorithm with parameter μ, denoted as *clock*(μ). This algorithm was first implemented and studied as the page replacement policy for the Multics operating system [Cor68a] and is generaly regarded as a practical approximation of the LRU algorithm. In its more general form, as introduced and investigated in [Eas79a], the algorithm has a parameter μ and functions as follows. At the time of a page fault, the pages of the program are examined sequentially (modulo m, the partition size) until the first page that has not been referenced during the time interval defined by the last μ examinations is found. In other words, each page frame has a modulo μ counter associated with it that is reset to μ each time the page is referenced and decremented by one each time the page is examined by the replacement algorithm. The page is selected for replacement only if its counter contains a zero at the time of examination. The special case μ=1 is particularly simple to implement since it requires only one bit per page, usually called the *reference bit*, that is set when the page is referenced and reset when it is examined. The algorithm selects for replacement the first page encountered with the reference bit off. In our environment where there are no reference bits, we simulate them by moving a page from the reclaimable state to the valid state whenever we want to set its reference bit and vice versa to reset it.

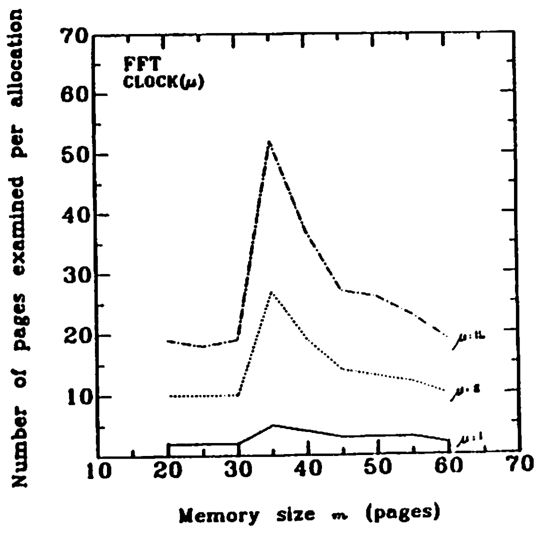
Contrary to what is suggested in [Cor68a], the performance of the clock(μ) policy does not approach that of the pure LRU asymptotically for all reference strings as μ tends to infinity (one can construct reference strings for which clock(μ) always replaces the most recently page regardless of how



(a)



(b)



(c)

Figure 5.3.2.4 Overhead associated with the clock(μ) algorithm for (a) WATFIV, (b) APL and (c) FFT.

large μ is) [Eas79a]. Empirically and analytically, increasing μ beyond 1 results in an insignificant improvement in performance but increases the cost of implementation noticeably.

Define the overhead associated with the implementation of page replacement algorithm A, denoted as $\Omega(A)$, to be equal to the work done during the selection of the page to be replaced. For the clock(μ) policy, this is reasonably measured by Ω (clock(μ)) = (mean number of pages scanned per page fault) * (number of page faults). Figure 5.3.2.4 plots this overhead normalized by the number of page faults (i.e., the mean number of pages examined per page fault) for the three programs for various values of μ. Since the page fault rates were indistinguishable for all the values of μ, the clock(1) policy is clearly superior to the others due to its uniformly low overhead. Based on this observation, we will only consider the clock(1) policy in the following and drop the parameter from our notation. Thus, "clock" stands for "clock(1)" unless otherwise noted.

The values of the weighted sum measure for the three programs under the clock and $H_{FIFO-LRU}$ policies are illustrated in Figure 5.3.2.5. As was also evident in Figure 5.3.2.3, the relative ordering of the $H_{FIFO-LRU}$ results for various values of the policy parameter changes as the memory size and the program vary. We note, however, that the weighted sum associated with the clock policy is always close to the minimum value attainable by the $H_{FIFO-LRU}$ policy for the entire range of memory sizes (especially for large memory sizes) and programs. In other words, with respect to this measure, the clock policy is much more robust to variations in the memory size, in the program characteristics and, although not displayed (for brevity), n the value of α. This is primarily due to the ability of the clock algorithm to dynamically par-

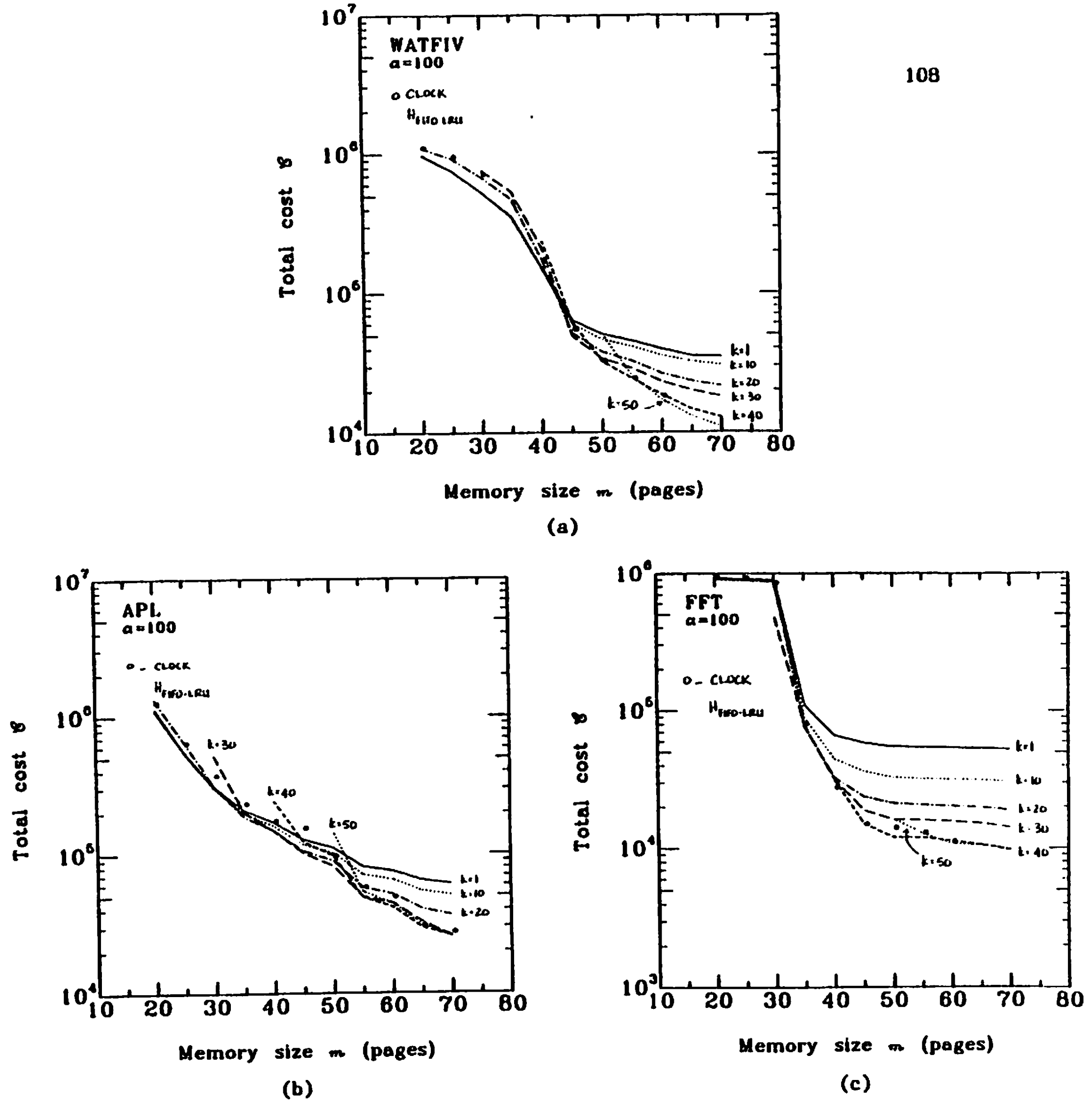


Figure 5.3.2.5 $H_{FIFO-LRU}$ (lines) and clock (circles) policy weighted sums as a function of the memory size m for (a) WATFIV, (b) APL and (c) FFT.

tition the available memory into the valid and reclaimable sets.

5.4. Variable-Size Partition Results

5.4.1. Hybrid Policies

Recall that variable-size partition hybrid policies result from the use of the working set policy for the management of the bottom. These policies have two parameter: the top size k and the window size τ.

Figure 5.4.1.6 compares the performance of the two variable-size partition hybrids with that of the fixed-size partition hybrid. For the three programs studied, the $H_{FIFO-WS}$ and $H_{RAND-WS}$ policies have similar performances, with neither exhibiting uniform superiority. This is not surprising if we recall our experience with the performance of the FIFO and RAND policies in conjunction with the LRU bottom. Both of these policies, however, outperform the fixed-size partition hybrid, by more than an order of magnitude in some cases, for equal values of the parameter and of the mean memory size in all three programs. Furthermore, these variable-size partition hybrids tend to have performances that are more uniform with respect to changes in the memory size than the fixed-size partition hybrid. Note that Figure 5.4.1.6 (c) contains mean memory sizes that are less than the parameter value for the case k=50. This results from the fact that our simulation studies assume an empty initial memory that may take a long time to fill.

In section 4.3.2.3, we discussed alternatives to the estimate of the time of last reference to a page during its residency in the top. Our simulations so far have assigned the time of departure from the top as this estimate. The proposed alternatives of the time of entry to the top and the arithmetic mean of the time of entry and exit were also simulated. These results did not

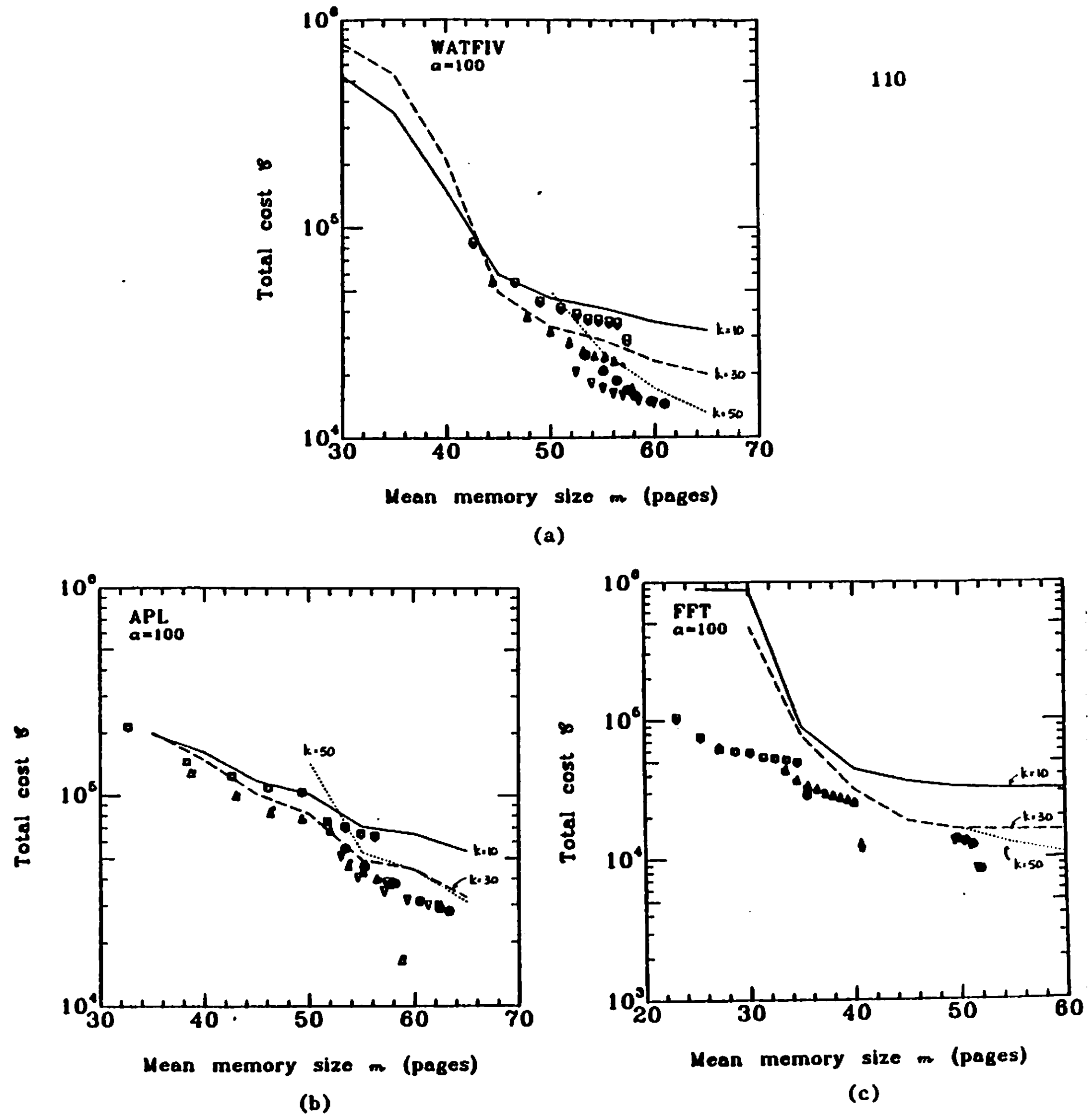


Figure 5.4.1.6 $H_{FIFO-LRU}$ (lines), $H_{FIFO-WS}$ and $H_{RAND-WS}$ policy total costs as a function of the memory size m for (a) WATFIV, (b) APL and (c) FFT. The window size, τ, was varied between 11730 and 117300 references to obtain the variable partition results. Key: ▪, Δ and ∇ represent $H_{FIFO-WS}$ with k=10,30,50 respectively whereas <>, o, and @ represent $H_{RAND-WS}$ with k=10,30,50 respectively.

show an appreciable difference in performance and are not presented for the sake of brevity.

5.4.2. The Sampled Working Set Algorithm

The Sampled Working Set (SWS) algorithm has been proposed as a practical implementation of the pure Working Set algorithm [Fog74a, Pri74a]. For simplicity, we assume that the sample interval T is equal to the working set window size τ. At the end of each sample interval the algorithm removes from the working set those pages that have gone unreferenced during this interval. The reference bits of all the remaining pages are reset. At the time of a page fault, the page that is referenced is added to the working set. Adapting this algorithm to an environment with no reference bits can be accomplished through the same scheme that we used with the clock algorithm where resetting the reference bit of a page is equivalent to changing its state from valid to reclaimable, while setting the reference bit is equivalent to changing its state from reclaimable to valid.

Given an n-page program modeled by the Independent Reference Model with parameters $(\beta_1, \beta_2, \ldots, \beta_n)$ operating under the SWS policy with sample interval T, the rate at which pages are reclaimed is given by

$$f(T) = \frac{1}{T}\sum_{i=1}^{n}(1-(1-\beta_i)^T)^2$$
$$= \frac{1}{T}(n - 2\sum_{i=1}^{n}(1-\beta_i)^T + \sum_{i=1}^{n}(1-\beta_i)^{2T}).$$

Note that, during a sample interval, a page reclaim occurs if a page has been referenced at least once during the last sample interval (thus it is an element of the working set at the beginning of the current interval) and is referenced at least once during the current interval. Since each of these events

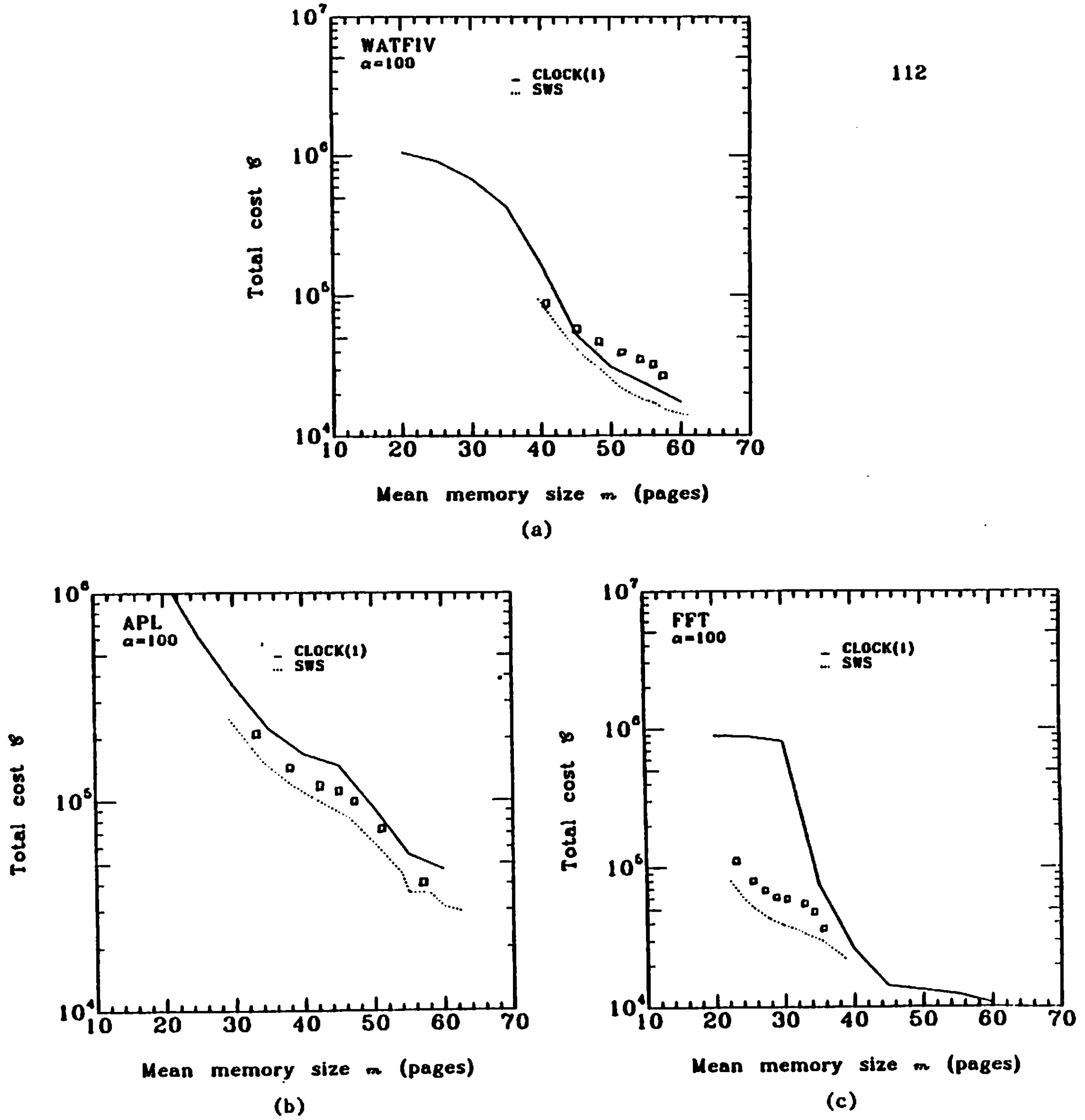


Figure 5.4.2.7 Weighted sums for the clock, SWS and $H_{FIFO-WS}$ (the square points) policies as a function of the memory size m for (a) WATFIV, (b) APL and (c) FFT.

occurs independently with probability $(1-(1-\beta_i)^T)$, the number of page reclaims within a sample interval is given by

$$N(T) = \sum_{i=1}^{n}(1-(1-\beta_i)^T)^2$$
$$= n-\sum_{i=1}^{n}(2(1-\beta_i)^T - (1-\beta_i)^{2T}).$$

Obviously, the reclaim rate is $f(T)=N(T)/T$.

The mean working set size, $\bar{w}(T)$, and the page fault rate, $F(T)$, under the SWS policy as obtained by Rafii [Raf76a] are:

$$\bar{w}(T) = n-\frac{1}{T}\sum_{i=1}^{n}\frac{(1-\beta_i)^T - (1-\beta_i)^{2T+1}}{\beta_i}$$

and

$$F(T) = \frac{1}{T}\sum_{i=1}^{n}((1-\beta_i)^T - (1-\beta_i)^{2T}).$$

Figure 5.4.2.7 compares the weighted sum measure under the SWS policy with those of the clock and $H_{FIFO-WS}$ policies. In all three programs, the SWS has uniformly better performance, especially for smaller mean memory occupancy values, than the clock policy with respect to this measure. While having comparable performances, the SWS policy has over the $H_{FIFO-WS}$ policy the advantage of not requiring the specification of a top size.

Recall that the overhead associated with an implementation of the clock policy was defined as $\Omega(clock)$= (mean number of pages examined per page fault) * (number of page faults). The equivalent measure for the SWS policy can be expressed as $\Omega(SWS)$= (mean working set size) * (number of sample intervals), assuming that the same unit of work is involved in performing the same operations under the two different algorithms.

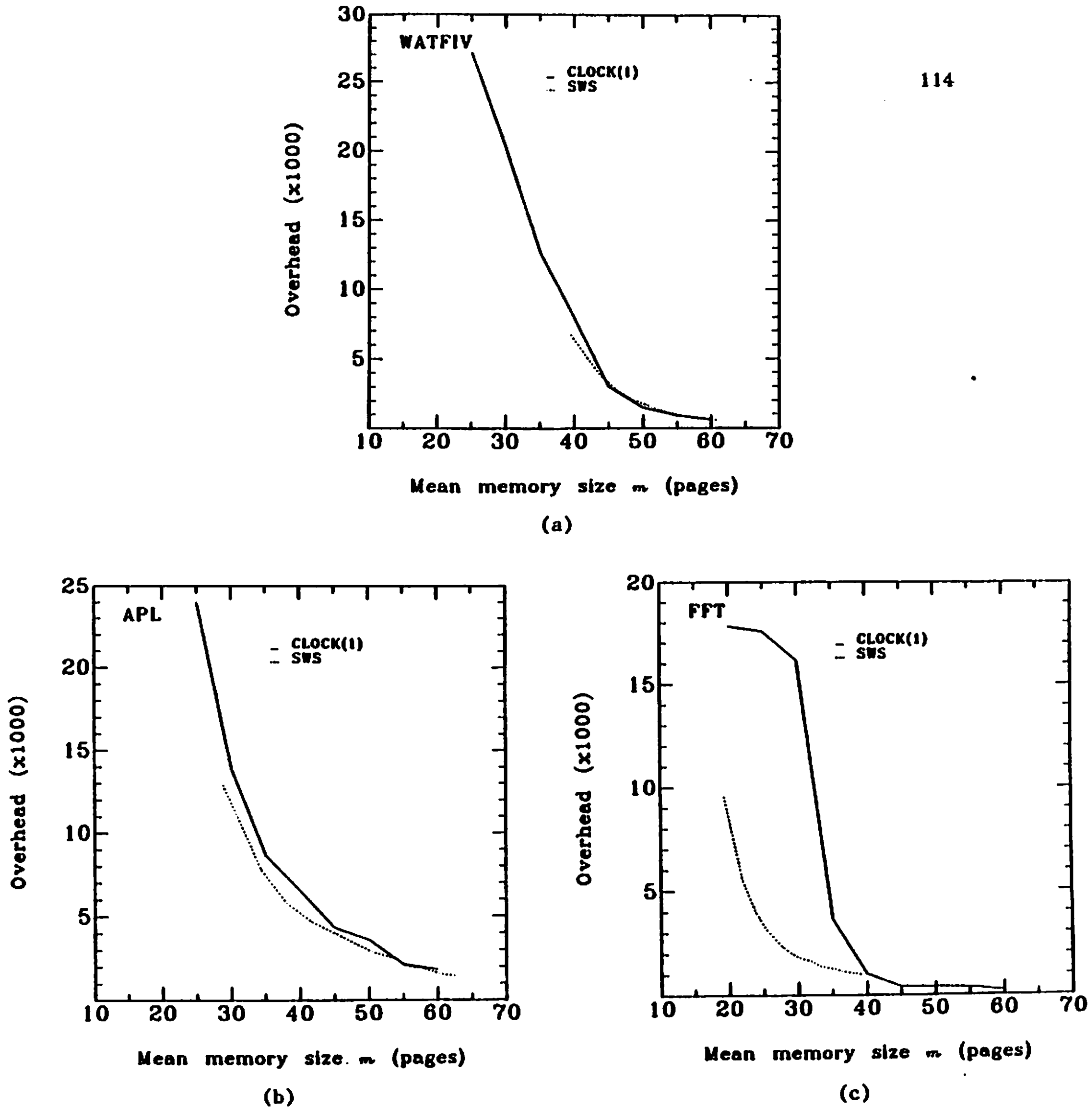


Figure 5.4.2.8 Overhead measures for the clock and SWS policies as a function of the memory size m for (a) WATFIV, (b) APL and (c) FFT.

This overhead measure for the three programs is displayed in Figure 5.4.2.8. The SWS algorithm incurs lower overhead uniformly for small values of mean memory occupancy. This overhead measure, while appropriate for an implementation where reference bits are available, is not capable of accounting for the time cost associated with a page reclaim operation in our environment. An algorithm that achieves good performance, as measured by our weighted sum function, may generate a very high page reclaim rate resulting in an unreasonable amount of CPU cycles devoted to servicing them.

A more appropriate measure of this overhead is the page reclaim rate generated by an algorithm to achieve a certain page fault rate. The SWS and clock algorithms are compared in this manner for the three programs in Figure 5.4.2.9. Observe that the SWS algorithm in fact generates a uniformly higher page reclaim rate than the clock algorithm. Let t_r denote the number of instructions required to service a page reclaim. For the SWS algorithm, $\frac{\bar{w}\cdot t_r}{T}$, where $\bar{w}$ is the mean working set size and T is the sample interval expressed in number of references, represents the mean fraction of the CPU cycles devoted to servicing page reclaims. In the next chapter, we report numerical results obtained from an implementation that show this fraction approaching 1 for reasonable values of $\bar{w}$ and T under SWS. To reduce this cost to levels comparable with that of the clock algorithm, we must increase the sample interval T of the SWS This reduces the frequency of sampling. However, Figures 5.4.2.7 and 5.4.2.8 show that for large values of T (thus large values of the mean memory occupancy), the SWS algorithm degenerates in performance and overhead measures such that it becomes indistinguishable from the clock algorithm.

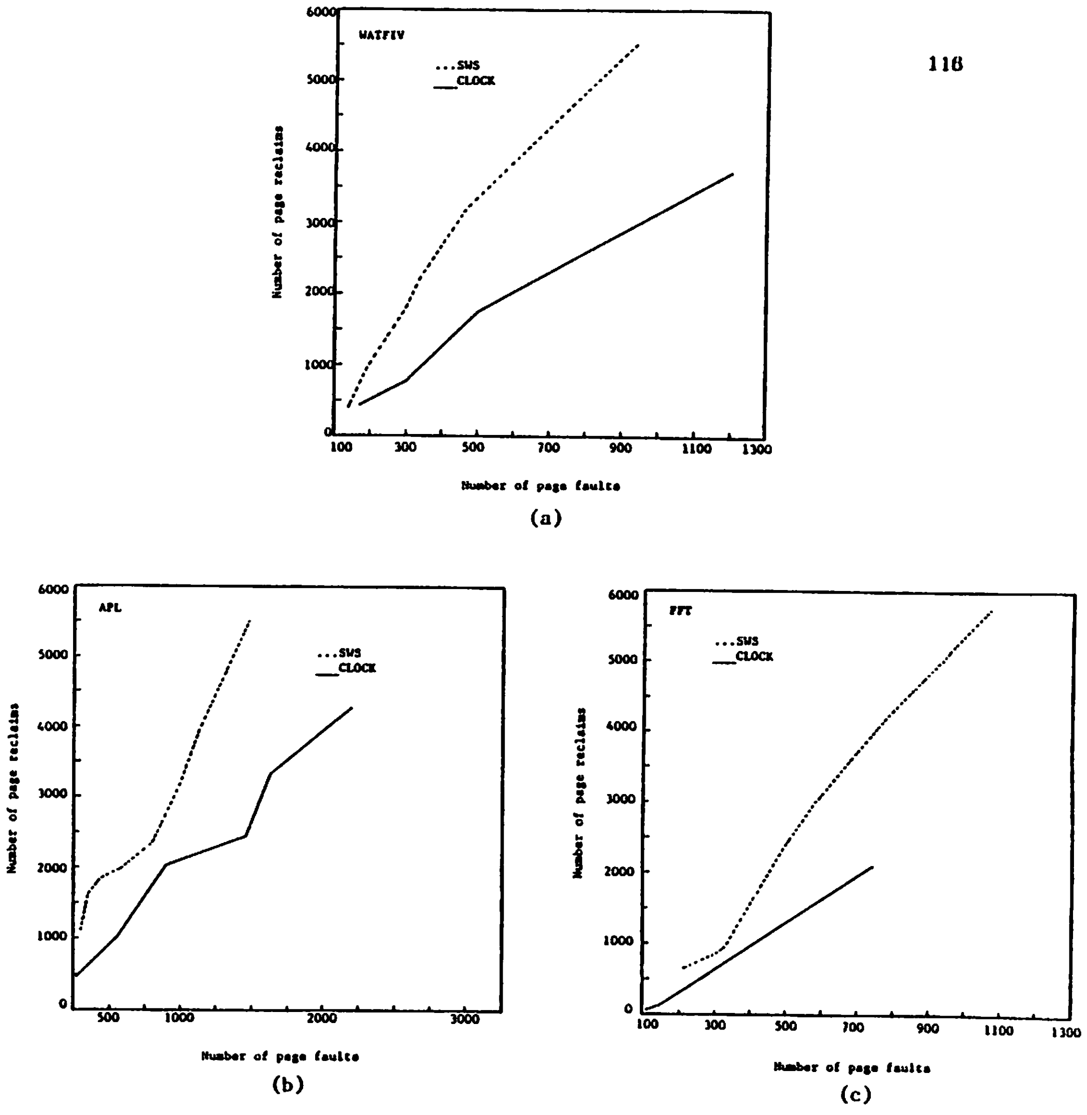


Figure 5.4.2.9 Number of page reclaims for the clock and SWS policies as a function of the number of page faults for (a) WATFIV, (b) APL and (c) FFT.

5.5. Conclusions

The observations of the previous chapter have been confirmed using trace-driven simulations. The fixed-size partition hybrid policies perform very well for a given program and memory size if the policy parameter is selected correctly. Under usual circumstances, the operating conditions of the policy are rarely constant. This requires that the policy parameter be modified to track these variations. The hybrid policies, however, have no built-in mechanism for doing this. External mechanisms must be introduced to vary the policy parameter based on certain heuristics.

The clock algorithm was observed to be much more robust with respect to the variations that affect the performance of the hybrid policies. The ability of this algorithm to dynamically partition the memory into the valid and reclaimable regions is at the root of its robustness.

The variable-size partition hybrid policies were observed to be uniformly superior in performance to their fixed-size partition counterparts. The use of FIFO or RAND replacement for the top had negligible influence on performance as did the choice of the estimate of the time of last reference to a page during its residency in the top. In addition to superior performance, the use of the WS algorithm for the management of the bottom results in completely local policies with total isolation amongst processes. The specification of the fixed top size remains the major drawback of these policies.

A truly variable-sized partition local page replacement algorithm was studied in the context of an environment lacking reference bits. The Sampled Working Set algorithm appears to have performance comparable to those of the variable-size partition hybrids and of the pure working set algorithm but does not have any of the problems associated with the fixed-size tops. However, the SWS algorithm results in page reclaim rates that are prohibitively high for reasonable values of the sample interval. Increasing the sample interval and, consequently, decreasing the frequency of samples reduces the fraction of CPU cycles spent servicing page reclaims. At the same time, this results in performances for the SWS that are similar to those for the clock algorithm. We further address this issue in the next chapter.

5.6. References

[Cor68a] F. J. Corbato, "A Paging Experiment with the Multics System," Project MAC Memo MAC-M-384 Mass. Inst. of Tech. (July 1968). Published in In Honor of P. M. Morse ed. Ingard MIT Press 1969, pp. 217-228

[Eas79a] M. Easton and P. A. Franaszek, "Use Bit Scanning in Replacement Decisions," *IEEE Trans. Comptrs.* C-28 pp. 133-141 (February 1979).

[Fog74a] M. H. Fogel, "The VMOS Paging Algorithm - A Practical Implementation of a Working Set Model," *Operating Systs. Rev.* 8 pp. 8-17 (January 1974).

[Lau79a] E. Lau, "Performance Improvement of Virtual Memory Systems by Restructuring and Prefetching," Ph.D. Th., Univ. California Berkeley, California (1979).

[Pri74a] B. G. Prieve, "A Page Partition Replacement Algorithm," Ph.D. Th., Univ. of California Berkeley, California (1974).

[Raf76a] A. Rafii, "Empirical and Analytical Studies of Program Reference Behavior," Ph.D. Th., SLAC Report 197, Stanford California (1976).

[Smi76a] A. J. Smith, "A Modified Working Set Paging Algorithm," *IEEE Trans. Comptrs.* C-25 pp. 907-914 (September 1976).

CHAPTER 6

VMUNIX- DESIGN, IMPLEMENTATION AND MEASUREMENTS

6.1. Introduction

In the fall of 1978 the Computer Science Division of the University of California at Berkeley purchased a VAX-11/780 and arranged to run an early version of UNIX for the VAX provided by Bell Laboratories under a cooperative research agreement. The VAX was purchased because it is a 32-bit machine with a large address space, and we had hopes of running UNIX, which was successfully being used on other smaller machines.

Except for the machine-dependent sections of code, UNIX for the VAX was quite similar to that for the PDP-11, which has a 16-bit address space and no paging hardware. It made no use of the memory-management hardware available on the VAX aside from simulating the PDP-11 segment registers with VAX page table entries. The main-memory management schemes employed by this first version of the system were identical to their PDP-11 counterparts-- processes were allocated contiguous blocks of real memory on a first-fit basis and were swapped in their entirety. A subsequent version of the system was capable of loading processes into noncontiguous real memory locations, an allocation policy called *scatter loading*, and was able to swap only portions of a process (*partial swapping*) as deemed necessary by the memory contention. This became the basis for the paging system we developed, called VMUNIX (for Virtual Memory UNIX), that is discussed in this chapter.

6.2. Search for a Replacement Policy

The VAX memory-management architecture supports paging within three segments (two for user processes, one for the system). The interesting aspect of the architecture is the lack of page-referenced bits (also called *use* bits). To remedy this situation, the dynamic address translation mechanism of the VAX was used to detect and record references to pages. With this scheme, a page for which reference information is to be gathered is marked as invalid although it remains in main memory. This state for a page is called the *reclaimable* state. A reference generated to a location within this page causes an *address-translation-not-valid* fault. However, the fault handler can detect this special state of the page and thus refrains from initiating the page transfer from secondary memory. In other words, the reclaimable state for a page corresponds to a valid page with the reference bit off, if a reference bit were available. Since this method of simulating page-referenced bits through software has a nonnegligible cost, the relative performances of some of the most popular replacement algorithms in this environment are no longer known.

In VMS, the vendor-supplied operating system for the VAX, the solution to the replacement decision is simple [Tur81a]. Each process is assigned a fixed-size memory partition, called a *resident set*, that is managed according to the FIFO policy. Pages that are not members of any of these resident sets are grouped together to constitute the global free list which functions as a disk cache. Although there is some isolation between the paging behavior of the various processes due to the strictly local resident sets, the coupling that is introduced through this global free list has significant performance implications. Lazowska [Laz79a] reports that in measurements based on a real workload, system performance was significantly improved by increasing the minimum size of the free list (a system generation parameter). An unfortunate consequence of allocating fixed-size partitions to processes is that a process has its pages taken away from its resident set (relatively small in size compared to the total real memory available on the machine) and placed in the free list to be subsequently reclaimed even though it may be the only active process in the system.

In the last two chapters we have studied the class of hybrid page replacement policies. This class includes the VMS algorithm described above, called Segmented FIFO in [Tur81a], as an instance where the resident set management is according to the FIFO policy and the free list management is approximately Least-Recently-Used (LRU).

UNIX is particularly ill-suited for such a scheme for several reasons. The UNIX system encourages the creation of a number of processes to accomplish most tasks-- processes are cheap. As in most systems, these processes are nonhomogeneous; they vary greatly in size and in the manner in which they access their address space. Furthermore, in certain processes the page reference behavior varies radically over time as the process enters different phases of execution. The LISP system, which initiates garbage collection after an interval of execution, is an example of such a process. Thus, in this environment, it is unlikely that we will find a single system-wide value for the fixed resident set size that will nearly optimize an objective function such as the weighted sum of the page fault rate and the rate at which reclaimable pages are referenced under the hybrid policy. In fact, even for a single process, the value of the resident set size must vary in time in order to track different phases of its execution and the varying amounts of real memory

available to it. As described earlier, the total number of pages from the free list belonging to a certain process is a dynamic quantity due to its sensitivity to the system-wide paging activity.

However, simulation studies based on actual program address traces, reported in the previous chapter, showed the clock page replacement algorithm [Cor68a] to be much more robust with respect to the objective function defined above to variations in the amount of memory available to the program, the relative costs of page faults and reclaims, and the nature of the program itself than the fixed-partition VMS scheme.

We now introduce some terminology associated with the algorithm that will be used in the remainder of this chapter. Recall that, under the simplest form of this policy, all the pages allocated to a program are thought of as ordered around the circumference of a circle, called the *loop*, according to their physical page frame number. In addition, there is pointer, called the *hand*, that is advanced circularly through them when page faults occur until a replacement candidate is located. A page is chosen for replacement if it has not been referenced during the time interval between two successive passages of the hand through this page. The movement of the hand to perform these functions is called the *scan* operation.

Another major departure in the VMUNIX memory management from the VMS design resulted from our decision to apply the clock page replacement algorithm globally to all pages in the system rather than locally to the pages of each process. Note that all of our studies in the previous chapters have assumed a uniprogramming environment, whereas this modification results in a variable-size memory partition for each process. This was motivated by studies where global versions of fixed-partition replacement policies had been found to have better performances than their local counterparts [Oli74a, Smi80a, Smi81a], and by the following considerations:

(i) The relative simplicity of the global clock policy and, consequently, the ease of implementation.

(ii) The projected workload for the system had no requirement of guaranteed response times as in time-critical applications. Whereas a local algorithm can allocate large amounts of memory to processes on a selective basis, a global algorithm cannot.

(iii) It was unreasonable to expect users to specify the sizes of the fixed program partitions since from the existing system they had little or no information about the memory requirements of their programs.

(iv) Without reference bits, the cost of implementing variable-partition local replacement policies such as SWS or Page Fault Frequency [Chu76a] was observed to be too high (see section 5.4.2 of the last chapter). We comment further on this point in the following section.

(v) UNIX encourages the construction of tasks consisting of two or more processes communicating through *pipes*, which must be co-scheduled if they are to execute efficiently. In most instances, the intensity of activity, thus the memory demand, shifts over time from the left-most process to the right-most process in the pipe while all of them remain active. It is our belief that, in such an environment, dynamic partitioning of memory amongst these processes in real time is more appropriate than having local partitions (working sets) that are maintained in process virtual time.

6.3. Memory Demand and Clock Triggering

The clock page replacement policy as described in the previous chapter is only engaged upon a page fault, at which time it selects a page to be replaced. Given that the demand for memory exhibits nonuniform patterns with occasional high spikes (see Figure 6.3.1), this strategy for the activation of the replacement policy is clearly suboptimal.

Having incurred the cost of page replacement policy activation, we would like to select more than a single page to be replaced in order to anticipate short-term demand for more memory. To this end, the system

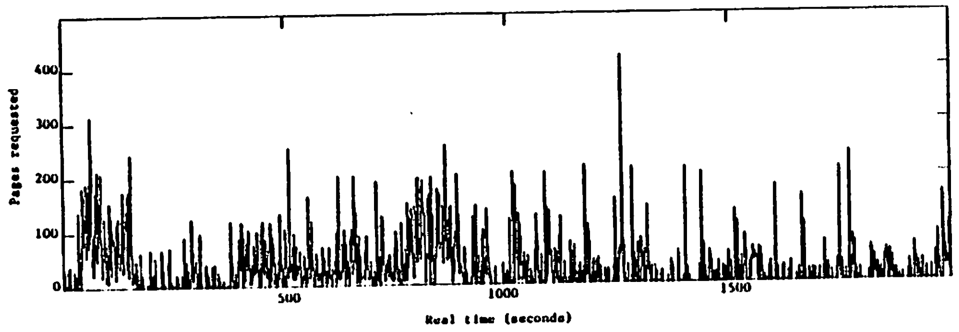


Figure 6.3.1. Number of page frames requested globally in one second intervals during a 33 minute observation period. The data was obtained by tracing the memory request events under the VMUNIX system with the performance enhancements (section 6.7) turned off.

maintains a free page pool containing all of the page frames that are currently not in the loop. Our version of the clock policy is triggered whenever the size of this pool drops below a threshold. Then, the algorithm scans a given number of pages per second of real time (a simplified version of this algorithm is discused in [Eas79a]). Currently, the default trigger point for the free page pool size is set at 1/4 of the real memory size and the default minimum scan rate of the hand is approximately 100 pages per second. As the free page pool size further drops below the threshold, the scan rate of the hand is increased linearly up to a given maximum value. The primary factor that determines this maximum value is the time it takes to service a page reclaim from the loop (i.e., the time to simulate the setting of a reference bit). Measurements based on the current system indicate that on the average this action consumes approximately 250 microseconds of processor time. The full distribution of the page reclaim service time is shown in Figure 6.3.2 (a). We note in passing that the ratio of the mean page fault service time to the mean page reclaim service time (α as defined in the previous chapter) is approximately 200 based on the data of Figure 6.3.2. Our use of α=100 for the simulation studies in Chapter 5 is, therefore, fully justified. Since the number of pages scanned by the clock algorithm provides an upper bound on the number of pages that can be reclaimed, the processor overhead due to the simulation of reference bits can be controlled by limiting this maximum scan rate. Currently, we allocate at most 10 percent of the available processor cycles to this function. This implies that the maximum scan rate of the hand is limited to approximately 400 pages per second. Due to the existence of the free page pool, however, short duration memory demands far in excess of this value can be satisfied. The problem of formalizing some of these decisions along with the selection of parameter values is

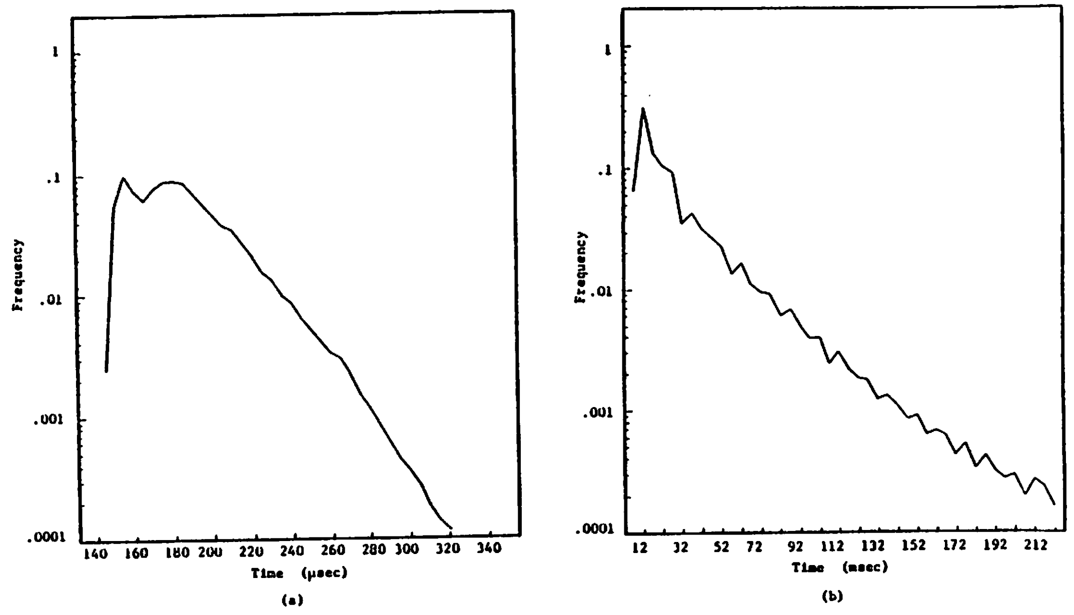


Figure 6.3.2. (a) Page reclaim service time distribution.
(b) Page fault service time distribution. Mean service time for a page reclaim is 250 microseconds while it is 50 milliseconds for a page fault. The page fault distribution was obtained under a two-disk drive system with the paging activity on a model RM03.
The page fault service time distribution includes all queueing delays (at the device and the processor) in addition to the device service time.

discussed in the next chapter.

The system maintains enough data to be able to reclaim any page from the free page pool regardless of how it arrived there. In addition to being replenished from the loop, the free page pool also receives pages of processes that are swapped out or completed. In both cases, these pages can be reclaimed by the process upon a subsequent swap in or a future incarnation of the same code, provided that in the meantime the pages have not been allocated for another purpose.

Given the cost to simulate the setting of a reference bit, our previous remarks concerning the unsuitability of local variable partition page replacement policies in the UNIX environment are justified. As an example, using the Sampled Working Set policy with a window size of 100,000 instructions (approximately 100 milliseconds on the VAX) operating with a program having a 400-page working set would consume 50 percent of the processor cycles just to simulate reference bits (assuming that the working set of the program remained unchanged between two consecutive sample points and that virtual time does not advance during the page reclaim service intervals).

The use of a modified clock page replacement algorithm where the scan rate is based on the available memory has several other advantages as well. The length of the free page pool becomes a natural indicator of the amount of memory contention in the system. As we shall see, the inability of the system to maintain some specified amount of free memory is the basis for load control, and causes process deactivation by swapping. Control of the scanning rate allows the page write-back activity, that is initiated when dirty pages are removed from the clock loop, to be spread more uniformly over time, thereby easing the contention for the disks.

6.4. Implementation; New System Facilities

The UNIX system memory-management facilities are particularly simple. Each user process has a read-only shared program area, a modifiable data area, and a stack. An *exec* system call overlays a process' address space with a particular program image from a file consisting of the shared code and the initialized data. New processes are created by the *fork* system call, which causes a process to clone itself. Usually, the command interpreter accomplishes its task by first creating a copy of itself to establish the context for the command and then causes this copy to overlay itself with the file that is the image of the command. Except for shared program areas, no other memory between processes can be shared. Access to files and devices is through *read* and *write* system calls; no segment-based or page-based shared access to file pages is available.

Consistent with our design goals, we wished to keep changes to the system as simple as possible and orthogonal to the rest of the system's design. Then, further changes to the UNIX system would be less likely to invalidate our efforts.

The conversion of the swap-based system to a paged system began in the late spring of 1979 and the first version of the paging system was put into production use on a single machine in September of 1979. At that time, the primitives for the swap-based UNIX system were still in use. Processes were created using the *fork* system call which copied a process' address space page by page to create the new address space. This newly-formed address space was then overlaid with a new image through the *exec* system call. These primitives, while simple to implement and relatively cheap (involving memory-to-memory copying and file reading) in a swap-based system, were very expensive under the new system, since programs could be partially loaded in memory and could be much larger.

We found that a vast majority (over 80 percent) of all *forks* executed in the system were due to the command interpreter. Since these *forks* only serve to establish the context for the new process, duplication of the entire address space was a wasted effort. Most of the sharp spikes in the global memory demand pattern of Figure 6.3.1 could be attributed to processes *forking* and/or *execing*. The nondemand nature of these requests for memory (in the sense that they are an implementation artifact) overtaxed the page replacement algorithm and had grave performance consequences.

A natural solution to the problem would have been to include a "copy-on-write" facility to implement a *fork* similar to that used in various PDP-10 operating systems (such as TENEX [Bob72a]). In this scheme, the two processes would be allowed to share the same address space and the copying at the page level would be deferred until the time of the first modification of a page by either process. However, this would have significantly increased the number of modifications to UNIX and hence delayed the completion of a workable and useful system. At the time, the desires of our user community did not indicate that shared-memory primitives would be necessary in the near future.

A new primitive to replace most instances of the *fork* system call was designed. This primitive, called ***virtual-fork***, allows the original process to establish the system context for the new process but refrains from creating the address space until the subsequent *exec* system call that is issued by the new process or until the completion of the new process. During this interval, the system context of the original process is dormant. To put it another way, the new process is allowed to run within the address space of the original process until it establishes its own address space through an *exec* system call or through completion, at which point the original process, which was dormant, regains its address space. Obviously, during this transition period the new process must not modify the contents of the address space that is "on loan" to it. This mechanism allows a new process to be created without any copying of address space and without requiring a mechanism like "copy-on-write."

Note that there are instances of process creation where the *virtual-fork* system call is inappropriate. An example of such a case occurs when commands are executed in the "background". Then, the new process is initiated, but the command interpreter does not wait for its completion and is ready to accept a new command line. However, all other instances of the *fork* system call could be (and were) replaced with the *virtual-fork* call without change to the calling program. It is quite easy to implement this primitive on non-paged machines as well as on paged machines, and there are strong indications that the overhead of process creation in the swap-based PDP-11 implementation of UNIX would be reduced if such a primitive were implemented.

A new load format was also provided to reduce the implied overhead of the *exec* call. Programs loaded using this new format would have their pages demand-loaded from the file system rather than pre-loaded as in the previous swap-based system. This reduced the overhead of process invocation, and was soon made the default load format.

6.5. Limiting Page Traffic and Controlling Multiprogramming Load

In addition to the processor overhead considerations which limit the scan rate of the clock replacement algorithm, there are global system considerations involved in limiting page traffic. Input-output activity generated by page replacement should not delay too much of the input-output activity generated by program request. UNIX typically runs on relatively small machines that often have only two moving head disk drives used for all system activity including paging, swapping and file system transfers. Special paging devices are rare in such systems. It is not practical to design a system that saturates one of these arms to maximize memory usage. Input-output bandwidth is often as precious as memory residency. Thus, load control mechanisms such as the "L=S" or the "50 percent" criterion [Den76a, Den77a] which assume the availability of a separate paging device, are inappropriate. We therefore decided to deactivate processes by swapping them to secondary storage when demand for main memory exceeded our ability to supply it.

Multiprogramming load control in our system is thus based on a desire to limit paging overhead. When the system finds that it cannot maintain an acceptable amount of free memory by consuming approximately 10 percent of the available processor time to sample page utilization, it lowers memory demand by removing a process from the set of runnable processes. The process to be swapped out is selected by choosing the oldest amongst the n largest resident processes. This policy represents a compromise between the *largest-first* and the *oldest-first* policies [Chu76a, Cof73a]. Neither of these policies was found to be satisfactory in its pure form; the former prohibits a large process from making any progress while the latter wastes effort by constantly swapping out small processes that do not contribute much to the memory demand. Currently, the default value for the variable n is 4. The pages of the swapped-out process are written to secondary storage if necessary, removed from the loop and returned to the free list. Processes that are swapped out are assigned priorities to return to the runnable set based on their size (smaller jobs have higher priority) and the amount of time they have been swapped out (priority increases as time goes by). Sufficient time delay is built into the swapping algorithm to ensure that useful work gets done between swaps. Since in a reasonably-configured system swapping a process out is a rare event, we do not swap in the resident set a process had at the time it is swapped out. In our environment, the long period of inactivity of the process that caused the swap out is usually a good indicator of a locality transition through the invocation of a new function (for example, a new input line to the command interpreter from a terminal). In such cases, the overlap between the old resident set and the new is minimal. However, chances are that the process will still find some of its pages in the free page pool, and can simply reclaim them by referencing them.

6.6. Comparison with the Swap-Based System

After two months of production use and a reasonable amount of tuning, we decided to compare the performance of the system running with and without the virtual-memory changes. A script-driven experiment was designed for a stand-alone configuration consisting of 1 megabyte of main memory, two disk arms on two different controllers, each with a peak transfer rate of 1 megabyte per second and a 40 millisecond average access time. For the comparison we used the version of the swap-based system that was the base for the paging extensions. The page size in use in the paging

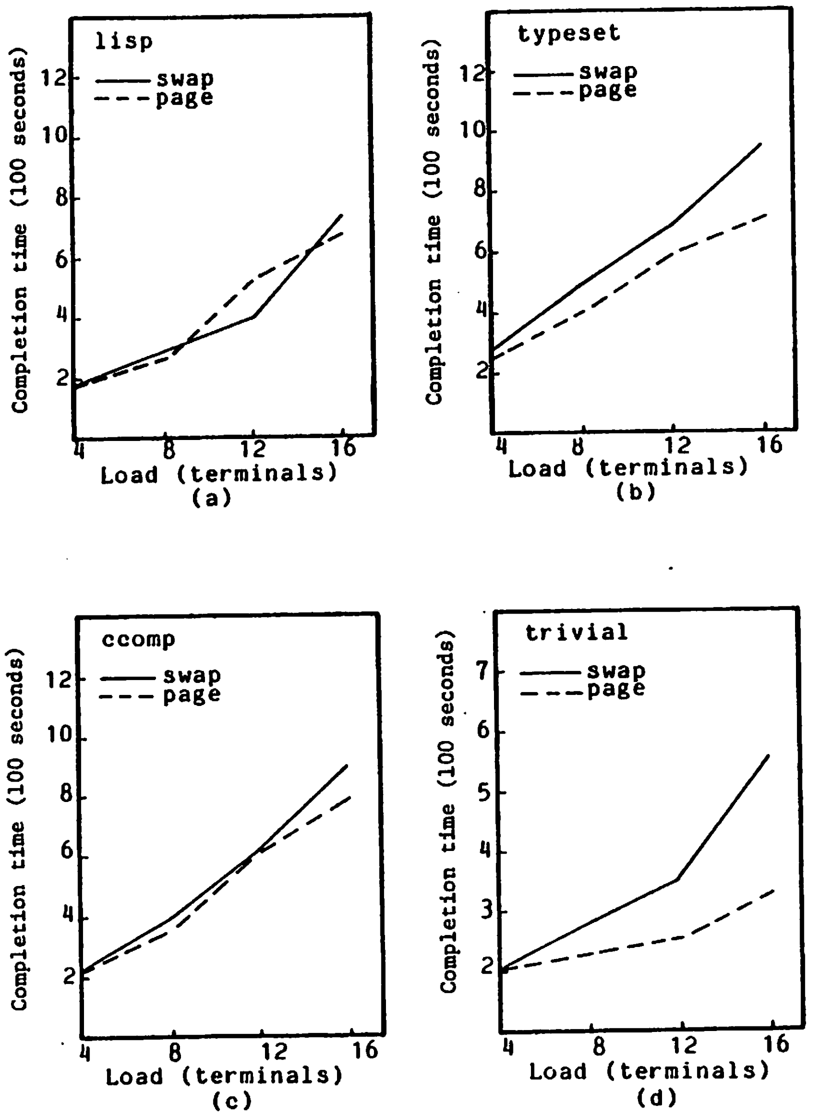


Figure 6.6.3. Average completion times
(a) lisp, (b) typeset, (c) ccomp, (d) trivial.

version of the system was 512 bytes.

The basic unit of work generated by the script was made up of four terminal sessions executed concurrently. The terminals are identified by the major task that the perform:

lisp	A LISP compilation of a portion of the LISP compiler, followed by a "dumplisp" using the lisp interpreter to create a new binary version of the compiler.
ccomp	An editor session followed by the compilation and loading of several small C programs that support the line-printer spooler.
typeset	An editor session followed by the typesetting of a mathematical paper and production of output for a raster plotter.
trivial	Repeated execution of a trivial command (printing the date) every few seconds.

Staggered multiple initiations of from one to four of these work units were used to create increasing levels of load on the system. Figure 6.6.3 gives the average completion times for each type of session under the two systems. For the nontrivial sessions, completion times were very similar under the two systems, with the paging version of the system running (in all but one case) faster. The interesting observation is that the swap-based system departed from linear degradation more rapidly, i.e., for a smaller number of active terminals. This trend is most noticeable in the response time for the trivial sessions.

Figure 6.6.4 gives system-wide measurements collected during the same experiments whose results were given in Figure 6.6.3. These measurements show the same trend for both the time when the last script completed execution and the average completion times for individual sessions, with the

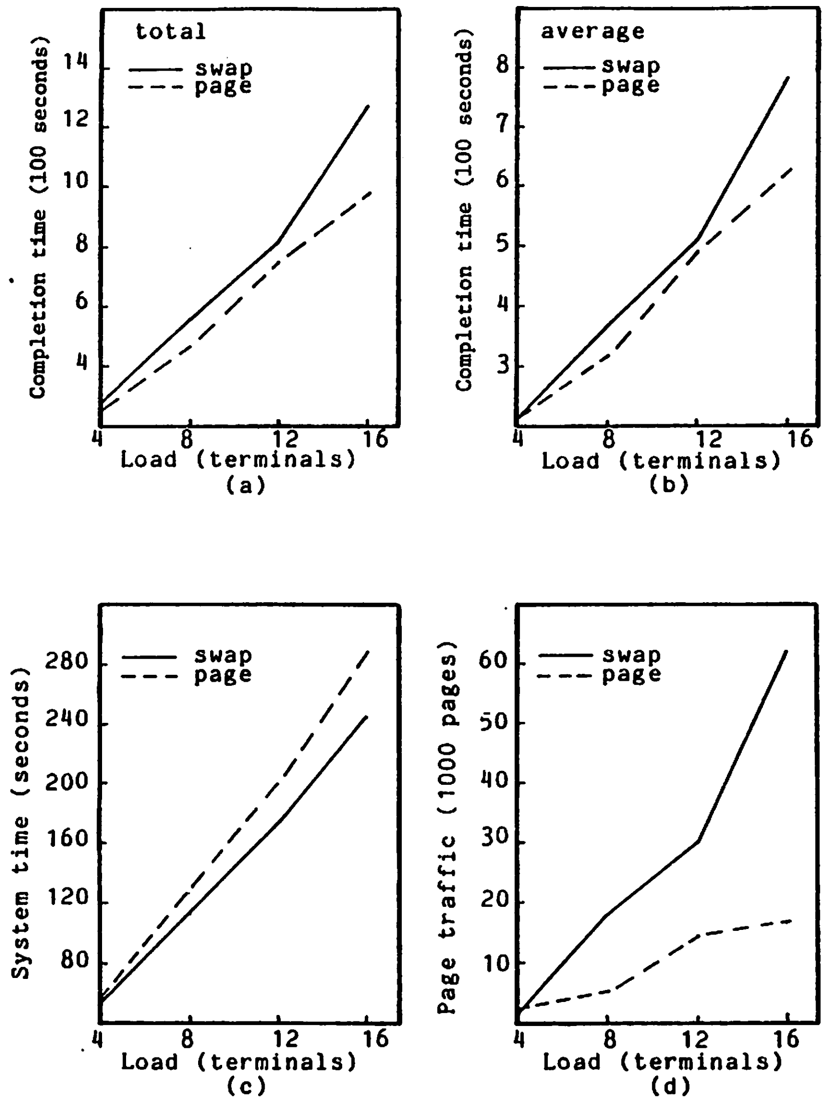


Figure 6.6.4. System-wide measurements (a) total completion time, (b) average completion time, (c) system time, (d) page traffic.

paging system slightly faster and degrading more linearly than the swap system within the measured range. Under heavy load, system overhead was uniformly greater under the paging system, constituting 26 percent of the CPU utilization as compared to 20 percent under the swap system. User-CPU utilization under this load was, however, uniformly greater for the paging system, averaging 48 percent, while the swap-based system averaged only 42 percent.

Finally, the total page traffic generated under the two systems was measured. The measurement accounts for both paging and swapping traffic under the paging system, as well as transfer of all system information (control blocks and page tables) under both systems. Although the paging system resulted in far fewer total pages transferred, the number of transactions required to accomplish this was much greater since most transfers under the paging system were due to paging activity rather than swapping activity. In this version of the paging system, all paging input/output activity dealt with single 512 byte pages.

6.7. Performance Enhancements and Comparisons with Hybrid Paging

After measuring the system and seeing that the performance was comparable with that of the swap system, we determined that there was a major bottleneck in the system due to the small page and file block size-- 512 bytes. Measurements of typical system programs which processed files one character at a time showed that the fastest such programs produced and consumed data at a rate of about 80 512-byte pages per second. The file system in use on VMUNIX at that time, however, could produce about 40 blocks per second on average, resulting in a factor of two mismatch between typical program speed and average file system throughput.

The file block size was increased from 512 to 1024 bytes and physically adjacent pages were grouped in pairs producing the current 1024-byte "pages". All future references to "pages" imply this new size, unless noted otherwise. With the new page and file block size, total system throughput on the script-driven benchmarks discussed above improved significantly, with the completion time dropping an average of 30 percent, user-CPU utilization rising nearly 20 percent and system overhead dropping below that of the swap-based system.

Benchmarks of paging intensive synthetic programs run on VMS and VMUNIX showed, however, that VMUNIX could not supply memory to heavily paging programs at a rate comparable to that of VMS [Kas80a]. Simple test programs that sequentially or randomly (with varying degrees of randomness) accessed virtual memory were run on both systems and ran much faster on the VMS system which clusters pages both for input and output. The problem, here, was similar to the problem with the file system: inadequate blocking. Transferring only 1024 bytes of data after incurring a 25-30 milliseconds delay while waiting for a moving-head storage device kept the bandwidth low.

To remedy the situation, a simple form of pre-paging was implemented [Smi78a, Lau79a]. Upon a page fault, the faulting page as well as the next several virtually (and physically) adjacent pages were read in as a single operation. Similarly, upon a page out decision, the set of modified pages would be searched to construct clusters of virtually (but also physically) adjacent pages that would be written back to disk in a single operation. Both the input and output cluster sizes are variables that can be varied while the system is in operation. This drastically improved system performance on the

simple test programs due to their sequential nature and the fact that they always dirtied pages by writing into them.

There remained, however, a performance gap between our system and VMS whose cause eluded us at the time. The problem was discovered to be the placement of pre-paged data. Such data was placed in the clock loop, but marked as being not referenced, so it would be moved to the free page pool in a single revolution of the clock if it remained unreferenced by the program. For programs similar to Kashtan's test programs, which have a very high data rate but do not use all the prefetched data, this resulted in an excessive load on the clock algorithm.

This flaw in the pre-paging algorithm was corrected by placing the pre-paged pages at the bottom of the free page pool list rather than in the clock loop. Recall that the system free page pool, which is implemented as a queue, is fairly long. Since page frames are allocated from the *head* of this queue, on a busy system, pages near the bottom may survive (i.e., remain reclaimable) for a few seconds before being re-used. Since the pages were pre-paged because they were adjacent to a recently referenced page, it is desirable to retain them only for a short while if they are not referenced. The modified pre-paging placement policy more closely reflected this intent.

A new system call was added to notify the system that a process would be exhibiting anomalous behavior. This call caused the reference bit simulation to be turned off resulting in approximately random page replacement (since the physical ordering of page frames in the free page pool from where they are allocated is destined to be random after a period of operation of the system) for these processes. Currently, the LISP system issues such a call before entering the garbage collection phase.

After these changes, the performance of the two systems on the test programs became comparable. In practice, however, the VMUNIX page replacement algorithm has the advantage that it does not give processes fixed partitions and therefore tends to avoid unnecessary processor overhead (a different form of thrashing [Den68a] that is unique to our environment) in a way that a fixed partition scheme cannot do.

6.8. Conclusions

The VAX-11/780 computer provided the initial motivation for the study of virtual storage management without reference bits. The preliminary studies of the problem were used to guide the selection of the algorithms to be employed in the actual implementation described. This system extended our understanding of the problem through measurements under real workloads. Some of the major observations we have to make about our experience follow.

A page replacement algorithm that is to function in a machine lacking reference bits must use a minimum of reference information because such information is expensive to gather. The global clock paging algorithm appears to satisfy this condition.

System performance under extreme paging load can be as good using the global clock algorithm as it is using a hybrid paging technique. In practice, the ability of the clock algorithm to vary the memory partitions dynamically increases memory utilization significantly over a scheme which allocates fixed partitions.

The global clock page replacement algorithm is limited in its ability to supply pages on a machine with no reference bits. This is normally not a problem under a time-sharing load, but can be when high data rate programs

are run.

Paging can result in performance enhancement over swapping in addition to the obvious increase in functionality.

6.9. References

[Bob72a] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10," *Comm. ACM* 15(3) pp. 135-143 (March 1972).

[Chu76a] W. W. Chu and H. Opderbeck, "Program Behavior and the Page Fault Frequency Replacement Algorithm," *Computer* 9 pp. 29-38 (November 1976).

[Cof73a] E. G. Coffman and P.J. Denning, *Operating Systems Theory*, Prentice-Hall, Enlewood Cliff, New Jersey (1973).

[Cor68a] F. J. Corbato, "A Paging Experiment with the Multics System," Project MAC Memo MAC-M-384 Mass. Inst. of Tech. (July 1968). Published in In Honor of P. M. Morse ed. Ingard MIT Press 1969, pp. 217-228

[Den68a] Peter J. Denning, "Thrashing: It's Causes and Prevention," *Proc. Fall Joint Comptr. Conf.*, pp. 915-922 (1968).

[Den76a] P. J. Denning, K. C. Kahn, J. Leroudier, D. Potier, and R. Suri, "Optimal Multiprogramming," *Acta Informatica* 7(2) pp. 197-216 (1976).

[Den77a] Peter J. Denning and Kevin Kahn, "An L=S Criterion for Optimal Multi-Programming," *Proc. Int. Symp. on Computer Performance Modeling Measurement and Evaluation*, pp. 219-229 , Cambridge, Mass.(August 1977).

[Eas79a] M. Easton and P. A. Franaszek, "Use Bit Scanning in Replacement Decisions," *IEEE Trans. Comptrs.* C-28 pp. 133-141 (February 1979).

[Kas80a] D. L. Kashtan, "UNIX and VMS: Some Performance Comparisons," SRI International Internal Report (1980).

[Lau79a] E. Lau, "Performance Improvement of Virtual Memory Systems by Restructuring and Prefetching," Ph.D. Th., Univ. California Berkeley, California (1979).

[Laz79a] Edward D. Lazowska, "The Benchmarking, Tuning and Analytical Modeling of VAX/VMS," *Proceedings of the Conference on Simulation, Measurement and Modeling of Computer Systems, Boulder, Colorado*, pp. 57-64 (August 1979).

[Oli74a] N. A. Oliver, "Experimental Data on Page Replacement Algorithm," *Proc. NCC*, pp. 179-184 (1974).

[Smi78a] Alan Jay Smith, "Sequential Program Prefetching in Memory Hierarchies," *Computer* 11(12) pp. 7-21 (December 1978).

[Smi80a] A. J. Smith, "Multiprogramming and Memory Contention," *Software- Practice and Experience* 10 pp. 531-552 (July 1980).

[Smi81a] A. J. Smith, "Internal Scheduling and Memory Contention," *IEEE Trans. Software Eng.* SE-7 pp. 135-146 (January 1981).

[Tur81a] R. Turner and H. Levy, "Segmented FIFO Page Replacement," *Performance Eval. Rev.* 10(3) pp. 48-51 (Fall 1981). Proceeding of ACM/SIGMETRICS Coference on Measurement and Modeling of Computer Systems, Las Vegas, Nevada

CHAPTER 7

MEMORY MANAGEMENT AS INVENTORY CONTROL

7.1. Introduction

Almost all virtual-memory management systems maintain a pool of free pages that are available for immediate allocation upon demand. In systems that employ variable-size partition local policies, this free page pool contains the balance of the available pages after allocating to each process admitted to memory its working set. In such systems, the size of the free page pool activates the process activation/deactivation mechanism. If the processes' working sets expand so as to consume the entire free page pool, one of them must be selected for deactivation. A process is considered for activation only if the free page pool is large enough to contain its working set. Under these conditions, Simon [Sim79a] has shown that the fraction of available memory is approximately given by

$$\frac{N}{N+(C^2+1)/2}$$

where N is the multiprogramming level and C is the coefficient of variation of the working set size of a program over time.

We are concerned with virtual-memory management systems that apply a replacement policy globally to the entire memory. Here, the free page pool does not arise naturally but must be maintained explicitly. Under normal operation, the sum of the processes' resident sets would be allowed to expand to fill the available memory. Any further expansion causes the replacement policy to select one page amongst the whole memory to satisfy

the new request. As we saw in the previous chapter, however, requests for memory are often nonuniform and there are instances when a large number of pages are required in rapid succession. The system should maintain a set of pages that are available immediately for allocation. This way, each request for memory will not cause activation of the replacement policy. The VMUNIX system described in the previous chapter does this. The memory manager tries to maintain some pages in the free page pool by initiating the clock replacement algorithm at varying rates depending on memory demand. This policy is described by three parameters: (i) the free page pool size at which the replacement algorithm starts to run, (ii) the rate of the clock scan at this point, and (iii) the maximum scan rate allowed before process deactivation is considered.

In this chapter, we present some preliminary results of our efforts to formalize the decisions made in the management of free page pools. Results from classical inventory theory are used to show how the problem at hand can be viewed within this framework. Although we will use the VMUNIX environment as an example of the application, the formulation is general enough to encompass other systems.

7.2. The Stock Room Problem

Consider the following problem: A stock room contains an inventory of a certain commodity that is for sale. There is a certain demand for the commodity, that may be characterized by the number of items requested per period, where the period is a fixed interval of time. There are also costs associated with maintaining the inventory at a certain level, ordering more items, and receiving requests that cannot be met due to depleted stock. At the beginning of each period we are faced with the choice of ordering additional items, if any, to minimize our total expected operating cost.

We formally define the *single period* model corresponding to this inventory problem [Ros70a]. Purchasing z units of the item incurs the cost

$$C(z) = \begin{cases} K+c\cdot z & , z>0 \\ 0 & , z=0 \end{cases}$$

where $K>0$. Note that c is the cost of each unit and K is the fixed setup cost for the transaction. We assume that an order is filled immediately. For each unit of the maximum level attained by the inventory during a period, we pay h dollars as a *holding cost*. Each unit of unmet demand is assumed to generate a *penalty cost* p. The planning horizon is assumed to be infinite. Therefore, there is no time in the future when the program stops and we are left with unsold inventory. Given that the per-period demands are independent and identically distributed with probability density $g(\xi)$, and the initial inventory level is x, we are interested in determining the amount of additional inventory to be ordered at the beginning of each period that will minimize the total expected cost.

Assume that the inventory level at the beginning of a period is x and we order $y-x$ additional items to bring it up to y. The expected costs incurred during the period can be expressed as

$$\mathbf{E}[\text{holding cost}] = h\cdot y$$

$$\mathbf{E}[\text{penalty cost}] = p\int_y^\infty(\xi-y)g(\xi)d\xi$$

since the maximum level of the inventory during a period is y. Let $L(y) = \mathbf{E}[\text{holding cost}]+\mathbf{E}[\text{penalty cost}]$. The total expected cost of ordering up to y is then given by

$$\begin{aligned} K+c\cdot(y-x)+L(y) \quad & \text{if } y>x \\ L(x) \quad & \text{if } y=x. \end{aligned}$$

An argument based on the convexity of the function $cy+L(y)$ can be used to show that the optimal policy is of the class (S,s) [Sca60a] that operates as follows:

if $x<s$, order up to S

if $x\geq s$, do not order.

S is the value of y that minimizes $cy+L(y)$ and s is the smallest value of y for which

$$cy+L(y) = K+cS+L(S). \tag{7.2.1}$$

By setting the first derivative of $cy+L(y)$ to zero and solving for S, we obtain

$$S = G^{-1}(\frac{p-c-h}{p})$$

where $G(x) = \int_0^x g(\xi)d\xi$ is the distribution of the per-period demand. Having obtained S, we can (in principle) determine s by solving equation 7.2.1. Note that, if K=0, i.e., there is no setup cost for ordering, then S=s and the optimal policy is particularly simple. Observe that, if $p\leq c+h$, then S=0 and no items are stocked.

7.3. Modeling the Demand Process

Demand is perhaps the most important and difficult aspect of inventory modeling. The stochastic processes that characterize demand, even for the simplest situations, are analytically intractable. Let the random variable N_i denote the quantity of items demanded during period i and let X_j denote the interval of time between the jth and $(j+1)$st demand points. In the continuous case, the assumption made in the previous section that the N_i's are independent and identically distributed is equivalent to assuming that the X_j's are independent and exponentially distributed, and that the quantity demanded at a given point is either fixed or is identically and independently distributed. In other words, we are assuming the demand process to be a Poisson process. This is a restrictive assumption and n section 7.4 we present empirical data to validate it.

In VMUNIX, demand for pages from the free page pool occur due to two types of events. A page fault always requests two 512-byte pages (since we simulate 1024-byte pages), whereas a process creation always requests eight 512-byte pages for establishing the process context (these pages constitute the so-called *u. area* and contain the kernel stack for the process, open file descriptors, register contents, etc.). Since about 94% of all page requests are of the first type (page faults), we will assume that each demand point represents a request for two pages. Based on this additional assumption about the stochastic process that generates the demand in our stock room model, the per-period demand divided by two (since we always allocate pages in pairs) is Poisson distributed:

$$\Pr(N_i/2=n) = e^{-\lambda t}\frac{(\lambda t)^n}{n!} \quad ,n=0,1,2,3,\ldots$$

where t is the length of the period and $1/\lambda$ is the mean interdemand time.

The memory allocation function of the VMUNIX system was instrumented to produce a trace record for each explicit request for memory from the free page pool. The trace record consisted of the amount requested as well as of a time stamp with microsecond resolution. As the tracing took place upon the arrival of a demand rather than at the delivery of the requested page frames, it approximately captured the intrinsic global memory demand of

the workload and not the artifacts of the particular policies employed at the time. There are, however, several peculiarities of the VMUNIX system that had to be dealt with. These peculiarities arose from the ability of the processes to reclaim pages from the free page pool. Since this action invokes a mechanism different from the normal memory request, it is not recorded in our trace as being part of the memory demand. In the stock room problem, this situation is equivalent to the inventory being depleted without demand. One can model such an environment as a stock room containing perishable items that become unusable at a certain rate. This would, however, further complicate the expressions for the order quantity S and the order point s.

Recall that in our system there are three sources of page reclaims:

(i) A page is removed from the loop and placed in the free page pool by the clock algorithm and is subsequently referenced. This type also includes the reclaiming of pages that were prefetched and placed in the free page pool.

(ii) A page belongs to a process that was swapped out and is referenced by that process after being swapped in.

(iii) The page belongs to a process that has completed and is referenced by a future incarnation of the same program.

Under normal operating conditions, the last two sources of reclaims account for a major percentage of all reclaims. To solve the problem of unaccounted allocations from the free page pool, the mechanism that allows reclaims of types (ii) and (iii) was turned off, causing a real page fault and consequently generation of the trace record for such references. The remaining source of reclaims represented only about 8% of all pages allocated during the measurement period. Their effect on the demand process will be ignored. This type of reclaim, however, will be accounted for in our definition of the holding cost for the pages.

7.4. Data Analysis

Using the above mechanism, the system was traced on August 18, 1981 at 6:28pm for a period of 195-seconds. The time series representing the number of pages requested during 100 millisecond intervals for the duration of the tracing period is displayed in Figure 7.4.1.

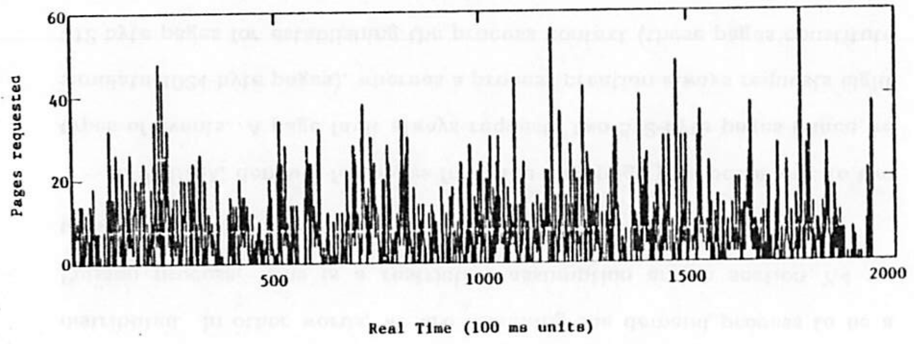


Figure 7.4.1. Number of pages requested within 100 ms intervals during a 195 second period.

7.4.1. Test for Trend

Many of the statistical tests that we are going to perform on the data are based on the assumption that the demand process is stationary. The (S,s) class of inventory policies with constant S and s can only be optimal when the stochastic process by which demand is generated has sufficient stationary properties. Since the lack of stationarity can result in misleading estimates of parameters, we must begin our data analysis with tests for trend. In case of nonstationarity of the data, it will have to be *detrended* before the analysis can proceed. This is a costly process and we will try to avoid it by selecting a portion of our data that appears to be stationary. The problem of applying forecasting techniques to the developed policy to deal with nonstationary demand will be discussed in section 7.7.

Let $W = 1/n \sum_{i=1}^{n} T_i$, where n is the number of demand points within the observation period and T_i is the time of the ith demand. For a Poisson process with a constant rate, this statistic is, conditional on observing n events in the fixed observation period t, asymptotically normally distributed with mean $t/2$ and variance $t^2/(12n)$ [Lew73a]. The normalized statistic

$$U = \frac{W - t/2}{t/\sqrt{12n}}$$

is standard normal. This statistic was evaluated for various portions of the trace data, and the null hypothesis $U=0$ was tested. Based on this approach, the first 3295 points of the trace, representing 102 seconds of real time, were selected for further analysis. This portion of the trace was judged to be sufficiently stationary since the U statistic had a value 0.008 and the critical values for a 5% level two-sided test under the null hypothesis are ± 1.96. Visual inspection of Figure 7.4.1 indeed supports the lack of any obvious trend during the first 100 seconds. Some of the other interesting characteristics of this subtrace are reported in Table 7.4.1.1.

For stochastic processes with variances greater than that of the Poisson process, the U statistic can produce inflated results. An alternate method of testing for trend involves a goodness of fit test based on the chi-square statistic. This statistic is obtained by comparing the empirical event counts within fixed length intervals with the uniform distribution [Bic77a].

7.4.2. Tests for Serial Independence

Recall that the assumption of independent and identically distributed per-period demands in our inventory model implied a Poisson process for the demand generation mechanism. This is equivalent to requiring the time interval between demand points, i.e., the X_i random variables, to be independent and exponentially distributed.

For the sequence of stationary random variables $X_1, X_2, X_3, \ldots, X_n$, the estimated serial correlation coefficient of lag k is defined as

number of demand points	3295
duration (real time)	102.5 seconds
total number of pages requested	6924
mean number of pages requested	67.4/second
number of page reclaims (type (i) only)	791
mean time interval between demand points	30.57 ms
coefficient of variation of time between demands	2.57
minimum time between demands	357 μs
maximum time between demands	1823 ms

Table 7.4.1.1. Characteristics of demand trace data.

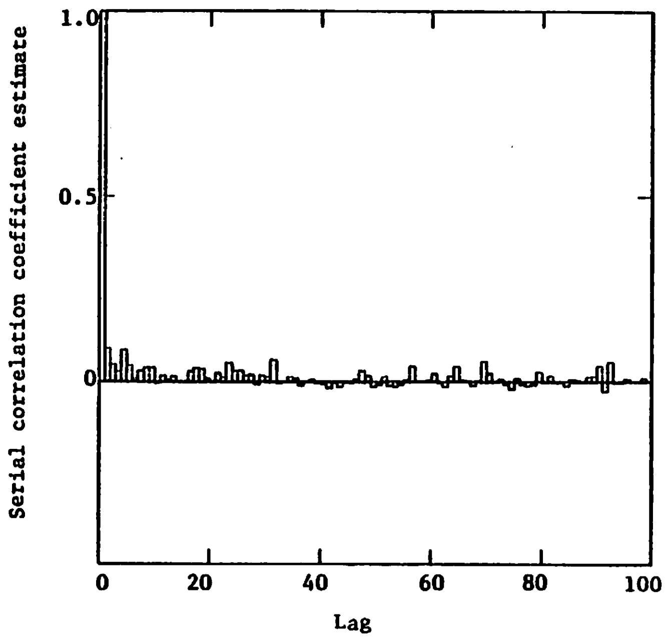


Figure 7.4.2.2. The first 100 serial correlation estimates for the times between demand points.

$$\hat{\rho}_k = \frac{\frac{1}{n}\sum_{i=1}^{n-k}(X_i-\bar{X})(X_{i+k}-\bar{X})}{\frac{1}{n}\sum_{i=1}^{n}(X_i-\bar{X})^2},$$

where $\bar{X} = 1/n\sum_{i=1}^{n} X_i$. Tests based on $\hat{\rho}_1$ are asymptotically most powerful for testing that a process is Poisson (independent exponentially distributed intervals between events) against the alternative that the intervals between events have exponential distributions, but first-order serial dependence [Lew73a].

When $\rho_1 = 0$, $\hat{\rho}_1$ is asymptotically normal with $E[\hat{\rho}_1] \approx 0$ and $Var(\hat{\rho}_1) \approx (n-1)^{-1}$ under very general conditions. The first 100 serial correlation coefficient estimates for the time interval between demands for the trace were calculated and are displayed in Figure 7.4.2.2. The point $\rho_0 = 1$ is shown to establish the scale. Before we comment on these results, a discussion of the physical process that generated the trace data is in order.

Recall that a demand for memory occurs upon a page fault (process creation also generates demand for memory, but as we saw in section 7.3, this accounts for a very small percentage of the total demand). In process virtual time (time that advances only when the process is executing), the time intervals between page faults have been observed to be serially correlated with highly skewed marginal distributions [Lew73a, Lew71a]. What we observe in our trace, however, is the composition of many stochastic processes, each corresponding to a particular program, as they interleave in real time. In our multiprogramming environment, a page fault causes the process to be blocked for the duration of the page transfer from secondary store and resumes another process from the set of runnable processes. The elapsed real time between the blocking (either due to a page fault or to quantum expiration) and resumption of a process is a random variable that is a function of many events. The global demand process is this complex interleaving of processes and should have little resemblance to its components. In other words, the blocking due to page faults in the multiprogramming environment has a *randomizing effect* on the individual page fault patterns of the programs. This is precisely the source of our hope for independence amongst the time intervals between demand points.

Our informal argument supporting serial independence is rejected by the formal test based on the estimate of the first serial correlation coefficient. We see that, for our 3295-point data, $\hat{\rho}_1 = 0.0887$. Although this estimate is not very different from zero, we must reject the hypothesis that $\rho_1 = 0$ at the 1% level (the upper critical point is 0.0452). This is a common

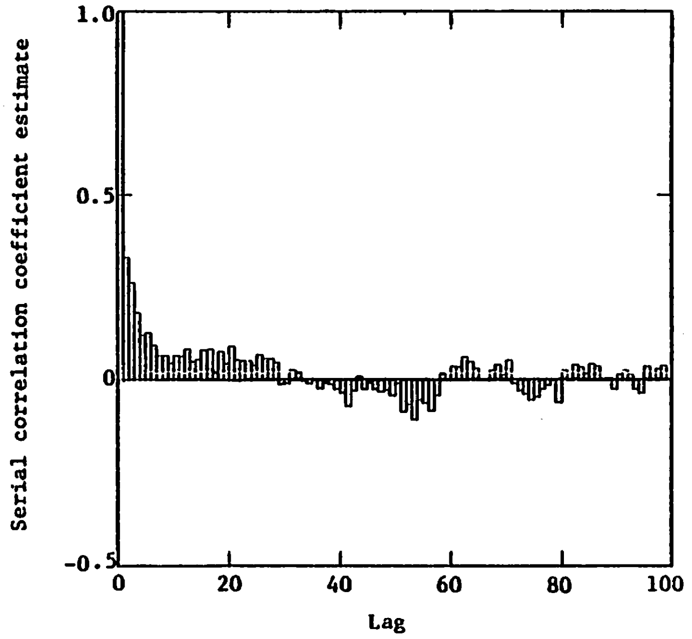


Figure 7.4.2.3. The first 100 serial correlation coefficient estimates for the number of pages requested within 100 millisecond intervals for the trace data (1025 points).

problem in hypothesis testing when the sample population is very large, resulting in a very small asymptotic variance. This observation is confirmed by the fact that, if we repeat the above test using only the first 104 points of the trace data (this represents a 4 second period where U=0.06), we obtain $\hat{\rho}_1$ = −0.029, and the null hypothesis is not rejected even at the 5% level (the critical points are ±0.196).

The nonzero value for $\hat{\rho}_1$ can be attributed to the following. In our system, there are two types of page faults that do not cause the process to block. These are due to the expansion of the data and stack segments of the process beyond their current sizes. Upon one of these faults, the process simply receives a page initialized with zeros from the free page pool. Processes executing many levels of nested procedure calls (perhaps recursively) will cause the stack segment to grow at a near constant rate since the procedure activation frames are stored there. Alternatively, a process executing code to initialize a large array will cause the data segment to grow at a constant rate. As the quantum size is large (16.7 milliseconds), a sequence of such page faults can be generated during an execution interval. The regular nature of the time interval between such demands is probably the source of the positive serial correlation of lag 1.

Having obtained inconclusive results about the hypothesis that the time intervals between demand points are serially independent, we examine the per-period demand patterns. Since the X_i's display nonzero serial dependence, we do not expect the N_i to be independent. Recall that the independence of the N_i implied a Poisson process as the demand generating process. Indeed, Figure 7.4.2.3 displays large values for the serial correlation estimates for small lags (for example, $\hat{\rho}_1$ = 0.3317).

7.4.3. Marginal Distributions

To complete our analysis of the data, we discuss the estimated distributions of the time intervals between demands and the per-period demands. For random variable X, the ***survivor*** function, $R(x)$, is defined as

$$R(x) = \Pr(X>x) = 1-F(x)$$

where $F(x)$ is the distribution function of X. For convenience, it is customary to plot the logarithm of $R(x)$ as a function of x. The estimate of this function for the time intervals between demands is shown in Figure 7.4.3.4.

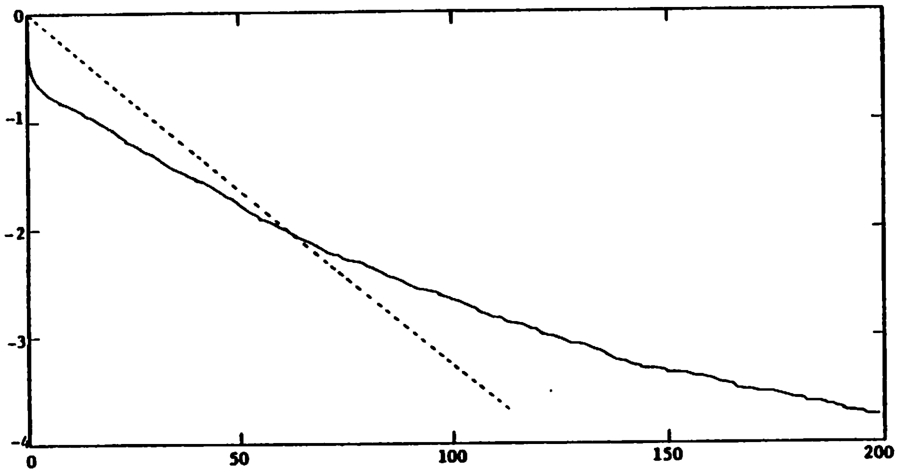
The Poisson process assumption implies an exponential marginal distribution for the X_i's. Note that if the X_i's were samples from an exponential distribution, then

$$R(x) = e^{-\lambda x} \quad ,x\geq 0$$

and

$$log(R(x)) = -\lambda x$$

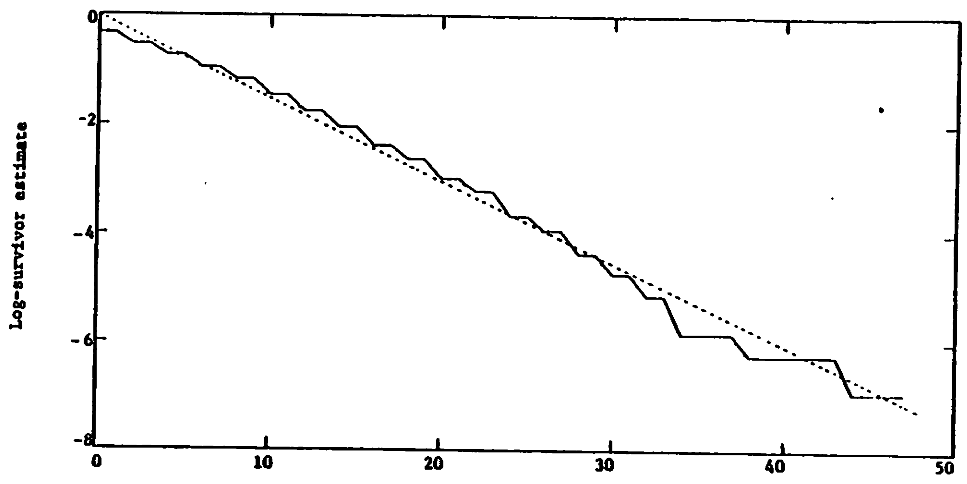
where $1/\lambda$ is the mean. In other words, the log-survivor plot of an exponential distribution is a straight line having a slope equal to the negative reciprocal of the mean. We see from Figure 7.4.3.4 that the empirical log-survivor function of X_i has a slope smaller than the exponential counterpart for large values of x. Based on the skewness of the data (the coefficient of variation is 2.57), it is obvious that any reasonable test would conclude that the distribution of times between demand points is not exponential. Guided by the shape of the empirical log-survivor curve, the distribution can probably be modeled by a ***mixed exponential*** distribution that is a weighted sum of n exponential functions for some $n\geq 2$. We will make no attempt at fitting distributions here.



Interarrival time (1 ms units)

Figure 7.4.3.4. Estimated log-survivor function for time intervals between demand points for trace data. The mean interval is 30.57 milliseconds and the coefficient of variation is 2.57. The dotted line is $log(R(x)) = -x/30.57$.

The time interval between demands has failed to satisfy both tests (independence and exponential distributedness) for being Poisson. Not surprisingly, the per-period demand process also exhibited serial dependence. To complete our data analysis, we examine the marginal distribution of the per-period demand, N_t.



Number of pages requested within 100ms intervals

Figure 7.4.3.5. Estimated log-survivor function for the number of pages requested during 100 millisecond intervals for trace data. The mean is 6.74 pages requested per millisecond and the coefficient of variation is 1.08. The dotted line is the plot of $log(R(x)) = -x/6.74$.

Figure 7.4.3.5 illustrates the empirical log-survivor function of the number of pages requested every 100 milliseconds. Again, on the same graph we have plotted the log-survivor function of the exponential distribution having the same mean as the empirical data. This time, however, we observe a good fit of the empirical distribution by the exponential distribution. This is also confirmed by the estimated coefficient of variation, which is very close to 1 as for the exponential distribution. Formal nonparametric test based on Kolmogorov's statistic [Bic77a], however, reject the hypothesis of exponential distribution. Compare the value 0.155 obtained for the Kolmogorov statistic with the critical value 0.032 at the 1% level for a sample size of 3295. As in the test for serial independence of the intervals between demands, reducing the number of points to the first 104 in the trace results in the non-rejection of the exponential hypothesis even at the 5% level. The critical value of 0.138 is significantly greater than the statistic's value of 0.073.

We summarize the findings of our data analysis efforts as follows. The time interval between demand points exhibits a distribution that is too skewed to be exponential (the coefficient of variation is 2.57). The test for serial independence of these times is inconclusive since the independence hypothesis is either rejected or not rejected depending on the sample size. The per-period demands generated within 100 millisecond intervals exhibit substantial serial correlation but appear to be well fitted by an exponential distribution (although formal tests reject or not reject this hypothesis depending on the population size as well). These findings lead us to reconsider our initial assumption of a Poisson process for the demand generation and consequently Poisson distributed per-period demands. However, due to its analytic tractability and partial support by the data analysis, we will go ahead and assume independent and exponentially distributed per-period demands and solve the stock room problem for the parameters S (the order quantity) and s (the order point) under this assumption.

7.5. Solution of Order Quantity and Order Point

Recall that in the general stock room formulation of the model, the optimal policy is such that at the beginning of each period the inventory is increased up to level S if it is below s and remains unchanged otherwise. The order quantity, S, is the solution to

$$G(S) = \frac{p-c-h}{p} \qquad (7.5.1)$$

and the order point, s, is such that

$$cs+L(s) = K+cS+L(S) \qquad (7.5.2)$$

where G(x) is the distribution of the per-period demand and $L(y) = hy+p\int_y^\infty(\xi-y)g(\xi)d\xi$. If the per-period demand is exponentially distributed with parameter λ, i.e., $G(x) = e^{-\lambda x}$, equation (7.5.1) can be inverted easily and we obtain $S = -\mu\ln(\frac{c+h}{p})$ where $\mu = 1/\lambda$. Substituting for S and $g(x)$ in equation (7.5.2) we obtain

$$sc+hs-e^{-\lambda s} = K' \qquad (7.5.3)$$

where $K' = K+cS+L(S) = K-\mu(c+h)(ln(\frac{c+h}{p})+\frac{1}{p})$. For u close to zero, $e^u \approx 1+u$. Making this substitution in equation (7.5.3) (this will require justification as we do not know the magnitude of λs) and solving for s, we obtain

$$s \approx \frac{K'+\mu}{c+h+1}.$$

Note that as the cost coefficients c, h, p, and K are constants, the logarithms can be computed a priori and the S and s values easily determined based on μ.

7.6. Cost Coefficient Estimation

To complete the model specification, we will have to interpret the various costs of the stock room problem in our environment. We will begin with the ordering cost.

In our environment, ordering more stock is equivalent to running the page replacement algorithm to obtain pages. This algorithm is implemented by a special process that is scheduled and run just as a user process, but at a high priority. Our model assumed that the order placed at the beginning of a period is fulfilled immediately. This does not reflect the behavior of our system since there may be a significant time delay between the initiation of the replacement algorithm (by *waking up* the process) and the delivery of the page frames. The ordering cost structure of the model has a component, K, that can account for the fixed overhead associated with context switching but is not capable of accounting for the possible lost sales due to demands during the lead time. The per-item cost, c, is interpreted as the average amount of work (some suitable units will have to be used consistently throughout these definitions) involved in selecting one page for replacement once the process implementing the algorithm has started executing.

The penalty cost, p, associated with each unit of lost sale due to depleted inventory is easily determined. The situation arises with a memory request when the free page pool is empty. Since all memory requests must be satisfied, i.e., we cannot really lose sales, the page replacement algorithm must be run just as for the normal ordering of pages. Now, however, we incur the fixed transaction cost, K, in addition to the usual cost c, for each page requested when the free page pool is empty. Therefore, the penalty cost per page, p, is equivalent to $K+c$.

The holding or storage cost, h, per unit of maximum inventory is perhaps the least obvious in our system. In the stock room problem, this cost is usually interpreted as the amount of money spent to rent the floor space that is used to contain the inventory. In our environment, we should ask ourselves what prevents us from placing all of the available memory in the free page pool. The answer is the reclaiming of pages from the free page pool. It is thus reasonable to define this cost as the reclaim rate times the cost of a single reclaim operation (in units consistent with the other costs). The larger the pool, the higher the reclaim rate and the associated processor overhead. To obtain a value for this cost we make the following observations. The global clock policy approximately behaves like the global LRU policy. If we consider all of the pages in real memory to be ordered according to their recency of usage (i.e., as the LRU stack), then all of the pages in the free page pool will be less recently used than those in the loop. Furthermore, we can assume that the pages in the free page pool are themselves ordered according to recency of usage. In other words, the free page pool represents the tail of the global LRU stack. Empirically, LRU stack position frequencies fall off very rapidly after the first few positions and are nearly constant for very large depths [Chu76a, Lau79a]. Note that a reclaim event is equivalent to a reference to this tail portion of the LRU stack. Under these assumptions, the reclaim rate increases linearly as the size of the free page pool increases consistent with our model holding cost structure. A possible scheme to determine a numerical value for h would be to observe the reclaim rate for a given free page pool size and then normalize this value to a unit length.

More general (and perhaps more realistic) functional dependencies of the holding cost on the maximum inventory level can be used at the expense of complicating the solution of the model for S and s.

7.7. Implementation Issues

The inventory model we proposed makes policy decisions only at regular time intervals and is called *periodic review*. Alternatively, we can allow the policy decisions to be made at any point in time, thereby obtaining the *continuous review* model. Note that the latter model requires knowledge of the system state (the inventory level) at all points in time. It can be shown that the continuous review policy results in a smaller expected cost (exclusive of the cost of gathering state information) than the periodic review policy [Bec61a]. For simplicity, we have assumed the periodic review model.

Based on the assumption that the per-period demands are independent exponential random variables with parameter λ, we have obtained expressions for S and s that are simple functions of λ. It is clear that the demand process in our system is not stationary. The rate of demand varies greatly in time depending on many factors. As the (S,s) policy with constant S and s can only be optimal under stationarity, the implementation must be able to track the variations in the demand and adjust S and s accordingly. Our system has a mechanism whereby the processor is interrupted every 16.7 milliseconds (1/60 seconds) to perform functions such as quantum updating, priority updating, etc. These times are natural candidates for the incorporation of our proposed policy. Given the estimates of c,h,p and K, the policy calculates the order point and the order quantity based on the current demand rate and places the order (if any) by initiating page replacement. We note that any number of forecasting methods such as maximum likelihood, exponential smoothing or Bayes procedure can be used in the updating of the demand rate (equivalently λ) [Gro74a].

Since the current system state is the size of the free page pool (contained in a variable) and the calculation of S and s involve trivial computations, the implementation of the policy will introduce insignificant additional overhead into the system.

7.8. Conclusions

The classical stock room problem was proposed as a suitable model for the general problem of free page pool size determination in a virtual-memory computer system. A specific instance of the problem was presented for the VMUNIX environment. The model is able to capture most of the essential characteristics of the problem but makes rather restrictive assumptions about the stochastic nature of the demand process. Formal statistical tests of empirical demand data supported the assumption of independently and exponentially distributed demands within fixed intervals. Based on this assumption, the model parameters were solved and shown to be simple functions of the distribution mean. We note in passing that there are inventory models that admit demand processes having arbitrary interval and quantity demanded distributions [Bec61a, Kao75a]. These models still require the time intervals to be independent but, more importantly, the S and s calculations are analytically intractable even for the simplest of distributions. Resorting to numerical methods each time the policy parameter need recalculation (every period, unless the demand process is judged to have been stationary) is prohibitive in terms of implementation overhead.

A trace-driven simulation of the proposed policy, using the trace data that was analyzed, will have to precede an actual implementation so that more exact procedures for the determination of the cost coefficients can be obtained and possible instability problems revealed.

7.9. References

[Bec61a] M. Beckmann, "An Inventory Model for Arbitrary Interval and Quantity Distribution of Demand," *Management Sci.* 8 pp. 35-57 (1961).

[Bic77a] P. J. Bickel and K. A. Doksum, *Mathematical Statistics: Basic Ideas and Selected Topics*, Holden-Day, San Francisco, California (1977).

[Chu76a] W. W. Chu and H. Opderbeck, "Program Behavior and the Page Fault Frequency Replacement Algorithm," *Computer* 9 pp. 29-38 (November 1976).

[Gro74a] D. Gross and R. J. Craig, "A Comparison of Maximum Likelihood, Exponential Smoothing and Bayes Forecasting Procedures in Inventory Modelling," *Int. J. Prod. Res.* 12(5) pp. 607-622 (1974).

[Kao75a] E. P. C. Kao, "A Discrete Time Inventory Model with Arbitrary Interval and Quantity Distributions of Demand," *Operations Research* 23(6) pp. 1131-1142 (November-December 1975).

[Lau79a] E. Lau, "Performance Improvement of Virtual Memory Systems by Restructuring and Prefetching," Ph.D. Th., Univ. California Berkeley, California (1979).

[Lew71a] P. A. W. Lewis and P. C. Yue, "Statistical Analysis of Program Reference Patterns in a Paging Environment," *Proc. IEEE Int. Comptr. Soc. Conf.*, pp. 133-134 , Boston, Mass.(September 1971).

[Lew73a] P. A. W. Lewis and G. S. Shedler, "Empirically Derived Micro Models for Sequences of Page Exceptions," *IBM J. Res. Develop.* 17(2) pp. 86-100 (March 1973).

[Ros70a] S. M. Ross, *Applied Probability Models with Optimization Applications*, Holden-Day, San Francisco (1970).

[Sca60a] H. Scarf, "The Optimality of (S,s) Policies in the Dynamic Inventory Problem," pp. 196-202 in *Mathematical Methods in the Social Sciences*, ed. K. J. Arrow, S. Karlin and P. Suppes, Stanford University Press, Stanford, California (1960).

[Sim79a] R. Simon, "The Modeling of Virtual Memory Systems," Ph.D. Th., Purdue U. Department of Computer Science (May 1979).

CHAPTER 8

SUMMARY AND CONCLUSIONS

8.1. Summary

In this thesis we have presented a comprehensive study of the problem of managing virtual storage systems that do not maintain page reference bits in hardware. Our study began with the development of some tools needed to evaluate virtual storage systems in general. The theoretical foundations of the synthetic program of Chapter 3 were laid in Chapter 2 based on certain observations about the LRU Stack Model of program behavior. The same result which allowed us to generate memory reference strings from this model in an efficient manner was exploited to realize a program that could reproduce automatically a given paging behavior when run in an environment that implemented an LRU-like replacement policy. The form of the program paging behavior specification is the lifetime curve. The synthetic program that was developed was used in the performance enhancement efforts for the VMUNIX system described in Chapter 6.

The main results of the thesis were developed in Chapters 4 and 5. In Chapter 4, we introduced the hybrid class of page replacement algorithms as a possible candidate for our environment. We obtained analytic expressions for their performances in terms of the Independent Reference Model of program behavior. From these results, it was evident that hybrid algorithms are in fact suitable for selecting pages to be replaced in the absence of reference bits under rather static conditions. Changes in the amount of memory allocated to the program, in the relative cost of a page fault and of a page

reclaim, in the characteristics of the program were amongst the variables that caused the hybrid policy parameter to require modification in order to maintain good performance. We pointed out that almost all of the above are in fact variables with very complex functional dependencies in a multiprogramming environment.

Chapter 5 extended the study of the hybrid policies through trace-driven simulations in a uniprogramming environment. These results confirmed our earlier observations based on the analytic expressions. A new performance measure based on the weighted sum of the page fault and the page reclaim rates was introduced that could be used to compare different algorithms in our environment. With respect to this measure, the clock algorithm was observed to have a better behavior than the fixed partition members of the hybrid class. Furthermore, the variable partition hybrid policies exhibited performances uniformly superior to those of the clock and of the fixed partition hybrid policies. This prompted the study of the Sampled Working Set algorithm within our environment. We derived an expression for the page reclaim rate under this algorithm, again based on the Independent Reference Model of program behavior. This algorithm, however, was observed to achieve its good performance at the cost of incurring uniformly higher page reclaim rates than those caused by the clock algorithm. Although not simulated, implications of a multiprogramming environment on our conclusions were discussed.

Chapter 6 reported on the design, implementation and measurement of the VMUNIX operating system. This system was derived from the UNIX operating system for the VAX-11/780 computer through the incorporation of paging. Since the VAX memory management does not support page reference bits, all of the results from the previous chapters were influential in the selection of the page replacement policy. Particularly, the high page reclaim rate due to the SWS policy would have resulted in prohibitively high fractions of the CPU cycles being used for handling the reclaim operations. In this chapter, we also discussed some of the other factors in choosing the global clock algorithm as the page replacement policy in the system. We observed that the interaction of the page replacement policy with the load control mechanism became increasingly important in our environment due to the possible collapse of system performance while trying to service the page reclaim rate generated by the multiprogramming load. Measurements obtained from a real work load showed that the paging system performed equally as well, if not better, as the swap system in addition to providing the obvious functional extensions. Further performance enhancements were achieved by an increase in the file and paging system block size and by the implementation of a simple prepaging mechanism.

Chapter 7 attempted to formalize some of the decisions and of the parameter selection procedures associated with the management of a global resource. The instance of the problem considered was the management of the free page pool in the VMUNIX system. We cast the problem as one of inventory control under a stochastic demand, so that results of previous studies could be used. To justify some of the simplifying assumptions made by the model about the demand process, we conducted a series of statistical tests on the trace data obtained from the VMUNIX system. Although only partially supported by the data analysis, the model parameters were solved under the assumption of independent exponential demands within fixed length intervals. We presented the procedures for estimating the various cost coefficients that are required for the model's specification and

concluded the chapter with a discussion of the implementation issues.

8.2. Conclusions and Topics for Future Research

Our study stopped short of extending the hybrid policies to remove the fixed size top restriction. As the simulation studies of Chapter 5 indicated, the variable size hybrid algorithms have very good performances conditional on the correct selection of the fixed top size. It appears that this drawback can be eliminated by dynamically varying the top size at the instants of page reclaims. A reasonable heuristic could be the observed frequency of page reclaims. Given a threshold (analogous to the parameter of the PFF algorithm), the top size could be increased by a small amount if the page reclaim frequency is observed to be above this threshold. Conversely, the top size could be reduced by a small amount if the observed frequency happens to be less than the threshold. This version of the algorithm could be characterized as the H_{PFF-WS}.

Some of the policies adapted in the VMUNIX system require more formal evaluation. Particularly, the decisions associated with the prepaging mechanism (such as the number of pages to prepage and where to place them) were resolved using intuitive arguments. The projected workload for the VMUNIX system consists of application programs for VLSI design and image understanding. It is suspected that these programs exhibit behaviors that are sufficiently different from those that have been studied during the page replacement policy selection phase. The design of algorithms to exploit these special programs can only be based on a better understanding of their behaviors through tracing their execution. It is unlikely that a new page replacement algorithm will be discovered that delivers uniformly good performance under all operating conditions (e.g., the normal time-sharing load as well as the dedicated use for the special programs mentioned). However, it is conceivable that good heuristics can be developed to allow a variety of replacement algorithms to be applied on a selective basis to different programs and perhaps to the same program at different times. A primitive form of this mechanism already exists as the system call which declares a process as being "anomalous", thus requesting FIFO replacement. The difference here is that a special behavior has to be signaled explicitly by the process and is not *recognized* by the system.

Perhaps the most rewarding future studies are those related to the material presented in Chapter 7. This chapter describes an initial attempt at formalizing the decisions associated with the free page pool management mechanisms of the VMUNIX system. As such, the model for the action is quite simple. The cost structures are restricted to linear forms and there is no explicit time delay due to ordering more inventory. The most severe simplifications, however, are those of the demand process. The independent and identically distributed assumptions for the per-period demand can be relaxed at the cost of rendering the model analytically intractable. However, there may be efficient numerical methods to solve for the model parameters even under the suitable generalization of independent and identically distributed intervals between demand points (i.e., a renewal process as the source of the demand). Any of these extensions should follow the evaluation of the model in its current form using trace data obtained from the live system. The current model, as simple as it is, might be found to produce costs sufficiently close to the stochastic optimum that there is no incentive to complicate it further. The criterion for introducing any new extension should be the net gain after the implementation overhead has been taken into account.