SLAC REPORT 244
STAN-CS-81-873
UC 32
(M)

VIRTUAL MEMORY MANAGEMENT*

RICHARD WILLIAM CARR
STANFORD LINEAR ACCELERATOR CENTER
STANFORD UNIVERSITY
Stanford, California 94305

September 1981

PREPARED FOR THE DEPARTMENT OF ENERGY
UNDER CONTRACT NO. DE-AC03-76SF00515

Printed in the United States of America. Available from National Technical Information Service, U.S. Dept. of Commerce, 5285 Port Royal Road, Springfield, Virginia 22161. Price: Printed Copy A11; Microfiche A01

*Ph.D. Dissertation

Acknowledgments

The author wishes to show his appreciation to all of his friends who have patiently endured the gestation of this thesis. To begin, I thank those who have contributed to the research:

John Hennessy, for his persistence, friendship, direction, and insight;

Forest Baskett, for his inspiration and example;

Ed McCluskey, for his patience and steady support and his knack of asking the embarassing question at the right time; and

Margaret Wright, for her ear, her shoulder, and her willingness to read and criticize this work when I needed it most.

This work was supported by the Computation Research Group of Stanford Linear Accelerator Center, Department of Energy Contract DE−AC03−76SF00515. I thank Jerry Friedman, Harriet Canfield, and all the others at SLAC who were exceedingly cooperative and helpful.

The Stanford Computation Center (later SCIP, now CIT, tomorrow ???) provided the incentive, opportunity, and environment to obtain the practical experience on which this research is based. I am particularly indebted to Mike Powell who helped me develop many of the ideas that appear in this work.

My friends in the C.S.D. have made this task eminently endurable if somewhat short of enjoyable. Particular thanks go to Tom Dietterich and Alfred Spector for their sanity, good counsel, and ability to restore confidence when all looked bleak.

A tribute is due to George Polya who once wrote: "The first rule of style is to have something to say. The second rule of style is to control yourself when, by chance, you have two things to say; say one first, then the other, not both at the same time."

My faithful Alto, #50#111#[Monterey], has been a delightful, if somewhat slow, contributor to this work. Even when the content of my writing was sorely lacking, the beauty of its form encouraged me to persevere. Thanks to the XEROX Corporation for providing these wonderful machines.

Among my long-suffering friends and family, my mother Lorraine, Shane Cortright, Linda Cahn, and, of course, Sasha deserve special awards for their love and support.

Finally, I want to thank Carolyn Tajnai for always having the right answers.

Table of Contents

List of Tables

List of Illustrations

Virtual Memory Management

Richard W. Carr
Computer Science Department
Stanford University

Abstract

This thesis studies methods of scheduling, memory management, and load control in a virtual memory computer system. It also studies the problem of modeling virtual memory systems. The work contributes new ideas and techniques in each of these areas. Finally, the thesis compares the two alternative classes—local and global—of virtual memory management policies.

In the area of scheduling, a *multi-level, load-balancing queue* is described; this mechanism is useful for maintaining good response time in interactive systems and for managing a central server in a distributed computer system. In the area of memory management, a new policy, WSCLOCK, is developed; this policy combines the natural load control of the WS (working set) policy and the simplicity and low cost of the CLOCK (approximate global LRU) policy.

Two contributions are made in the area of load control. First, a general *loading-task/running-task* (*LT/RT*) control alleviates congestion when multiple tasks are newly-activated at the same time. Second, a load control for the global policy CLOCK is presented; this adaptive feedback control uses information collected during the page replacement process to estimate the overall level of main memory commitment and, then, adjusts the multiprogramming level for best performance. This control employs exponentially-smoothed confidence intervals—a statistic that is also developed in the thesis.

Models to compare local and global memory management policies that are both accurate and efficient have been difficult to construct. The thesis 1) develops a trace-driven model of program behavior that greatly reduces the cost of simulation with a negligible loss of accuracy and 2) constructs a detailed simulation model of multiprogrammed virtual memory systems. The program behavior model is parameterized by measurements of real programs; it reflects the behavior of those programs at a very precise level. The system model is driven by a heterogeneous collection of program models that are typical of many computer systems.

The thesis uses the model to make a direct comparison of the WS, CLOCK, and WSCLOCK policies. The study indicates that neither of the classes of memory management policies are *inherently* superior. Disregarding the time spent executing the algorithms themselves (i.e., operating system overhead), no significant differences in system performance could be detected. On a practical level, CLOCK has considerably less overhead than either WS or WSCLOCK and, thus, represents the best alternative.

Chapter 1

Introduction

1.1 Introduction

The efficient use of main memory is a problem that has remained important since the development of the stored-program computer. When computer memories were small and very expensive, each programmer was encouraged to develop and use techniques that minimized memory use. This situation produced a number of elegant mechanisms such as stacks, hashing, and list processing. Sophisticated algorithms were developed to manipulate sparse matrices, sort large files, execute large programs and deal with many other problems made difficult by small memories. This environment also encouraged obscure and cryptic programming techniques, the use of low-level languages and other practices now considered harmful.

Today, things have changed. The cost of hardware has fallen, while the expense of producing and maintaining software has risen sharply. While the techniques developed in earlier times are still useful, there is a greater emphasis on clarity in programming to reduce the cost of developing and maintaining programs. In the majority of programs written today, complex programming techniques for conserving memory are rightly discouraged as uneconomical.

Computer systems with millions of words of main memory are now commonplace. While these memories cost less per word, their total cost is still substantial. The growth of main memory has been matched, step for step, by consumption of memory for larger and more complex undertakings. Thus, the problem has merely changed form. Instead of trying to save tens or hundreds of words in a memory of a few thousand words, our goal is to save a substantial portion of memories with millions of words. Instead of requiring each programmer to minimize use of memory, we now develop techniques for efficient memory allocation by the operating system. These techniques, even if costly to develop and maintain, will result in a continuing and substantial increase in system capacity and reduction in costs.

In most large computer systems the primary technique used by the operating system to manage main memory is *virtual memory.* In this chapter, we review, briefly, the basic elements of virtual memory and present the motivations and goals of virtual memory. For a complete description of the detailed mechanics of virtual memory, the reader should consult [DENN70].

1.2 Virtual Memory Overview

Due to the large variety of architectures and the range of sizes of modern computer system, it is difficult to select a *typical* virtual memory computer system. The model we present here is based roughly on the IBM 370 computer, but not on any particular operating system. To be quite specific, neither of IBM's standard operating systems, MVS and VM/370, have had a predominant influence on the model. Some virtual systems, such as MULTICS, provide more sophistication than the one presented; some, such as the DEC VAX 11/780, provide less. Most virtual systems on large-scale computers are quite comparable to the one presented here.

The three basic elements of virtual memory, as illustrated in Figure 1.1, are the *virtual address space*, the *main memory*, and the *auxiliary memory.* Each task's program and data are contained in a virtual memory address space, which appears to be a private and dedicated area of the computer's main random-access memory. In reality, the address space is only an *illusion* of physical memory that is created by the coordinated action of specialized hardware and operating system software. In general, the illusion is not detectible by the task.

To the task, the address space is simply a continuous array of words or bytes, each with a unique virtual address. To the virtual memory hardware, the address space is partitioned into equal-sized blocks called *pages*, while the main memory is partitioned into similarly sized blocks, or *frames.* A secondary or *auxiliary* memory, which has a much longer access time (and is much larger and cheaper) than main memory, is also organized into page-sized blocks called *slots.* Conceptually, one should think of a page as a collection of information, and frames and slots as places to store that information.

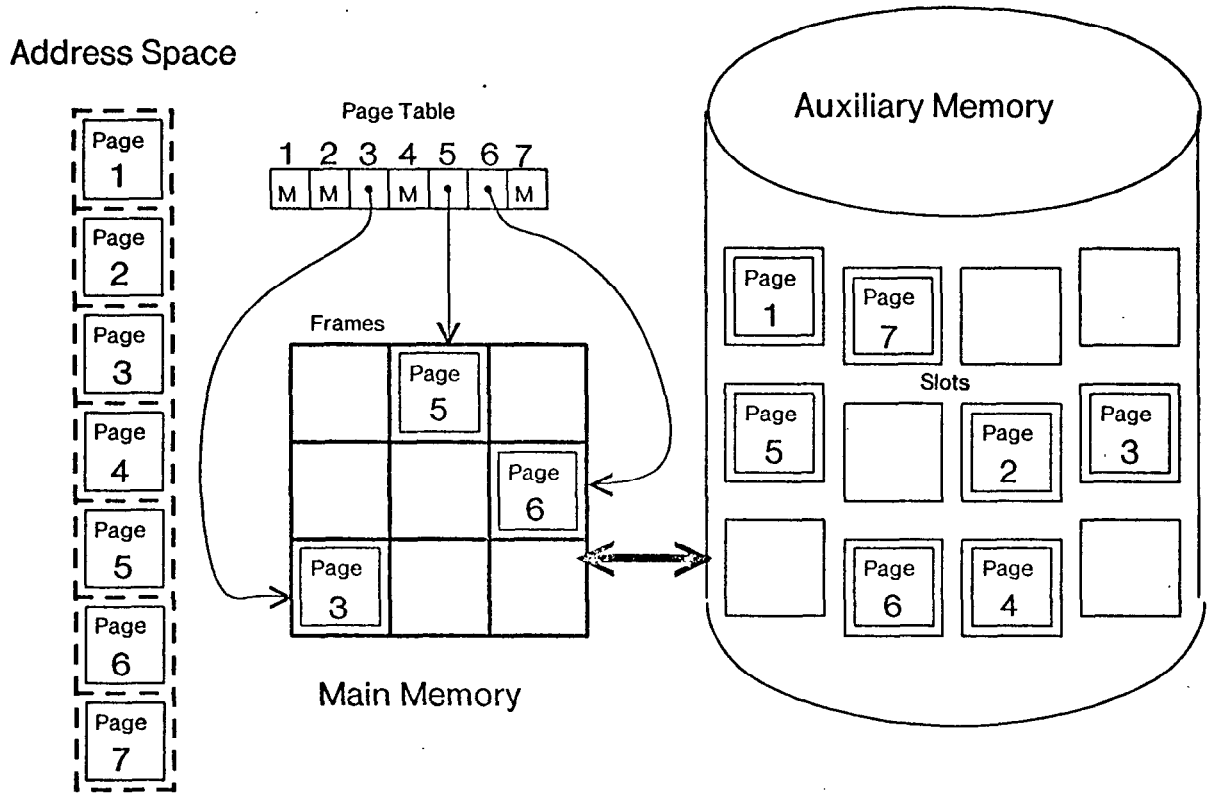


Figure 1.1 - Virtual Memory Elements

Each page of each task's address space is assigned an auxiliary memory slot. Some of the task's pages may also be located, or be *resident*, in main memory frames. As a program executes, it accesses, or *references*, a sequence of virtual addresses in the address space. If the page containing a virtual address is resident in main memory, the information at the virtual address is properly obtained or updated. If a referenced page is not resident, it is *missing*, and a *page fault* occurs; the page must be moved from auxiliary memory to a frame before execution can proceed.

At any given time, the pages that are placed in frames are known as the task's *resident set.* The *page table* is a data structure that identifies the resident set pages and the frames in which they reside. Missing pages are identified by a particular value (M). At any time, a page may removed from the resident set by placing an M in the associated element of the page table; the page may then be *replaced* by moving it to auxiliary memory, if necessary, and assigning its frame to hold some other page.

1.2.1 Hardware

The essential hardware element in a virtual memory computer is an *address mapping device* which is logically located between the processor and the main memory, as shown in Figure 1.2. The device translates each virtual address in the address space, using the page table, to a *real address* in main memory. Special techniques, analogous to memory caching, make the delay required for the translation reasonable, if not negligible. When a page is resident, a reference to it is translated successfully. If the page is missing, the address mapping device interrupts execution to signal the operating system that a page fault has occurred.

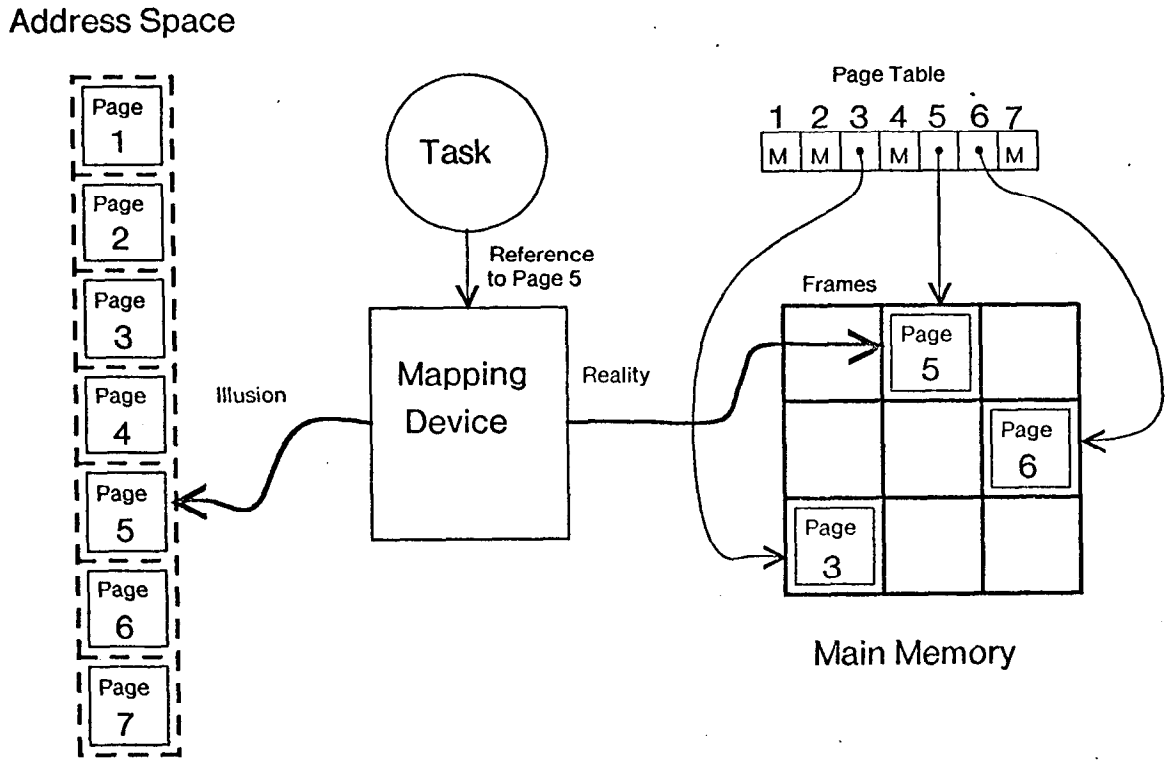


Figure 1.2 - Virtual Address Translation

A minor, but important, hardware element in most virtual systems is the memory activity indicators known as the use-bit and the dirty-bit. These bits are in a separate memory, one pair for each main memory frame, and are updated in parallel with each successful reference. A

frame's use-bit is set each time the frame is referenced; the dirty-bit is set whenever the frame is updated. These bits are examined and cleared by the operating system: the use-bit indicates a page is active and, most likely, should not be replaced; the dirty-bit indicates that the page must be moved to auxiliary memory before it can be replaced.

The use-bit and dirty-bit are not absolutely required to support virtual memory. Indeed, they are not found on the recently introduced DEC VAX 11/780. They are, however, essential for the modern virtual memory management methods described in this thesis; the VAX operating system uses fixed-space memory allocation and a FIFO replacement algorithm, both of which are known to be inferior. [DEC78]

The auxiliary memory is commonly a physically-rotating magnetic disk or drum device, with either a moving or a fixed access-head assembly. Other technologies used for auxiliary memory include electronically-rotating devices, such as magnetic bubbles or charge-coupled devices, and random-access memory.

1.2.2 Software: Operating System

To manage virtual memory, the operating system creates and maintains the page table for each task and processes each page fault when a missing page is referenced. To process a page fault, the system selects a main memory frame to hold the missing page and performs an I/O operation to move the page from its slot to the frame. During this time, the task is blocked. The most problematical element of this process is selecting a frame to hold the missing page. If there are any frames which have not been previously allocated, any one of them can be chosen with equal effect; each frame is physically identical except for its real address, which has no importance. When every frame is allocated, which is the typical case, it is necessary to replace a resident page with the missing page. The method used to select a page to replace is the *page replacement algorithm.*

The process of moving pages between frames and slots is known as *paging*. Two elements of paging are the *loading policy*, which moves pages from slots to frames, and the *cleaning policy*, which moves pages from frames to slots. As indicated above, each page fault requires the operating system to move, or *load*, a missing page from its slot to a frame. This is known as a *demand-paging* operation. Certain economies may be realized if many pages are loaded at the same time and the operating system may load, or *pre-page*, additional pages in anticipation that they may be referenced in the future. Systems that use only the first type are known as demand-paging systems, while pre-paging systems use both types.

When a page is loaded the system clears the frame dirty-bit. As long as the page is not changed, the dirty-bit remains unset and the auxiliary memory slot contains a good copy of the page; the operating system can replace it simply by loading another page in its place. If the page is changed and the dirty-bit is set, the system must clean the page, by writing it to the slot, before the page is replaced. Similar to the situation with page loading, pages can be cleaned only when they have been chosen for replacement (*demand cleaning*) or can be cleaned in advance (*pre-cleaning*).

Finally, virtual-memory systems require some form of *load control*. Although there is no physical upper bound on the number of concurrently-executing tasks, each additional task reduces the size of the average resident set, and increases the rate of page faults. Contention for memory eventually produces a condition known as *thrashing* in which page faults are the predominant activity and very little useful work is performed. To prevent this situation, but still gain the maximum benefits of multiprogramming, a load control mechanism is required to determine the proper number of concurrently-executing tasks and maximize system performance.

1.2.3 Software: Tasks

The programs executed by tasks are expected to be ignorant of virtual memory and to assume that the address space is a real, physical memory allocation. Rather than depending on any explicit programming rules, virtual memory is effective due to the naturally-occurring property of

locality. The typical program tends to reference a subset of its address space for significant intervals and, during these intervals, only the necessary subset need be resident for efficient processing of the task.

Although there are programming techniques that improve a program's locality and, therefore, make it possible to process the program more efficiently, we view these as useful only in special instances. Since the ratio of software costs to hardware costs is constantly increasing, any requirement to program for efficiency in a virtual memory environment will become less and less useful. Virtual memory should simplify the task of programming, not complicate it.

1.3 Motivation and Goals of Virtual Memory

There are two fundamental motivations for virtual memory:

1) To enable processing of a task whose program and data are larger than the real main memory, and

2) To increase the performance of a multiprogrammed system by increasing the number of concurrently-executed tasks.

The first motivation is obvious. Anyone familiar with computer programming can appreciate the effect a main memory limit has on program design, complexity and execution time. In some cases, such as the classical example of sorting (see [KNUT73]), the inability to hold all data being processed in main memory complicates the programming enormously. In other cases, such as language translation, the size of the program may, in a limited-memory environment, require the division of processing into passes or overlays. Virtual memory can increase the apparent size of main memory and relieve many of these difficulties.

In earlier days, when main memory was extremely limited, this first motivation was quite important and was the prime factor in the development of virtual memory. It is still important in the development of virtual memory for limited-memory microcomputer systems, but it hardly

applies to today's large systems which rarely execute any single program that is larger than real memory. Instead, the prime motivation is to execute more programs than is possible without virtual memory. Of course, virtual memory does permit most installations to grant larger memory allocations to each individual program, but these are commonly limited to some fraction of the real memory size.

The benefits of virtual memory in a multiprogrammed system arise in many ways. First, it eliminates memory fragmentation that occurs when variable-sized contiguous memory areas are allocated and deallocated when tasks start and stop. Second, dynamic memory requests, during task execution, can be processed without the need to allocate the maximum, high-water-mark, memory extent for the life of the task. Third, the locality of references permits the system to remove inactive areas of memory, temporarily, and reduce the average main memory required to execute the program.

The first two benefits can be obtained without additional overhead and with a minimum of complexity; they alone would probably justify the cost of the virtual memory hardware. The third benefit, however, is the most important, but it is obtained only at the cost of (1) a considerable increase in the complexity and manageability of the operating system, (2) increased overhead to process page faults and other virtual memory functions, (3) increased task blocking and task switching when page faults occur, and (4) increased I/O traffic contention due to paging I/O. Each of these costs is significant. Each can be affected by the choice of strategies to manage virtual memory.

Virtual memory is particularly useful in interactive systems. An interactive system seeks to support a large number of concurrently-executing tasks and these tasks must be moved frequently between main and auxiliary memory. A virtual system loads only the address space pages required to process each interactive request and unloads only the pages which have been changed; furthermore, a virtual system can placed the needed pages in any collection of frames. The alternative to virtual memory, *swapping*, requires the movement of the entire address space and must allocate a contiguous memory area to hold it.

1.4 Thesis Overview

The Primary Goal

Research in virtual memory computer systems has been substantial over the last two decades. In October, 1961, the *Communications of the ACM* contained seven articles on the subject of virtual memory management, including a description of ATLAS, the first virtual memory computer. Since then, thousands of published papers, doctoral theses, conference reports and unpublished studies have addressed the subject of virtual memory. In spite of this prodigious and sustained effort, understanding of how virtual systems work, or ought to work, is still incomplete.

A single topic has received much of the attention in virtual memory research: the *page replacement algorithm.* In the first decade, slow, but steady, progress was made in both the sophistication of replacement algorithms and in the ability to model and evaluate them. Two major results were established: (1) algorithms that use past program behavior to predict future behavior are superior to those that do not and (2) replacement algorithms that vary the memory allocation of each program according to its needs result in better resource utilization than fixed-space algorithms. [BELA69] These results were obtained through the analysis of a uniprogram model, but have clear applicability to a multiprogram system.

In the past 10 years, however, there has been little visible progress in improving the theory of page replacement algorithms or in developing methods to analyze them. This situation is due, largely, to the complexity of multiprogram models.

Modern memory management policies comprise both a page replacement algorithm and a load control mechanism, as well as a few subsidiary techniques. The most capable policies can be divided into two classes: *local* and *global.* Both of these classes vary the amount of memory allocated to each task according to its needs using measurements of past program behavior; both classes adjust the number of executing programs to maximize performance. An effective comparison of these two classes is absent from the research literature; due to the complexity of multiprogram models, the two classes have never been quantitatively compared.

Numerous models, both analytic and simulation, have been constructed to evaluate a system using one class of algorithm or the other, but various restrictions in the modeling process have prevented a head-to-head comparison. There have been indirect attempts to establish the general superiority of the primary local policy, Working Set (WS). Studies have been made to show that WS (1) has an effective natural load control, (2) is very close to a semi-optimal replacement strategy, and (3) can make substantial improvements in the performance of real systems. None of these studies, however, makes a direct or conclusive argument that WS is superior to global policies.

The primary goal of this thesis is to develop an efficient virtual memory computer system model that will make possible an accurate, unbiased, and quantitative evaluation of local and global memory management policies.

The Plan of the Thesis

In Chapter 2, we describe the central management policies of a virtual memory computer system. We discuss task scheduling disciplines, both local and global policies of virtual memory management, and load control mechanisms. We describe a number of useful new algorithms and practical techniques that were discovered and investigated in the course of this research. In particular, we describe WSCLOCK, an algorithm that is a simple, efficient, and effective implementation of the WS policy.

In Chapter 3, we review many of the previous efforts to model and evaluate virtual memory systems in general and paging algorithms in particular. From this review, we conclude that present analytical models are not likely to permit the direct comparison of local and global memory management policies. An *effective* comparison of these policies requires both an accurate representation of program behavior and a detailed model of the dynamics of memory allocation. An analytical model that properly characterized the differences between local and global memory allocation would be inordinately complex and intractable.

The alternative to analytical modeling is simulation. Simulation models are capable of representing the complexity of program behavior and dynamic memory allocation accurately, but they tend to be difficult and time-consuming to construct and validate and, in addition, are usually very expensive to run. A number of simulation models are described in Chapter 3; some of these reduce the expense by using simple stochastic models of program behavior. Unfortunately, these simple models are inaccurate and make it difficult or impossible to model the differences between local and global policies. An alternative to the stochastic program behavior model is a trace-driven, reference string model, which is more accurate but is very expensive to use in a full-scale system model. Chapter 3 describes two methods for "filtering" reference strings to make them more compact and efficient to use; these two methods introduce more inaccuracy than is justified by their cost savings, but this basic approach is the appropriate method for studying the complex behavior of page replacement algorithms.

In Chapter 4, we describe a new method for filtering reference strings that realizes an accurate representation of program behavior and results in a substantial reduction in the string size and in the cost of using it to model programs in a simulation model. Chapter 4 also describes a computer system simulation model that is designed to permit an direct comparison of a large variety of memory management policies and other system control strategies. In addition to modeling various practical local and global algorithms, the simulator also models an unrealizeable look-ahead algorithm, VMIN. The model demonstrates the feasibility of using simulation to model both real and theoretical virtual memory management policies accurately and inexpensively.

In Chapter 5, we study the efficiency and accuracy of the simulation model and then use it to investigate the performance of memory management policies. First, we make an intensive study to tune the theoretical "exact" WS policy for best performance. Then, the exact WS policy is compared to practical algorithms that approximate the WS policy and to variants of WS. A similar process is carried out for a global policy and the resulting "best" performances are compared. We conclude that, ignoring the time spent in the algorithms themselves, the global

replacement algorithm performs about as well as the WS algorithm. A final study indicates that the global policy results in a substantial reduction in operating system overhead costs.

In Chapter 6, we review the conclusions of our experimental results and the other contributions of this thesis.

Chapter 2

Virtual Memory System Management Methods

In this chapter, we set forth a collection of processor and memory allocation methods that we wish to study with a virtual memory computer system model. These methods include the implementation of policies, the design of algorithms, the use of heuristics, and the tuning of parameters. Some of the issues presented here, such as scheduling disciplines and replacement algorithms, are described and evaluated in standard operating system texts. [COFF73, HABE76, SHAW74] We also introduce a variety of methods that are general in nature, but are rarely presented in texts, since they represent practical techniques whose effects are not easily analyzed or understood.

This chapter is divided into three sections. Section 2.1 describes scheduling of both processor and memory. We present numerous scheduling methods and introduce a new method that combines the features of load balancing and the multi-level queue. Section 2.2 describes virtual memory management, particularly emphasizing local and global memory management policies, and also treating a few neglected issues, such as page cleaning strategies. Local policies are represented by the working set (WS) policy. Global policies are represented by the global least-recently-used (LRU) approximation algorithm, CLOCK. A special subsection describes the generalized CLOCK algorithm, which is shown to be a simple and elegant mechanism that coalesces and simplifies virtual memory management for both local and global replacement policies. We describe a new memory management algorithm, WSCLOCK, which uses the CLOCK algorithm to approximate the WS policy simply and efficiently.

Section 2.3 describes load control as the interface between scheduling and memory management. First, we describe a new load control *LT/RT* that improves both local and global policies. For local memory management policies, we present the standard WS load control. For global memory management policies, we present a new adaptive load control based on the CLOCK

algorithm and on exponentially-smoothed confidence intervals, which are also developed in this section. Finally, we discuss policies of choosing a task to demote when the load control determines that memory is overcommitted.

2.1 Scheduling

A general schema for task scheduling is illustrated in Figure 2.1. As tasks arrive from some input source, they may be placed in an *admission queue* and required to wait for further processing. Once admitted, a task waits in a *memory allocation queue*, commonly called the *ready queue*, until sufficient main memory is available to load the task. The task is then *activated* and moves to the *processor allocation queue*, or *active queue*, to await execution on the processor.

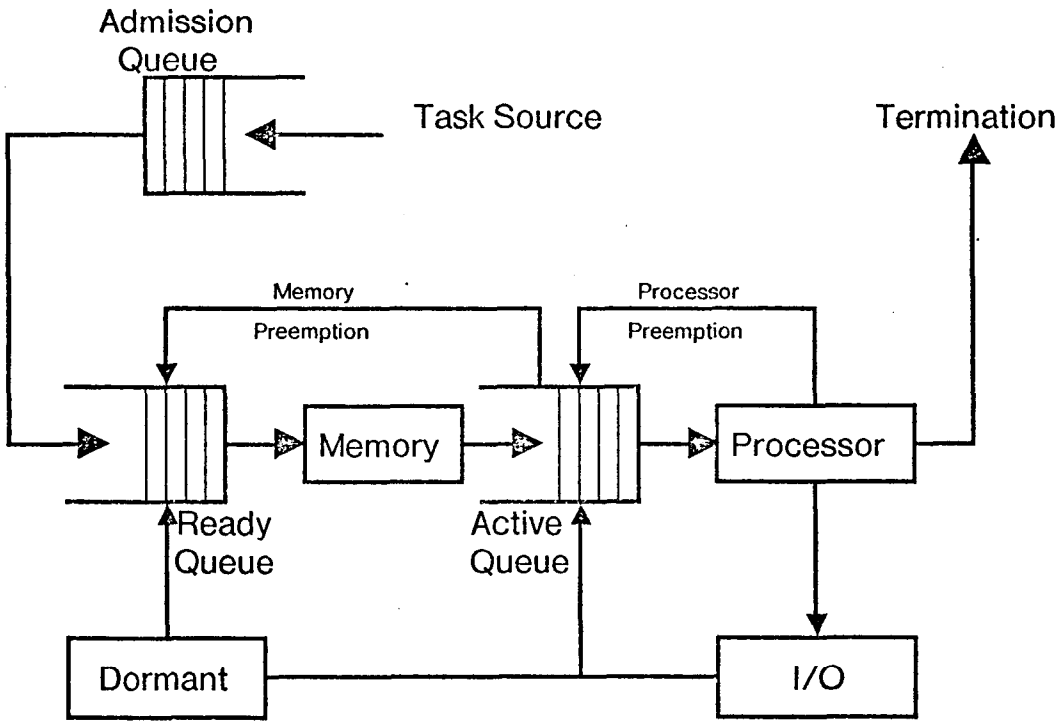


Figure 2.1 - Task Scheduling

Each task in the active queue is given a turn using the processor until the task requests an I/O operation or the scheduler gives another task a turn; in either case, the task usually remains in the active queue. Tasks are *deactivated* for various reasons; these are removed from the active queue, relinquish their memory allocation, and are placed in the ready queue. Tasks performing long-waiting operations (e.g., terminal interactions) are removed from the active queue and placed in a *dormant* state until the I/O operation is completed, after which they join the ready queue.

2.1.1 Admission Scheduling

In batch processing, the number of tasks that are processed concurrently is often limited. There is little to be gained by admitting tasks once the processor and I/O system have become saturated. Even if additional tasks do not cause degradation, withholding tasks from processing can provide greater scheduling flexibility. Within the normal objectives of the computer service center, tasks may be admitted with higher or lower priority based on various policies:

- System maintenance functions, such as the timely repair of damaged information, may require immediate admission.

- Priority may be granted based on a user's willingness to pay for it or on some other budgetary control scheme.

- Shorter tasks may be given priority over long tasks in order to maximize the task completion rate and please the maximum number of users.

To maximize system capacity, tasks can be admitted on the basis of their predicted resource requirements. A mixture of I/O-bound and processor-bound tasks will make better use of the system as a whole. Tasks with special requirements, such as exclusive access to a resource, can be delayed until the requirement can be met. Tasks requiring operator action, such as a tape mount, can be admitted at a rate in keeping with the ability of the operator to respond.

In interactive systems, the number of tasks admitted is limited by the number of terminals that can be connected to the system. However, since terminals are relatively inexpensive and it is

desirable to make them conveniently available to those who will use them, it is common to obtain and distribute more terminals than the system can support at one time without being overloaded. It may be necessary to impose an artificial limit on the number of active terminals. It is a disservice to all users to permit more terminals to be active than can be supported with good response time; good service to a few users is preferable to poor service to many users.

Within the constraints of policy, operational, and service objectives, tasks are usually admitted on a first-come first-served basis. Once admitted, tasks do not normally rejoin the admission queue, even if higher priority tasks arrive.

2.1.2 Memory Scheduling

Once placed in the ready queue, a task must receive a memory allocation before it can be executed. Here we distinguish conventional batch and swapping systems from virtual memory systems. In a conventional system, the decision to admit a task for processing and the decision to allocate memory for the task are made at the same time. A task is considered both admitted and activated when an explicit region of memory is reserved for the task; the ready queue is superfluous. In a swapping system, a task may be admitted to the system before a region of memory is available for it, but an explicit memory allocation is required before the task can be activated. In a virtual memory system, memory allocation is made implicitly and any task can be activated at any time. The system allocates memory as the task demands it, one page at a time.

A local memory management policy budgets or *commits* an explicit number of memory frames to each active task, without selecting the specific frames. This budgeting process is dynamic and commitments are frequently altered as each task's apparent needs change. A global policy does not have an explicit budget for each task; it commits all of memory to a set of active tasks without pre-specifying the number of frames committed to each. In our model of a virtual system, the scheduler determines only the *ordering* of tasks in the ready queue awaiting allocation of main memory. The load control decides when tasks should be activated, based on an implicit or explicit estimate of the amount of uncommitted memory.

Both the scheduler and the load control make decisions to deactivate tasks. The scheduler deactivates a task when an allotment of processor time (a *time-slice*) has been consumed or the task becomes dormant. The load control deactivates, or *demotes*, a task when it determines that memory is overcommitted. Deallocation of memory, when a task is deactivated, is implicit since task pages remain resident until they are replaced by other tasks. Deactivation reduces the level of memory commitment and increases the probability of activating a new task.

Methods used to schedule the tasks in the ready queue are described below. A full discussion of load control, which coordinates scheduling and memory management is deferred to Section 2.3, following the description of memory management. Note that a task's main memory commitment requirement is not a factor in scheduling the task for memory allocation. Instead, the goals of sharing the processor equitably, maximizing throughput, minimizing response time, or some combination of these are used to order the tasks in the ready queue.

2.1.3 Processor Scheduling

Once a task has been activated, it is given turns using the processor according to one of the queueing disciplines described below. As a task executes, it demands pages of memory and its implicit memory commitment is converted to an explicit allocation, which is the *resident set.*

2.1.4 Scheduling Methods

First-Come First-Served Scheduling

First-come first-served (FCFS) is the simplest scheduling discipline to implement. It orders tasks by their time of arrival and gives service to the oldest task in the queue. FCFS is satisfactory if all tasks are relatively I/O-bound and they all receive reasonable service. When a processor-bound task attains a higher priority than other tasks, it will monopolize the processor and starve the tasks below it. Not only is it unfair for one task to monopolize the processor, there is also a reduction in system throughput because of reduced processor-I/O concurrency.

Round-Robin Scheduling

The round-robin (RR) discipline attacks the fairness problem directly and the throughput problem indirectly. When a task joins a queue, it receives service in its normal turn for one *quantum, Q*. If a task consumes its entire quantum, it is preempted and placed at the end of the processor queue. If a task blocks for an I/O request, it retains it place in the queue along with the unused portion of its quantum. The scheduler simply processes the first unblocked task in the queue. When a processor-bound task is executed, it consumes its quantum and is placed behind I/O-bound tasks; thus, it can monopolize the processor for at most Q before all other tasks are given a turn. When an I/O-bound task is unblocked, it usually receives service at a high priority, allowing it to make an additional I/O request and maintain a high processor-I/O concurrency.

This simple mechanism may be all that is required to achieve both fairness and good throughput. Of course, I/O-bound tasks can consume an entire quantum over several processor-I/O sequences and be moved behind processor-bound tasks. This situation soon corrects itself and, on average, tasks are ordered in direct relation to their I/O boundedness.

Q must be chosen to balance system objectives. As $Q \rightarrow 0$, the scheduler will approach the *processor sharing* discipline, where each of n active tasks has, apparently, a processor with $1/n^{th}$ the capacity of the real processor. Unfortunately, task switching overhead becomes an limiting factor as $Q \rightarrow 0$ and the number of switches increases. As $Q \rightarrow \infty$, the scheduler approaches FCFS.

Chanson and Bishop [CHAN77] studied various methods of varying the quantum size dynamically to balance performance criteria. For example, the *90% rule* adjusts the quantum so that 90% of all interactive requests are completed in a single quantum. Thus, trivial requests (by definition, the smallest 90% of all requests) receive high priority while keeping the quantum as large and efficient as possible.

Load-Balance Scheduling

Load balancing is a more complex mechanism for achieving high processor-I/O concurrency. The relative processor-I/O use of each task is measured and used to predict its future behavior. Tasks which are predicted to be I/O-bound are scheduled before processor-bound tasks. A common goal of load-balancing is to execute tasks which will have the shortest processing time until the next I/O request. The shortest-processing-time-first (SPTF) queueing discipline is known, analytically, to achieve higher throughput than FCFS or RR [COFF73]. It is also insensitive to task-switching overhead, since a switch occurs only when a task blocks.

Future processor-I/O use is usually assumed to be the same as the past behavior. Past behavior can be estimated using an historical average over the life of the task [MARS69] or over the last quantum [RYDE70], using a moving average [WULF69], or using an exponentially-smoothed average [SHER72].

Sherman et al. [SHER72], use a trace-driven model to compare load-balancing (which they called *dynamic prediction*) methods with RR scheduling. To obtain good results with load-balancing, they show that it is necessary to bound the quantum size because bad predictions will keep I/O-bound tasks waiting and reduce throughput. This study shows that RR scheduling with a small quantum is almost as good as the load-balancing algorithms, but only if the task switching cost is low.

The Multi-Level Queue

The multi-level queue is typically found in interactive systems. It was first reported in 1962 by Corbató et al. [CORB62], in a description of the CTSS scheduler, and has been used in Multics, VM/370 and other systems..

A special case of scheduling arises in interactive systems, which process many small, *trivial* requests and fewer large ones. Although the large requests can often be processed more efficiently (i.e., with less capacity lost to overhead functions), good response time for trivial

requests is usually the more important goal. One can view the interactive user as a slow, but expensive, I/O device which should be kept as busy as possible.

From the scheduler's viewpoint, the hallmark of an interactive system is the need to process many more tasks than can reside in main memory and, consequently, the need to make many more memory allocation and deallocation decisions. If a simple RR discipline is used to move tasks between the memory and processor queues, there will be a need to use a large quantum to keep the rate of memory allocations acceptable. Unfortunately, a large quantum is not compatible with the desire to maintain good response to trivial commands, since any non-trivial request will delay trivial ones that arrive after it.

The multi-level queue is a mechanism to order tasks by the triviality of their requests. Since it is not practical to predict which requests will be trivial, it is assumed initially that every request is trivial, and the non-trivial ones are sorted out as soon as possible.

The multi-level queue is a set of $k-1$ FIFO task queues, L_1, L_2, ..., L_{k-1}, and a RR queue, L_k, as illustrated in Figure 2.2. Each queue has an associated quantum, Q_i, such that $Q_i \leq Q_{i+1}$. When a terminal user makes a request, the request is assumed to be trivial and the task is placed in L_1 where it receives a Q_1 processing quantum before all tasks in L_2, ..., L_k. If the task does not complete the request, it moves to L_2 where it receives an Q_2 quantum before tasks in L_3, ..., L_k, and so on. If the task does not complete the request by the time it moves to L_k, it shares the available processor time with the other L_k tasks in a RR discipline.

A common formula for choosing the Q_i is $Q_i = n\ Q_{i-1}$ (or $Q_i = n^{i-1}\ Q_1$). Q_1 should be sufficient to process the longest trivial request (see the 90% rule above). The choice of the Q_i represents a trade-off between the gain in efficiency when the quantum is large, and the degradation of response for non-trivial tasks. A common choice is $n=2$ [CORB62] although the practical effect of the parameter (or other distibutional formulae) is not well understood [CHAN77]. Adiri derives analytical formulas for expected processing time as a function of the Q_i in a queue with an infinite number of levels [ADIR71]. The analysis is complex and not immediately applicable to the problem of chosing the Q_i.

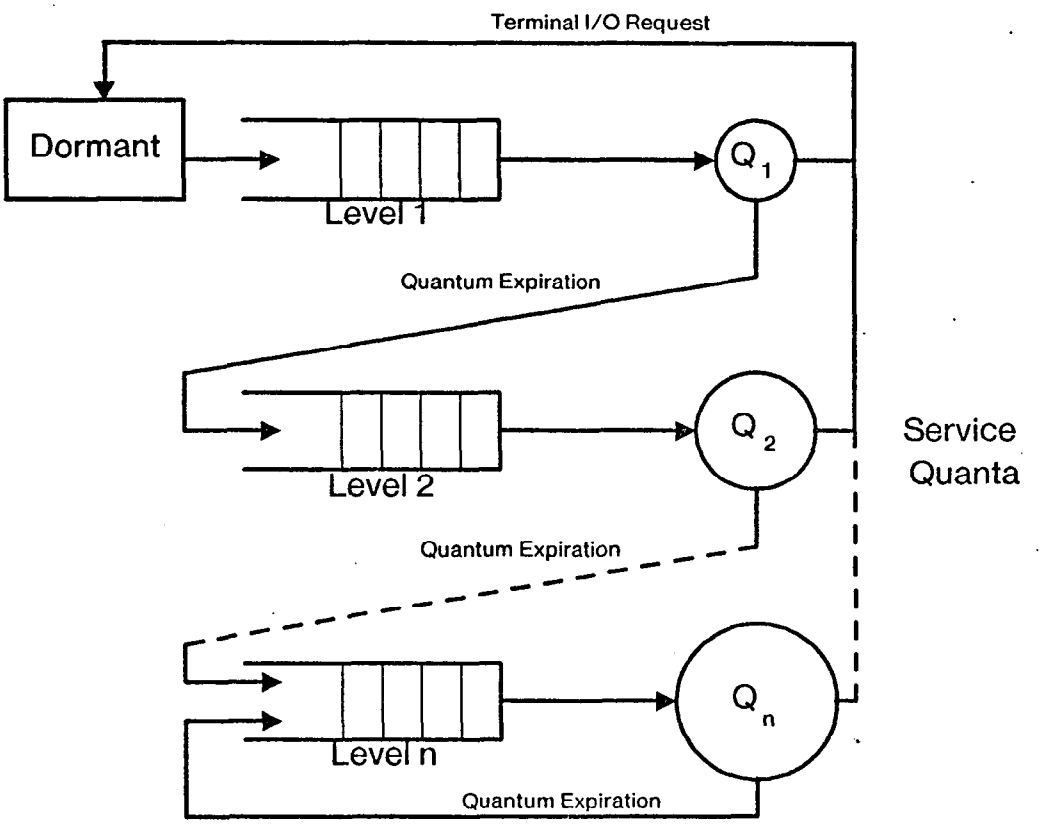


Figure 2.2 - Multi-Level Queue

In the CTSS system, the initial level for a task was determined by its memory allocation so that the effort to load the task and the initial quantum were commensurate. This refinement has not survived in current descriptions of the multi-level queue [COFF73, CHAN77].

A Multi-Level Load-Balancing Queue

The major advantage of interactive systems is the ability for the terminal user to interact with tasks for relatively short processing requests. Editing, debugging, database inquiries and the like are performed much more rapidly in an interactive environment. In modern systems, however, interactive access may be the primary or sole means of processing all types of work, including work which has the characteristics of batch tasks. The multi-level queue of the previous section

makes the implicit assumption that most I/O operations are terminal transmissions and that most requests are trivial; it does not address the problem of leveling the load when there are a significant number of non-trivial requests or when some tasks make extensive use of faster I/O devices..

To complicate the problem further, some *terminal* transmissions may have characteristics similar to fast (e.g., disk) I/O operations. If a task is writing a text file, a line at a time, to a high-speed display device, each line may require only milliseconds to transmit. Classifying such an operation with terminal I/O's that involve true human interaction, which usually take many seconds, will lead to two problems. First, the task will remain at high priority even though it is using a significant portion of the processor, slowing the response to other, more trivial, requests and starving tasks with moderate processing requests. Second, each terminal I/O usually indicates that the task is dormant and should relinquish its memory allocation and be reloaded when the I/O completes; a rapid sequence of these interactions will severely congest the system's capacity to load tasks.

This problem is growing worse with the growth of distributed processing. Small processors are often connected to large central processors that provide services such as large, cheap, file storage; the small processor signs on to the large processor as a terminal and communicates with it over a high-bandwidth link to send or receive information at speeds comparable to disk access times. At other times, the small processor may act as a pass-through interface to support human interaction with the central processor. Thus, the central processor cannot predict the rate at which the small processor will make requests, or how trivial they are likely to be. The simple multi-level queue does not discriminate the infrequent terminal interaction which should be given high priority from the frequent one that should be treated like a disk access.

The capability for an individual task to overlap processing with I/O operations is useful, but further complicates the scheduling problem. A user who wishes to use a large amount of processing time can "cheat" by performing terminal I/O operations while the task continues to execute and, consequently, keep the task in a higher level queue.

A solution to these problems is to combine the features of load-balancing and the multi-level queue. In the ordinary multi-level queue, a task is forced to lower priority levels as it consumes processor time and is catapulted to the highest priority level when it performs a terminal I/O operation. To incorporate load-balancing, the type (i.e., terminal vs. disk) of each I/O operation is ignored; only its duration is considered. In addition to the quantum, Q_i, each queue level has an associated *waiting-time quota*, W_i, such that $W_i \geq W_{i+1}$. When a task performs an I/O operation and waits for a time, w, the task becomes entitled to join the i^{th} level queue if $W_{i-1} > w \geq W_i$. By convention, $W_0 = \infty$. If a task is already at a higher level, it is not moved.

As with ordinary load-balancing, the waiting time, w, can also be calculated by averaging multiple observations of past behavior. Depending on the method used to calculate w, the W_i are chosen to discriminate, and properly order, tasks according to their predicted processor demand. For example, one value of W_i might discriminate between terminal I/O's which involve human interaction and those which do not; another W_i might discriminate terminal and disk I/O's.

This mechanism has considerable robustness. A task can join the first-level queue at most once every W_1 time units and, thus, its maximum processor use in that queue is Q_1/W_1. By setting W_1 large with respect to Q_1, good response to trivial requests is guaranteed to users who "think" for at least W_1 time units. Furthermore, the mean number of tasks at the highest level is limited by the W_1/Q_1 ratio, and service for tasks at the next level can be similarly guaranteed, but at a proportionately longer response time. Processor-bound tasks are given poorer service but receive large quanta and execute efficiently when they do receive service. A proper setting of the Q_i and W_i, relative to the number of tasks, will guarantee some service to the lowest priority tasks while maintaining good response for higher (i.e., trivial request) tasks.

Preemption Policies

There are, essentially, two very different types of preemption to be considered. The first, *processor preemption*, merely halts processor use by the currently executing task so that another task can receive its share of the processor. If tasks are ordered by priority, the scheduler can preempt one task whenever a higher task becomes ready to execute. Processor preemptions

typically consume only a small amount of processor time to save the state of the task and initiate the execution of another. Given their low cost, they are particularly useful for ensuring an equitable distribution of processor time among active tasks, and to perform load-balancing.

The second type, *memory preemption*, requires, in effect, the unloading and deallocation of main memory for one task and the subsequent allocation and loading of memory for another task. In demand paging systems, the actual switching cost for memory preemption may be low, but there is a high implicit cost for processing page faults, replacing pages, performing paging I/O operations, and incurring task-block delays. Much of the time required to load and unload tasks can be overlapped by processing of other tasks, but each page fault requires an appreciable amount of processor time and I/O capacity to resolve.

A memory preemption is an expensive operation and should occur infrequently. In particular, it may be a poor policy to preempt a task from memory when a higher-priority task becomes ready. This situation arises in a system where low-priority batch tasks compete with high-priority interactive tasks. When a batch task is activated, it will require a non-trivial amount of time and effort to load its resident set; if it is often preempted during this process, or shortly thereafter, by interactive tasks, much work is lost and throughput is affected adversely. It is often preferable to accept a small degradation in response time to permit activated tasks to complete their allotted quanta before being preempted.

2.2 Virtual Memory Management

2.2.1 Memory Management Policies

We consider only modern, automatic, variable-space, memory management policies. For a survey of some historically significant, but poorer performance, policies, the reader should consult Denning and Graham [DENN75c]. Among modern memory management policies there is a sharp division between local and global policies.

First, we should clarify the common attributes of these policies. Both local and global policies attempt to guage the locality of active tasks and vary the size of each task's resident set to execute the task efficiently, but without wasting memory. Both policies control the number of active tasks to strike a balance between underutilization and overcommitment of memory. Both policies have practical implementations on conventional virtual memory computers.

A local policy has three distinguishing principles:

1) The rule or algorithm that estimates each task's locality, or working set, is based solely on the behavior of the task itself.

2) The memory allocation policy ensures that each active task is granted sufficient memory to hold its working set.

3) The load control finds the maximum multiprogramming level (the number of active tasks) that is consistent with the first two principles.

The three principles form an integrated whole; the load control is a natural product of the working set estimation and memory allocation algorithms. The major drawback of local policies is the cost of estimating each task's locality. The commonly-used working set scan (described in the next section) must be invoked at reasonable intervals and has a non-trivial computation overhead.

A global policy selects a set of active tasks and allocates memory to them as if they were a single composite task. In some sense, a global policy is a fixed-space allocation policy applied to the composite task. The benefits of variable-space allocation are obtained by adjusting the number of tasks in set and by altering the partitioning of main memory as tasks expand or contract their localities.

Since there is no direct measurement of each task's locality, the natural WS load control cannopt be used for a global policy. Some global systems operate satisfactorily with a fixed multiprogramming level or static estimates of each task's locality. An adaptive global load control

estimates the level of memory commitment indirectly, and adjusts the multiprogramming level when it appears that main memory is either underutilized or overcommitted. In general, static systems have poorer performance than those that adapt to changes in the memory requirements of the active task set.

Relative to local policies, the apparent drawbacks of a global policy are (1) the lack of a guarantee to reserve sufficient memory for each active task to execute efficiently and (2) the unstable or laggard nature of indirect feedback control systems. The first concern is a problem if memory is overcommitted and certain "aggressive" tasks monopolize memory and prevent other tasks from making progress. The first concern is really a sub-problem of the second concern, since both aggressive and non-aggressive tasks will execute efficiently if the system operates at the proper multiprogramming level. The problem of designing a stable and prompt feedback control is non-trivial, but there is no a priori argument that a feedback control is inferior to a direct control.

Ignoring the considerations of overhead, difficulty of implementation, and so on, there does not appear to be a conclusive argument for any "natural" superiority of one policy over the other. Local algorithms have gained considerable popularity among the research community because they are easier to analyze and use in multiprogram system models. Global algorithms are more popular with operating system designers because of their simplicity of implementation and minimal overhead. In Chapter 3, we discuss the efforts that have been made to analyze these two classes of algorithms and to compare their performance.

2.2.2 Working Set – A Local Policy

Let P be the set of all pages of a task. At virtual time t, the program's working set $W_t(\theta)$, is the subset of P which have been referenced in the previous θ virtual time units. θ is a parameter, usually fixed for the duration of the task. The task's virtual time is a measure of the duration the task has control of the processor and is executing instructions; in models, virtual time is often approximated by the number of instructions executed or memory references performed.

A virtual system is said to use a working set replacement algorithm if it operates according to the following rule:

> [A] page may not be removed if it is the member of a working set of a running program. [DENN70]

To implement WS, as defined above, we require the ability to compute the last reference time of each page $p \in \mathbf{P}$, including non-resident pages,

$$LR(p) = \max \{u \mid u \leq t \,\&\, p \text{ is referenced at time } u\},$$

at any arbitrary virtual time t. The task's working set is defined by

$$W_t(\theta) = \{p \mid t - LR(p) < \theta\}.$$

Morris [MORR72] designed a hardware mechanism to aid the computation of working sets. A small register is associated with each page frame which is set to zero each time the page is referenced. At regular intervals, the register of each frame belonging to the currently active task is incremented; the value in the register is an approximation to the amount of virtual time since the last reference. When the register equals a value equivalent to θ, the page can be removed from the working set.

In this work, we consider an equivalent mechanism that computes $LR(p)$ directly. We assume a clock which contains the virtual time of the current task; the clock value is set each time a task is executed and stored each time the task is interrupted. Each time the task references a page, the virtual time clock value is stored, in parallel, in a register associated with the page frame. We do not propose this mechanism as superior to the Morris mechanism; it is just simpler to incorporate in the system model that we present in Chapter 4.

In the absence of special hardware, there is no practical method to compute a precise $LR(p)$ to implement true WS replacement (designated as WSEXACT when necessary to make a distinction). Instead, we approximate WSEXACT by performing *WS scans* of each task at various intervals. In a WS scan, each page p in P is examined, including those that are not in the resident set. If p is

resident, the frame use-bit is tested and, if it is set, the bit is cleared and the approximate $LR\ (p)$ is updated. If $t_1, t_2, t_3,...$ are a sequence of increasing times at which the WS scan is performed, the estimated $LR\ (p)$ is usually the last $t_i < t$, where t is the current virtual time, at which the use-bit was observed to be set. If the scan intervals (t_{i-1}, t_i) are not equally sized, the midpoint of the last referenced interval may be a more accurate estimate, although the reasoning for this choice is far from obvious. After the use-bit test, the page is removed from the working set if $t - LR\ (p) \geq \theta$.

We note that WS scans are required even if special WS hardware is constructed. Some mechanism is required to determine which (and how many) pages are in a task's working set. Unless the WS hardware mechanism causes an interrupt each time a page leaves the working set, some sort of scanning procedure is required. Even with an interrupting mechanism (which implies a non-trivial cost for processing the interrupts) the problem is not entirely solved. In particular, after a task is activated it's working set pages are usually non-resident; if some of these pages are not referenced, the hardware mechanism cannot detect their departure from the working set.

We present three algorithms for scheduling the WS scan. The first, WSINTERRUPT, scans each task page whenever the task is interrupted. This method has the highest overhead and, moreover, has a variable scan interval which is largely unrelated to the needs of the replacement algorithm. It should be considered only if it is suspected that the less costly methods are inaccurate.

The second WS approximation, WSFAULT, performs a WS scan at each page fault, with the limitation that the inter-scan time is bounded, $t_i - t_{i-1} \leq u$, where u is a system parameter. This method usually scans the pages often enough to maintain a good estimate of the working set. If each change in locality is signalled by a sequence of page faults, a page's $LR\ (p)$ is set close to the time it ceases to be referenced. The inter-scan time bound is necessary because a task's working set may contain all of its pages at one time. In this case, it would never fault and no page would ever be removed from the working set. The main drawback to WSFAULT is the repetitive scanning (and the resultant overhead) that occurs when the program changes locality. A possible

refinement is an additional limitation that $t_i - t_{i-1} \geq v$, where v is another system parameter, that defines the minimum inter-scan time.

The third method, WSINTERVAL, performs a WS scan at regular virtual-time intervals, $t_i - t_{i-1} = u$. This method has the least overhead, but may not detect locality changes as accurately as WSFAULT. Generally, $u = \theta / n$ for some small integer n.

There have been many suggested refinements to the basic working set algorithm. Denning [DENN75] suggested using two values of θ, one for program pages and one for data pages, since the reference patterns of programs and data are different. Prieve [PRIE73] suggested using a different θ for each page. Chu and Opderbeck [CHU72] proposed the Page Fault Frequency algorithm which, in effect, alters θ to achieve a given page fault rate. Smith [SMIT76] developed the Damped Working Set algorithm, which is a method for reducing the variance in the working set size due to abrupt locality changes.

Prieve and Fabry [PRIE76] describe a lookahead algorithm, VMIN, which is claimed to be optimal. It is the same as WS except that it removes a page from the working set whenever the page is predicted to be unreferenced for θ or more references. The proof of optimality is a "local" proof; it *assumes* that a local memory management strategy is used without any demonstration that a local strategy is better than a global strategy. Additional simplifying assumptions that are required are described in Appendix B. VMIN probably represents the optimal local replacement algorithm that uses lookahead information and can be computed in linear time.

2.2.3 CLOCK – A Global Policy

The definition and implementation of the global replacement algorithm is not well-defined; it is most often described as an algorithm that lacks certain features of local replacement algorithms. Nevertheless, global replacement is commonly based on the least-recently-used (LRU) replacement algorithm applied to all resident pages. As is the case with WS replacement, modern computers make no provision for determining a precise LRU ordering of pages and approximation methods must be used. Easton and Franaszek [EAST76] discuss three approaches to approximating LRU with the commonly-available frame use-bit that is set whenever a page is referenced.

The *scheduled sweep* algorithm examines all frame use-bits at fixed times, $\{T, 2T, 3T,...\}$, where T is a system parameter. If, at time t, the use-bit is set the page has been referenced in the interval $(t-T, t]$; the page is considered a recently-used page and the use-bit is cleared. If the use-bit is not set the page is placed in a pool of not-recently-used (NRU) pages, all of whom are considered replaceable. If the NRU pool becomes empty before it is replenished by the next scheduled sweep, an *unscheduled sweep* is made to fill the pool. T should be small enough that unscheduled sweeps are rare but large enough to avoid high sweep overhead.

The *triggered sweep* algorithm performs only unscheduled sweeps. That is, it scans all resident pages whenever the NRU pool is empty.

The *CLOCK* algorithm considers the collection of system page frames to be arranged about the circumference of a circle as illustrated in Figure 2.3. The CLOCK pointer (or "hand") points at the last page replaced by the algorithm, and is advanced "clockwise" when the algorithm is invoked to find a replaceable page. When a frame is examined for replacement, the frame use-bit is tested and cleared. If the page has been referenced since the last examination, the pointer is advanced to the following frame; otherwise the page is eligible for replacement and the pointer is left pointing to that frame. CLOCK does not maintain a NRU pool; it finds a single replaceable page each time it is invoked.

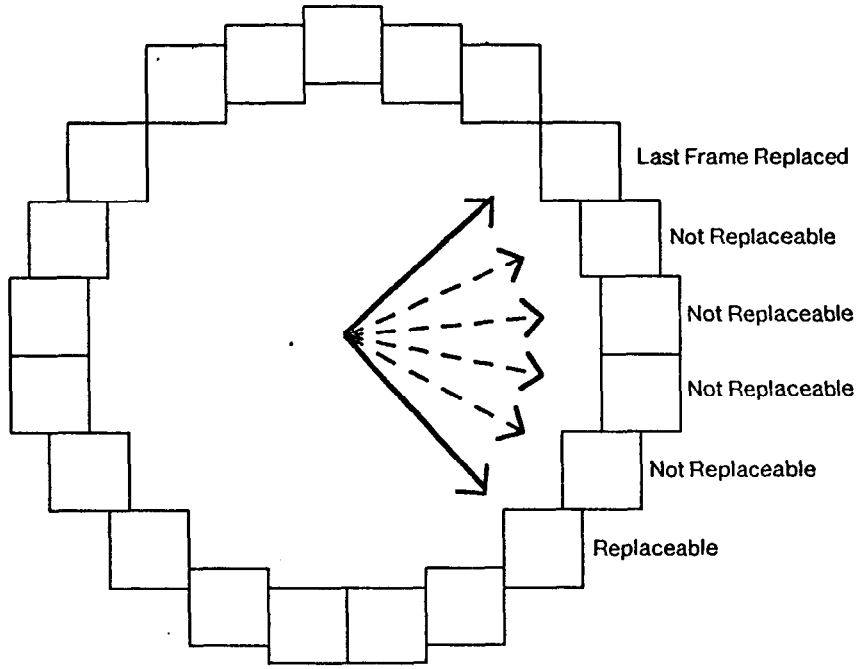


Figure 2.3 - Clock Algorithm

Easton and Franaszek derive upper and lower bounds for the *overhead* of each algorithm, defined as the number of pages scanned per replacement. Their analysis indicates that the overhead of CLOCK is less than the overhead of triggered sweep, which is less than the overhead of scheduled sweep. Unfortunately, the bounds are very loose and diverge rapidly as the main memory allocation increases. Interpolating average behavior from the boundary values is an unreliable method of comparing algorithms which are so similar. Their experiments, however, do support the intuitive belief that CLOCK has less overhead than triggered sweep.

Easton and Franaszek considered a refinement of these algorithms in which a page must be unreferenced for k consecutive scans before it can be replaced. It has been frequently conjectured that, as k approaches the number of page frames, the algorithms will approximate LRU more and more closely. Unfortunately, the refinement does not work as expected. The

algorithms creates clusters of pages, all of which are not referenced for the same number of scans; within each cluster, pages cannot be LRU ordered, and the main effect is to increase overhead. Empirical measurements by Corbató [CORB68] and Easton and Franaszek, as well as unpublished experiments by this author, indicate that k has little effect on the missing page rate and that the algorithms approximate LRU extremely well with $k=1$. It may be postulated that there is nothing unique about the *least*-recently-used page; any page which is not recently used may be an equally good choice for replacement.

2.2.4 The Generalized Clock Algorithm

The previous section introduced the CLOCK algorithm as a simple and efficient method for approximating the global LRU replacement algorithm. CLOCK has two advantages over the sweep algorithms. First, it eliminates the data structure for the NRU pool. Second, it finds more replaceable pages in each full circuit of the frames. Each frame being examined has had a longer real time since its last examination than other frames in the system and has had a greater opportunity to be referenced in the interval. An unreferenced page found in the CLOCK-scan has a higher probability of being least-recently-used than other unreferenced pages.

The density of replaceable pages is highest immediately in front of the CLOCK pointer. Knuth applied the analogy of a snowplow moving around a circular track to a similar situation [KNUT73]. If, as illustrated in Figure 2.4, snow is falling on the track at a constant rate, the snowplow always finds the deepest snow directly in front of it. In fact, the depth of the snow in front of the plow is twice the average depth on the track as a whole. By this analogy, the number of frames replaced by CLOCK on a single circuit should be twice the number that are replaceable at a random time. The analogy is imperfect because the CLOCK pointer does not move at a constant rate, but the intuitive idea remains.

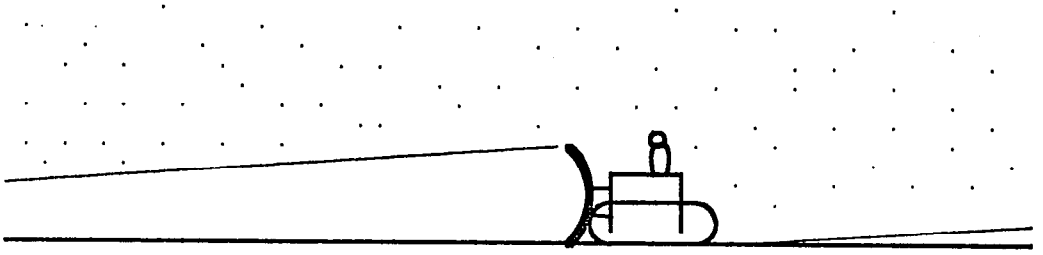


Figure 2.4 - Snowplow analogy

The definition of CLOCK can be expanded to become a universal mechanism that is adaptable for use with any of the proposed replacement algorithms. The basic circular list and pointer are retained, but the LRU replacement criteria, based on setting and clearing the frame use-bit, is altered to suit the desired replacement strategy. For example, a global FIFO replacement algorithm substitutes a null criterion: each page is replaceable as soon as it is examined regardless of the state of the use-bit.

To implement an ordinary WS replacement algorithm, the replacement criteria is straightforward: a page is replaceable if its owning task is not active or if the page is not in the task's working set. Determination of a working set page can take two forms: (1) periodic WS scans simply mark pages as either in or out of the working set; (2) periodic WS scans estimate the last reference time of a page and, when the page is examined in the CLOCK scan, the elapsed virtual time since the last reference is compared to θ.

Most descriptions of WS include a replaceable page list which is added to when tasks are de-activated or when pages are trimmed from active task working sets. CLOCK eliminates this list; replaceable pages are found when necessary by clock-scan search. The cost of a CLOCK scan is inversely proportional to the number of replaceable pages in the circular list. If no pages are replaceable, the algorithm would loop endlessly; this condition must be checked for explicitly. In the normal case, there should be a sufficient number of replaceable pages so that an extensive search is rarely required; otherwise, the system has so few available frames that it is on the verge of thrashing.

An additional advantage of the CLOCK algorithm for WS is the approximate LRU ordering of replacement pages in addition to the basic WS criteria.

Page Reclamation

As discussed previously, a virtual memory system commonly maintains a list, or lists, of pages which are considered replaceable. When a task is deactivated, its pages can be stolen and placed on an available page list. With WS replacement, each WS scan removes pages from the working set and places them on the available list. The scheduled and triggered sweep methods of global replacement also place pages on the available list. When a page is placed on the available list, it is made inaccessible by the owning task, so that it can be replaced without the need to remove it from the owning task.

If a task faults for a page which has been removed from its resident set but is still on the available list, a *page reclamation* is possible. By removing the page from the available list and restoring it to the task, a page transfer is eliminated and the faulting task can continue processing without delay. For each fault, the list must be checked for the desired page. Although the available page list and page reclamation can be implemented with minimal overhead, they are an added complexity which provides no particular advantage. It is our view that they are wholly unnecessary and are naturally eliminated by the additional algorithm. The CLOCK algorithm finds a single replaceable page each time one is required; this page is replaced immediately and irrevocably.

Page Cleaning

When a page is being cleaned, it is not necessary to remove the page from use by the task. It is sufficient to clear the dirty-bit immediately before the transfer to auxiliary memory. The task can access the page during the transfer and if the page is changed, the dirty-bit is set, indicating that the page must be recleaned.

When CLOCK (implementing either a WS or global policy) determines that a page is eligible for replacement, its dirty-bit is tested; dirty pages are placed on a paging I/O request queue for cleaning and skipped in the CLOCK scan. When compared to the sweep and conventional WS algorithms, CLOCK generates a more orderly flow of page cleaning requests and minimize fluctuations in the request queue. If forced writing (see Section 2.2.7) is employed, the sweep algorithms may force large numbers of page writes at the same time, resulting in long processor idle periods if tasks are waiting for page reads.

In the most common case, a page which is queued for cleaning by CLOCK is cleaned during the next circuit of the CLOCK pointer and replaced when it is reexamined. The bias for replacing clean pages more quickly than dirty pages may be significant since the "grace period" for a dirty page is greater than the time required to clean the page. A possible refinement would maintain a list of recently cleaned pages. When a page replacement is requested, the pages on this list are selected first. CLOCK is invoked only when this list is empty.

2.2.5 The WSCLOCK Algorithm

The use of the CLOCK algorithm to implement a WS policy has suggested an entirely new replacement policy which may combine the replacement rules and load control policy of WS with the simplicity of global replacement. We call this algorithm WSCLOCK. WSCLOCK eliminates the periodic WS scans to determine each task's working set. The working set is approximated using observations made during the CLOCK scan. Each time a page is considered for replacement, the use-bit is tested and cleared as in global replacement. If the page is recently referenced, the last reference time, $LR\ (p)$ is estimated to be the current virtual time, VT. If the page is not referenced, the estimated page idle time, $VT - LR\ (p)$, is compared to θ and, if greater, the page is removed from the working set and is replaceable.

WSCLOCK has two fundamental departures from the standard WS algorithm. First, the accuracy of the last reference time approximation depends on the rate at which the CLOCK pointer moves around the circular list. Thus the algorithm is affected by the general level of system activity and,

thus, is not strictly "local." Second, the algorithm does not examine pages which are not resident. If a newly activated task does not reference some of the pages in its working set, there is no mechanism to remove those pages after they have been unreferenced for θ. As a consequence, WSCLOCK's estimate of the task's working set size, which is needed for load control, may be inaccurate. Practical techniques for dealing with this problem are presented in Chapter 4.

2.2.6 Page Loading

We consider two methods of loading the pages of a task when it is activated. As stated earlier, *demand paging* loads a page only when that page is required for a task's continued processing. *Prepaging* loads a collection of pages (the prepage set) when a program is activated and loads additional pages on demand. Prepaging is a compromise between *swapping*, in which entire address spaces are loaded whenever a task is activated, and demand paging, which loads only the active pages of a task and minimizes its use of main memory.

The selection of the *prepage set*, the pages to be loaded when a program is activated, is usually based upon the task's working set or its resident set when the task was deactivated. Other, more exotic, methods have been devised and implemented, but they usually depend upon detecting special attributes of the programs being executed or some unusual feature of the computer's architecture. (For example, IBM 370 programs load addressing registers for each 4K bytes the program wishes to access. A prepage set could consist of pages addressible by registers at the time of reactivation.)

The advantage of prepaging is to load pages more efficiently and to reduce page fault interruptions. Loading efficiency occurs when many pages are transfered with a single I/O operation, which reduces system overhead. If the prepage set is arranged in carefully-chosen auxiliary-memory slots when the task is deactivated, the prepage I/O operation can transfer a number of pages with negligible latency for all but the first page. Eliminating some page fault interruptions, which are often a major contributor to system overhead, also improves performance.

Prepaging may also improve the effectiveness of the page replacement algorithm. If the prepage set has n pages, then one call is made to the replacement algorithm to find n replaceable pages. In addition to the obvious reduction in system overhead, the choice of pages to replace may be made more intelligently when it is known that many are being replaced simultaneously. This question deserves further study.

Unfortunately, prepaging has negative effects on performance. First, it is impossible to predict which pages will be needed and when they will be needed. Some useful pages will be replaced by pages which are never referenced, causing additional faults when the replaced pages are referenced again. Further, the efficiency of prepaging is inversely proportional to the elapsed time until each pre-loaded page is actually referenced.

Second, the processing of paging I/O can be affected adversely. Prepaging unneeded pages wastes paging I/O capacity. A transfer of a large prepage set denies access to auxiliary memory for demand paging and other I/O requests; tasks which page-fault during a prepage operation are blocked longer, increasing system idle time. Arranging the prepage set in auxiliary memory for efficient loading is not trivial and may require extra paging I/O operations to accomplish, particularly if the prepage set contains pages which are never dirtied.

Finally, prepaging is complex and its effect on system performance is difficult to predict. Prepaging is difficult to implement and maintain; it is safe to assume that many systems avoid using prepaging simply because of its added complexity. The net effect of prepaging will depend on the paging storage devices, the overhead of a page fault, and the behavior of programs being executed. If the paging storage device has a large latency (e.g., a disk instead of a drum), prepaging becomes more attractive.

A prepage policy should be tunable. For example, if the working set (i.e., all task pages referenced in the last θ references) is the basis for replacement and prepaging, the prepage set might be defined as all pages referenced in the last $p\theta$ references, where p is an adjustable parameter between 0 and 1. Thus, the degree of prepaging can be controlled in small steps.

2.2.7 Page Cleaning

In earlier designs of virtual systems, it was often envisioned that one would maintain a continuous stream of page transfers, and transfer capacity which was not used for page reads would be used to clean dirty pages. With this technique, and a fast auxiliary memory, random cleaning of dirty pages was considered sufficient. This technique may be useful in particular applications but is not, in general, a satisfactory method. Modern systems have increasingly faster central processors and larger main memories, while the auxiliary memory, in terms of absolute speed and cost-effectiveness, has lagged behind. It takes a considerable amount of time to write out, blindly, hundreds or thousands of dirty pages only to find that the majority of them have become dirty again. The transfer capacity of auxiliary memory is limited and should not be wasted with unnecessary cleaning operations.

An opposite extreme to this wasteful approach is to clean pages only when a dirty page is chosen for replacement by another page. The writing of the first page is coupled to, and precedes, the reading of the second. This method may minimize page writes but the requirement to write the dirty page before reading the second page lengthens the interval between a task page fault and the unblocking of the task. If faulting tasks are blocked for two page transfers instead of one, processor utilization will be decreased.

A better approach is to clean only pages that are replaceable, but to decouple the cleaning and replacement operations. A common technique places replaceable pages on two lists, one for clean pages and one for dirty pages. Only pages on the clean list are replaced, while the pages on the dirty list are cleaned and moved to the clean list.

In general, page reads should be processed before dirty pages are cleaned. This policy is a greedy algorithm for reducing the time that tasks are blocked for page faults. If, however, paging devices are saturated with page reads, dirty pages will remain dirty and will not be eligible for replacement. A bias to replace unmodified pages is created, and, particularly under the global replacement algorithm described below, this bias will force the replacement of active pages,

increasing the fault rate and further aggravating the paging device saturation. Thus, the system may thrash even though memory is not overcommitted.

A simple heuristic control deals with the problem by forcing the writing of dirty pages even though there may be page read requests outstanding. We assume that the page replacement algorithm examines each frame from time to time to determine if it is replaceable. Alternatively, the algorithm must "examine" each frame on a replaceable list until it finds that the page is clean and replaces it. The first time a page is found to be replaceable, it is replaced if it is clean and queued for cleaning if it is dirty. Subsequent examinations increment a count associated with the page each time the page is still dirty. When this count reaches a particular value (a tuneable system parameter) the page is placed at the head of the paging I/O request queue and forced out even if page reads are queued.

This heuristic control is robust since a general scarcity of clean replaceable pages causes the algorithm to force the writing of dirty pages more vigorously. If there are sufficient numbers of clean replaceable pages, the forcing of page writes occurs less frequently but still proceeds at a rate commensurate with the general level of replacement activity if the paging channel remains saturated with read requests.

2.2.8 Auxiliary Memory Management

Although it is of considerable importance to the effectiveness of a virtual memory system, the management of the physical medium for page storage is beyond the scope of this thesis. A proper treatment of this subject would include various techniques for reducing page transfer latency and migration of pages between fast and slow auxiliary memory devices.

2.3 Load Control

2.3.1 Introduction

The primary objective of load control is to maximize the effectiveness of main memory in meeting the overall goals of sharing, throughput, and responsiveness. If main memory is underutilized, there are fewer active tasks and the possibility of all tasks being blocked, leaving the processor idle, is increased. An excess of active tasks will overcommit memory and many tasks will block for page faults, leading to the same problem. A secondary objective is to optimize the use of auxiliary memory. If devices and access paths are dedicated to paging, there is a need to control congestion which can be caused by page traffic even when main memory is not overcommitted. If other system functions share use of the paging devices and access paths, the load may have to be controlled to limit paging traffic and obtain maximum system performance.

At this point, we should clarify the distinction between memory scheduling and load control, since they appear to have similar objectives and overlapping functions: the scheduler determines *which* tasks are to be activated and receive memory allocations; load control decides *when* the activation may be made. The scheduler considers factors such as external priority, equitable processor sharing, load balancing, and interactive response time. Load control considers only the optimal use of main memory and auxiliary memory, given the ordering of tasks by the scheduler. A proper load control achieves a balance between 1) underutilization, which occurs when too few tasks are executing, and 2) overcommitment, which occurs when too many tasks are executing.

It is clearly possible to mingle the functions of scheduling and load control. There may be some advantage to considering a task's main memory and paging I/O requirements when scheduling it for memory allocation; there may be methods of partitioning tasks into groups which make balanced use of the memory management resources. At the present time, however, the complexity of designing such a scheduler, not to mention the difficulty of measuring and analyzing its effectiveness, weighs heavily against its consideration in this thesis.

For local memory management policies, the time of activation depends on the task which is scheduled to be activated; for global policies it does not. Once activated, a task is processed until its quantum is exhausted, it terminates or goes dormant, or it is demoted when load control detects overcommitment. As stated earlier, preempting tasks from memory is costly and has a deleterious effect on performance. Tasks demoted by load control may be scheduled for reactivation differently from the tasks that are deactivated by the scheduler. We assume that a demoted task may still have a number of resident pages, when the next time to activate a program occurs. Thus, a recently demoted task is the load control's "natural" choice to reactivate; this choice also implements a fairness doctrine, since the demoted task did not complete its allocated quantum before it was deactivated.

In the course of this research, we have discovered a new general load control which is independent of the local-global policy division. The *loading-task/running-task* control is described in Section 2.3.2. We discuss the attributes of load control for the WS policy in Section 2.3.3. In Section 2.3.4, we examine a new CLOCK load control for global policies that use the clock replacement algorithm, which we have found to compete favorably with load controls based on local replacement algorithms. In Section 2.3.5, we present an effective load control for the WSCLOCK policy. Finally, in Section 2.3.6 we discuss the *demotion-task* policy to choose a task to demote when the load control detects overcommitment.

2.3.2 The Loading-Task/Running-Task Load Control

When a task is activated, it normally encounters two processing phases. In the *loading* phase, the task has few resident pages and must load them before it can execute efficiently. In the *running* phase, the task has loaded a sufficient resident set and encounters relatively few page faults. In a swapping system these phases would be clear and distinct, since the system must load the entire program before execution can proceed. In a demand paging system, the demarcation is poorly defined and, as a consequence, usually ignored. The ability to distinguish loading tasks from running tasks, however, has two important load-control uses.

Loading tasks compete intensively for service by the auxiliary memory manager to process page-in requests. If many loading tasks are active, the time required for each task to reach the running phase will be increased; as running tasks leave the active queue, loading tasks will predominate and the processor will be idle for much of the time. It is preferable to service only a few loading tasks at one time, and maintain a balance of loading tasks and running tasks.

The loading-task running-task (*LT/RT*) load-control mechanism limits the number of concurrent loading tasks to some number, *L*. This control can be easily combined with an overall multiprogramming level control as described in the following sections.

The transition from the loading phase to the running phase can be determined heuristically, by monitoring the virtual time of the program after activation. Whenever this time exceeds τ, a system parameter, the task is assumed to be running. This heuristic is robust since it is largely independent of the working set size and the state of the task's virtual memory at activation. If most or all of the working set is already loaded at activation time, the tasks simply executes for τ and is considered a running task. If a task is still faulting often after τ, it is assumed to have poor locality; such a task should not be allowed to prevent the activation of other tasks.

A more complex *LT/RT* discriminator could measure the inter-fault times and predict future inter-fault times. When the predictor reached a threshold, the task would be considered running. This work will only consider the simpler heuristic.

In Section 2.2.2, we discussed the policy of processing page-ins before page-outs in order to minimize the time a faulting task is blocked and to increase processor utilization. It may also be a good policy to process page-ins for running tasks before page-ins for loading tasks, since these tasks are more likely to execute for longer intervals and increase processor utilization.

Together, the *LT/RT* control and paging-I/O queue policies form the basis of a sub-optimal load control to attenuate, but not prevent, overcommitment. If the system is overcommitted, loading tasks will have difficulty loading their resident sets and reaching the running phase; thus, a loading-task limit will prevent the activation of additional tasks. As long as running tasks are

faulting, the paging I/O queue policy will process their page-in requests first and prevent loading tasks from obtaining their missing pages and reaching the running phase. When overcommitment abates, the loading tasks will obtain their missing pages and reach the running phase, permitting new loading tasks to be activated. Thus, the system will operate, at times, in a partially-overcommitted state. This mechanism is not an optimal load control, but it does prevent the complete collapse of an overcommitted system.

2.3.3 Working Set Load Control

Denning provides the following description of WS load control (slightly paraphrased to use the terms defined in this thesis) [DENN80]:

> The load control maintains an uncommitted frame pool, which is a list of available page frames, and a count K of the pool's (non-negative) size. The highest priority ready task may be activated only if that task's working set size w satisfies
>
> $$w \leq K - K_0$$
>
> where K_0 is a constant specifying the desired minimum on the pool. The purpose of K_0 is to prevent needless overhead of dealing with memory overflow shortly after a new task is activated. ... Note that $K < K_0$ may occur because working sets may expand after loading. [When a page fault occurs] the page fault handler subtracts 1 from the count K. If K is already 0 the page fault handler will first cause the load control to preempt a page from the lowest priority active task; this implies that the lowest priority active task may not have its working set fully resident. A deactivate decision may be issued ... by the page fault handler if the lowest priority active task has its resident set reduced to naught.

In Denning's description, it is not clear why the low priority task which has lost working set pages should continue executing; it will fault often trying to reload its working set, but never succeeding, and will cause useless congestion on the auxiliary memory.

WS load control has two parameters θ and K_0. Most analyses of WS have focused on the effect of varying θ and obtaining an optimal value for this parameter. If θ is small, the average working set size of each task is smaller and the multiprogramming level is increased. A small θ, however, increases the fault rate of each task and can lead to thrashing. A large θ reduces the fault rate but causes the task resident set to grow and depresses the multiprogramming level.

Selecting a value for the uncommitted pool size K_0 represents a trade-off between maximal use of main memory and reducing the overhead that occurs when the system becomes overcommitted. If K_0 is very small, load control allocates nearly all of main memory to active tasks; each time a task expands its working set, which is a frequent event, there is a high probability that memory will become overcommitted and demotion will be required. A large K_0 has an obvious effect of depressing the multiprogramming level.

The optimal value of K_0 is affected by the interactiveness of the load and by the task scheduler policy. When a task is newly activated, its resident set is usually a subset of its working set. The difference between these two sets represents pages that have been reserved for the task but not yet claimed; they function, temporarily, as part of the uncommitted pool. In a batch system with large time-slices, it is expected that the mean task resident set will approach the amount of memory committed for its working set, and a positive K_0 is necessary to prevent frequent demotion.

In a highly interactive system, or one in which time-slices are relatively small, there may always be a number of newly-activated tasks. Such tasks will have memory committed for their working sets, but will not have claimed all of it for their resident sets; such a system should have a zero or negative K_0. Care must be taken to prevent a deadlock situation in the rare cases when all tasks claim their committed frames.

2.3.4 CLOCK Load Control

Introduction

Most global load controls are based on measuring some operational variable (usually related to the rate of paging activity), comparing the variable to some nominal value that is associated with the optimal multiprogramming level, and making an appropriate change to the multiprogramming level if the two are significantly different.

Denning et al. [DENN76] study two similar global load controls, the *$L=S$ criterion* and the *50% criterion.* The $L=S$ criterion adjusts the multiprogramming level until the mean time between faults is approximately the same as the mean page fault service time. The 50% criterion adjusts the multiprogramming level so that the paging device is busy about half the time. The study showed these load controls to be effective under certain modeled workloads but to lack a certain robustness.

An interesting departure from these strategies occurs in the work of Badel et al. [BADE75], which studied a control mechanism that hunted for maximum performance by continually trying different levels of multiprogramming. The study showed that the control adapted to drastic changes in the load (in which the optimal multiprogramming level changes from 7 to 17) and did so very sluggishly. This method requires the system to operate at multiprogramming levels that are *known* to be non-optimal to find the optimal level.

Overview

The *CLOCK load control* depends on the use of the CLOCK replacement algorithm. It is based on observations of the rate at which the CLOCK pointer travels around the circular page frame list. This heuristic is an attempt to measure the current main memory commitment more directly than the previous global load controls (but considerably less directly than WS load control).

Let $\hat{C}$ be an estimate of the CLOCK pointer travel rate C, expressed in revolutions (a scan of every page frame) per interval of real time. Suppose that there is a value C_0 which is indicative of an optimal load. If $\hat{C} < C_0$, the pointer is moving slowly, which indicates one, or both, of two circumstances:

- Few page faults are occurring, resulting in few requests to move the pointer.
- For each request, the mean pointer travel is small, indicating (probabilistically) that there are many resident pages that are not being referenced and are readily replaceable.

These conditions present an opportunity to raise the multiprogramming level.

The opposite relationship, $\hat{C} > C_0$, indicates rapid pointer movement and either a high fault rate, or difficulty in locating replaceable pages in the CLOCK scan, or both. Either condition, especially the latter, implies that the multiprogramming level is too high.

Implementation

The CLOCK load control has four parameters. The main parameter C_0 is the nominal value of C that is associated with the optimal multiprogramming level. Like the WS parameter θ, C_0 is the primary tuning parameter of the global replacement algorithm. Larger values of C_0 permit the system to operate at higher levels of multiprogramming and, therefore, with more page faulting and closer to overcommitment. Smaller values of C_0 lead to undercommitment. As with the WS parameter, tuning is performed through experimentation. We have no theory, as yet, relating various system parameters (particularly the configuration) and the optimal value of C_0.

The remaining three parameters control the computation of $\hat{C}$, which is the estimate of C, and the manner in which it is compared to C_0. The first parameter δ is the estimation time interval. A load control routine is invoked at times t_1, t_2, t_3, ..., where $t_i = t_{i-1} + \delta$. At time t_i, the routine examines information maintained by the operating system to estimate the value of C over the interval $(t_{i-1}, t_i]$, which we designate c_i. The second parameter α is the exponential smoothing

weight which is used to average c_i with c_{i-1}, c_{i-2}, ..., and derive an estimated $\hat{C}_i$. The third parameter φ is a confidence level that is used to compute a confidence interval about $\hat{C}_i$. If C_0 is within this confidence interval, it is assumed that the system is sufficiently close to the optimal multiprogramming level; otherwise, depending upon the relative values of C_0 and C_i, the appropriate change in the multiprogramming level is made.

The effects of δ, α, and φ are interdependent. A small δ causes frequent observations of c_i and induces a larger amount of variability in the estimates; a large δ reduces the variability but causes the load control to react more slowly to changes in the system's locality. (A small δ would also increase the cost of the load control function, but we have found this cost to be negligible for all reasonable δ.) A large α (closer to 1) gives more weight to the latest observation c_i, which causes $\hat{C}_i$ to react more quickly to changes but also increases estimation variability and allows short-term effects to have increased influence on the load control. A large φ (closer to 1) reduces the size of each confidence interval and increases the frequency of multiprogramming level changes; a small φ causes the load control to be more stable at the expense of slow reaction to locality changes.

Exponential Smoothing

Conventional global load control methods attempt to forecast one or more operational variables, and adjust the load whenever the forecast variable differs significantly from a target value which has been associated with the optimum loading. The system is tuned by varying the target value until peak performance is observed.

Predictions can be made by determining trends in observations of the operational variable(s) and extrapolating to future values. Most load control predictions, however, assume that the operational variables are stationary, and future behavior is presumed to be the same as some measure of past behavior.

Typical operational variables, or predictors, have a great deal of variability, even when the load is static. Consider, for example, an predictor based on auxiliary memory traffic intensity. Each task

activation causes a transient increase in traffic as the task's locality is loaded. This may make the system appear overcommitted when it is not. The predictors can be observed over longer periods, but at the expense of reducing responsiveness to over- or undercommitment. It is a particularly poor idea to use a running or historical mean of all observations to predict future behavior. If the system is stable for a time and then makes a change in its locality size, the time required to respond to the change will be proportional to the time the system was stable.

Two methods are used to reduce the variability of, or *smooth*, observations made over short intervals. The first is the *moving average*, which is the (possibly weighted) mean of the last n observations. Moving averages are particularly well suited to smoothing highly periodic variables where the period is fixed and known. The standard example is the removal of seasonal variation from economic time series. This method would be satisfactory, but a second method, *exponential smoothing*, has a number of significant advantages.

Suppose that the current load is established at time t_0 and the control variable, χ, is observed at equally spaced times, t_1, t_2, t_3,..., and x_i is the mean value of χ during the interval (t_{i-1},t_i). At time t_n, the exponentially weighted mean of χ is

$$X_n = \sum_i \beta_i \, x_i$$

where $\{\beta_i\}$ is a set of exponentially decreasing weights such that $\Sigma \beta_i = 1$. Let α be the weighting parameter, $0 < \alpha < 1$. If β_i is defined by

$$\beta_i = \alpha^{n-i} \; / \; \sum_j \alpha^{n-j},$$

then the β_i sum to 1 and $\beta_{i-1} = \alpha \, \beta_i$. If the x_i are stationary, that is, $E(x_i) = E(x_j) = E(\chi)$ for all i, j, then expected the value of X_n is

$$E(X_n) = \sum_i \beta_i E(x_i) = \sum_i \beta_i E(\chi) = E(\chi) \sum_i \beta_i = E(\chi).$$

The exponentially weighted mean incorporates all past observations but gives more weight to observations from the recent past. If the mean is used to forecast the next observation(s), the optimal value for α will minimize the sums of squares of the prediction errors,

$$\sum_i \left\{ x_i - \sum_j \beta_j x_j \right\}^2 .$$

Calculating the optimal α by differentiating the above expression is impractical; since we do not require or expect great forecasting accuracy, calculating a near optimal α by successive approximations is a reasonable procedure. This calculation is unwieldy and too expensive to imbed in a global load control, so a static analysis should be used to select α. Further experimentation and analysis are needed to determine the effect of α on the accuracy of the forecast and, more importantly, its effect on performance when used in a global load control.

The major operational advantage of the exponentially weighted mean is its computational simplicity. No record of past x_i's is required. For each observation, calculate

$$Y_n = x_n + \alpha\, Y_{n-1}, \text{ and}$$

$$Z_n = 1 + \alpha\, Z_{n-1}.$$

It easily verified that

$$Y_n = \sum_i \alpha^{n-i} x_i, \text{ and}$$

$$Z_n = \sum_i \alpha^{n-i}, \text{ so}$$

$$X_n = Y_n / Z_n.$$

Since Z_n is a power series, it can be calculated directly; in this case it is just as convenient, and more efficient, to use the recursive formulae to calculate each X_n as x_i is observed.

When used to control the load, the forecast X_n is compared to some target value, X_0, and the load is adjusted according to their relative values. For example, if X_n is the auxiliary memory

traffic, $X_n < X_0$ indicates overcommitment and a need to reduce the load. $X_n > X_0$ indicates an opportunity to increase the load. Since X_n will equal X_0 only on rare occasions, this scheme will activate or deactivate a task each time X_n is evaluated; on balance, it cause a task deactivation every second observation, and the cost of memory preemption will be higher than necessary. Since X_n is only an approximate forecast, an interval, (X_{min}, X_{max}), around X_0 is a better approach. If X_n is within the interval, the system is either in equilibrium (neither over- nor undercommitted) or X_n is not sufficiently far from X_0 to make a reliable judgment that the load needs to be changed. Unfortunately, this method is sensitive to the width of the interval and provides another parameter to vary in searching for maximum performance.

A more sophisticated method uses the knowledge that X_n becomes a more reliable predictor as n grows and as the variance of the x_i decreases. Given the definition of X_n above, the variance is defined as

$$\begin{aligned} \text{var}(X_n) &= \sum_i \beta_i \, (x_i - X_n)^2 \, , \\ &= \sum_i \beta_i \, x_i^2 - 2 \sum_i \beta_i \, x_i \, X_n + \sum_i \beta_i \, X_n^2 \\ &= \sum_i \beta_i \, x_i^2 - 2 \, X_n^2 + X_n^2 \sum_i \beta_i \end{aligned}$$

Since $\Sigma \, \beta_i = 1$,

$$\text{var}(X_n) = \sum_i \beta_i \, x_i^2 - X_n^2 \, .$$

The variance can be calculated recursively as

$$\text{var}(X_n) = W_n \, / \, Z_n \, , \text{ where}$$

$$W_n = x_i^2 + \alpha \, W_{n-1} = \sum_i \alpha^{n-i} \, x_i^2$$

The variance is exponentially weighted so that it is affected more by recent observations. If we assume that χ is independent and distributed normally, we can make the confidence statement

$$\text{Prob}\left(X_n - K\sqrt{\text{var}(X_n)/n} < \chi < X_n + K\sqrt{\text{var}(X_n)/n}\right) \simeq \varphi ,$$

where K is the point of the unit normal distribution for which the probability that $-K < N(0,1) < K$ is also φ.

Even if the x_i are not normal, the method is a robust heuristic to determine the variance of the predictor; load adjustments can be delayed until the predictor has "settled down" and is not dominated by transient behavior. The variance is exponentially weighted so that it is affected more by recent observations.

2.3.5 WSCLOCK Load Control

The WSCLOCK replacement algorithm approximates WSEXACT; it makes periodic scans of each resident page to estimate $LR(p)$ and then applies to the standard WS rule to remove pages from working sets and make them replaceable. The major difference between WSCLOCK and WSEXACT is the treatment of non-resident working set pages. When a task is activated, all, or almost all, of its pages are usually missing. As it executes, the task demands the pages it needs, which may not include all of the task's current working set. There may be a number of working set pages that are about to leave the working set and will not be referenced for some time. Without the WS scan used by the other WS policies, WSCLOCK has no mechanism to detect the departure of these pages; after a number of activation/deactivation sequences, the calculation of the task's working set size may be very inaccurate.

If WSCLOCK does not determine an accurate working set size, the WS load control mechanism will be unstable and useless. Fortunately, a solution to this problem exists in the proper use of the *LT/RT* control. Consider the state of a task after it has been active for θ of virtual time. The WSCLOCK algorithm will have identified, approximately, those resident-set pages that are in

the working set and those that are not. *All non-resident pages are, by definition, not in the working set.* Thus, θ virtual time after activation the resident working set is the true working set. In practice, the resident working set closely approaches the true working set when the loading phase (as defined by the *LT/RT* control) has been completed.

Thus, if we define the WSCLOCK working set to be the same as the resident working set, significant inaccuracy is expected only when the task is inactive or during the loading phase. When a task is inactive, therefore, its working set size should remain constant and equal to the value at its last deactivation. To eliminate, or attenuate the effects of, inaccuracy during the loading phase, we see two solutions. First, we can fix the working set size – to the same value it had at activation – until the loading phase is complete or until the number of resident working set pages exceeds the initial value. Second, during the loading phase we can depend on the *LT/RT* load control to prevent the WS load control from overcommitting memory. Either solution may be sufficient; a combination of the two is better.

2.3.6 Demotion-Task Policy

Demotion is a task deactivation that occurs when load control detects overcommitment and directs a reduction in the multiprogramming level. We assume that a demoted task is given a special status and will be reactivated at the first opportunity; not only does this seem to be a fair policy, it also increases performance to the extent that the demoted task still has resident pages when reactivated.

We note that a demotion can easily be made to coincide with a page fault. With WS replacement this occurs naturally since demotion is signalled by a page fault which cannot be processed due to a lack of uncommitted pages. With global replacement, load control can decide to lower the multiprogramming level asynchronous to a fault; there is, however, no motivation to select a task for demotion until there is an event which will alter the allocation of main memory, that is, a page fault. Although the number of active tasks may be reduced by attrition, this is unlikely because lowering of the multiprogramming level implies that the fault rate is high.

The main issue of demotion is the choice of task to demote. In [DENN80], Denning suggested the *lowest priority task.* Actually, he suggests that, when overcommitment occurs, the lowest priority task should give up its pages, one by one, without being demoted until all resident pages are stolen. This would appear to be a mistake, since the lowest active task will be forced to execute with a restricted resident set and will fault often (replacing its own pages) and generate a great deal of paging I/O without making much progress. In any case, demoting the lowest priority task may be a proper choice from a policy standpoint, but is not necessarily the best from a performance standpoint.

A common suggestion is to demote the *faulting task* [FOGE74]. This has some intuitive appeal since there is a greater probability that a faulting task does not have its working set resident, and performance would suffer least by demoting it. This choice also has an immediate payoff because it blocks a task which is about to be blocked anyway and it eliminates the overhead of a page replacement and I/O operation.

A third choice for demotion is the *last task activated.* If the loading task heuristic of the previous section is employed, the last activated task is least likely to have loaded its working set. Not only is it the task that executing least efficiently (in terms of its page fault rate), it is also likely to have the fewest resident pages and requires the least amount of effort to reload. For this last reason, a fourth choice for demotion is the *smallest task*, that is, the one with the fewest resident pages. Unfortunately, this policy would penalize any program that happened to have a small locality and should probably be avoided unless it results in a substantial improvement in performance; system policies should encourage programmers to reduce program locality.

A fifth choice is to demote the *largest task*; since demotion occurs when memory is overcommitted, this choice will reduce that overcommitment by the largest amount, making an additional demotion unlikely. It would appear that the largest task is also the most expensive one to reload. A counter-argument is that the largest task will lose only enough resident pages to relieve the overcommitment; if it is the next task activated, it may have most of its working set still resident. On a policy level, the largest task might also be thought of as the greediest and that

demoting this task would encourage programmers to keep their localities as small as possible. Not only is this latter argument narrow-minded, it appears (as shown in Chapter 5) that this demotion choice has the worst effect on performance.

A sixth demotion choice is the task with the *largest remaining quantum*, thus effecting a shortest-processing-time-first scheduling discipline. Finally, *random* selection can be considered to distinguish which of the above policies have some sort of positive effect on performance.

2.4 Conclusion

In this chapter, we have described detailed methods of scheduling, memory management, and load control in a virtual memory system. Numerous new algorithms and techniques were introduced. This collection of methods form the basis of the operating system model to be described in Chapter 4.

Chapter 3

Virtual Memory Models

The past 15 years have produced an astonishing variety of virtual memory models. These models can be generally divided into simpler ones that model an individual program's behavior under some virtual memory management strategy, and more complex ones that model a set of programs being executed concurrently in a virtual memory computer system. Program behavior models can be divided into stochastic models, which characterize program execution with an analytic or Markov model, and deterministic models that use a program trace or reference string to drive the model. System models are similarly divided into analytical models and simulation models. Each system model usually incorporates one of the program behavior models.

3.1 Preliminary Assumptions and Definitions

Almost all models of virtual memory decompose the execution of each program instruction into a set of main memory accesses. Furthermore, each memory access, in the absence of a page fault, is assumed to require the same time interval to complete. This interval, the *reference*, is the basic model time unit. A program is defined by a *page-set*, $P=\{p_i \mid i=1,2,...,m\}$, and a *reference string*, $\{r_t \mid t=1,2,...,T\}$. Each r_t is a pair, (p,d); p is the page addressed by memory access t; the boolean d is **true** if the reference updates (dirties) the page. The integer m is the number of distinct pages referenced and T is the total number of memory accesses.

A *resident set*, R_t is the subset of P present in main memory when reference r_t is completed. In order to *execute* a program, it is necessary to construct R_t at each t such that $r_t \in R_t$. The construction is dependent upon the paging algorithm used. The fact that the execution of most programs can proceed efficiently when R_t is only a subset of P is due to the notion of *locality*: at a given time, a program tends to reference only a few of its pages and the set of pages being referenced changes slowly.

If $r_t \notin R_{t-1}$, then a *page fault* occurs and a *page-in* is required to place r_t in R_t. Define $F_t=1$ if a fault occurs for reference t and $F_t=0$ otherwise. *Demand* paging algorithms never add a page to the resident set until there is a fault for the page, i.e.,

$$r_t \in R_t - R_{t-1} \Rightarrow F_t=1 \text{ and } (R_t - R_{t-1}) = r_t .$$

By implication, each reference can cause at most one page fault and page-in.

Let D be the mean page-in delay time. The mean real time required to process each reference is

$$RT_{ref}(t) = (1 + D\ F_t),$$

and the total real execution time is the sum of these

$$RT = \sum_{t=1}^{T} RT_{ref}(t).$$

It is often assumed that the efficiency of a paging algorithm is inversely related to the *real space-time product* required to execute a program. The space-time product of each reference is defined by

$$ST_{ref}(t) = RT_{ref}(t)\ |R_t|,$$

where $|R_t|$ is the size of the resident set at time t. The total space-time product is the sum over all reference times,

$$ST_{real} = \sum_{t=1}^{T} ST_{ref}(t).$$

Alternatively, we can partition ST_{real} into the *virtual* space-time product and the *fault* space-time product by

$$ST_{virtual} = \sum_{t=1}^{T} |R_t|,$$

$$ST_{fault} = \sum_{t=1}^{T} |R_t| \; D \; F_t \;, \; and$$

$$ST_{real} = ST_{virtual} + ST_{fault} \;.$$

Let f be the mean number of faults per reference, $(0 \leq f \leq 1)$. We define the program *lifetime*, $L = 1/f$, as the mean reference times between faults.

There are three different measures of memory usage when a program is executed. $S_{virtual}$ is the mean virtual resident set size,

$$S_{virtual} = ST_{virtual} / T = \sum_{t=1}^{T} |R_t| / T.$$

S_{fault} is the mean resident set size at fault times,

$$S_{fault} = ST_{fault} / fT = \sum_{t=1}^{T} (|R_t| \; D \; F_t) / fT.$$

S_{real} is the mean real resident set size,

$$S_{real} = ST_{real} / RT.$$

3.2 Stochastic Models of Program Behavior

3.2.1 Lifetime Models

Lifetime Curves

The simplest models of program behavior are measured *lifetime curves.* The execution of a sample program is simulated (usually using one of the program reference strings described in Section 3.3) using a chosen page replacement algorithm; the mean resident set, m, and mean lifetime between page faults, $L(m)$, are calculated and tabulated for different values of m. Different values of m are either generated directly, as with fixed-space replacement, or by varying a parameter such as θ for WS. The set of points $(m,L(m))$ form a "curve." An ideal lifetime curve has the convex-concave shape illustrated in Figure 3.1. Below some critical value of m the program faults very often and $L(m)$ rises slowly. Past this value $L(m)$ increases rapidly until it reaches a "knee" beyond which additional increases in m have a small effect and the curve flattens out. Denning [DENN80] defines the knee as the point $(m,L(m))$ that maximizes the value of the ratio, $L(m)/m$, and cites studies that associate this point with optimal performance.

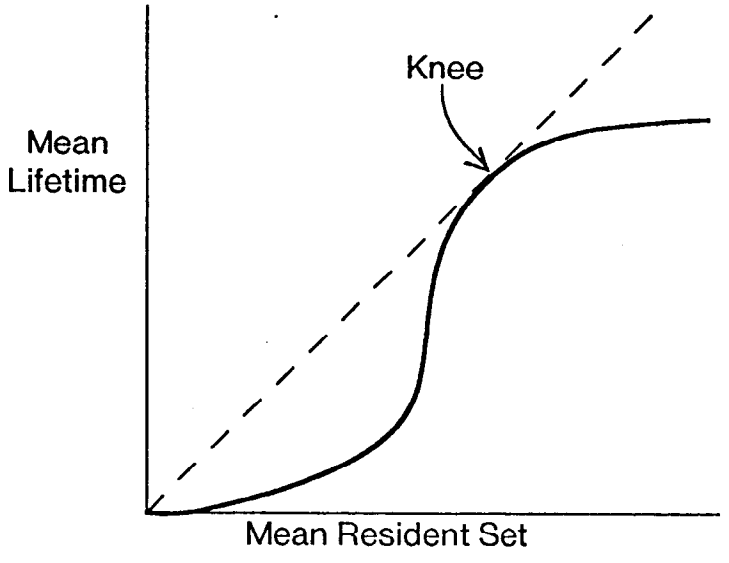


Figure 3.1 - Ideal Lifetime Curve

Efficient methods for generating sufficient points to produce smooth curves have been devised. Slutz and Traiger [SLUT74] describe a method for generating WS lifetime curves in a single pass over the reference trace. Denning [DENN74] extends this method to VMIN. Bard [BARD73] measures lifetime curves for global LRU replacement.

Lifetime Functions

The logical progression from lifetime curves is to lifetime functions. A simple function which fits the points of the lifetime curve is not only a more compact representation but also may indicate some fundamental aspect of program behavior. Saltzer [SALT74] proposed a linear function

$$L(m) = a\,m$$

where a is a constant, which he observed on the MULTICS system. Denning [DENN74] objected strenuously to Saltzer's model as unsupportable by existing data.

Belady [BELA69] suggested a model of the form

$$L(m) = a\,m^k$$

where k is dependent on the locality of the program and is normally in the range [1.5, 3]. Alderson et al. [ALDE71] used the model

$$L(m) = a\,2^{km}$$

in a virtual memory simulation study.

The two previous models model the concave part of the lifetime curve but do not exhibit the leveling off of real curves. Chamberlin et al. [CHAM73] evaulated a model of the form

$$L(m) = a / [1 + (b/m)^2]$$

which has the desired convex-concave shape.

These functions are only moderately successful in representing measured lifetime curves. Actual program lifetime curves are rarely smooth and often have several convex-concave regions. The

formulas of lifetime functions cannot be used dependably to infer more subtle properties of program behavior. There is no basis for assuming that these formulas capture any *inherent* properties of a program's lifetime curve other than its general shape. For example, Chamberlin et al. *differentiated* the ratio $L(m)/(L(m)+W)$ to find the value of m which maximized the rate of a program's progress. Such a technique is ingenious but perilous, since both the ratio and the differentiation compound the errors in the lifetime model.

Ambiguity in Lifetime Models

Lifetime curves and functions have become a common method for characterizing program behavior. As described in the following sections, lifetime models are often incorporated in computer system models as the sole model of program behavior. In many other cases, lifetime curves form the basis of more complex program models such as the phase-transition model described in Section 3.2.5. Thus, we should expect that a given program's lifetime curve will be well-defined and that the method of calculating it should be relatively unambiguous. Unfortunately, this is not the case.

We observe that both the choice of page size and the amount of time the program is measured can drastically alter both the shape of the lifetime curve and the range of lifetime values. Suppose that a program has m pages and is measured over T references. Since the program faults at least once for each page, its maximum lifetime is T/m. If either the page size is reduced by a factor of k, or the measurement period is increased by a factor of k, the maximum lifetime will be kT/m.

It is not unreasonable for the lifetime to change when the page size is changed, but one should be careful to choose a page size which is representative of the computer system environment to be modeled. Quite often, an unrealistically small page size is chosen because the resulting lifetime curve is more interesting or fits some preconceived notion of what a lifetime curve should look like. For example, models by Kahn [KAHN76] and Simon [SIMO79] both depend on lifetime measurements that use a page size of 64 words. Kahn stated that he "settled on a 64 word page

size ... because some early analysis showed little sensitivity of program fault characteristics to page size." The lower section of the curve may be insensitive to page size, but the upper section is directly, linearly, dependent on page size.

It would appear much less reasonable to accept the lifetime curve as a fundamental description of program behavior when that curve's upper bound varies directly, linearly, with the amount of time the program is measured. An analysis of either the Kahn or Simon models using the same sample programs, but measured for twice (or half) as long as the original sample, would produce startlingly different results. The fault is not that lifetime curves are difficult to measure, but that lifetime curves themselves are a weak and poorly-defined model of program behavior.

Another, related, problem with the ordinary definition of lifetime curves is the inclusion of the initial reference to each page when counting page faults. Most of these initial faults occur early in the program's execution and are part of a startup transient which should be eliminated from the measurement of the program's long-run behavior. When the mean resident set, m, is large, initial faults dominate over faults due to replacement and are the main cause of the concave part of the lifetime curve.

Consider, for example, a program which has its entire m page address space resident. What is its expected lifetime? The lifetime curve model says T/m, where T happens to be the amount of time the program is measured. In reality, the lifetime is infinite because no fault can occur. Thus, lifetime curves have considerable bias, particularly in the upper sections, past the knee. As described in Section 3.4.1 and in Appendix A, the results of Simon's model, which indicated WS to be near-optimal, depend almost entirely on the upper sections of the lifetime curves.

We suggest that researchers who build models depending on lifetime curves consider eliminating initial faults; of course, the resulting lifetime curves will be knee-less. A number of studies [KAHN76, DENN76, GRAH76] have associated near-optimal behavior with memory management policies that attempt to keep a program operating at its lifetime knee. These studies are based on analyzing system models that incorporate lifetime models of program behavior. If, as this section

suggests, the knee is an ephemeral artifact of the program measurement process, these studies need to be re-examined.

An alternative to eliminating initial faults is to delay counting faults until a measured program has executed for some time, thus eliminating the startup transient. We know of no studies in which either of these methods has been used to remove the bias present in ordinary lifetime curves. In Section 5.1, we illustrate the bias in lifetime curves with measurements of real programs.

3.2.2 The Coffman-Ryan Model

Coffman and Ryan [COFF72] characterize program behavior by the distribution of working set sizes: a program's working set is assumed to vary by a stationary, normal, stochastic process with mean m and variance σ^2. The model is validated by analyzing reference strings produced by the LRU stack model described below, which is itself open to some validity questions. The model is then used to show that variable-space allocation is better than fixed-space if σ^2 is relatively large.

Lifetime curves and functions, and the Coffman-Ryan model, are not program behavior models as much as they are *replacement* behavior models. Replacement algorithms can be compared to a limited extent by comparing their lifetime curves on sample programs. Chu and Opderbeck [CHU72], and Bard [BARD73] have established the superiority of WS and global LRU, respectively, over fixed-space LRU, but WS and global LRU have not been compared to each other.

3.2.3 Reference Probability Models

Unlike lifetime models, which simply predict the occurrence of the next page fault, *reference probability* models generate page references that have some properties similar to page references generated by real programs. Each model is parameterized by the probability vector

$$\{ \rho_i \mid i=1,\ldots,m\}, \quad \sum_{i=1}^{m} \rho_i = 1.$$

In the Independent Reference Model (IRM) [AHO71], each page p_i is simply referenced with probability ρ_i. This model only incorporates the fact that some pages are referenced more often than others. In effect, it models a program as having a single locality. The parameters of the IRM are determined by counting the number of references to each distinct page.

Distance string models, or LRU stack models (LRUSM) [SPIR76], incorporate the tendency of programs to re-reference pages that have been referenced in the recent past. As the program "executes", the model maintains a stack of the program's m pages in order of recency of use. At each reference, r_t, the probability that the page in stack position i will be referenced is ρ_i. Each time a page is referenced, that page is moved on the top of the stack, displacing the pages below it. The process then continues with the updated stack.

Measuring the parameters of the LRUSM is more complex than measuring IRM parameters. As a reference string is scanned, an LRU stack is maintained and the distance of each reference is determined by searching the stack. Since most references are to pages near the top of the stack, the average number of search steps is usually small. Tabulating the frequency of each distance determines the ρ_i's. The probabilities tend to follow the rule that $\rho_i \geq \rho_{i+1}$, but this is not a hard and fast requirement. If references tend to follow an "instruction, data, instruction, data, ..." sequence, ρ_1 may be quite small while ρ_2 will be larger.

If we assume that the ρ_i's have a natural monotonicity, it is natural (as with lifetime curves) to look for a simple function to describe them [LEWI73]. Linear, exponential and geometric relationships between i and ρ_i have been postulated, but this line of research has little promise. Both lifetime curves and the LRUSM assume that a program's locality changes at a fixed rate; neither models the abrupt locality shifts present in real programs.

3.2.4 First-Order Markov Model

In a model of a multiprogrammed system [SEKI72], Sekino uses a first-order Markov model of program behavior. The model has m states, one for each page of the program. The transition probability matrix $\{ \rho_{i,j} \mid 1 \leq i,j \leq m\}$ is defined such that

$$\sum_{j=1}^{m} \rho_{i,j} = 1, \text{ for all } i.$$

Each step of the Markov process represents one reference. If the process is in state i at step t, page p_i is referenced; at step $t+1$, the process transitions to state j, and references page p_j, with probability $\rho_{i,j}$.

This model is capable of exhibiting abrupt locality changes, but each locality set is fixed for the duration of the program and may not intersect any other locality set.

3.2.5 Phase-Transition Model

Denning and Kahn [DENN75a] developed the phase-transition model which reflects abrupt changes in locality. The model consists of a macro-model and a micro-model. The macro-model is a two-state semi-Markov chain. The states of the macro-model are the *phase*, in which the program locality is varying slowly and few faults occur, and the *transition* in which the program is making an abrupt change to a new locality. The micro-model describes the lifetime (or holding time) in each state. As illustrated in Figure 3.2, the model parameters are:

- L_{phase} and $L_{\text{transition}}$ - the lifetime between faults (holding time) in each state
- P_{phase} and $P_{\text{transition}}$ - the probabilities of changing state after a page fault

When the program enters a phase, it executes for an interval with mean L_{phase} before it encounters a page fault. Then it enters a transition with probability $P_{\text{transition}}$ and remains in the phase with probability $1-P_{\text{transition}}$. Once in a transition, it executes for an interval with mean $L_{\text{transition}}$ (typically $L_{\text{transition}} << L_{\text{phase}}$) and then re-enters the phase with probability P_{phase}.

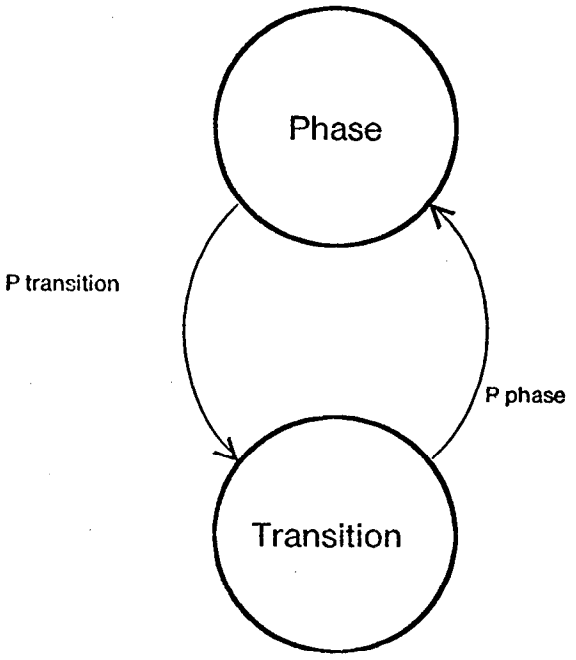


Figure 3.2 - Phase-Transition Model

The parameters can be specified as various distributions and are derived by measurement [KAHN76] or simply postulated [DENN75a]. The definition of phase and transition, as well as the derivation of the parameters, assumes a prior characterization of locality. In the work by Kahn, locality is defined by the WS model and the resulting phase-transition model assumes that WS replacement is used. Modifying this type of model to incorporate some form of global replacement does not appear to be feasible.

Simon [SIMO79] extended the two-state phase-transition macro-model to a three-state model which includes system-induced program swapping, and then incorporated it in a queueing network model of multiprogramming. We discuss this model further in Section 3.4.1.

The concepts of phase-transition and the distance string model were combined in the earlier Shedler-Tung model [SHED72]. In their model, a phase is characterized by references with small stack distances, that is, a small and stable locality. A transition is characterized by a sequence of references with stack distances of $l+1$, $l+2$,..., and $l+n$, where l exceeds the size of the current locality. Thus, at a transition the model moves n pages from a lower part of the stack to the top of the stack. The movement between phases and transitions is controlled by a Markov model.

3.3 Deterministic Models of Program Behavior

3.3.1 The Program as its own Model

The most *precise* model of a program's behavior is the program itself. The program behavior of most programs is, in most cases, repeatable at a very fine level of detail. If the program's origin remains fixed, relative to a page boundary, the page referencing behavior will also show great repeatability.

Canon et al. describe a simulation of VM/370 [CANO80] that used an enhanced version of the VM/370 operating system control program. All timing, such as the instruction processing rate and I/O service times, was simulated to permit the controlled evaluation of various real or hypothetical hardware configurations under specific loads. The loads that were executed under the enhanced control program were various collections of real programs.

While appropriate in some instances, this approach has numerous disadvantages. First, it entails the construction or adaptation of a real operating system capable of processing real programs. This is a forbidding undertaking for the vast majority of those interested in performance evaluation. It is particularly ill-suited for the evaluation of alternative resource allocation and scheduling strategies; the complexity of most real operating systems makes it exceedingly difficult to modify the central algorithms.

Without the virtual machine facility of VM/370, which permits the on-line development and testing of real operating systems, the simulator would be very difficult and costly to implement. The simulation runs would require stand-alone time on a large-scale machine, which may be inconvenient, costly, or impossible to obtain. Even with the availability of virtual machines or stand-alone time, the real machine used to run the simulation must have a larger main memory than the computer being simulated. If the simulated virtual system is operating efficiently, most of its simulated main memory is occupied by pages that are active and are being referenced frequently. Thus, the "working set" of the simulation program will be greater than the size of the

simulated main memory. If the real computer's main memory is smaller than the simulated computer's, thrashing will occur. Further, if the system is simulating a thrashing condition, the real computer will be thrashing at a much higher rate. When Canon et al. simulated a 370/145 on a larger 370/168, they were able to run the simulation in a virtual machine. When they simulated a 370/168 on the 370/168, stand-alone time was required.

Another disadvantage to this approach is the necessity of processing the real workload in addition to performing all of the operations usually associated with a computer system simulation; not only do the programs have to be executed, all of the I/O traffic of the simulated system must be faithfully performed by the simulator. This makes each simulation run quite expensive. For example, Canon et al. simulated a 370/168 computer on a real 370/168 computer; stand-alone time was required and the real computer run time was 2.5 times the simulated time. This ratio is considered to be "substantially less than most other types of simulation."

The simulator was used to evaluate 370 hardware systems using a standard operating system provided for those machines. It was not capable of studying radically different system management strategies or computer architectures. There was, apparently, no attempt to simulate a system larger than the real computer available to perform the simulation. Since there is a strong tendency towards larger and larger main memories, the ability to simulate their effect under various management policies is an important need that is not feasible with this approach.

3.3.2 Reference String Models

The basic alternative to using a program to model its own behavior is to use a *trace* of the program. A *trace generator* is a program that processes another program by emulating the central processor functions of instruction fetch and decode, operand fetch, execution, and operand storage and, at the same time, produces a history, or trace, of the program. The information recorded for each instruction may include the instruction address, the operation, the operand address(es), length(s), and value(s), and the result address and value. Shustek describes the construction of such an emulator and its use to evaluate machine architectures [SHUS78]. Given a model of a

computer which specifies the duration of each instruction type, the memory access time, cache structure, address translation overhead and other relevant factors, an extremely precise evaluation of central processor operation can be performed.

Use of a trace, when compared to the program itself, has some immediate operational benefits. The size of main memory used by the simulator drops dramatically. Each page frame of the modeled system can be represented with a few words; very large main memories can be simulated on a small computer. All paging and user I/O operations can be simulated without the need for actual I/O transfers. The complexity of the simulator is sharply reduced, since the need to support a full operating system is eliminated, including concerns such as protection and error recovery. Shustek points out, however, that the trace generator must have explicit knowledge of the interface between the operating system and the program being traced and, in some cases, must emulate that interface.

There are several operational disadvantages to using traces. When using traces of programs to model behavior, it soon becomes apparent that the simple *volume* of the trace is a considerable limitation to its usefulness. The disk storage for a few minutes of traced execution may be prohibitively large. Tape storage may be more cost-effective but will limit the modeling of multiprogramming to one task per tape drive used. The simulator must read the traces, generating a significant and costly number of I/O requests. The time to simulate each element of the trace may be many times the original execution time.

For the purpose of modeling program behavior under virtual memory, these disadvantages can be reduced by using the simpler, more compact, reference string model presented in Section 3.1. In this model, program execution is modeled as a sequence of memory references, $\{r_1, r_2, ...\}$. Each r_t is a pair, (p,d); p is the page referenced by memory access t; the boolean d is *true* if the reference updates (dirties) the page. As the reference string is generated, it is useful to transform the page addresses isomorphically to the integers $\{1, 2, ..., m\}$, where m is the number of distinct pages in the program. Thus, the representation of the page in the reference string is made as compact as possible.

Naturally, we are concerned with the nature of errors induced by this simplification. Some instructions, such as floating-point division, may have an execution time that is disproportionate to the number of memory references. On sophisticated computers, instruction-fetch references are often overlapped with execution and references for operand access. In interleaved memory systems, the access time for multi-word operands is not a simple multiple of the single word access time. In a system with a memory cache, the pattern of memory references will affect the mean time for each reference. If a lookaside translation scheme is used, the pattern of access will also affect the address translation delay. To model these low-level effects would require a many-fold increase in the complexity and cost of modeling program behavior that, particularly in the study of virtual memory management, would not improve the model significantly. We shall assume that these effects contribute to the mean time of each reference but that all references are identical.

Another source of model error is the independence of successive references. In reality, all pages referenced during any instruction (by both instruction fetch and operand access) must be resident simultaneously, so a faithful model would require all references generated by a single instruction to be processed together. For example, the execution of a single IBM 370 instruction may reference as many as 8 pages; if a program is being executed in a fixed-space memory allocation of 7 pages, the independent reference assumption can lead to a considerable error. Fortunately, this worst case does not occur naturally, and the overwhelming majority of instructions reference fewer than 4 pages. Further, if a program's memory allocation were limited to 7 pages (or otherwise constrained so that the error became significant), the system would be thrashing heavily and the accuracy of the measurement would have little value.

Finally, one detail of instruction processing, variable length operands, can have a significant effect on the execution time of some instructions and can be easily retained in the reference string model. An extreme example is the IBM 370 MOVE LONG, which can access every word of main memory (either real or virtual) with a single instruction. A straightforward solution is to segment the operand and generate one reference for each full or partial segment. In our studies, we use a

segment size of 8 bytes since this matches the memory architecture of the class of computers we are modeling.

3.3.3 Stack Distance Strings

In Section 3.2.3, we describe the stochastic stack distance model of program behavior. Each page reference is characterized by the distance or position of the page in an LRU stack. The stochastic model assigned a referencing probability to each stack position; the model can be used to estimate page fault frequencies for LRU replacement directly or to generate synthetic reference strings.

The stack distance model can also be used as an alternative representation for reference strings. Instead of specifying the page being referenced, each element of the stack distance string specifies the stack distance of the page. The reduced reference string and a stack distance string are equivalent representations of program behavior. For example, the reference string, {a,b,c,c,d,c,d,b,d,a}, and the distance string, {1,2,3,1,4,2,2,3,2,4}, are equivalent; each can be automatically derived from the other.

Since the maximum stack distance is the same as the number of distinct pages, there is no volumetric advantage to the distance string. (Since the majority of distances are small numbers, we grant that a variable length encoding scheme may result in a reduction, but at a non-trivial cost in decoding each reference.) The main advantage to the use of distance strings would occur in studies which would otherwise have to maintain an LRU stack of pages. In studies which use page names explicitly, the overhead of maintaining the stack, required to convert distances to page names, is a distinct disadvantage.

3.3.4 Filtered String Models

When used to examine realistic virtual memory policies, much of the information in the reduced reference string is redundant. The vast majority of references are to pages that were referenced in

the very recent past. Unless the program's main memory allocation is unreasonably constrained, these references will never cause page faults. We examine two methods described by Smith [SMIT77] that eliminate redundant information. In Chapter 4, we introduce an improved method for the same purpose.

Snapshot Strings

Reference strings typically have long intervals in which a few pages are repeatedly referenced. In such an interval, the actual order and frequency of reference to each page has a negligible effect on virtual memory management. Thus the first, and most obvious, technique divides each string into fixed-length segments and reduces each segment to a list of pages referenced in the segment. Let the segment length be ω. Define a *snapshot* at time t,

$$I_t(\omega) = \{ (p,d) \in [r_t r_{t+\omega-1}] \} .$$

The *snapshot string* is $\{I_{n\omega}(\omega),\ n=0,1,...,\lfloor T/\omega \rfloor\}$.

The snapshot string is easily produced in a single pass of the reference string by simulating the hardware use and dirty bits for each page. At the end of each virtual time interval, each page whose use bit is set becomes a reference in the snapshot string; if the dirty bit is set, the reference is so marked; all bits are reset. A special code, such as (0,true), is inserted to delimit one snapshot from the next.

Snapshot strings retain some of the redundancy of ordinary reference strings. In order for the string to be an accurate model of locality changes, the interval size, ω, must be much smaller than the age of a typical locality set. Consecutive snapshot intervals are usually in the same locality and are often identical. The informational redundancy is reduced but not eliminated. Compared to more elaborate filtering methods, however, the snapshot string is simple and compact.

Smith also determines that lexicographic ordering of page references in the snapshot induces an undesirable bias in the referencing pattern; randomizing the order eliminates this bias.

Boksenbaum et al. [BOKS73] devised a similar method, but each snapshot (which is called a *load macro*) represents a variable amount of virtual time. The snapshot contains a fixed number of distinct pages; a new snapshot interval begins whenever the previous snapshot cannot accommodate the next reference. An elaborate method of simulating the references within a snapshot was devised so that virtual time could proceed in increments smaller than the snapshot interval.

Filtered Stack Distance Strings

Smith observes that most references in the stack distance string are to the top positions of the stack and can be considered redundant references. He proposes elimination of all references with stack distances less than a parameter, *D*. The parameter controls the trade-off between accuracy and volume reduction. The resulting distance string can then be re-converted to a reference string as indicated in Section 3.3.3.

Smith distinguishes time-dependent paging policies, such as WS, from time-independent policies, such as LRU. Study of time-dependent policies would require that each retained reference be associated with its virtual execution time. We note that realistic multiprogramming models would invariably require virtual-time information even if the paging policy were ostensibly time-independent.

Smith compared both snapshot strings and filtered stack distance strings to the standard reference string in calculating fault rates for fixed-space (LRU, MIN, CLOCK) and variable-space (WS) replacement. Both methods resulted in a 25:1 reduction in trace length with a maximum page fault error rate of about 7% in the regions of interest. Smaller maximum errors were obtained when the trace was reduced by a smaller factor.

3.4 Computer System Models

Models of multiprogrammed computer systems fall into two broad categories: analytic and simulation. Of the many computer system models reported, only a handful have either implicit or explicit representations of automatic memory management using virtual memory. This section describes the general failure of analytic models to represent the dynamics of virtual memory accurately or to provide a vehicle for the comparison of management policies. Simulation models, while more appropriate for modeling the complexity of virtual memory, are usually simplified to reduce the modeling cost, and suffer from the same problems as analytical models.

3.4.1 Analytical Models

Sekino's Model

Sekino's model [SEKI72] is an elaborate set of analytic queueing models and other submodels representing the physical hardware, system management policies, including memory management, and program behavior. As will be typical of the analytical models that follow, memory management in Sekino's model is severely limited. In particular, the model assumes that:

1. All programs are identical.
2. Main memory is partitioned equally among the active programs.
3. Each program always makes full use of its memory allocation.
4. Replacement is local.

Sekino's model considers only fixed-space replacement, such as LRU, FIFO, and RAND, so that these limitations are not entirely unreasonable. The necessity for these limitations indicates the difficulty of extending analytic queueing models to systems that run different programs and have unbalanced memory partitions.

Program behavior is represented by the first-order Markov model described in Section 3.2.4. The system model is used to analyze hardware configurations and to optimize operating system policies.

Chamberlin's Model

Chamberlin, Fuller, and Liu [CHAM73] provide the only analytic model of virtual system behavior which does not use queueing theory. It assumes the lifetime model,

$$L(m) = a \;/\; [1 + (b/m)^2]$$

for program behavior which has the convex-concave shape. By differentiating the ratio $L(m)/(L(m)+W)$, where W is the page fetch time, a program's maximum rate of progress relative to its memory allocation is obtained. This result is calculated for various memory partitions to find the optimal multiprogramming level for a system with identical programs and equal partitions.

Tripathi's Model

Tripathi [TRIP77] uses an analytic queueing network model to study virtual systems with a fixed multiprogramming limit and to compare two fixed-space page replacement algorithms, LRU and OPT. Program behavior is represented by a lifetime curve derived from measuring real programs under LRU and OPT replacement for a range of memory allocations. All programs are identical and receive an equal share of main memory.

The OPT and LRU paging algorithms can be compared by their lifetime functions, . OPT, the best possible page replacement in a fixed memory allocation, generally has lifetimes 50 to 200% longer than LRU. Predictably, a system with OPT reaches a higher multiprogramming level than an LRU system before lifetimes become short enough to lower throughput. It is suprising, however, that the peak OPT throughput is only a few percent better than LRU throughput. OPT does have a distinct advantage over LRU because it remains near its peak over a wider range of the multiprogramming level.

Simon's Model

In addition to being the most modern analytical model of virtual memory systems, Simon's model [SIMO80] also incorporates the most complex program behavior model. Further, it is the only model which is claimed to indicate, albeit indirectly, the relative performance of local and global replacement algorithms. Thus, we have examined this model in great detail.

Simon constructs a simple queueing network of a computer system having a central processor, one paging I/O server, and one task I/O server. However, he incorporates an extended version of the phase-transition model of program behavior described in Section 3.2.5. The extension adds a third state, swapping, to the macro-model. By considering each of the three states (phase, transition, and swapping) as a separate customer class, the network is solved using the theorem of Baskett et al. [BASK75].

The phase-transition micro-model uses measured WS and VMIN lifetime values for various real programs. Although VMIN (see Section 2.2.4) is claimed to be an optimal paging algorithm in general, it is actually optimal only among *local* replacement algorithms and then only under restrictive assumptions. Simon solves the model for various loads and compares WS and VMIN performance. As reported by Denning [DENN80]:

> Simon compared the optimum throughput from the tuned WS policy to the optimum from the VMIN policy. He found that VMIN improved the optimum throughput from 5 percent to 30 percent depending on the workload, the average improvement being about 10 percent. ... This is the most compelling evidence available that no one is likely to find a policy that improves significantly over the performance of the tuned WS policy.

Unfortunately, the compelling evidence is illusory. In Appendix B, we include a paper entitled "Is the Working Set Policy Nearly Optimal?", which analyzes the Simon's model and the basis for concluding the near-optimality of WS. We discuss five major reasons why this model does not constitute compelling, or even plausible, evidence of WS superiority. For example, it is shown that the lifetime measurements are badly biased by measuring short reference strings (see Sections

3.2.1 and 5.1) and by using a small page size. The WS-VMIN comparison depends mainly on lifetime measurements where the bias is most pronounced.

Simon's model is a good illustration of both the strengths and weaknesses of the analytic queueing model. The customer class concept is a powerful technique that permitted modeling of the three different states of program behavior. Unfortunately, the queueing model is unable to model the dynamics of local memory management faithfully because they require a model load of statistically identical programs, the equal partitioning of memory, and other unrealistic assumptions; no one has reported attempting using queueing models to model global memory management.

3.4.2 Simulation Models

Most simulation models of virtual memory systems have used crude models of program behavior to reduce programming complexity and to avoid long simulation running times. Typically, memory management is simplified so that the simulator keeps track of only the number of associated with pages each task and uses random variates to determine when a task will fault or alter the status of its resident set.

Alderson et al. [ALDE71] use the lifetime function

$$L(m) = a\,2^{km}$$

in a simulation to compare some primitive page replacement algorithms, load control strategies and paging device organizations. Winograd et al. [WINO71] simulate a time-sharing system using a task model that assumes that working sets are normally distributed and that inter-fault times are described by the Fine's curve, which is a precursor of the lifetime models. Badel et al. [BADE75] combine Belady's lifetime function,

$$L(m) = a\,m^{k},$$

with the assumptions that all programs are identical and share memory equally to study adaptive load control mechanisms.

Chanson and Bishop [CHAN77] model the Michigan Terminal System. They recognize that traditional workload models are not realistic:

> In an interactive multiprogramming environment, where many resources are competing for limited resources, the order in which the demands arrive significantly influences the performance of the system and most resource demand models do not model this well. The problem is in the assumption of the independence of the statistical distributions used. What is needed is joint distributions relating the different variables, but these are not practically constructable because of the interactions amongst these variables.

They then proceed to model task behavior such that, "page faults are generated from a distribution rather than by a page reference string simulation because of the costs involved," and admit that "[t]his can have an adverse effect on the validity of simulated results in certain areas of study."

Grit [GRIT77] performs a simple simulation of two programs competing for memory using global LRU replacement. Memory references are explicitly generated by a stochastic stack distance model for each program and it is shown that the program with more locality (i.e., more references to the top stack positions) pushes the less local program out of memory.

Masuda [MASU77] uses a phase-transition type of program behavior model in a complex computer system model. Transitions are predetermined by some unspecified process; program behavior in each phase is specified by a stack distance model with artificially generated probabilities. The lifetime curve can be derived from the stack probabilities and the need to simulate individual page references is eliminated.

Gomaa [GOMA79] models the IBM VM/370 system in three different ways: (1) by using the Belady lifetime function, (2) by using another lifetime predictor described in [BARD73a] and (3) by assuming that program paging rates are known precisely. By comparing these models, he

concludes that:

> The substantial increase in error in the model, when either the [Belady model] or the [Bard model] is used, indicates that both are crude methods of predicting paging rates.

The folowing two models incorporate explicit representation of pages and frames, and model program behavior more accurately than the previous system models. They can be considered the direct antecedants of the model described in this thesis and are used to illustrate the improvements represented by that model.

Nielsen's Model

Nielsen [NIEL66, NIEL67] develops a finely detailed simulation model to predict the performance of the IBM 360/67 time-sharing system, TSS, on various hardware configurations. The model is particularly interesting to us because it represents each task page and each main memory frame explicitly and, thus, has the means of modeling the dynamics of memory management under various policies. The model can simulate the sharing of memory, such as reentrant programs, between tasks. The model contains explicit algorithms for task scheduling, paging I/O, memory management, and multiprocessor management, all reflecting the algorithms used in the initial version of TSS.

Nielsen considers using a stochastic model of program behavior. He rejects the idea because he considers such a model to be both inaccurate and exceedingly expensive to use as a memory reference generator. The model he considers would have required the generation of two random numbers for each reference. Apparently, he also rejectsusing measured reference strings, probably due as much to the difficulty of obtaining such strings as to their cost when used to drive a simulation.

The task model used by Nielsen is a set of synthetic program *loops*. Each loop contains actions, typically to reference a single page for some multiple of 100 μsec. The task cycles through a loop for a number of times and then moves on to some other loop. A set of prototype jobs are

constructed by rough *judgments* of what a typical task would do. As stated in [NIEL66], "no claim is made that a prototype represents the behavior of any real job."

According to the model description, TSS used a primitive local replacement algorithm. A task's pages are replaced only if the task was not in the active set. No examination of the use bit, or any other page activity measurement, is employed and a task's working set would be all of the pages it referenced since it was activated. Nielsen uses the simulator to predict bottlenecks due to paging device performance and main memory size. All of his experiments predict high system overhead and generally poor performance. These predictions proved true when TSS was released. No experiments are performed to compare system resource allocation or scheduling policies

Boksenbaum's Model

The model most similar to the one presented in this thesis is a simulation model of IBM's CP-67, a forerunner of VM/370, by Boksenbaum, Greenberg, and Tillman [BOKS73]. First, the structure of the model mirrors the structure of a real operating system. Second, the model represents task pages and main memory page frames explicitly, and manages them as a real system would. Third, the task model is trace-driven, using a reduced form of instruction traces of real programs. Finally, the model is used to compare replacement algorithms.

The full instruction traces are reduced to a sequence of records called load macros. The predominant type of load macro contains the addresses of a fixed number of pages (typically 5 or 10) and an instruction count. The task model processes each load macro by simulating references to these pages for the specified number of instruction times. Although the method is similar to snapshot strings (Section 3.3.4), elaborate mechanisms are provided to reference subsets of the pages for fractions of the specified instruction count. Measurements show close agreement between the load macro model and ordinary reference strings when used to simulate FIFO replacement in a fixed-space memory allocation. Boksenbaum et al. do not describe the efficiency of the load macro approach either in terms of reference string compaction or simulation cost reduction. We surmise that compaction is similar to snapshot strings (e.g., 25:1) and that the cost

reduction in simulating references is somewhat less than that due to the elaborate mechanisms to reference various subsets of the pages in each load macro.

The system model is used to compare two CP-67 replacement algorithms, version 3.0, which uses global FIFO replacement, and version 3.1, which employs the frame use-bits to approximate global LRU replacement. Version 3.1 is a marked improvement over 3.0.

The calibration process revealed an important factor in simulating multiprogrammed computer systems. In both real and simulated systems, very minor changes in some event sequence can alter the task dispatching sequence and lead to large differences in the measured system performance. Of course, these fortuitous differences should disappear in the long run, but extraordinarily long simulationsmay be required to eliminate their effect.

3.5 Conclusions

We have examined an extensive array of program behavior models and computer system models. The lifetime model may be appropriate for some types of studies, but is too weak and too ambiguously defined for analyzing the subtle differences between virtual memory management policies. The other simple stochastic models do not appear to be any better at capturing program behavior. The phase-transition model is an improvement over the simpler models but it is based on the lifetime model and inherits much of its weakness. We know of no instance in which a system model, either analytic or simulation, based on a stochastic program model has been validated by comparison with a real system or model using non-stochastic program models. Until such validation is performed, there is little justification for placing confidence in such models.

More precise program models are more expensive to use and may have a limited range of usefulness. An extreme example is the model by Canon et al. which executes programs to model their behavior. The cost of this approach is extremely high and precludes modeling large systems. Constructing the model, by modifying an operating system, is also prohibitively expensive. Use of ordinary reference strings is also expensive since it requires a minimum of 15 to 20 reference

times to simulate each reference. Techniques to reduce the reference string can bring this approach into an economically feasible range; the methods which have been reported previously can be expected to produce about a 25:1 reduction in trace length with acceptable modelling error. Since such strings are more complex to use in simulation, we expect to the net cost to be on the order of 1 reference time per simulated reference.

The analytical system models that are described above have severe limitations. None of them model the allocation of memory explicitly. Most of them assume that all programs are identical, that the multiprogramming level is fixed, that page replacement is local, and that memory is equally divided among programs. These limitations and the models' dependencies on lifetime models make it unlikely that they can be used to evaluate memory management policies effectively. Simulation models which depend on lifetime models and do not represent memory allocation or page referencing explicitly suffer from many of the same weaknesses as the analytical models.

We conclude that only a simulation model with an explicit representation of page referencing and main memory allocation can be relied upon to give trustworthy evaluations of virtual memory management policies. In Chapter 4, this thesis presents a model that not only uses a highly accurate model of program behavior and a finely detailed model of virtual memory, but also achieves simulation run times that are a small fraction of simulated time.

Chapter 4

A Simulation Model of a Virtual Memory Computer System

This chapter describes an elaborate, trace-driven, discrete-event simulation model of a hypothetical virtual memory computer system. The distinguishing characteristics of this model are (1) a precise, but efficient, model of program behavior and (2) an operating system model that is, for all intents and purposes, a finely-detailed facsimile of not one, but many, different operating systems.

Program behavior is modeled with a new, trace-driven, Inter-Reference Interval Model (IRIM) which is several orders of magnitude more efficient than the reference string model, but which has a negligible loss of accuracy. The IRIM makes practical the precise simulation modeling of virtual memory management and, in many cases, is preferable to analytical models.

The operating system model, while comparatively simple in design, incorporates many different strategies for scheduling, memory management, and load control. It permits the direct comparison of local and global memory management policies; coupled with the IRIM, it also permits precise calculations of the task working set (impractical on standard computers) and the optimal, lookahead, algorithm, VMIN. All of these capabilities are implemented so as to eliminate superficial implementation disparities and allow a comparison of their essential differences.

Section 4.1 describes the basic methodology, structure, and implementation of the model. Section 4.2 describes the configuration model which defines the hardware elements of processor, main memory, and I/O devices. Section 4.3 describes the task model, including the definition of the IRIM and its use to model memory referencing. Section 4.4 describes the operating system model.

4.1 Introduction

The methodology of discrete-event simulation modeling is well established. We have depended heavily on the excellent texts by Fishman [FISH73], Shannon [SHAN75], and Kleijnen [KLEI75], and refer the reader to them for more detailed accounts of the methods used in this work.

4.1.1 Simulator Construction

The basic structure of the simulator is shown in Figure 4.1. There are four major components. The *input processor* accepts statements in a simple, free-form specification language described in Appendix A. These statements can cause the loading of previously created files of other statements, such as standard configurations, workloads, and system parameters. The input processor creates data structures to represent the various objects, such as main memory, tasks, and I/O devices, which comprise the system configuration and workload. The *initialization processor* checks the generated data structures for consistency, assigns default values for unspecified options, and generates a pre-simulation report.

The *execution processor*, which is described in the following sections, performs the simulation. It accesses files of program traces (reference or IRIM strings) to drive the task models. As the simulation proceeds, samples of processor utilization and detailed operation traces can be generated. The *report generator* analyzes and displays the results of the simulation and resets the simulator to accept a new set of specification statements. The summary report of a typical simulation is reproduced in Figure 4.2.

The simulator is programmed in HIBAL, a medium-level, block-structured, system implementation language. It has 5400 lines of source code and occupies about 30,000 bytes of storage, exclusive of dynamically allocated data structures and I/O buffers. It required about 1/2 person-year to design, write, and debug.

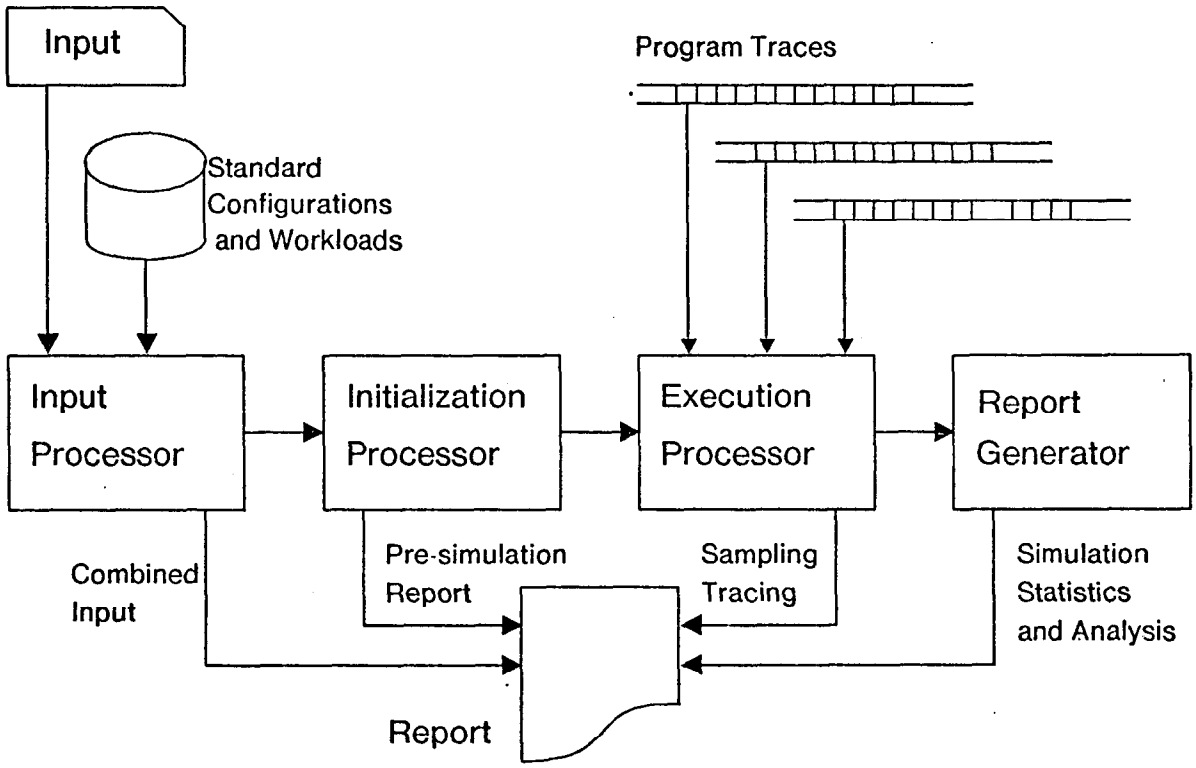


Figure 4.1 - Simulation Program Structure

```
SIMULATION COMPLETE. COST=  10.96 CPU SECONDS, TOTAL SIMULATED TIME = 438.752066
MEASUREMENT START TIME=      .000000, MEASURED TIME= 438.752066, IDLETIME= 166.220613
UTILIZATION= .6211 +|-  .0398 AT CONFIDENCE LEVEL=.95      (EXP. STD. UTIL= .625 +|- .0472)
   55 TASKS EXECUTED.

----------PROCESSOR SCHEDULING----------
ACTIVE QUEUE SIZE:            4    5    6    7
MEAN=  5.25 N=   259      .166 .420 .335 .060

ACTIVE TASKS EXECUTING:       0    1    2    3    4    5
MEAN=  1.49 N= 26134      .378 .179 .169 .142 .096 .029

LOADING TASK COUNT:           0    1
MEAN=   .75 N=   263      .241 .758

PROMOTIONS=132, DEMOTIONS=77

----------PAGING I/O AND REPLACEMENT----------
PAGING I/O: FAULT-READS= 3684, LOAD-READS= 1753, WRITES= 3556, FORCED WRITES=    0

RUNNING FAULT QUEUE:          0    1    2
MEAN=   .29 N=  3684      .740 .227 .028

LOADING FAULT QUEUE           0    1
MEAN=   .21 N=  1753      .784 .215

PAGE WRITE QUEUE:             0    1    2    3    4    5    6    7
MEAN= 19.22 N=  7810      .316 .046 .036 .032 .030 .025 .022 .021

CLOCK SCAN TRAVEL:            1    2    3    4    5    6    7    8    9   10   11
MEAN= 15.92 N=  5438      .200 .122 .084 .068 .059 .047 .036 .032 .024 .024 .022

--------------WORKING SET SUMMARY--------------
WORKING SET SCANS: INTERVAL      FAULT      TOTAL
                   1286(100%)       0(0%)    1286

ACTUAL SYSTEM WS SUM:      247  249  250
MEAN=204.69 N=  7458       .020 .020 .024

ESTIMATED SYS WS SUM:      247  249  250
MEAN=204.69 N=  7458       .020 .020 .024

WORKING SET FRAME USE:     223
MEAN=200.10 N=  8096       .022

---------- DEVICE SUMMARY ----------
DEVICE      I/O'S   UTIL    MEAN-IOTIME
UNITREC      2924   1.26      190509
DISK1         551    .23      190783
DISK2         679    .29      189035
DISK3         814    .35      191307
DISK4         426    .18      189010
DISK5         384    .16      192425
DRUM1        8993    .81       39867

PAGFILE     I/O       UTILIZATION
 DEVICE   COUNT    RUNFS  LOADS WRITES
DRUM1       8992   .335   .158   .323
```

Figure 4.2 - Sample Summary Report

4.1.2 Verification

Fishman [FISH73] identifies a number of circumstances which may cause a simulation to fail to agree with the underlying model:

- A poorly chosen pseudo-random number generator
- Inappropriate approximate random variate generation techniques
- Input parameter misspecification
- Programming errors
- Measurement errors

We shall consider each of these in turn.

Pseudo-Random Number Generation

The simulator uses the multiplicative pseudo-random number generator

$$r_i = 7^5\, r_{i-1} \text{ modulo } 2^{31}-1,$$

which has a maximal period of $2^{31}-1$. This generator is suggested by Lewis, Goodman, and Miller [LEWI69]. It has been checked using a variety of statistical tests by Lewis et al. and, more recently, by Fishman [FISH76]. Although it does not pass all tests, it performs well in comparison to the other commonly used generators.

Each simulated random process, such as a task execution or an I/O device, is provided with an independent stream of pseudo-random numbers. Each process uses the same generation formula but maintains its own generation seed. Initial seeds for the independent generators are themselves generated with the mixed congruential generator,

$$s_i = (314159621\, s_{i-1} + 1) \text{ modulo } 10^9,$$

suggested by Knuth [KNUT69]. By using a seed generator of an entirely different type, we avoid introducing any systematic overlap between pseudo-random number streams. In order to test the simulation's sensitivity to the randomness of the stochastic components, the seed of each component's generator, as well as the seed of the seed generator, can be specified explicitly.

Random Variate Generation

The pseudo-random integer r_i is converted to a real number uniformly distributed on [0,1) which is used to produce integer-valued samples of four distributions: constant, uniform, exponential, and truncated exponential.

Table 4.1 - Random Variate Specification

Parameters	Distribution	Value
min	constant	*min*
$min = max$	constant	*min*
$min = mean$	constant	*min*
$min = mean = max$	constant	*min*
max	uniform	S(1,*max*)
$min < max$	uniform	S(*min*,*max*)
mean	exponential	1+E(*mean*−1)
$min < mean$	exponential	*min*+E(*mean*−*min*)
$min < mean \leq max$	truncated exponential	**min**(*min*+E(*mean*−*min*),*max*)
$mean \leq max$	truncated exponential	**min**(1+E(*mean*−1),*max*)

The type of distribution is specified implicitly with three parameters: *min*, *mean*, and *max*. Only a subset of the parameters need be specified, but it is required that $1 \leq min \leq mean \leq max$. Table 4.1 gives the distribution implied by each legal combination of parameters. S(*a*,*b*) is a sample of the distribution in which the integers $a, a+1, \ldots, b$, are chosen with equal probability. E(*a*) is a sample of the exponential distribution with mean *a*, rounded to the nearest integer.

The generation of the constant distribution is trivial. The generation of $S(a,b)$, on $[a,b]$ from a uniform random number, u, on $[0,1)$ is straightforward:

$$S(a,b) = \lfloor u\ (b-a+1) \rfloor + a.$$

The generation of a non-uniform distribution, such as an exponential distribution, can be performed with the *inverse transformational method.* If χ is a random variable with pdf $f(x)$ and cdf $F(x)$, then x is a random sample of χ if

$$\text{Prob}[\chi < x] = F(x) = \int_{-\infty}^{x} f(y)\ dy$$

is uniformly distributed over (0,1). That is, if u is a sample of U(0,1), then x, such that $F(x)=u$, is a sample of χ. Since the pdf of the exponential distribution with mean λ,

$$f(x) = 1/\lambda\ e^{-x/\lambda} \text{ for } x \geq 0,$$

$$f(x) = 0 \text{ for } x < 0,$$

is integrable, $F^{-1}(x)$ can be evaluated directly:

$$u = F(x) = \int_0^x 1/\lambda\ e^{-y/\lambda} dy = -e^{-y/\lambda} \Big|_0^x = 1 - e^{-x/\lambda},$$

$$e^{-x/\lambda} = 1-u,$$

$$x = -\lambda\ \ln(1-u).$$

If u is U(0,1), then so is $1-u$, and

$$E(a) = -a\ \ln(u).$$

Simulation Input Specification

The simulation input parameter specification language is described in Appendix A. The format of the language is designed to minimize the possibility of accidental misspecification. Input

parameters that are common to a set of simulations, such as the configuration and workload, can be stored in a file and accessed with a single input statement. Thus, errors due to mistakes in duplicating a standard environment are easily avoided.

Following the input phase, a pre-simulation report is generated. This report describes the configuration, workload, and system options in effect for the simulation run. It is a simple matter to check this report for agreement with the intended input specification.

Programming Errors

The problem of detecting and eliminating programming errors is one which pervades the use of computers in any endeavor; we have no special methods of attacking the problem. Even if we were able to validate the simulator by comparison with real systems or independent models, there is no guarantee that some new combination of input specifications would be processed without error.

As the simulator was programmed, numerous tests were included in logic sections that would otherwise continue to function in the presence of errors. There was also a conscious effort to eliminate programming robustness; it is desirable that any errors propagate themselves and cause obvious output inconsistencies or outright failures.

The simulator has an extensive execution trace capability that records a selectable set of simulated events such as task scheduling, page replacement, load control, etc. The detailed operation of the simulator was examined at a very low level for accuracy and consistency. Numerous programming errors were detected and eliminated by use of this facility.

Measurement Errors

Verifying the accuracy of the performance measurement is a difficult task. We have examined the output of hundreds of simulation runs of varying length and complexity, and have ensured that the output has face validity. The simulator has been tested with parameter sets that have

extreme values; the behavior of the simulator has been consistent with the generally expected behavior of the modeled system.

Some consideration was given to comparing the behavior of the simulation model to an analytical model or to a real system. The fundamental reason for using simulation is to represent the dynamic program behavior and complex memory management interaction which are beyond the capability of any reasonable analytical model. In fact, we would consider it more appropriate to use the simulation model to analyze the errors and lack of faithfulness in such analytical models.

An attempt, by the author, to study a real computer system, with the intention of comparing its behavior with that of the simulator, has only served to illustrate the usefulness of the simulator. It was impossible to obtain reproducible results on the real system due to the vagaries of the physical devices and the difficulty of controlling the scheduling processes in real operating system. In real systems, very small perturbations in time lead to large differences in measured performance. The work of Canon et al. [CANO79], described in Chapter 3, illustrates the complexity and expense required to achieve reproducible results with a real system.

4.1.3 Model Structure

Descriptions

The significant elements of the model, particularly the program behavior and memory management models, are described in considerable detail, both in ordinary prose and by program fragments. These fragments have been taken from simulation code, omitting pedestrian details and translating the implementation language to one resembling Pascal. Although these fragments are faithful representations of the simulation processing, the reader should be warned that they are not verbatim and have never been compiled or executed.

Program fragments, while not the most elegant way to convey information to the reader, perform three important functions. First, they are precise descriptions of the model. Our attempts to

understand many previous studies have often been frustrated by a high degree of abstraction and omission of important details of the modeling process; thus, we have tended to provide an unusually high degree of detail for those who might wish to understand or extend this work. Second, the fragments serve to illustrate the relative simplicity of the algorithms used to model the computer system and to increase confidence that the model has been implemented correctly. Finally, they may be useful to both operating system designers as well as the simulation practitioner; they demonstrate that it is practical for very different (e.g., local and global memory management) strategies to be implemented simultaneously in the same operating system.

Time and Clocks

Time is measured in discrete multiples of a single memory reference time. The system real-time clock measures the passage of total or "wall clock" time. Each task has a virtual-time clock which measures the number of references successfully completed.

On typical modern computers, the average reference time ranges from tens of nanoseconds to several microseconds. Thus, all times expressed in the model are relative to the speed of the processor. To simplify the following descriptions, we assume a standard reference time equivalent to one microsecond. When the simulator is parameterized to model a particular computer, all other time values are scaled appropriately.

In this model, we do not simulate overhead. The real-time clock is advanced only when a reference is successfully processed or when the system is idle. The inclusion of overhead in the model is not difficult and would make an interesting followup study, especially in light of the results of Chapter 5. Our intent in this study is to measure the essential difference between alternative strategies without clouding the results with the effects of overhead.

Events and Interrupts

The simulation model is *event-driven.* Events are specific actions which the simulator is to perform at some future real time. The simulator maintains an *EventQueue* which is a list of

Event records as shown below. The *Event* records are ordered by the time they are scheduled to occur. When the system is *busy* (i.e., processing program references) the real- and virtual-time clocks are updated until the real-time clock equals the time of the next scheduled event; then an *interruption* occurs. Whenever the system has no work to perform, it *idles* until the next event. Idling is a simple and efficient matter of advancing the real-time clock to the time of the next event.

```
type Event = record
      Next: ↑ Event; {EventQueue link}
      Time: time; {scheduled real time of event}
      Action: procedure {called when event occurs}
end ;
```

Submodels

The model is conveniently described as a set of three submodels: the *configuration model* includes the basic hardware (processor, memory, and I/O devices) and fundamental concepts such as time; the *task model* represents the execution of programs as a sequential memory referencing and I/O requesting process; the *operating system model* performs the functions of allocating hardware resources in order to execute tasks. While the configuration model and task models are simplified versions of their real counterparts, the operating system model is a detailed replica of the central allocation and scheduling mechanisms of a real system.

4.2 Configuration Model

4.2.1 Processor Model

This model contains no explicit representation of the processor. The operations normally associated with a processor, such as task execution, are performed as part of the other models.

4.2.2 Main Memory

Main memory is a collection of *frames*, each of which can contain a single page. The size of a frame or page is not specified but is implicitly characterized by the task model. The content of each frame is irrelevant and requires no representation in the model or storage allocation in the simulator. Each frame is identical and can be assigned to contain any task page. The memory required for the operating system is assumed to be fixed and suplemental to the memory represented in the model. Each frame is represented by the following data structure:

```
type Frame = record

        {hardware functions}
        Use, {page has been referenced}
        Dirty: Boolean; {page has been modified}
        LastReferenceTime: time; {special hardware capability}

        {memory management functions}
        Free: Boolean; {does not contain any page}
        Owner: ↑ Task;
        VirtualAddress: ↑ Page;

        {paging I/O control functions}
        Next, Previous: ↑ Frame; {paging I/O queue links}
        PageIn, PageOut: Boolean {paging I/O status}
end;
```

and main memory is a fixed array:

```
const MemorySize = 250;
var MainMemory: array [1..MemorySize] of Frame;
```

Associated with each frame are the *Use* and *Dirty* bits which are commonly found on virtual memory computers; these are set when a page is referenced or changed, respectively, and are cleared by the operating system.

Under the WSEXACT replacement algorithm, the variable *LastReferenceTime* is modeled as part of the configuration model – a hardware register is updated automatically each time the frame is referenced. For the other replacement algorithms, *LastReferenceTime* is a simple software entity that is the operating system's approximation of the last time that the frame was referenced.

The remaining variables associated with each frame are software artifacts that are maintained and used by the operating system, and are described in Section 4.4.

4.2.3 I/O Subsytem Model

In direct contrast to the level of detail used in simulating memory and processor management, I/O devices are modeled very simplistically. In real systems, many factors contribute to the time required to complete an I/O operation. The operating system must interpret, verify, schedule, and initiate each I/O request. Queueing delays for obtaining an access path depend on the processor-channel-controller-device configuration. The operation itself is subject to seek and latency delays; the transfer time depends on the device speed and the length of the data record. I/O completion requires processing to detect errors, release the access path, and schedule the waiting task for execution. Realistic modeling of device access time should also consider serial correlation of sequential file accesses and the lack of correlation for random accesses.

In the study of device scheduling policies, optimal hardware configurations, etc., some or all of these factors should be modeled faithfully. In our study of virtual memory, the I/O device model is intentionally simplified so that its effect on system performance is uniform. The elimination of perturbations caused by the physical details permits a clearer focus on the memory management policy questions that are studied in this work.

The following data structure describes the characteristics of each device:

```
type Device = record
        Busy,
        Serial, Boolean;
        Queue: ↑ IORequest; {unprocessed requests}
        MinIOTime, MeanIOTime, MaxIOTime: time;
        Seed: integer {random variate seed}
end;
```

Each device is modeled as an independent entity which operates in parallel with the processor and other devices. When scheduling of I/O devices, the operating system model which processes requests for both task and paging I/O in the same way. Each request specifies a particular device, and is either processed immediately or is placed in the device queue for processing when the device is available. When a request is completed, a simulated interrupt tansfers control to the operating system.

A device which is designated as *Serial* can process a single I/O request at a time; the queue of pending requests is processed in FIFO order. Otherwise, a non-*Serial* device can process any number of I/O requests concurrently. This capability is useful for modeling an entire terminal system with a single device. The service time of the device is independent of the requesting entity and is modeled stochastically. Access time distributions can be specified as constant, uniform, or exponential.

4.3 Task Model

The *task model* consists of the general task state, the virtual memory state, and the execution (or program behavior) model. Execution is modeled as a sequence of memory references punctuated by stochastically-generated I/O requests. Memory references are generated by a trace-driven model such as the reference string or IRIM. Task termination occurs deterministically when the reference string is exhausted. Figure 4.3 provides an overview of the task model.

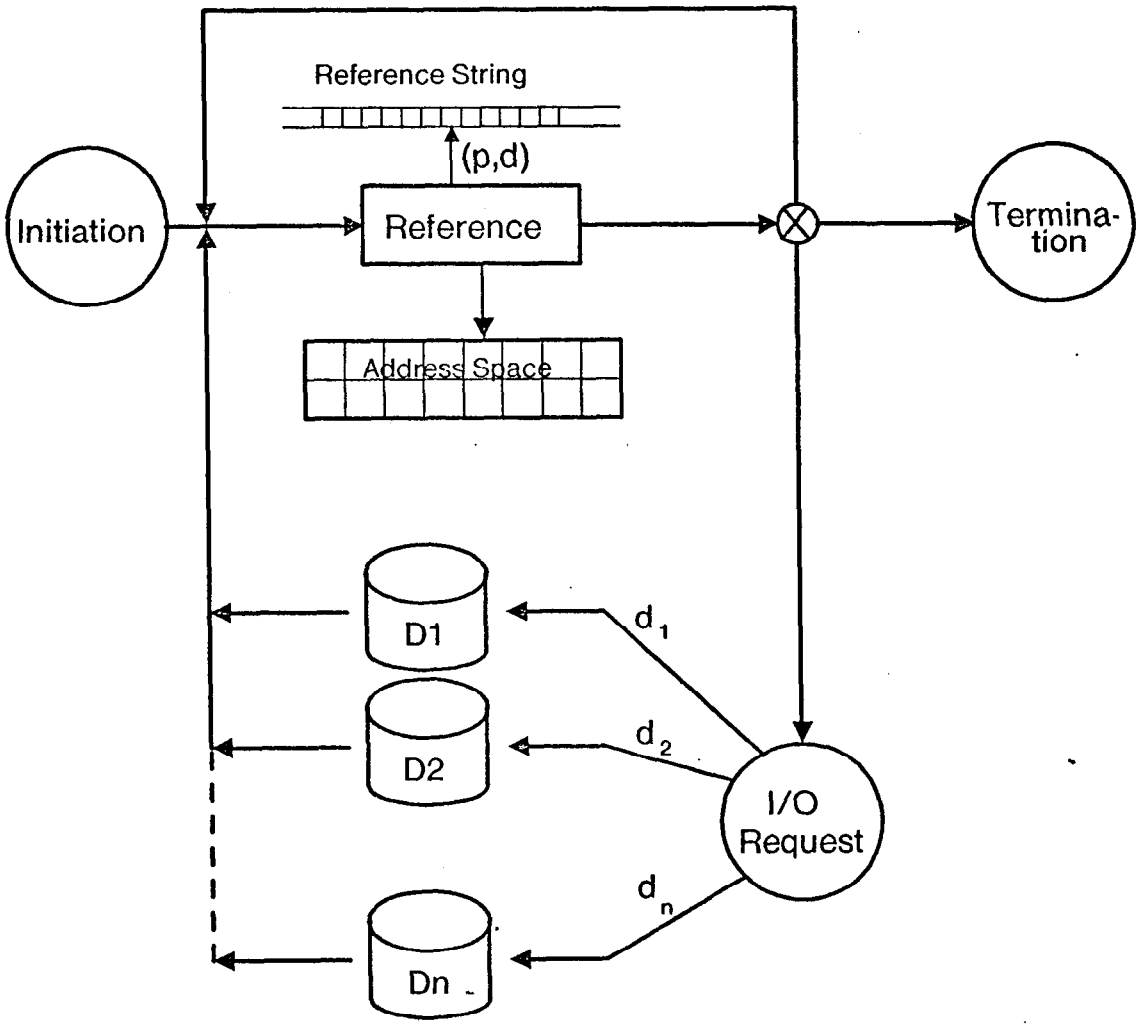


Figure 4.3 - Task Model

4.3.1 Task State

The task state is described by the following record:

```
type Task = record
      {virtual memory state}
      PageTable: array [1..MaxPage] of Page;
      WorkingSetSize: 1..MaxPage;
      WSScanTime: time;
```

```
        {execution state}
        VirtualTime: time;
        {ReferenceStringModel;}
        NextIOTime: integer;

        {I/O request model}
        MinInterIOTime, MeanInterIOTime, MaxInterIOTime: integer;
        DeviceUse: array [1..MaxDevice] of real;
        CurrentDevice: ↑ Device;
        Seed: integer;

        {scheduling state}
        Active, Ready: Boolean;
        Blocked: Boolean;
        WaitingForFrame: Boolean;
        Loading, Swapin: Boolean;
        QuantumEnd: time;
        LastWaitTime: time;
        RealActivationTime,
        VirtualActivationTime: time
end;
```

4.3.2 Virtual Memory State

The task's virtual memory state is described by a *PageTable* array, which contains a model of each task *Page*, and a number of state variables maintained by the operating system. Each *Page* is modeled by the following data structure:

```
type Page = record
        RealAddress: ↑ Frame;
        Resident: Boolean; {page is resident}
        SlotAddress: integer;
        InWorkingSet: Boolean; {used by replacement algorithm}
        LastReferenceTime: time; {used by replacement algorithm}
        IRIbusy, IRIdirty: Boolean; {used by IRIM}
end;
```

In a real system, a *PageTable* element usually contains only the *RealAddress* of each resident page in a minimally coded format. Since the address of each frame is constrained, by the hardware, to being a multiple of the page size, the address translation hardware can transform a relative frame number to the actual memory address of the frame. In the model, since no memory is actually allocated for the frame or the page, the *RealAddress* is simply a pointer to the data structure used to model the frame. The *Resident* boolean variable in the *Page* element signals the model of the address translation hardware that a page is or is not resident.

The *SlotAddress* contains a coded description of the page's location in auxiliary memory. This model, however, does not manage auxiliary memory explicitly and this element is largely unused. The elements *InWorkingSet* and *LastReferenceTime* are maintained by the replacement algorithm being modeled. *IRIbusy* and *IRIdirty* are special indicators used only for the simulation of the IRIM of program behavior and are not usually represented in a real system.

4.3.3 Inter-Reference Interval Model

Introduction

In this section, we introduce the Inter-Reference Interval Model (IRIM) of program behavior. The IRIM is a deterministic, trace-driven, model of the program reference string that is used to simulate virtual memory referencing. The IRIM is derived by a computational procedure that, compared to the processing required to collect the original reference string, is efficient and applicable to very long strings. When used to study modern virtual memory management strategies, the IRIM is an accurate and efficient method for modeling program behavior.

The effectiveness of the IRIM is based on the existence of program locality and the behavior of practical memory management policies. Locality implies that, at any given time in a tasks's execution, most program pages fall into two groups: those that are being referenced frequently and those that are not being referenced at all. We consider a practical replacement algorithm to be one that rarely or never replaces a page that is being referenced "frequently", unless the task

that owns the page is deactivated. In the IRIM, *frequently* is defined to mean "at least once every ω references", where ω is a model parameter similar to the WS model parameter θ. At each point in virtual time, the IRIM divides the task's pages into a *busy* set and an *idle* set, such that the busy pages are being referenced frequently and the idle pages are not.

Since the need to clean dirty pages before replacing them has an effect on performance and, further, since there are alternative cleaning strategies to be evaluated, a program model should also provide information about the nature of page dirtying. The IRIM divides the set of busy pages into clean pages and dirty pages: a dirty page is one which is being updated at least every ω references.

Although the IRIM is similar to the WS model, in that a page not referenced for ω (θ) consecutive references will be removed from the busy (working) set, there are several important differences. If r_t and r_u are two successive references to the same page, such that $u-t \geq \omega$ (or $\geq \theta$), the IRIM removes the page from the busy set immediately after r_t—at time $t+1$—while WS removes the page from the working set at time $t+\theta$. The IRIM also distinguishes page changes from simple accesses; it even models the fact that a page may be first changed, then simply accessed for a time, and then changed again without passing through an idle period.

Formal Definition

We describe the IRIM formally as follows:

Let $\mathrm{P} = \{p_i \mid i=1,2,...,m\}$ be a task's page set and $\{r_t \mid t=1,2,...,T\}$ be its reference string as defined in Section 3.1. Let I, C, and D be three mutually-exclusive sets of page-time pairs (p, t) from $\mathrm{P} \times \{1,2,...T\}$ such that $\mathrm{P} \times \{1,2,...,T\} = \mathrm{I} \cup \mathrm{C} \cup \mathrm{D}$. The elements of each set are defined by the mapping

$$\Phi_\omega: \mathrm{P} \times \{1,2,...,T\} \rightarrow \{\mathrm{I}, \mathrm{C}, \mathrm{D}\},$$

where:

1) $\Phi_\omega (p, t) \in C \cup D$ iff there exist a t_1 and t_2 such that:
   a) $t_1 \leq t \leq t_2$,
   b) $r_{t_1} = r_{t_2} = p$, and
   c) $t_1 - t_2 \leq \omega$;

2) $\Phi_\omega (p, t) \in D$ iff there exist a t_1 and t_2 such that:
   a) $t_1 \leq t \leq t_2$,
   b) $r_{t_1} = r_{t_2} = (p, true)$, and
   c) $t_1 - t_2 \leq \omega$.

The three sets contain page-time pairs that are *idle* (I), *clean* (C), and *dirty* (D). For convenience, we define a new set B, $B = C \cup D$, containing *busy* pages. Thus, at time t, page p is idle if t is in some interval $[t_1, t_2]$ of at least ω references and p is not referenced in the interval; otherwise p is busy. Likewise, if p is busy, it is also clean if t is in an interval $[t_1, t_2]$ of at least ω references and p is not modified in the interval; otherwise p is dirty.

Next, we define the set I_t by

$$I_t = \{p \mid \Phi_\omega (p, t) \in I \},$$

which contains all idle pages at time t. C_t, D_t, and B_t are defined similarly. At time t, we refer to the triple $\{I_t, C_t, D_t\}$ as the model state S_t.

To represent the IRIM by recording S_t for each t is both impractical and unneccessary. Due to the locality of references, pages are either busy or idle for long periods. They are also, in the IRIM sense, either clean or dirty for long periods. Thus, for practical values of ω, there are relatively few times t when $S_t \neq S_{t+1}$. When successive states S_t and S_{t+1} are not the same, there are at most two pages ($p = r_t$ and $p' = r_{t+1}$) which change sets:

1) p moves from C_t or D_t to I_{t+1} if $p \notin \{r_{t+1}, \ldots, r_{t+\omega}\}$; or
   p moves from D_t to C_{t+1} if p is not dirtied in $\{r_{t+1}, \ldots, r_{t+\omega}\}$.

2) p' moves from I_t to C_{t+1} or D_{t+1} if $p \in I_t$; or
   p' moves from C_t to D_{t+1} if r_{t+1} is a dirty reference.

In Section 5.1.3, it is shown that, for practical values or ω, S changes infrequently; a compact representation of the IRIM, known as the IRIM string, is a sequence of transition records that specifies each change to S. These records have the forms:

$(t, p, \text{"I"}, n)$ – p is idle in $[t, t+n)$
$(t, p, \text{"C"}, n)$ – p is clean in $[t, t+n)$
$(t, p, \text{"D"}, n)$ – p is dirty in $[t, t+n)$

The records are ordered by time t. The fourth element, n, is the *tenure* of the page in a particular state; it is used to predict the next transition of a page for lookahead algorithms.

IRIM String Computation

The computation of the IRIM string has two processing steps. In the first step, transition records are produced in a single pass over the reference string. These records are only partially time-ordered and must, in the second step, be sorted by time.

As each reference is processed, the state of each page p is defined by three variables:

- *Start* (p) – last time page entered either the **C** or **D** set,
- *LastUse* (p) – time of latest reference to page, and
- *LastDirty* (p) – time of latest page change.

Immediately prior to processing reference t, the following relationships hold for each p:

- $t - LastUse\ (p) \geq \omega \Rightarrow$ p is idle in the interval $(LastUse\ (p) + 1,\ t - 1)$
- $LastDirty\ (p) < Start\ (p) \Rightarrow$ p is clean in the interval $(Start\ (p),\ LastUse\ (p))$
- $Start\ (p) \leq LastDirty\ (p) \Rightarrow$ p is dirty in the interval $(Start\ (p),\ LastUse\ (p))$

When executed for each reference (p,d) in the original reference string, the following program fragment creates the IRIM transition records.

```
if t − LastUse (p) ≥ ω then   {page has completed idle state}
   begin if Start (p) ≤ LastDirty (p) then   {previous state is dirty}
      write (Start (p), p, "D", LastUse (p) − Start (p) + 1)
   else write (Start (p), p, "C", LastUse (p) − Start (p) + 1);
   write (LastUse (p) + 1, p, "I", t−LastUse (p) −1;
   Start (p) := t
   end

else begin   {page not idle; check for clean/dirty transition}
   {dirty reference may signal a clean→dirty transition}
   if d and (LastDirty (p) < Start (p)) and (t −Start (p) ≥ ω) then
      begin write (Start (p), p, "C", t − Start (p));
      Start (p) := t
      end;
   {clean reference may signal a dirty→clean transition}
   if not d and (Start (p) ≤ LastDirty (p)) and (LastDirty (p) − t > ω) then
      begin write (Start (p), p, "D", LastDirty (p) − Start (p) + 1);
      Start (p) := LastDirty (p) + 1
      end
   end;

LastUse (p) := t;
if d then LastDirty (p) := t;
```

Briefly, if two successive references to a given page are at least ω references apart, we know that (1) a busy state terminated at the time of the first reference, (2) an idle state occurred during the interval between the references, and (3) a new busy state began at the second reference. Records that describe both the previous busy state (either clean or dirty) and the idle state are written to an output file.

If a page is referenced in a busy state, it is tested for a **C** $\rightleftarrows$ **D** transition. This occurs whenever (1) a dirty page remains busy but is not dirtied for ω or more references, or (2) a page receives only clean references for ω or more references and is then dirtied. When a **C** $\rightleftarrows$ **D** transition occurs, any time *between* the states (i.e., after the last clean reference and before the first dirty reference, or vice-versa) is included in the **C** state record. Thus, the exaggeration of page dirtying in the model is minimized.

An additional mechanism detects the first reference to each page and avoids generating records for the period before that reference. When the reference string is exhausted, records are written for each page to describe a final return to the idle state.

Although the records for each page are produced in time-sequenced order, the IRIM string as a whole is not time-ordered. Since the beginning of an idle period is signalled by the *absence* of references to a page, it is not detected for at least ω references. Furthermore, it is much more efficient to detect the idle period when the page is referenced at the beginning of the *next* busy period. Thus, successive **B**$\rightarrow$**I** and **I**$\rightarrow$**B** transitions are both detected at the time of the latter transition.

To use the IRIM string to model a program, all records must be sorted by time. Although the non-linear nature of the sort phase might appear to be a drawback when processing a long reference string, the number of transition records remaining after the first processing phase is a small fraction of the original reference string. In our tests on strings of 10,000,000 references, the sorting cost is negligible compared to the the cost of generating the original reference string.

In the description of the two previous string reduction techniques in Section 3.3.4, Smith stresses the importance of methods which have processing times that are linear in the length of the string. He cites studies [HORO66, YU76] in which non-linear processing times limited the analysis to strings of 10,000 references, which, in the lifetime of ordinary programs, is an insignificantly short time. Although the analysis of the IRIM requires a non-linear sorting step, it is eminently practical for extremely long reference strings.

Modeling Program Behavior with the IRIM

To use the IRIM to model program behavior in a simulation model, we use the following two rules:

- At virtual reference time t, every page in C_t is referenced;
- At virtual reference time t, every page in D_t is both referenced and dirtied.

The sequence of states S_t is easily reconstructed from the IRIM string. We simply define an array to describe each page as being in one of the states: I, C or D. Initially, all $p \in I$. As reference r_t is simulated, the time is compared to the time of the next IRIM transition record; if they are equal, the transition is processed by making the appropriate change to the page description.

To simulate a reference, our model implies that each page in C is referenced and each page in D is referenced and dirtied. To perform this operation literally would be far less efficient than using the original reference string. We are easily rescued from this problem. When a page moves to C or D, a reference or dirtying is simulated, setting the page use- and dirty-bits. Subsequent references to the page within the current state have no effect (in our model) unless the operating system clears the frame activity bits or removes the page from the resident set. Proper use of the IRIM implies detection of these occurrences, and the model must reference and dirty the pages in C and D before task execution continues. A detailed algorithm for using the IRIM is presented in section 4.3.4.

Precise Modeling of VMIN and WS Policies

With the IRIM we model the precise calculation of the working set without introducing inaccuracies or incurring high cost. The IRIM may also be used to model the practical, approximate, implementations of WS. Finally, the IRIM can be used to model the lookahead VMIN algorithm.

To model WS and the other policies precisely, we merely require that $\omega \leq \theta$. (As shown in Section 5.1, practical values of ω are considerable smaller than practical values of θ.) When a page is busy ($p \in$ C or $p \in$ D) it is referenced at least once every ω references and, thus, at least once every θ references; a busy page is always a working set page. When a page transitions to state I, the state tenure, w, is recorded in the transition record. If $w \geq \theta$, then the page will leave the working set at time $t+\theta$; otherwise, it remains in the working set..

Suprisingly, the lookahead VMIN algorithm can be simulated *more* efficiently than the WSEXACT algorithm. Each time an idle state record is processed, the state tenure is compared to the VMIN parameter θ. If the page will be idle for more than θ references, it is removed from the working set immediately. Unlike the WSEXACT algorithm, the VMIN algorithm requires no WS scans to remove pages from the working set.

Accurate Modeling of Fixed-Space and Global Policies

The IRIM is also suitable for simulating fixed-space or global replacement, but with some simple safeguards. Consider the simulation of fixed-space LRU replacement. Each $p \in$ B is referenced within the last ω references, but the time of the last reference is unknown. Each $p \in$ I has not been referenced for at least ω references, and the last reference time is known exactly. If at least one resident page is idle, the LRU page can be found and replaced. If, however, all resident $p \in$ B, then the LRU page cannot be determined, and the IRIM, rather than introducing inaccuracies, fails altogether.

Suppose that an algorithm tries to avoid this problem by choosing a random $p \in$ B and replacing it. The model assumes that all $p \in$ B are referenced *constantly* when the task is executing. The task will fault immediately for the replaced page, without advancing the virtual time of the task, and some other busy page will need to be replaced. The task simulation will loop endlessly.

The reader may consider this feature of the IRIM to be a weakness; actually it is a great strength. LRU replacement is modeled with absolute precision unless this failure occurs. When the failure occurs, we can conclude that either (1) the parameter, ω, is too large or (2) the modeled system

has entered a thrashing state and further simulation is probably useless. Detecting the failure condition requires no special processing: the normal scan for the LRU page will reveal that no page is idle and the simulation can be halted at that point.

Simulating global page replacement with the IRIM is more complex but has the same problem. For reasonably small values of ω, the simulation fails only when memory is grossly constrained or overcommitted. We conclude that the IRIM is precise except for simulating memory partitions that would cause a real system to thrash badly. In this case, the model fails. If it is desired to model these situations, the IRIM of a program can be computed using a smaller ω, which increases the modeling cost, but provides the necessary accuracy.

There is a final, and most useful, property of the IRIM. It enables the accurate simulation of lookahead algorithms such as OPT and VMIN, both of which require knowledge of the time of next reference for each page. The tenure element of each idle state record is the virtual time that the page will be unreferenced. With this information, we can easily calculate the time of next reference.

The OPT (optimal fixed-space) algorithm can be simulated as efficiently as the LRU algorithm. When the program faults, a scan of its resident pages locates the page with the longest time *until the next reference* instead of the page with the longest time since the last reference. The IRIM will cause the OPT algorithm to fail in the same manner as the LRU algorithm: whenever the number of simultaneously busy pages exceeds the fixed-space memory allocation, it is impossible to determine which has the longest time until the next reference.

The IRIM would be highly feasible for evaluating global memory management strategies that use lookahead information. As far as we can determine, no such strategy has ever been proposed; one reason for this omission is the difficulty of analyzing or simulating such a strategy.

4.3.4 Memory Referencing Model

The program behavior model generates the equivalent of reference pairs (*p*, *d*), where *p* is a *Page* and *d* is a Boolean dirty reference indicator. The following routine, which simulates a single reference, is common to all of the program behavior models:

```
function PageIsResident (var Page: Page, d: Boolean): Boolean;
begin if not Page.Resident then PageFault (Page);
   if Page.Resident then
      begin if WSExact
         then Page.Frame.LastReferenceTime := Task.VirtualTime
         else Page.Frame.Use := true;
      if d then Page.Frame.Dirty := true;
      PageIsResident := true   {reference succeeds}
      end
   else PageIsResident := false;   {reference fails}
end;
```

The routine *PageIsResident* simulates a page reference and detects a page fault if the page is not resident. If the page is resident, the routine sets the hardware frame activity bits appropriately and returns a Boolean value *true*. If the page is not resident, the page fault processing routine is invoked and the *PageIsResident* routine returns the value *false*.

Note that the routine tests the *Resident* status of the page twice: once to call the *PageFault* routine if the page is not resident, and then to set the activity bits and determine the value (*true* or *false*) to return. In some circumstances, the missing page is made *Resident* immediately by the *PageFault* routine; thus, the second test of *Resident* detects this fortunate occurrence.

If the WSEXACT replacement algorithm is modeled, the hypothetical hardware facility is simulated to store the precise *VirtualTime* of the reference. The decision not to set the use-bit in this case is intentional as described in section 4.4.4.

Reference String Model

The representation of the reference string model is straightforward, as shown in the following data structure:

```
type Reference = record
    p: PageNumber;
    d: Boolean
end;
var ReferenceString: array [1..T] of Reference;
```

For descriptive purposes, the reference string is represented as an array. In actual use, this array would require too much storage and, therefore, is implemented by reading file blocks as necessary. The referencing simulation is described by the following routine:

```
while Task.VirtualTime < InterruptTime and not Blocked do
    begin with ReferenceString (Task.VirtualTime) do {next (p,d)}
        if PageIsResident (Task.PageTable(p), d) then
        begin Task.VirtualTime := Task.VirtualTime + 1;
            RealTime := RealTime + 1
        end
        else Task.Blocked := true
    end;
{return to scheduler}
```

The variable *InterruptTime* is set by the operating system model as described in Section 4.4.3. It limits the number of references a task can perform before control must be returned to the scheduler. Since *VirtualTime* is defined to be the number of completed references, it is used as the *ReferenceString* array index. Whenever a reference is unsuccessful, the task is blocked and control is returned to the scheduler.

Inter-Reference Interval Model Strings

Each element of the IRIM string is a 4-tuple and is described by the following data structure:

```
type IRIMstate = {idle, clean, dirty};

    IRIM = record
    time: time;      {time of state change}
    p: PageNumber;
    state: IRIMstate;
    tenure: time {state holding time}
end;
```

The model of each virtual memory page, described in Section 5.3.4, includes two Boolean variables, *IRIbusy* and *IRIdirty*. *IRIbusy* is set whenever the page is in either a clean or dirty IRIM state, while *IRIdirty* is set whenever the page is in a dirty state. The routine to maintain these indicators and to simulate memory referencing has two parts. The first part ensures that the virtual memory model is consistent with the current state of the IRIM, while the second part processes new IRIM records and updates the virtual memory state.

Although it is not necessary to reference each busy page at every simulated reference time, we must ensure that each busy page is resident and that its frame activity bits are set whenever the task is executed. Since the IRIM contains predictive information that is not normally known to the operating system the model should not prevent the operating system from changing these bits or removing busy pages unless a lookahead algorithm such as VMIN is modeled.

In this model, the IRIM information is hidden from the operating system model, but the operating system model informs the memory referencing model whenever it resets a use- or dirty-bit or replaces a busy page. It does this by setting the task's *SwapIn* Boolean variable. When the task is next executed, the execution model simulates references to all busy pages to ensure that they are resident and to set their activity bits. The routine to perform this immediately precedes the basic referencing routine:

```
if Task.SwapIn then
   for i := 1 to MaxPage do  {ensure that all busy pages are resident and referenced}
   begin with Task.PageTable (i) do
      if Page.IRIBusy then
         if not PageIsResident(Page, Page.IRIDirty) then goto "scheduler"
   end;
Task.Swapin := false;
```

If a busy page has been removed from the resident set, a page fault occurs and the task is blocked until the page is made resident. When the task is unblocked, there will be another scan of the busy pages until it is verified that all are simultaneously resident.

Once all busy pages are resident, memory referencing is simulated until a page fault occurs or the next *InterruptTime* is reached. *VirtualTime* is incremented in large steps corresponding to the transitions of the IRIM. If an IRIM record occurs before *InterruptTime*, *VirtualTime* is updated to the time of the record which is accepted and processed. Otherwise, *VirtualTime* is updated to *InterruptTime* and the interrupt is simulated. The routine to process the IRIM string follows:

```
while (Task.VirtualTime < InterruptTime) and not Task.Blocked do
  with Task.IRIMString [Task.StringPointer] do {select next IRIM record}
    if IRIM.time > InterruptTime then     {interrupt before next IRIM record}
      begin RealTime := RealTime + (InterruptTime – Task.VirtualTime);
      Task.VirtualTime := InterruptTime
      end
    else     {process IRIM record}
      begin RealTime := RealTime + (IRIM.time – Task.VirtualTime);
      Task.VirtualTime := IRIM.time;
      with Task.PageTable (IRIM.p) do {select Page}
        case IRIM.state of
        clean :
          begin Page.IRIbusy := true;   Page.IRIdirty := false;
          if not PageIsResident (Page, false) then Task.Blocked := true
          end
        dirty :
          begin Page.IRIbusy := true;   Page.IRIdirty := true;
          if not PageIsResident (Page, true) then Task.Blocked := true
          end
        idle :
          begin Task.VirtualTime := Task.VirtualTime – 1;
          RealTime := RealTime – 1;
          if not Page.Resident and Page.Frame.Used then LogicError;
          if Page.IRIdirty and not Page.Frame.Dirty then LogicError;
          Page.IRIbusy := false;   Page.IRIdirty := false;
          if WSExact then
            begin if IRIM.tenure > Theta
              then Page.Frame.LastReferenceTime := VirtualTime
              else Page.Frame.LastReferenceTime := infinity;
            Task.WSScanTime :=
                  min(Task.WSScanTime, Page.LastReferenceTime + Theta)
            end
          if VMIN and (IRIM.tenure > Theta) then
            begin Page.InWorkingSet := false;
            Page.Frame.Used := false;
            end
          end;
      Task.StringPointer := Task.StringPointer + 1
      end;  {return to scheduler}
```

A *clean* or *dirty* IRIM record causes a page reference and, possibly, a page fault and return to the scheduler. If the page is resident, the *IRIbusy* and, if appropriate, *IRIdirty* indicators are set. Note, in particular, that a transition from the dirty to the clean IRIM state has no direct effect on the frame dirty-bit; the responsibility to clean the page and clear the dirty-bit lies with the operating system model which does not access the IRIM state information.

An *idle* IRIM record marks the end of a busy state and clears the *IRIBusy* and *IRIDirty* indicators. If the page is not available and active (*Used* and *InWorkingSet*) at this time, there is an inconsistency in the model of task execution.

In the modeling of *practical* replacement algorithms, the entering of the idle state is not an event which is detectable and should not have any direct affect on those algorithms. On the other hand, the idle state record is the key to efficient calculation of the working set under the WSEXACT and VMIN replacement algorithms. If the idle time (*IRIM.tenure*) exceeds θ, the WSEXACT algorithm sets *LastReferenceTime* exactly and schedules an operation to remove the page from the working set in exactly θ references; the VMIN algorithm removes the page from the working set immediately.

4.3.5 I/O Request Model

Task I/O requests are generated stochastically with inter-I/O times measured in the virtual time of the task. Inter-I/O times are independent and identically distributed for each task; the distribution is defined by the task parameters: *MinInterIOTime*, *MeanInterIOTime*, and *MaxInterIOTime*, which specify a constant, uniform, or exponential distribution.

When an I/O request occurs, the task device selection vector, $\{p_i \mid i=1,2,...,d\}$, $0 \leq p_i \leq 1$, and $\Sigma p_i = 1$, specifies the frequency of access to each device, D_i. The service time of each device is determined by the I/O device model, which is independent of the task model.

4.4 Operating System Model

4.4.1 Overview

The design and implementation of the operating system model are major factors in the fidelity and credibility of this computer system model. Most of the mechanisms used to model the scheduling of tasks and the management of virtual memory are exact representations of the corresponding mechanisms that would be required in a real operating system.

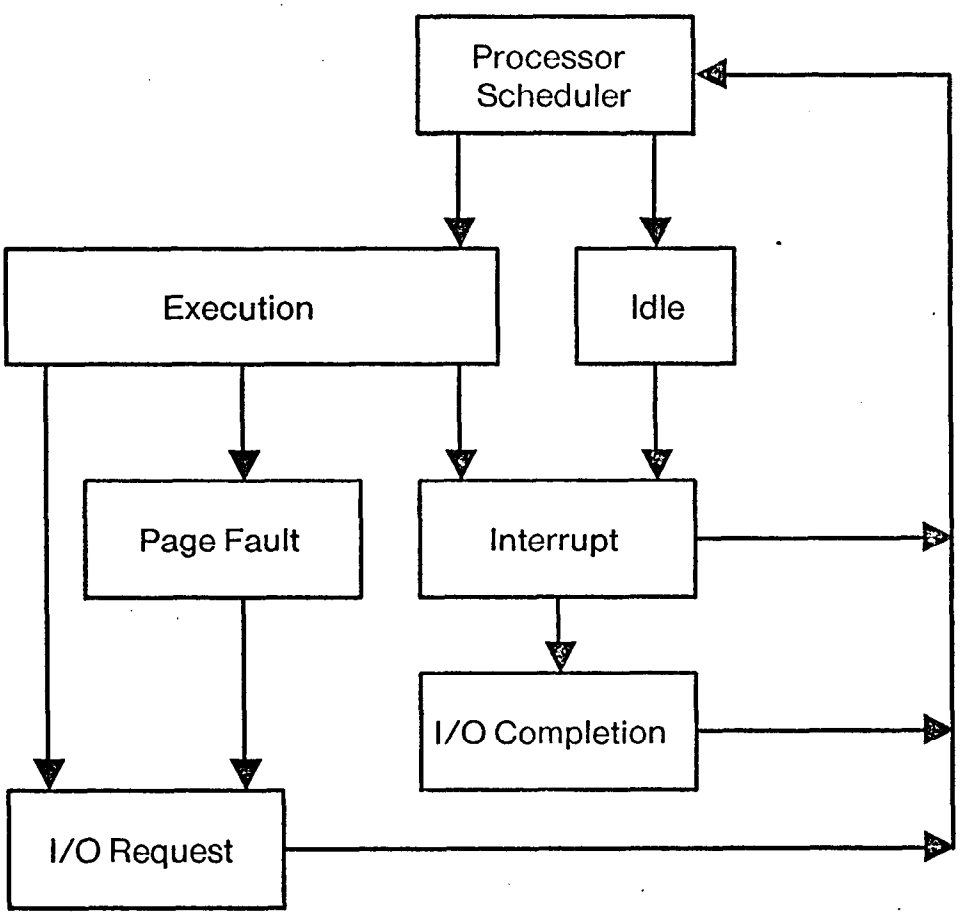


Figure 4.4 - Operating System Model

The flow of control in the operating system model is illustrated in Figure 4.4. When the scheduler gains control, it attempts to select a task for execution. If no task is eligible to execute, the system enters an idle state until an interrupt, which is caused by the next event on the event

queue, occurs. When a task is selected, it is executed until it (1) page faults, (2) makes an I/O request, or (3) is interrupted by the next event. A page fault typically causes the system to make an I/O request and return control to the scheduler. Most interrupt events are I/O completions or scheduler functions such as quantum expiration.

The operating system model has four major components:

1) The *scheduler* allocates task priorities, time-slices, and quantums, selects tasks to execute and invokes functions such as load control. It also models idle periods when no tasks are executing.

2) The *task execution processor* models memory referencing and detects page faults and interruptions.

3) The *virtual memory manager* processes page faults, allocates page frames, makes replacement decisions, performs some load control functions, and manages auxiliary memory.

4) The *device scheduler* processes task and paging I/O requests.

4.4.2 Scheduler

The development of the scheduler model follows the description of scheduling methods in Chapter 2. The admission queue is modeled as a never-empty, randomly-ordered queue of tasks chosen from a set of task prototypes. Each task prototype contains a program behavior model and an I/O request model dervived from measurement of a real program. A model parameter *MaxTasks* specifies the number of simultaneously-admitted tasks; when any task terminates, a new task from the admission queue takes its place.

Once admitted, tasks are scheduled using the ready queue-active queue structure described in Sections 2.1.2 and 2.1.3. Figure 4.5 illustrates the flow of tasks in the model. The ready queue is a multi-level load-balancing queue which can also be parameterized to operate as a FCFS or a

RR queue. The active queue is managed to effect an approximate processor-sharing discipline by using a RR queue with a small quantum.

Tasks are activated in accordance with the load control. A task remains in the active queue until it (1) terminates, (2) consumes its time-slice, or (3) is demoted by the load control. A demoted task is given a special priority which keeps it at the front of the ready queue until it can be re-activated; its priority in the multi-level queue then reverts to the previous value. The model incorporates the various methods of choosing a task to demote described in section 2.3.5.

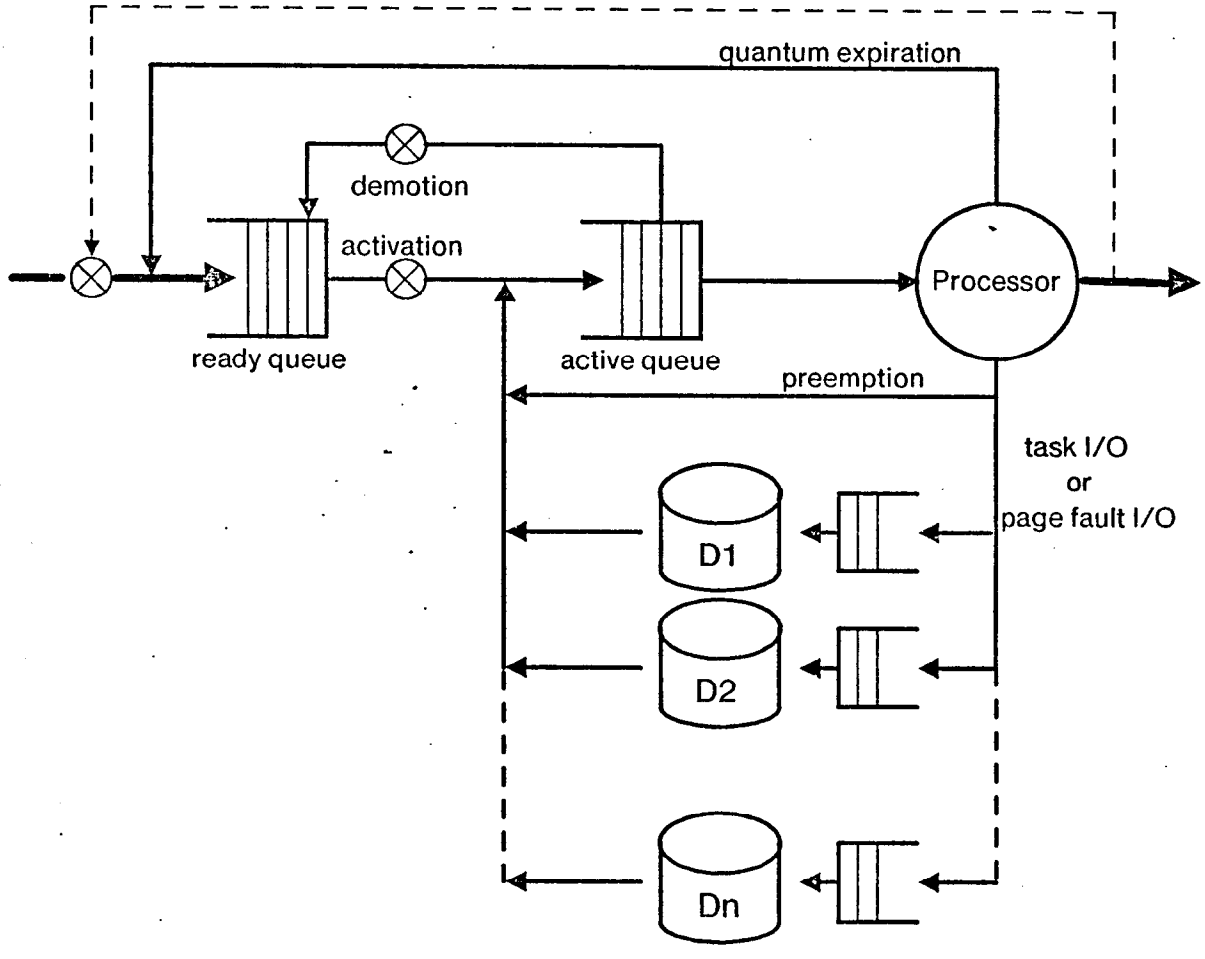


Figure 4.5 - Scheduler Model

4.4.3 Task Execution

Once a task is selected for execution, the scheduler calculates a virtual *InterruptTime*. *InterruptTime* is the time that the first of the following events is predicted to occur:

1) The next event on the simulation event queue occurs;
2) The task time-slice or quantum is consumed;
3) The task makes an I/O request;
4) The task requires a working set scan;
5) The task terminates.

Execution proceeds by processing the memory referencing model as described in section 4.3 until the virtual *InterruptTime* is reached or a page fault occurs. Once the interrupt or fault is processed, control is returned to the scheduler to select a task to execute.

4.4.4 Virtual Memory Management

The management of virtual memory includes page fault processing, page replacement, auxiliary memory allocation and paging I/O, and either working set estimation or an adaptive load control. The main data structures used to model the virtual memory hierarchy are the task virtual memory pages, main memory frames, and auxiliary memory slots. Figure 4.6 illustrates the relationship between these elements.

As described in section 4.2.2, the frame model contains a number of variables which describe its memory management state. A frame is *Free* whenever it does not contain a task page; it can be simply assigned to a task. If a frame is not *Free*, it is in the virtual memory of some task, as defined by *Owner* and *VirtualAddress*; this is the essential information required to remove the page from the owning task's resident set. The variables *Next*, *Previous*, *PageIn*, *PageOut*, and *PageOutCount* are used by the auxiliary memory manager to schedule paging I/O operations.

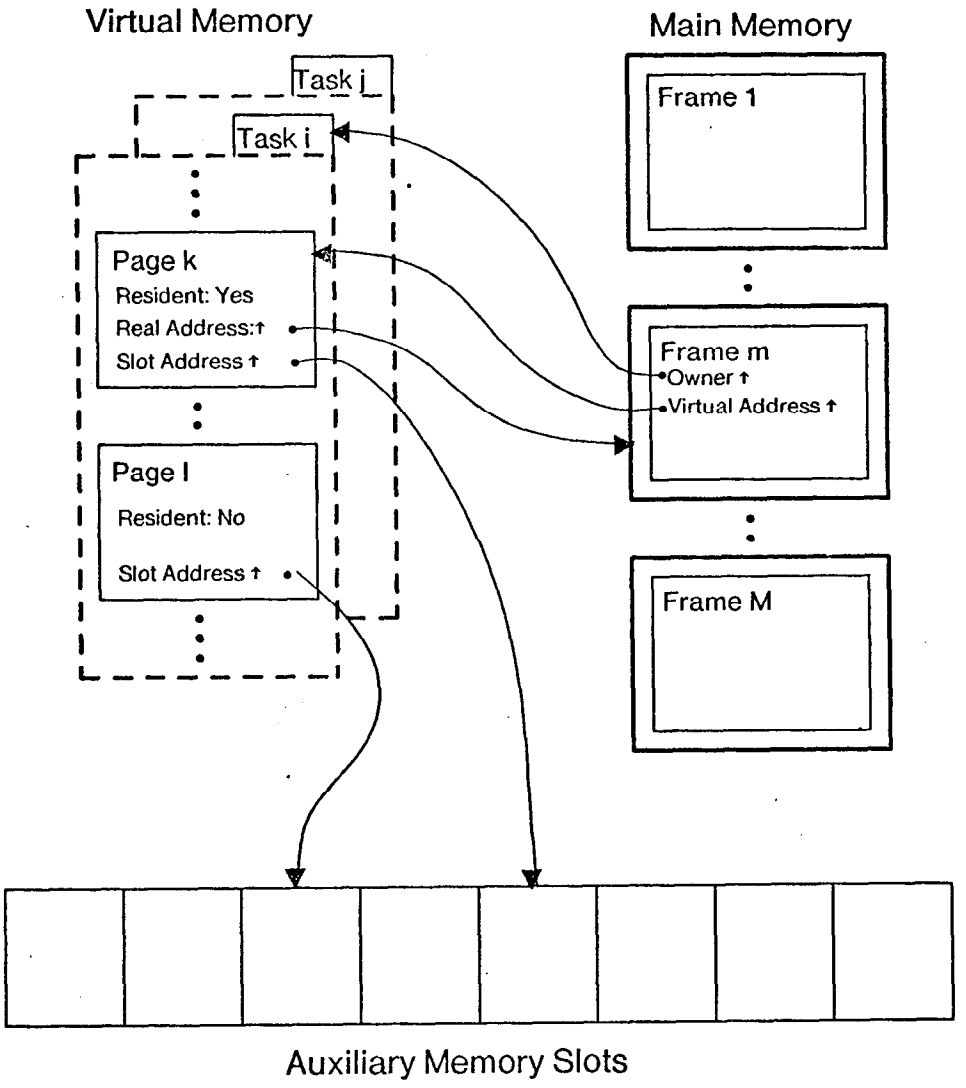


Figure 4.6 - Memory Management Structures

Page Fault Processing

To process a page fault, the system must allocate a frame and perform a paging I/O operation to load the missing page. The basic routine used in the model to process a page fault follows:

```
procedure PageFault (var Page: Page);
    begin if WSFault then WSScan;
    if Replacement (Frame) then
        begin Frame.Free := false;
        Frame.Owner := ↑ Task;
        Frame.VirtualAddress := ↑ Page;
        Page.RealAddress := ↑ Frame
        if not Page.Resident then
            begin PageIn (Frame);
            Task.Blocked := true
            end
        end
    else {no frame is replaceable}
        begin Task.Blocked := true;
        Task.WaitingForFrame := true;
        SomeTaskWaitingForFrame := true {system-wide indicator}
        end
    end;
```

Under the WSFAULT replacement algorithm, the system performs a scan of the task working set at each page fault prior to the request for a frame. The procedure *Replacement*, described in the next subsection, attempts to find a frame to hold the missing page, and returns the value *true* (and a pointer to the frame) if it is successful. The frame is assigned to the faulting task by setting the frame's *Owner* and *VirtualAddress* and by updating the page table entry with the *RealAddress* of the frame. The *Resident* indicator is not set until the page is transfered by the paging I/O operation that is requested in a call to the auxiliary memory manager routine *PageIn*. The task is *Blocked* until the paging I/O is completed.

If the call to *Replacement* returns the value *false*, no frame is available and the task is *Blocked* in a special *WaitingForFrame* state. This condition usually occurs only when it is necessary to wait until some dirty pages have been cleaned before they can be replaced. The operating system remembers that this unusual, but not necessarily rare, condition has occurred by setting a system *SomeTaskWaitingForFrame* indicator. As long as this indicator is set, all newly faulting tasks are also placed in the *WaitingForFrame* state. When the completion of a page-out operation makes a frame replaceable, *WaitingForFrame* tasks are unblocked to retry the faulting reference.

Page Replacement

In Chapter 2, we presented WS and CLOCK as two diverse and distinct strategies of managing virtual memory. In this chapter, we demonstrate that, due to the universality of the clock algorithm, their implementation can have much in common. Each class of algorithm can benefit from techniques normally used in the other without losing its identity or any of its basic advantages. We also describe the new algorithm, WSCLOCK, which attempts to combine the best features of both WS and CLOCK.

The model selectively incorporates either of the two main replacement algorithms, WS and CLOCK, in one of the variations, WSEXACT, WSFAULT, etc. Every algorithm uses the clock algorithm to locate a single replaceable page each time a fault occurs. Every algorithm uses the same procedure, *Replacement*, to find a replaceable page. This algorithm is shown on the next page.

If the system has previously determined that there is a scarcity of clean replaceable frames, as indicated by the Boolean *SomeTaskWaitingForFrame*, this routine exits immediately. In this case, *Replacement* returns a *false* value and the faulting task is blocked until the scarcity is relieved. Otherwise, the routine scans frames in the circular clock order until it finds one to assign to the faulting task. Since this routine is complex and is central to the development of the operating system model, we provide a flowchart of the replacement scan in Figure 4.7.

```
function Replacement (var Frame: Frame): Boolean;
begin Replacement := false;
if not SomeTaskWaitingForFrame then
   begin Frame := LastFrameReplaced;
   repeat Frame := Frame.Next;
      Page := Frame.VirtualAddress;

      if Clock or WSClock then
         if Frame.Used then {test frame activity indicator}
            begin Page.LastReferenceTime := Owner.VirtualTime;
            Page.InWorkingSet := true;
            Frame.Used := false
            end
         else if Owner.VirtualTime– Page.LastReferenceTime < Theta
            then Page.InWorkingSet := false;

      if Frame.Free or not Page.InWorkingSet or not Owner.Active then
         if Frame.Dirty then PageOut (Frame)
         else Replacement := true
   until Replacement or Frame = LastFrameScanned;

   {check for failure and remedy if possible}
   if not Replacement then
      if Clock then {CLOCK: choose any replaceable frame}
         begin repeat Frame := Frame.Next
         until not Frame.Dirty or Frame= LastFrameScanned;
         if not Frame.Dirty then Replacement := true
         end
      else if (MemorySize– WSPool ≤ MemoryCommitment) then
         begin Demote;
         if CurrentTask.Active then Replacement := Replacement (Frame)
            end;

   {steal page from curent owner}
   if Replacement and not Frame.Free then
      begin Page.Resident := false;
      Frame.Free := true
      end;
   LastFrameScanned := Frame
end;
```

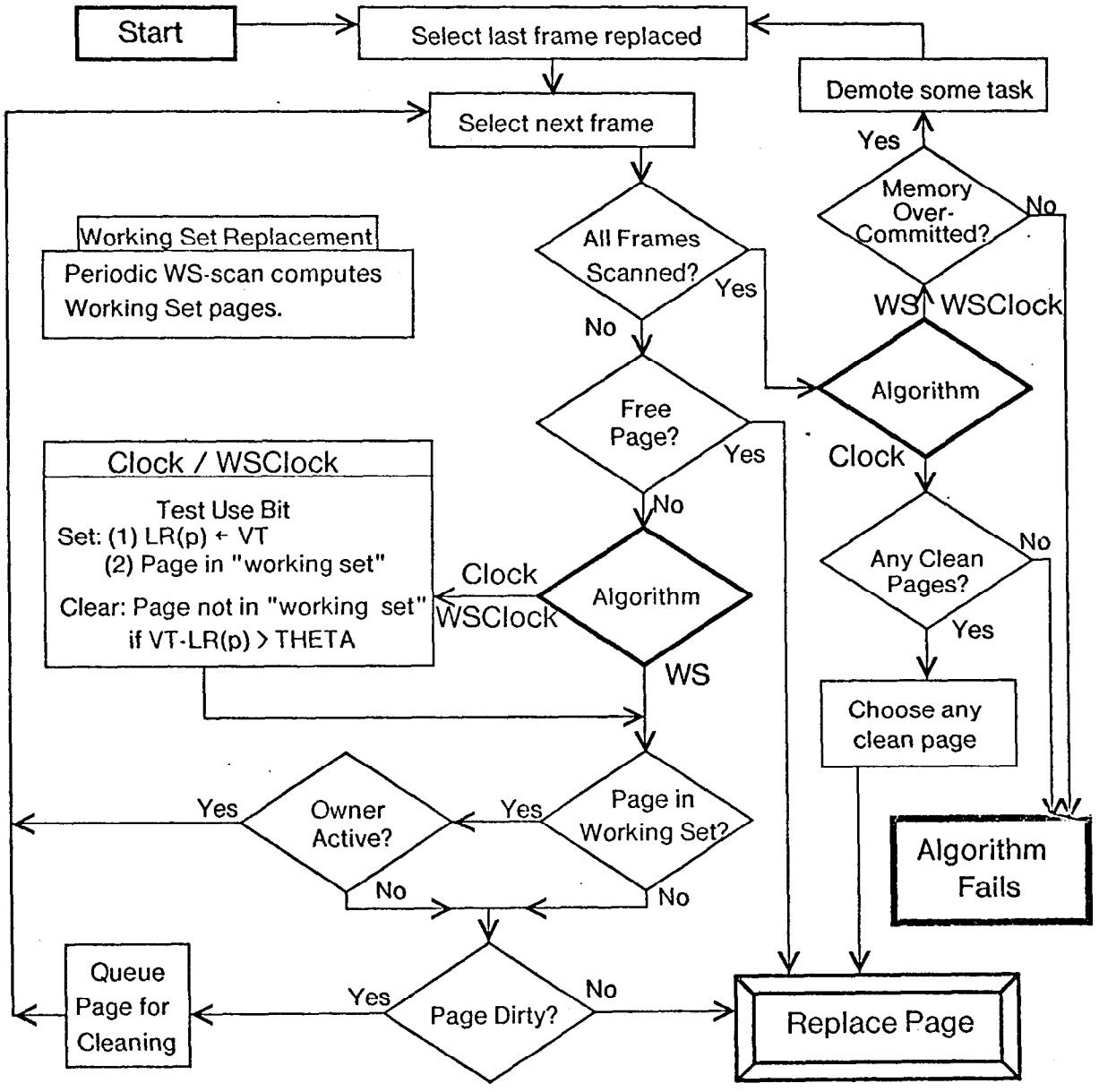


Figure 4.7 - Replacement Algorithm

Briefly, the routine uses the clock scan algorithm to search for the first clean page that is not in an active-task's working set. The essential difference between WS and CLOCK lies in (1) the definition of "working set" and (2) the action taken when no page is found to replace. Each page has an associated *InWorkingSet* indicator which is not specific to WS, but, rather, is also used to mark pages that CLOCK should not replace unless necessary.

Under WS, the indicator is, in effect, maintained as follows:

1) *InWorkingSet* is set whenever the page is referenced;
2) *InWorkingSet* is cleared whenever the page is unreferenced for more than θ.

The task variable *WorkingSetSize*, which contains the number of *InWorkingSet* pages, is the amount of memory committed to the task.

The implementation of the first rule is not as straightfoward as it seems. If a referenced page is already *InWorkingSet* or is not resident, no problem arises. If, however, a referenced page has been removed from the working set, but has not been removed from the resident set and replaced, the system will not receive an indication that the page has returned to the working set until the working set is scanned (see the next subsection). This may cause the WS load control principle to be violated, since the amount of committed memory will be underestimated.

One solution is to remove a non-working-set page from the resident set even though it has not been replaced; if it is referenced, a page fault will occur and the page can be reclaimed and made both resident and *InWorkingSet* at the same time. The disadvantage of this method is the cost of processing the page fault and the unnecessary mechanism to reclaim the page. See Section 2.2.5 for a general discussion of page reclamation.

In our model, we do not attempt to analyze algorithmic overhead, and so we assume that the first rule can be implemented accurately without using a page reclamation mechanism. This is a good example of the practical difficulties encountered in the implementation of WSEXACT even if the *LastReferenceTime* register is implemented in the hardware.

Clearing the *InWorkingSet* indicator is performed by the *WSScan* routine described below, which is invoked independent of the replacement algorithm.

Under CLOCK, we also use *InWorkingSet* in a simpler mechanism that sets and clears the indicator as part of the replacement clock-scan. Since CLOCK load control does not use an estimation of the task's working set size, the setting of *InWorkingSet* is irrelevant except when the page is scanned for replacement. In this case, the frame use-bit indicates any recent references. If the use-bit is set, then (1) *InWorkingSet* is set, (2) the task's current *VirtualTime* is saved as the page's *LastReferenceTime*, and (3) the use-bit is cleared. *InWorkingSet* is cleared if the page was not referenced since the last replacement scan and the elapsed virtual time since the *LastReferenceTime* is greater than θ.

Recall, from Chapter 2, that this last criterion has a very different purpose from the similarly-defined WS criterion. The typical CLOCK θ is very small and is intended to ensure that an active but I/O-bound or low priority task is given some opportunity to reference its resident pages between successive replacement scans. The parameter is not used to distinguish active from inactive pages.

Returning to the basic replacement algorithm for both WS and CLOCK, a page should be replaced if it is either (1) not *InWorkingSet* or (2) owned by an inactive task. If, however, the page is dirty, it must be cleaned before being replaced; it is queued for a writing to a slot and skipped in the replacement scan until it is clean. Otherwise, the replacement algorithm is complete: the page is removed from the current owner's resident set and the clock-scan pointer is set to start with frame following the chosen frame.

WS and CLOCK diverge again when a complete scan of all frames fails to find a replaceable frame. To WS, this signals one of two conditions. If (almost) all frames contain active-task working-set pages, then main memory is overcommited. This causes a task to be demoted, which should make some pages replaceable. Unless the faulting task is itself demoted, the replacement algorithm is then invoked recursively. If, however, there are a number of dirty-but-otherwise-

replaceable pages (the minimum number being equal to the WS free page pool size), then memory is not overcommitted and the faulting task must wait for a frame to be cleaned.

If CLOCK cannot find a replaceable frame in a complete scan, it simply chooses the first clean frame ahead of the clock-scan pointer. CLOCK fails only when all frames contain dirty pages.

Working Set Scanning

Each WS algorithm, except VMIN and WSCLOCK, performs the *WSScan* routine at various times to remove non-working set pages and to calculate the task's *WorkingSetSize*:

```
procedure WSScan;
var LRUTime: time;
begin
Task.WorkingSetSize := 0; LRUTime := infinite;
for Page in Task.PageTable do {scan all task Pages}
      begin
      if Page.Resident and Page.RealAddress ↑ Frame.Used then
            begin Page.RealAddress ↑ Frame.Used := false;
            Page.LastReferenceTime := Task.VirtualTime;
            Page.InWorkingSet := true
            end
      else if Task.VirtualTime − Page.LastReferenceTime ≥ Theta
            then Page.InWorkingSet := false;
      if Page.InWorkingSet then
            begin Task.WorkingSetSize := Task.WorkingSetSize + 1;
            if Page.LastReferenceTime ≤ LRUTime then
                  begin Task.LRUPage := ↑ Page;
                  LRUTime := Page.LastReferenceTime
                  end
            end
      end;
if WSFault or WSInterval then Task.WSScanTime := Task.VirtualTime + WSScanInterval;
if WSExact then Task.WSScanTime := LRUTime + Theta
end;
```

WSScan examines all task pages, including non-resident ones. If a page is resident, its use-bit is checked; if it has been referenced since the last *WSScan*, the page's *LastReferenceTime* is estimated to be the current *VirtualTime*. Any page which has been referenced within the last *Theta* (θ) references is *InWorkingSet*.

Under WSEXACT, the execution model does not set the frame use-bit, but, instead, always computes the appropriate *LastReferenceTime*. With the ordinary reference string model, the *LastReferenceTime* is set correctly for every page. With the IRIM, the *LastReferenceTime* is set only for pages that are predicted to be unreferenced for θ or more references; for other pages the value is "infinity".

The scheduling of the working set scan is the fundamental difference between the WS variants. The practical variants (WSFAULT, WSINTERVAL) schedule a scan at a fixed *WSScanInterval*. *WSScan* determines the task's *LRUPage*, which is the *InWorkingSet* page which has not been referenced for the longest time. WSEXACT schedules the next scan for θ references from the *LastReferenceTime* of the *LRUPage*, which is the soonest possible time a page can leave the working set. If the IRIM is used to model WSEXACT, the *LRUPage* is certain to leave the working set at that time; thus, the IRIM minimizes the number of *WSScans*.

VMIN and WSCLOCK Working Set Estimation

The VMIN algorithm can be modeled only if the IRIM is used. Whenever the execution model processes an idle state transition and $IRIM.tenure > \theta$, the page is immediately removed from the working set. The set of *InWorkingSet* pages and the task's *WorkingSetSize* are maintained precisely. Thus, the lookahead VMIN algorithm is modeled simply, accurately, and more efficiently than the ordinary WS algorithms.

WSCLOCK retains the essential nature of the local WS policy, while eliminating working set scans. It calculates the task's working set by examining each resident page in each circuit of the replacement clock-scan. WSCLOCK uses the same criteria as CLOCK, but with a value of θ that is more suited to a WS algorithm.

The primary problem with WSCLOCK occurs each time a task is activated. Since WSCLOCK examines only resident pages, it will not remove pages that are not resident at activation and leave the working set after a short while. After a task has been active for θ, the resident working-set pages will be the true working set. Thus, there is an interval following each task activation in which WSCLOCK has no mechanism to calculate the task's *WorkingSetSize*, which may lead to the improper activation of new tasks and overcommitment of memory.

The following solution is used in this model. When a task is deactivated, its *WorkingSetSize* is the number of resident working-set pages; this value is used when the task is considered for a subsequent activation. After the task is re-activated, its *WorkingSetSize* is the number of *remaining* resident working-set pages, if any. As the task executes, it loads additional working-set pages and the *WorkingSetSize* soon becomes a more accurate estimate. The performance of the WSCLOCK algorithm may be especially dependent on the use of the load control heuristic which limits the number of loading tasks; when the loading phase has ended, the task's *WorkingSetSize* should be reasonably accurate to permit the WS load control to estimate the amount of uncommitted memory. A more complete description of WSCLOCK and its load control may be found in [CARR81].

Load Control

The remaining load control methods are implemented as they are described in Section 2.3. WS load control (2.3.2) has two main parameters, *Theta* (θ) and *WSPool* (K_0). CLOCK's load control (2.3.4) has one main parameter C_0 and the smoothing and confidence interval parameters, δ, α, and φ. The *LT/RT* control (2.3.5) and demotion task choices (2.3.6) are also implemented as described.

4.4.5 Auxiliary Memory Management

Overview

Except for the order in which page transfer requests are processed, the structure of the auxiliary memory manager (AMM) model is highly simplified. There is no explicit representation of slots; it is assumed that they are an unlimited resource. Only one page can be transferred in each paging I/O operation. If multiple paging I/O devices exist, it is assumed that paging I/O requests are balanced across the devices; the model will use any available paging device to process a transfer request. As with the I/O device model itself, a more complex model would be useful to study questions that are not considered here. For example, a comparison of demand paging with either pre-paging or swapping would require modeling the concurrent transfer of many pages in a single I/O operation.

The interface to the AMM is through two routines: *PageIn* and *PageOut*, both of which specify a single *Frame* as an argument. The *Frame* model contains the identity of the *Page* to be transfered. The Frames are placed in a *PageIOQueue* according to one of three disciplines: (1) the FIFO discipline is ordered by the time of the transfer request; (2) the Reads-First discipline places all *PageIn* requests before all *PageOut* requests; (3) the Running-Reads-First discipline places *PageIn* requests for running tasks first, then *PageIn* requests for loading tasks, followed by *PageOut* requests. The distinction between running and loading tasks is described in Section 2.3.4.

Page Files

A page file is not a device in its own right; it is simply an indirect reference to an ordinary device managed by the I/O subsystem. The page I/O scheduler makes requests to the I/O subsystem in the same manner as tasks make I/O requests. If the model is configured to permit it, both task and page I/O can contend for the same devices; if the AMM makes an I/O request on a device that is processing task I/O requests, all requests are processed in a FIFO order.

```
type PageFile = record
        Busy : Boolean;
        Device: ↑ Device;
        Frame: ↑ Frame;
        IORequest: IORequest
end;
```

The *PageFile.Busy* indicator is separate from the *Device.Busy* indicator in the I/O device model. *PageFile.Busy* signifies that a paging I/O transfer request is being processed on the device; a given *PageFile* can make only one request at a time and must keep other requests pending in the *PagingIOQueue*. If more than one *PageFile* specifies the same *Device*, this indirectly specifies the number of simultaneous page I/O requests that can be queued to the *Device*.

Whenever a *PageFile* is not busy and there is an outstanding request in the *PagingIOQueue*, the AMM initiates a transfer by making an I/O request. If the request is a *PageOut*, the frame dirty-bit is cleared at this time. Note that the operating system never removes a page from a task's resident set until that page is actually replaced. Thus, the page can be referenced and dirtied during the *PageOut* transfer, setting the use- and dirty-bits once again.

The AMM processes a completed *PageIn* request by marking the page resident and unblocking the task. When a *PageOut* completes, the AMM checks the *SomeTaskWaitingForFrame* indicator; if it is set, the AMM unblocks a task which is *WaitingForFrame* since the *PageOut* operation has (probably) made a page clean and replaceable; if there are no *WaitingForFrame* tasks, the system indicator is cleared.

Chapter 5

Empirical Studies

This chapter describes the use of the model presented in Chapter 4 to study policies of virtual memory management. In Section 5.1, we describe the selection, measurement, and analysis of sample programs. Execution traces of these programs are processed to create reference strings and IRIM strings. In Section 5.2, we describe the preparation of the simulation model. A sample load is constructed using the sample programs. The IRIM is validated as an accurate and highly efficient replacement for the reference string model.

In Section 5.3, we study the WSEXACT memory management policy. WSEXACT is tuned for best performance over various strategies and parameter settings. We also study WSFAULT and WSINTERVAL, the commonly-used, practical approximations to WSEXACT, and VMIN, the unrealizable, lookahead, version of WSEXACT. In Section 5.4, we study WSCLOCK, the newly-developed WS policy.

In Section 5.5, we study the GLOBAL replacement algorithm and load control, and tune them for best performance over various strategies and parameter settings. In Section 5.6 we compare the results of the three previous sections and draw conclusions about the relative performance of the algorithms studied.

5.1 Task Model Preparation

5.1.1 Program Sample

Eight programs are selected for use in this study. These programs are among the most commonly used at the central computing facility of Stanford Linear Accelerator Center (SLAC). Although computing at SLAC is heavily scientific, the sample contains mostly utility programs that are in common use on educational and commercial systems as well. As far as we can determine, none of these programs have been explicitly written or adapted for a virtual memory environment. Each program has a four-letter indentification for use in later descriptions. The programs are:

1. LKED – IBM 370 Linkage Editor: This program links compiler output (object modules) and previously compiled library routines to create a load module. This program is often the most frequently executed program on IBM 370 systems.

2. SORT – SYNCSORT utility program: This program is an improved version of IBM SORT/MERGE. The program is measured while sorting some 10,000 randomly-ordered records. A small virtual memory allocation is specified to force the program to use a combination of internal and external sorting methods.

3. FORC – IBM FORTRAN H compiler: This compiler is very large and, particularly when optimizing, slow. Most of the compiler itself is written in FORTRAN. The compiler is measured compiling 210 statements in 2 subroutines, using the highest optimization level (OPT=2).

   Due to its large size and a multi-phase compilation procedure, the compiler is usually overlaid explicitly. For this study, however, the compiler is not overlaid and is executed in a large memory allocation.

4. WATF – University of Waterloo FORTRAN IV compiler, WATFIV: This compiler is about 1/5th the size and 5 to 10 times as fast as the FORTRAN H compiler. It is a debugging

compiler, which produces unoptimized code that makes many run-time error checks. The program is measured compiling 1800 statements in 20 subroutines.

5. ASMC – IBM Assembler H: This program translates a 575 statement program in 370 Assembler Language.

6. SCRP – University of Waterloo SCRIPT text processor: This program formats text, including right-margin justification and hyphenation.

7. PASC – Stanford Pascal compiler: This program is a top-down compiler written in Pascal. The program is measured compiling a large part (about 1100 lines) of the compiler itself.

8. DRAW – SLAC TOPDRAW graphics utility program: This is a FORTRAN program which is measured while producing some of the graphs in this chapter.

Each program is executed twice, once in a normal fashion on an IBM 370/168 processor, and once under the program execution tracer described in [SHUS78]. In the untraced mode, the entire sample requires 10.37 seconds of CPU time. The tracing generates over 100 million bytes of trace data (in a highly encoded format) and reveals that a total of 26,713,734 instructions are executed.

The trace data is reduced to reference strings, which are made as compact and as efficient to process as possible. Since no sample program has more than 127 pages, each reference requires one byte: 7 bits for the page number and 1 for the clean/dirty indicator. A total of 44,991,850 references are generated from the instruction traces. Table 5.1 gives statistics for each program.

A simple analysis of this data reveals some interesting facts. Table 5.2 gives the instruction-processing rate, memory-referencing rate, and the ratio of references to instructions for each sample program. Programs written in higher-level languages (FORC, PASC, and DRAW) have the highest instruction-processing rate; this is predictable, since compilers tend to generate code that uses simpler instructions. These programs also have a higher memory-referencing rate, which is less understandable.

Table 5.1 – Sample Program Data

Program	Time	Instructions	References	Pages
LKED	1.12	2,129,602	3,510,900	51
SORT	1.86	2,828,556	4,818,090	73
FORC	1.24	4,216,349	7,218,382	124
WATF	1.01	2,683,951	4,551,154	73
ASMC	0.97	2,276,941	3,831,515	63
SCRP	1.39	3,354,062	6,049,228	55
PASC	1.42	5,157,284	7,970,015	30
DRAW	1.36	4,066,989	7,042,566	62

Table 5.2 – Sample Program Statistics

Program	Instructions per sec.($x10^6$)	References per sec.($x10^6$)	References per Instruction
LKED	1.900	3.135	1.65
SORT	1.520	2.590	1.70
FORC	3.400	5.821	1.71
WATF	2.656	4.505	1.70
ASMC	2.347	3.950	1.69
SCRP	2.412	4.352	1.80
PASC	3.632	5.613	1.55
DRAW	2.990	5.178	1.73

Six of the eight programs have virtually identical references-per-instruction ratios of about 1.7. SCRP has a higher number of references per instruction, probably due to manipulation of variable-length text strings. PASC has a much lower number of references per instruction, probably due to the simpler nature of compiler-generated code. It is interesting that the FORTRAN-generated code in both FORC and DRAW has the same reference-per-instruction ratio as the assembler language programs.

5.1.2 Lifetime Curves

We measure the lifetime curves of the eight sample programs. Lifetime curves are not used in the simulation model, but they serve both to ensure that our sample programs are typical and to illustrate our earlier comment about lifetime curves. Figures 5.1(a) through 5.1(h) are the fixed-space LRU, WS, and VMIN lifetime curves of the eight programs. As expected, VMIN achieves the highest lifetime at most memory allocations, while WS is typically, but not always, higher than fixed-space LRU. The horizontal limit line in each figure represents a lifetime of T/m, where T is the number of measured references and m is the number of distinct pages. This value is the maximum possible lifetime since execution of the program requires a minimum of m page faults

Figures 5.2(a) through 5.2(h) are long-term lifetime curves generated by ignoring the page fault caused by the first reference to each page. For reference purposes, the maximum lifetime of the curves in Figure 5.1 is indicated on each figure. As described in Section 3.2.1, the long-run curves are unbounded because no faults occur at the maximum memory allocation. We also observe that the curves have no knees. We conclude that knees are produced by measuring programs for a limited time and including the initial page faults. If a program is measured for an indefinitely long time (assuming that it does not terminate), the long-run behavior is knee-less.

Figure 5.1(a) - Lifetime Curves of LKED

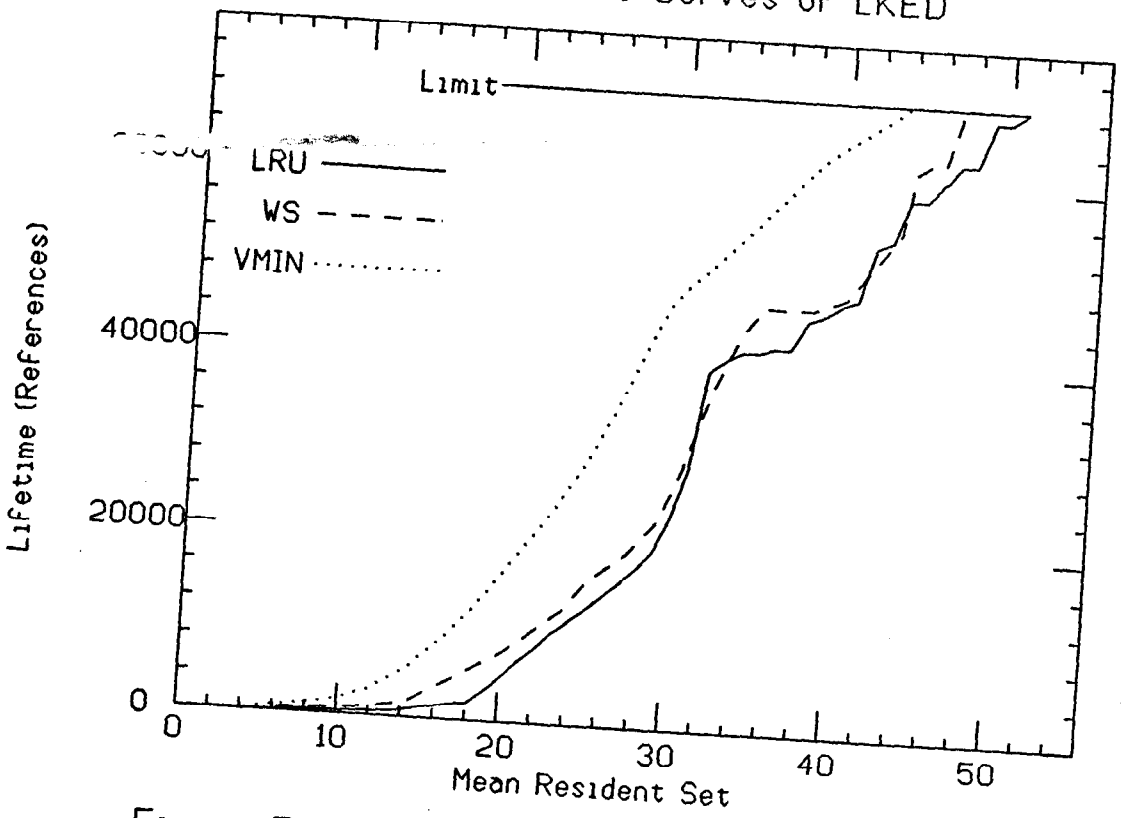


Figure 5.1(b) - Lifetime Curves of SØRT

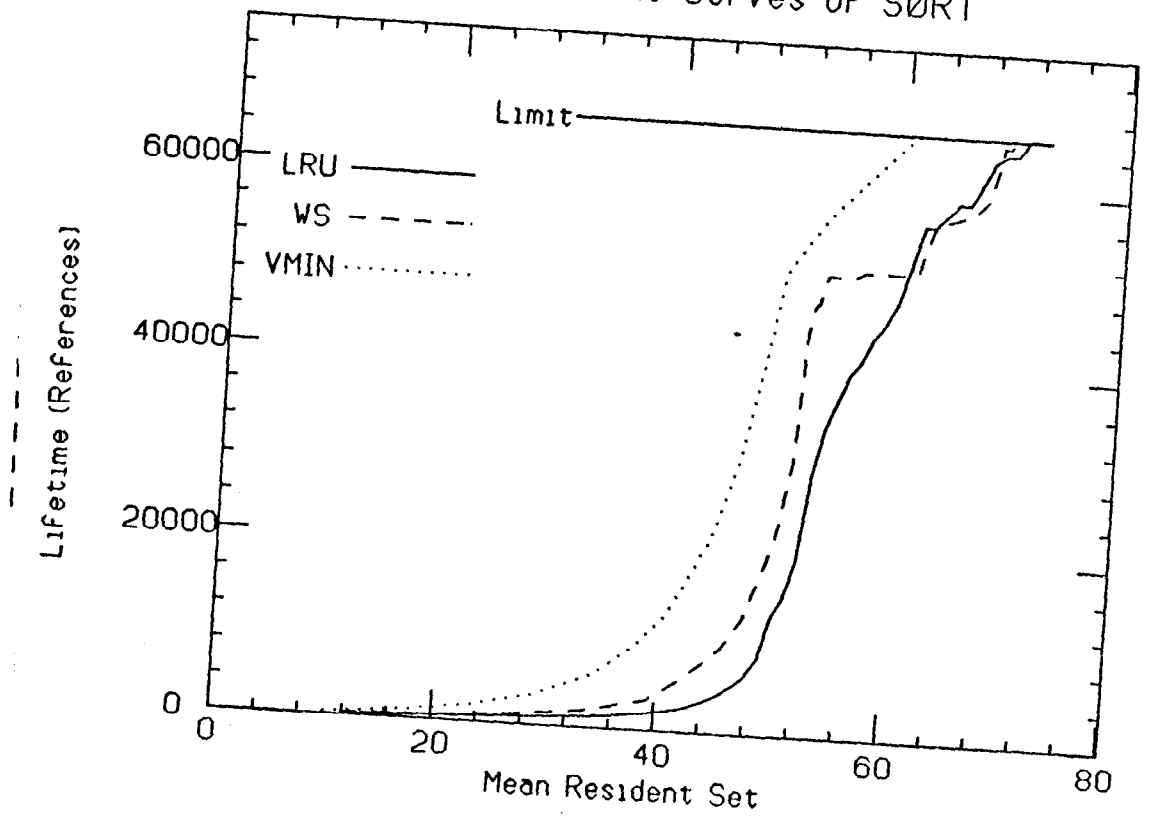


Figure 5.1(c) - Lifetime Curves of FØRC

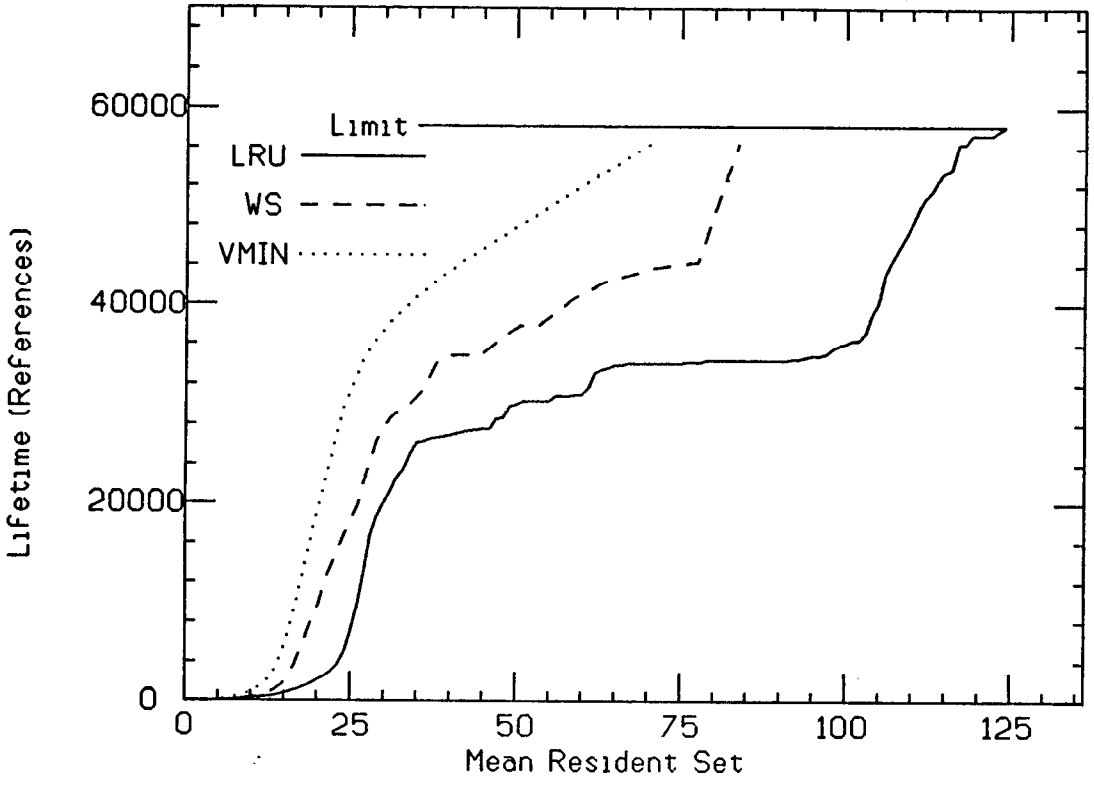


Figure 5.1(d) - Lifetime Curves of WATF

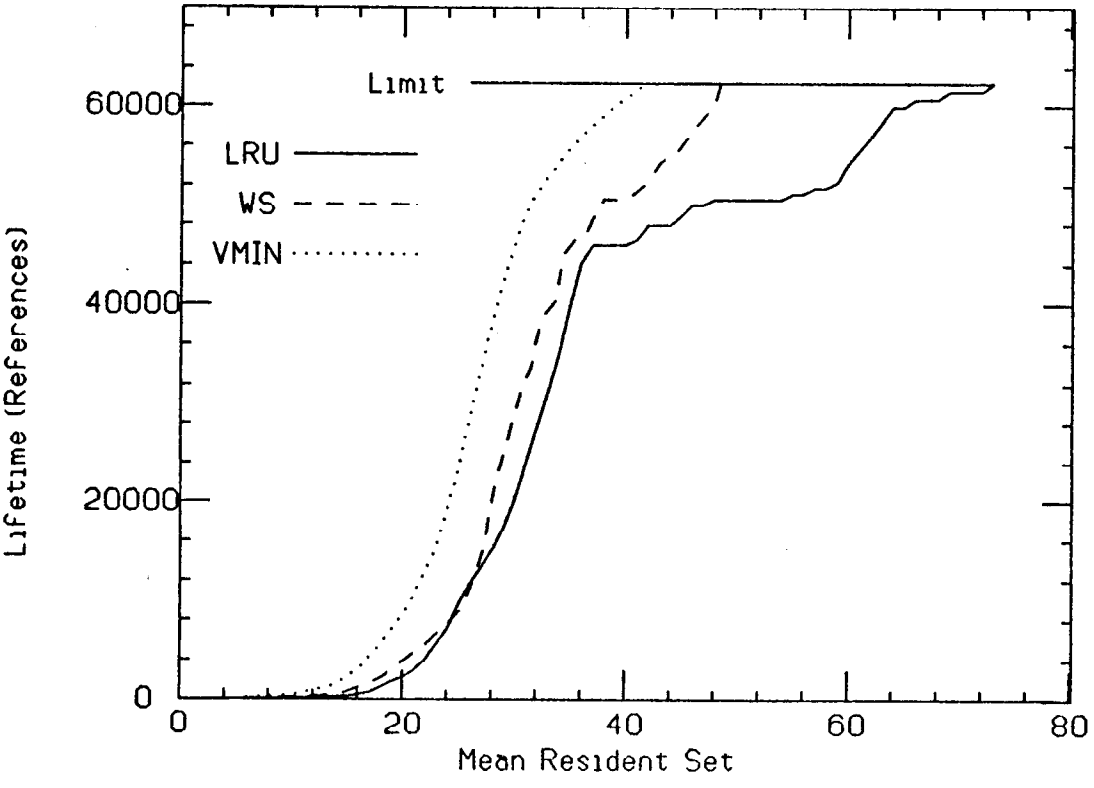


Figure 5.1(e) - Lifetime Curves of ASMC

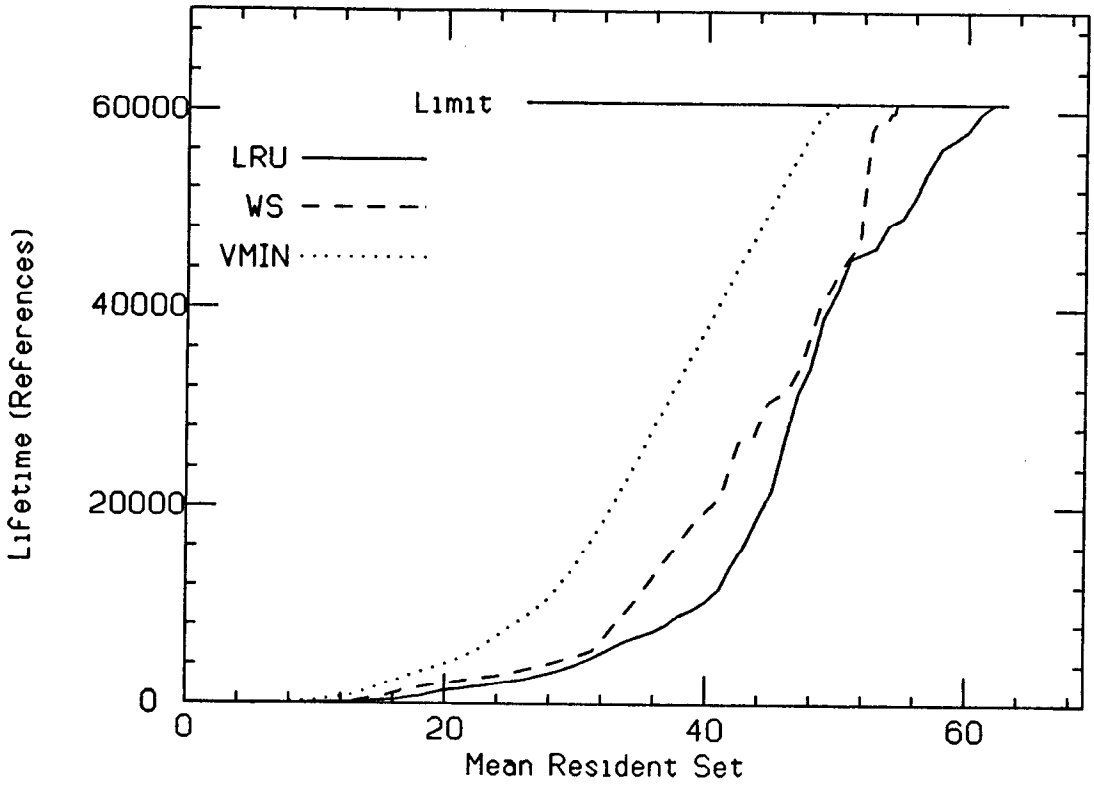


Figure 5.1(f) - Lifetime Curves of SCRP

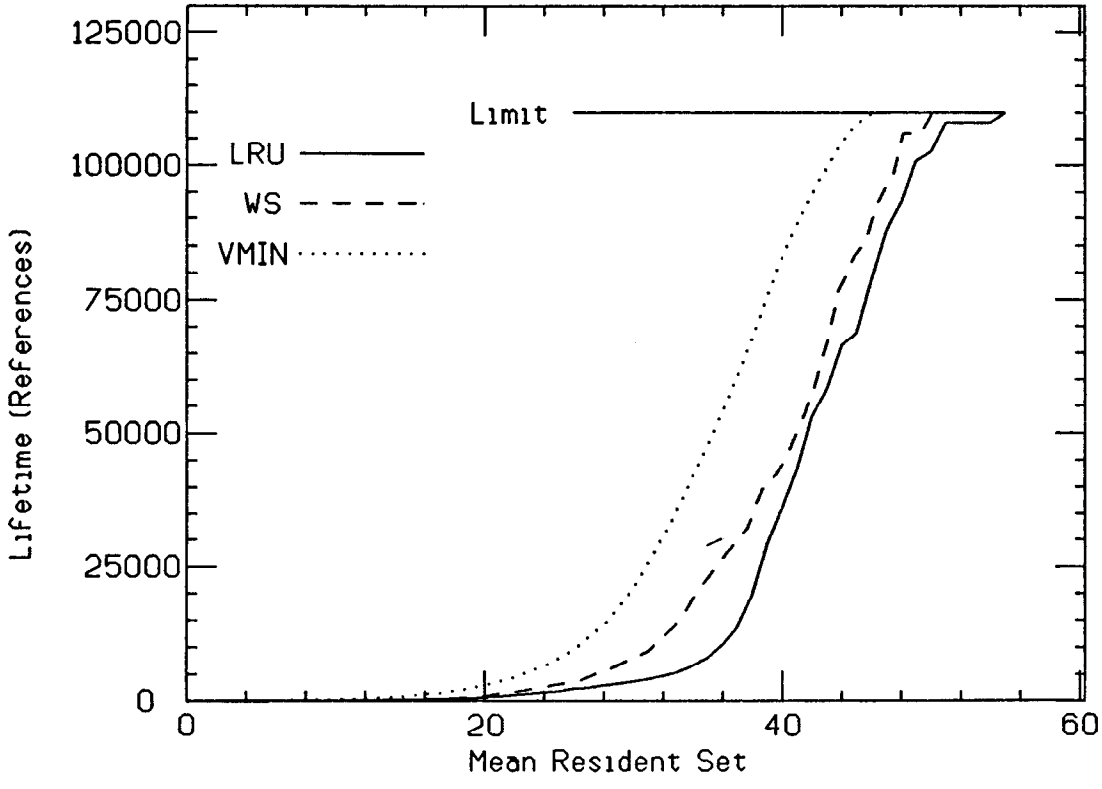


Figure 5.1(g) - Lifetime Curves of PASC

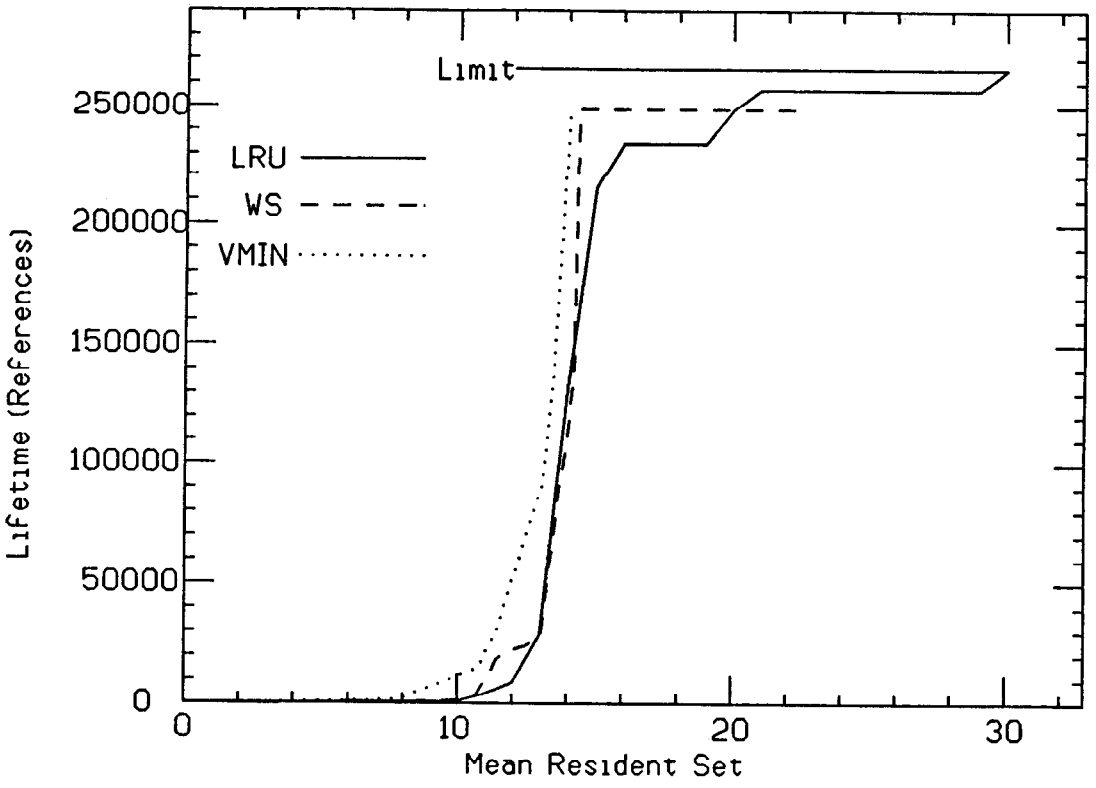


Figure 5.1(h) - Lifetime Curves of DRAW

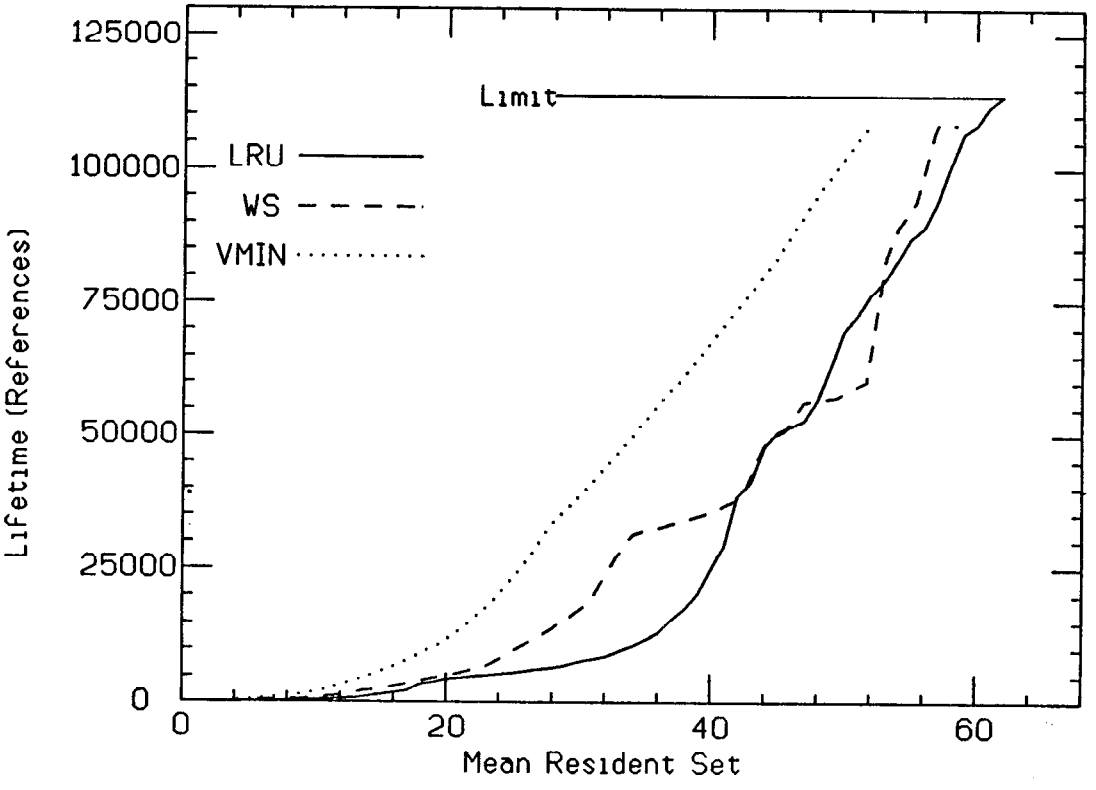


Figure 5.2(a) - Long-Term Lifetime Curves of LKED

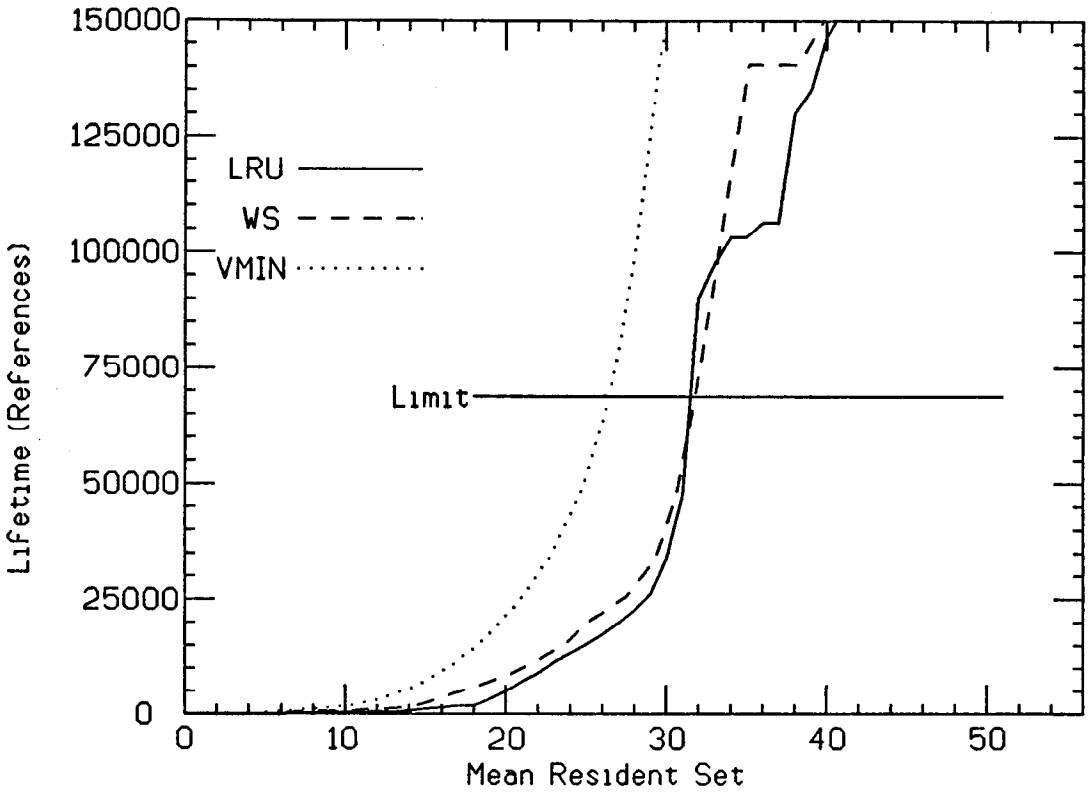


Figure 5.2(b) - Long-Term Lifetime Curves of SØRT

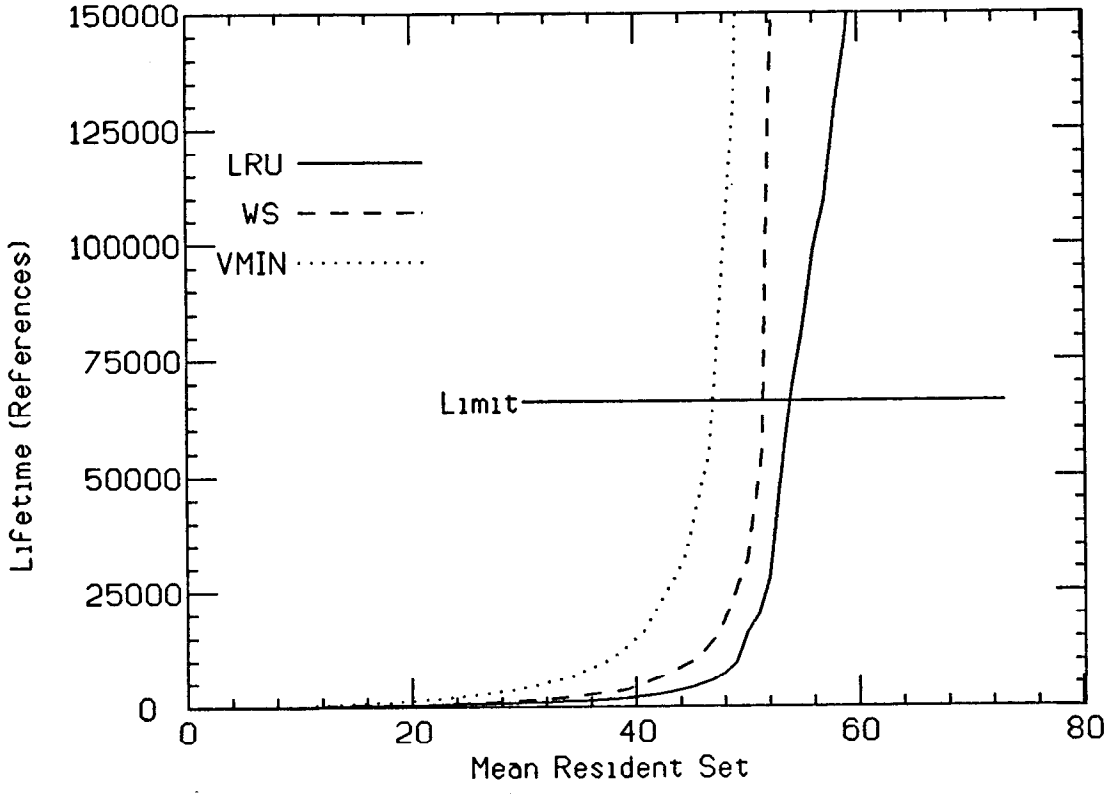


Figure 5.2(c) - Long-Term Lifetime Curves of FØRC

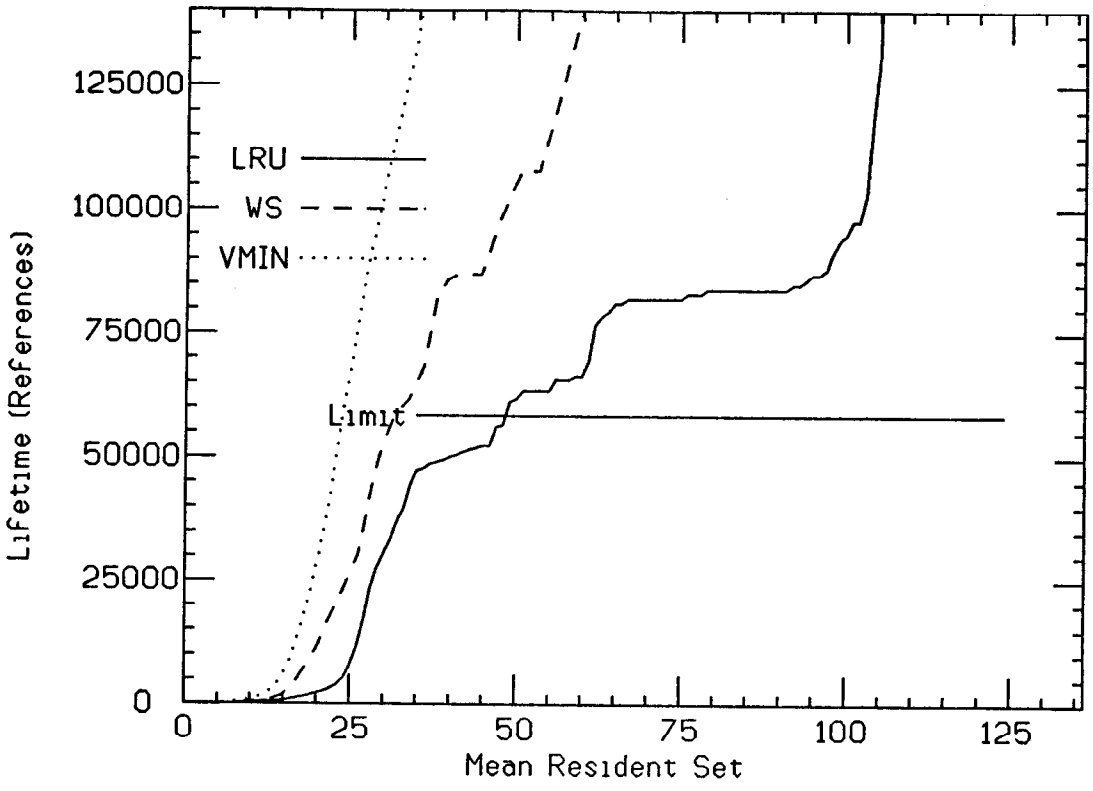


Figure 5.2(d) - Long-Term Lifetime Curves of WATF

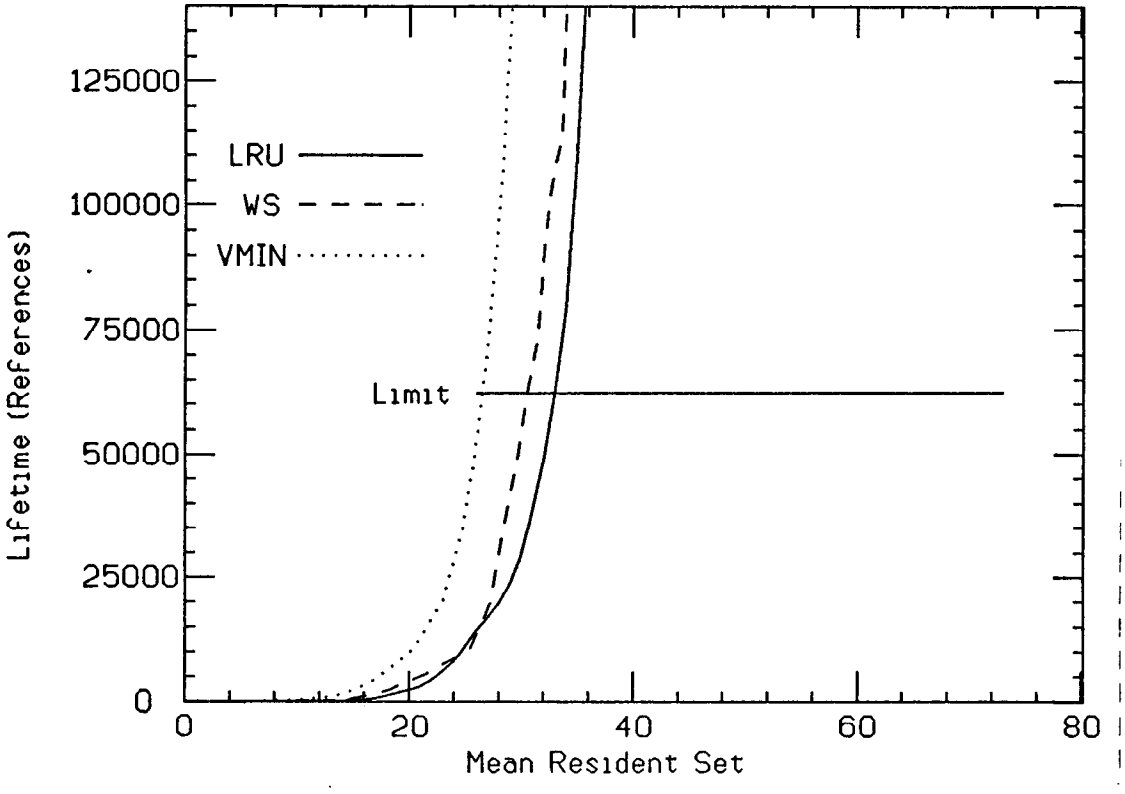


Figure 5.2(e) - Long-Term Lifetime Curves of ASMC

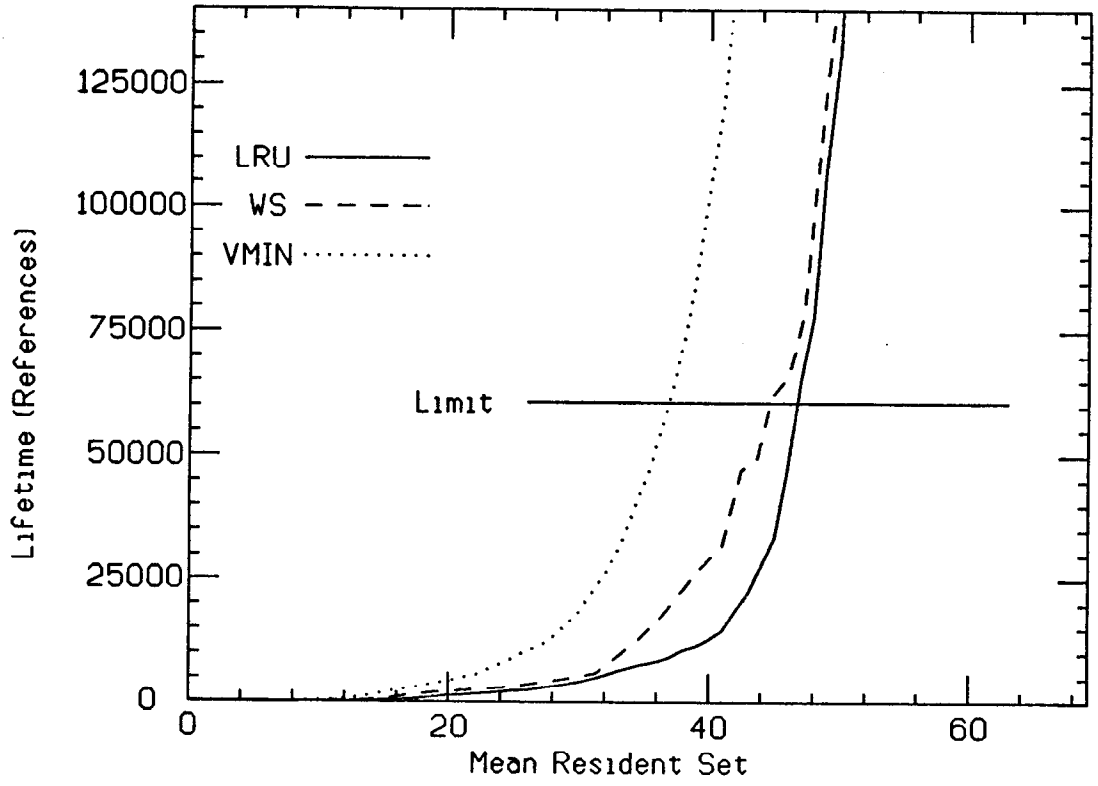


Figure 5.2(f) - Long-Term Lifetime Curves of SCRP

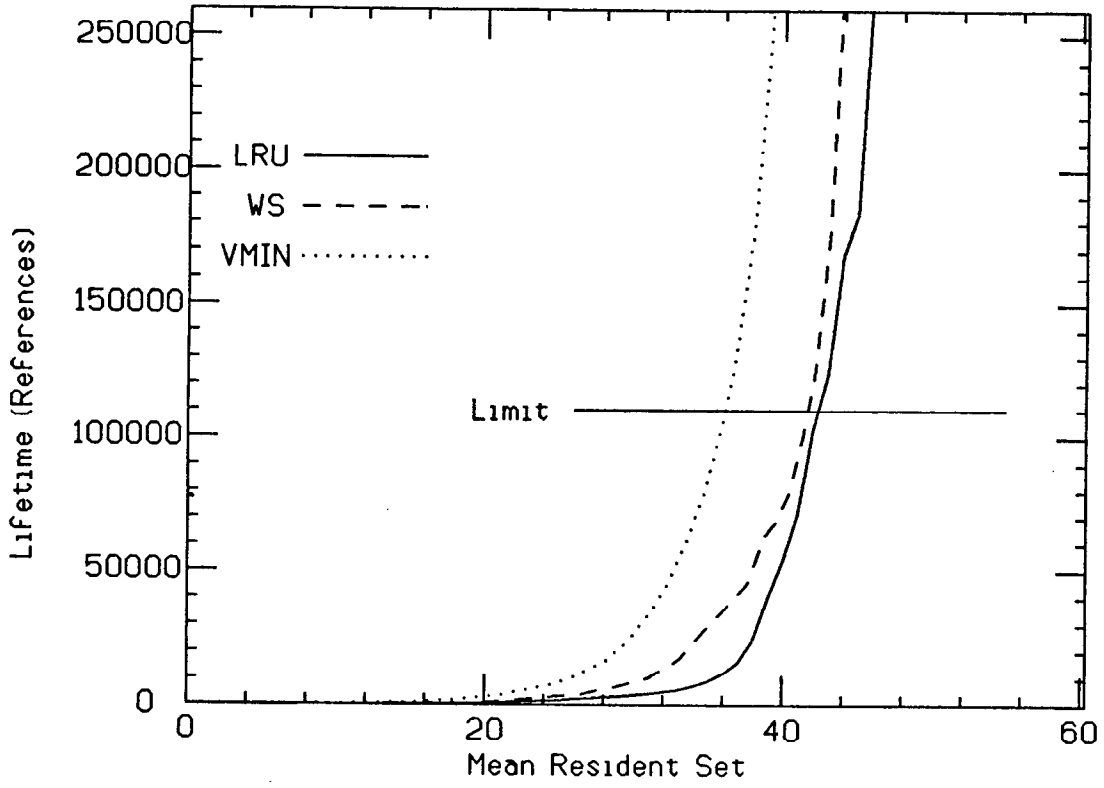


Figure 5.2(g) - Long-Term Lifetime Curves of PASC

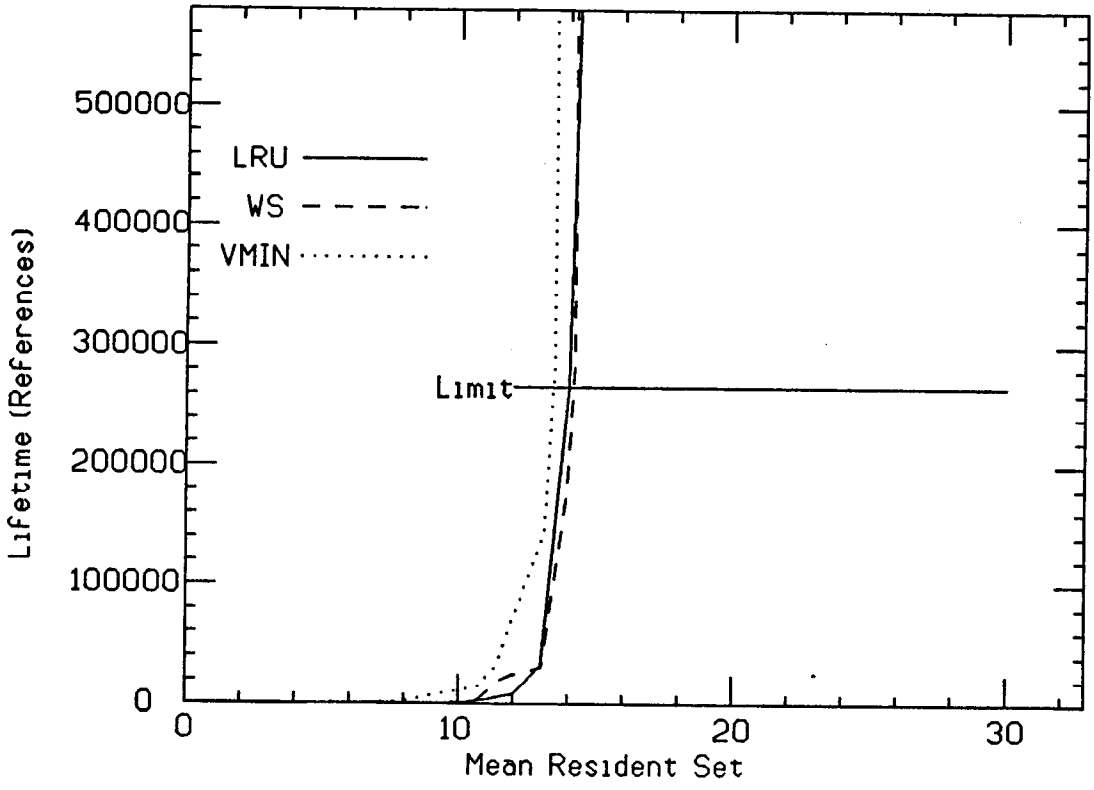
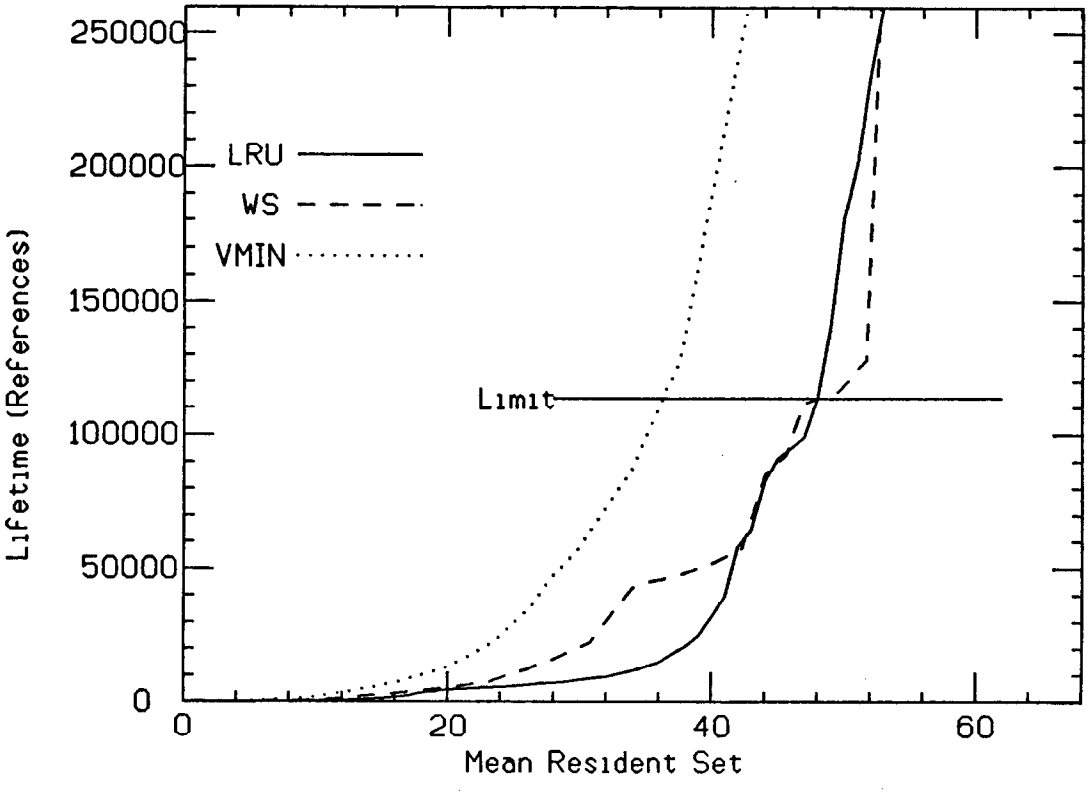


Figure 5.2(h) - Long-Term Lifetime Curves of DRAW



5.1.3 IRIM String Preparation

Before creating the IRIM strings for use in the simulation model, we perform a preliminary analysis of the relationship between ω, the IRIM window size, and the amount of string reduction compared to reference strings. Figure 5.3 illustrates this relationship for each of the eight programs. In Figure 5.4, the curves are combined to illustrate the similarity in the reduction factors for most programs.

Tables 5.3 through 5.6 present detailed measurements of the IRIM for $\omega =$ 500, 1000, 5,000, and 10,000.

Table 5.3 – IRIM Records – $\omega = 500$

Program	Record Type			Total Records	Reduction Factor
	Busy-Clean	Busy-Dirty	Idle		
LKED	5092	3867	8850	17809	197
SORT	9392	7690	15312	32394	149
FORC	20338	15173	34384	69895	103
WATF	14349	11512	25760	51621	88
ASMC	17162	7897	24900	49959	77
SCRP	27859	13939	41504	83302	73
PASC	14057	5516	17227	36800	217
DRAW	11434	11139	22321	44894	157

Table 5.4 – IRIM Records – $\omega = 1000$

Program	Record Type			Total Records	Reduction Factor
	Busy-Clean	Busy-Dirty	Idle		
LKED	4074	2983	6983	14040	250
SORT	5925	6487	11279	23691	203
FORC	11108	9357	19857	40322	179
WATF	11029	7927	18810	37766	121
ASMC	9195	4741	13807	27743	138
SCRP	19623	7699	26939	54261	111
PASC	11208	5332	14240	30780	259
DRAW	7721	8317	15739	31777	222

Table 5.5 – IRIM Records – $\omega = 5000$

Program	Record Type			Total Records	Reduction Factor
	Busy-Clean	Busy-Dirty	Idle		
LKED	1846	1201	3034	6081	577
SORT	3311	4937	7981	16229	297
FORC	1960	2488	4234	8682	831
WATF	1170	1417	2547	5134	886
ASMC	1753	729	2462	4944	775
SCRP	5058	1547	6470	13075	463
PASC	3949	398	4347	8694	917
DRAW	2084	2952	4888	9924	710

Table 5.6 – IRIM Records – $\omega = 10{,}000$

Program	Record Type			Total Records	Reduction Factor
	Busy-Clean	Busy-Dirty	Idle		
LKED	670	613	1273	2556	1374
SORT	1308	3227	4378	8913	541
FORC	744	1249	1880	3873	1864
WATF	571	818	1379	2768	1644
ASMC	1305	478	1769	3552	1079
SCRP	2348	500	2737	5585	1083
PASC	221	206	427	854	9333
DRAW	1018	1827	2779	5624	1252

In the simulation studies, we use IRIM strings generated with $\omega = 5000$ for two reasons. First, this value is smaller than any WS parameter, θ, we will consider. Practical values of θ usually exceed 5000 by an order of magnitude. Second, the value of 5000 is sufficiently smaller than the minimum time a program usually executes between I/O requests or quantum expirations. Thus, the granularity of the IRIM is small compared to time-scale of events that cause task deactivations that, in turn, lead to errors in the IRIM.

At $\omega = 5000$, the IRIM reduces the trace length for all eight programs from 44,991,850 references to 72,763 IRIM records, a factor of 618. Of course, each IRIM record is 8 bytes, so the net volume is reduced by a factor of 77. An analysis of the reduction in simulation cost appears in Section 5.2.

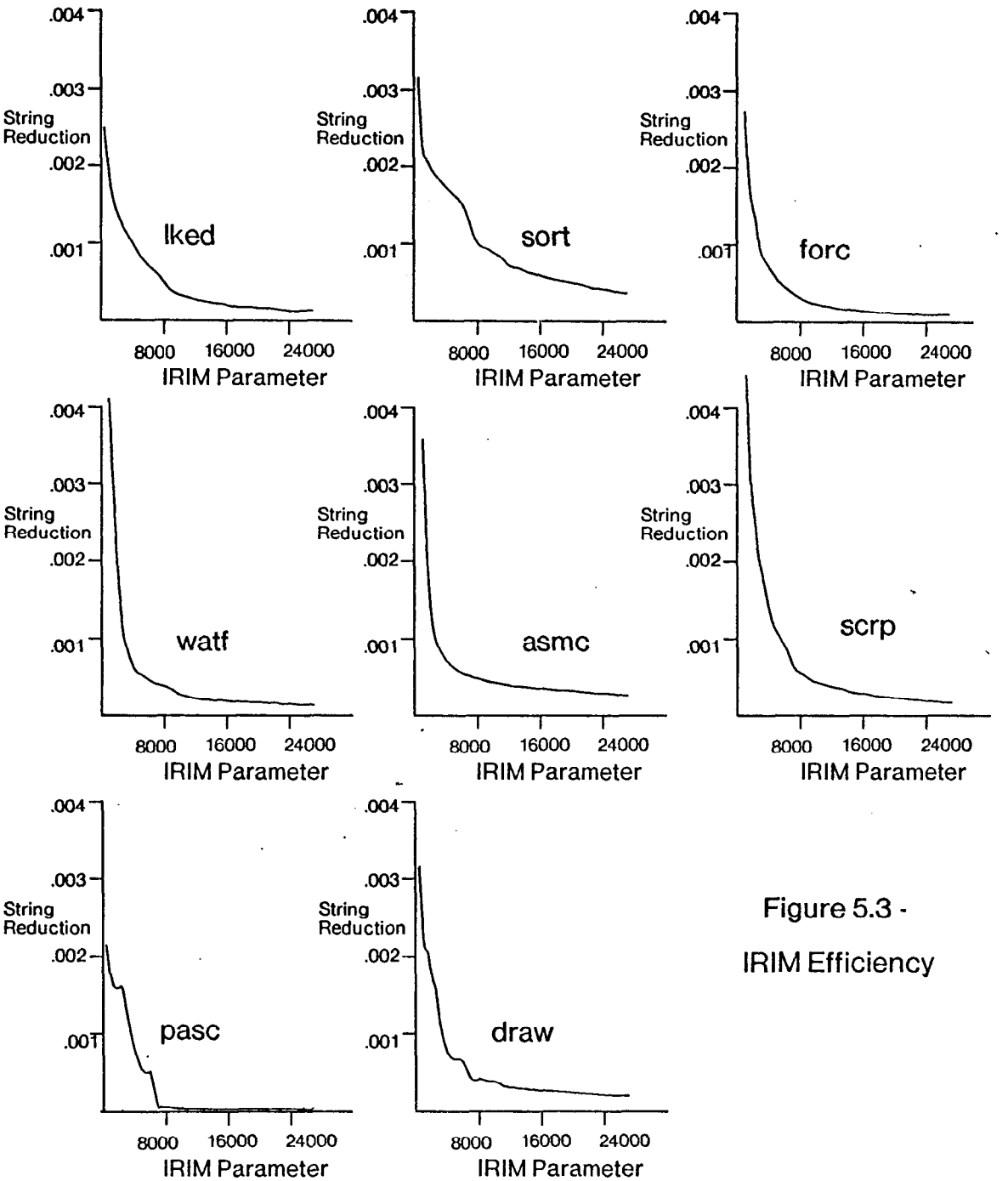


Figure 5.3 -
IRIM Efficiency

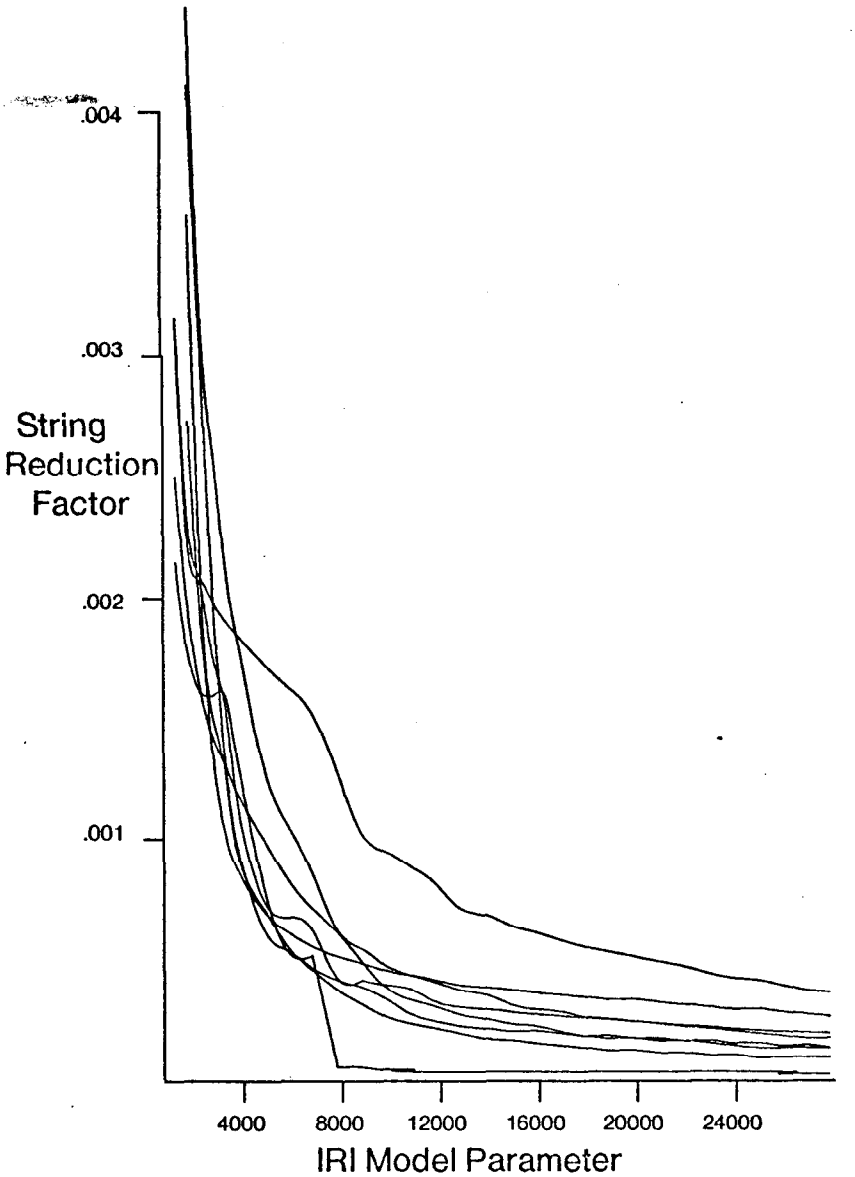


Figure 5.4 - Composite IRIM Efficiency

5.1.4 Task I/O Request Model

The occurrence of task I/O requests is modeled stochastically. When each of the sample programs is traced, we obtain counts of disk I/O operations. By dividing the program execution time (i.e., total references) by the I/O count, we determine the average execution time between I/O requests as shown in Table 5.7. We assume that the time between I/O requests is exponentially distributed (i.e., the requests form a Poisson process) and have a minimum value of

5000 references. This minimum value is a realistic assumption for modeling real systems, due to the extensive processing usually required to request, verify, initiate and terminate an I/O operation.

Table 5.7 – I/O Request Model

Program	Trace Measurements		Processing Interval Model		
	References	I/Os	Minimum	Mean	Distribution
LKED	3,510,900	239	5000	14686	Exponential
SORT	4,818,090	227	5000	21233	"
FORC	7,218,382	53	5000	136226	"
WATF	4,551,154	74	5000	61486	"
ASMC	3,831,515	89	5000	43033	"
SCRP	6,049,228	56	5000	108035	"
PASC	7,970,015	325	5000	24493	"
DRAW	7,042,566	31	5000	227096	"

5.1.5 Workload Model

The workload of the simulation model is specified as a set of task prototypes, one of which is chosen to describe each newly admitted task. In the studies that follow, there are eight prototype tasks, corresponding to the eight sample programs. A pseudo-random number generator is used to select each of the eight task prototypes with equal probability. This generator is independent of all other random generators; its seed can be specified as part of the simulation input. Thus, the workload is easily controlled and duplicated.

Preliminary tests have demonstrated that both the distribution of prototype tasks in the workload and the order in which they are chosen have a great effect on performance in the simulated system. The variation due to workload differences can overshadow the variation due to any of the algorithm and policy alternatives we seek to evaluate. If the purpose of the model were to evaluate a computer hardware configuration for some real installation, we would have to be very sensitive to this problem. Long simulation run times and multiple replications would be required

to remove effects due to variations in the load. It would also be necessary to determine whether our sample programs were typical of the workload at the installation to be evaluated.

Since the object of this work is to evaluate algorithms and policies, we are concerned with the relative, not absolute, level of performance with any realistic, but precisely repeatable, workload. In the studies that follow, the order in which tasks are admitted is the same in each run. The simulation is run until the first 50 tasks are completed. Since the typical multiprogramming level in the model system is about 5, many different groupings of the eight prototypes are processed concurrently during each simulation run.

5.2 System Model Preparation

5.2.1 Model Configuration

We model a computer system which has a processor speed roughly the same as the IBM 370/168. The basic time unit of the simulation model is a memory reference and our program sample conveniently reveals that the 370/168 processes about 4,000,000 references per second. The selected configuration contains two types of I/O devices: disks, which are used for task I/O requests, and drums, which are used for paging I/O. The service times for each device are assumed to be uniformly distributed between the minimum and maximum values shown in Table 5.8.

Table 5.8 - I/O Devices Model

	Service Time (370/168)		Model Parameters		
Device	Min.	Max.	Min.	Max.	Distribution
DISK	37.5 msec	57.5 msec	150000	230000	Uniform
DRUM	7.5 msec	12.5 msec	30000	50000	Uniform

Although the simulator is capable of modeling contention for each disk, including the maintenance and measurement of request queues, the experiments in this study assume that the

system is disk-rich and that no queueing of task I/O requests occurs. Contention for drums, on the other hand, often has a significant effect on virtual memory management. The model assumes a finite number of drums (typically one) which process one paging I/O request at a time.

The remaining configuration parameter is the size of main memory. We performed a number of simulation runs to obtain a rough estimate of the effect of main memory size on performance. The runs used the WSEXACT replacement algorithm for a range of values of θ, and for memory sizes of 200, 250, 300, and 350 pages. As shown in Figure 5.5, there is a roughly linear relationship between system performance and main memory in this range. We desire a configuration which is not abnormally constrained by memory size, but one in which a more efficient use of main memory would result in a significant improvement. We selected 250 page frames for our standard memory configuration.

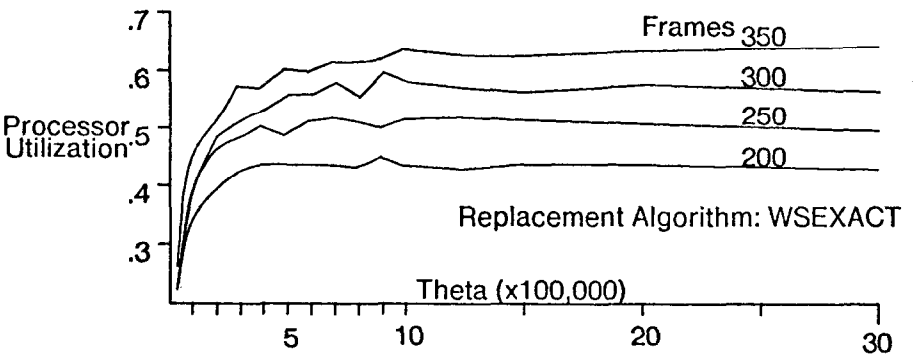


Figure 5.5 - Main Memory Size vs. Utilization

5.2.2 Validating the IRIM

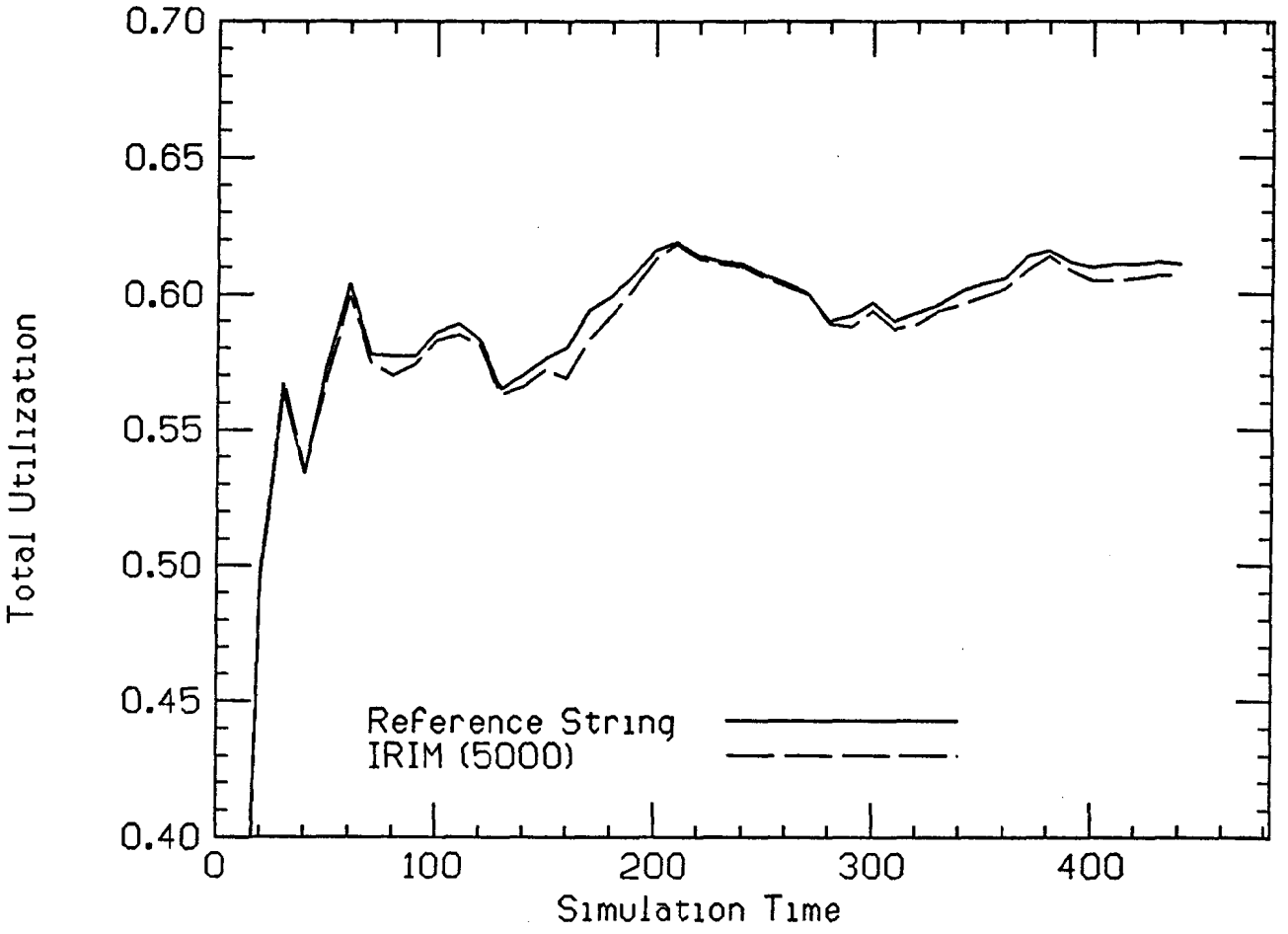
The task model can be driven either by ordinary reference strings or by IRIM strings, although the former are too costly for practical use. As described in Chapter 4, the model is precise except when a IRIM busy page is removed from the resident set when the task is inactive. Errors can occur when a program is deactivated and later reactivated, or when a global replacement algorithm is forced, by overcommitment, to remove a busy page.

In Chapter 4, we argue that the effect of such errors will be small; in this chapter, we present evidence to support that argument. We simulate our standard configuration model using both WS and CLOCK replacement, repeating each run twice—once using ordinary reference strings and once using IRIM strings. As indicated in the next section, reference string simulation is very inefficient and the number of validation runs is limited by the cost.

Figures 5.6 and 5.7 display the mean processor utilization as a function of total simulation time. There is close agreement between the two models in each case. When a disagreement does occur, IRIM performance is below than the reference string model. This is expected since, at each activation, the IRIM simulates immediate page faults on any busy pages not in the resident set; when execution is modeled by the reference string, these faults may not occur for up to ω references after reactivation. Note, also, that the IRIM recovers from its underestimate of performance since the reference string model encounters most of the same faults at a later time. We do not claim that the IRIM will always recover from underestimation, but the empirical results are encouraging.

Figures 5.8 and 5.9 compare system utilization for the two models in each 10-second period of simulation time. These graphs show that both models have similar short-term behavior patterns.

Figure 5.6 - IRIM Validation - Working Set



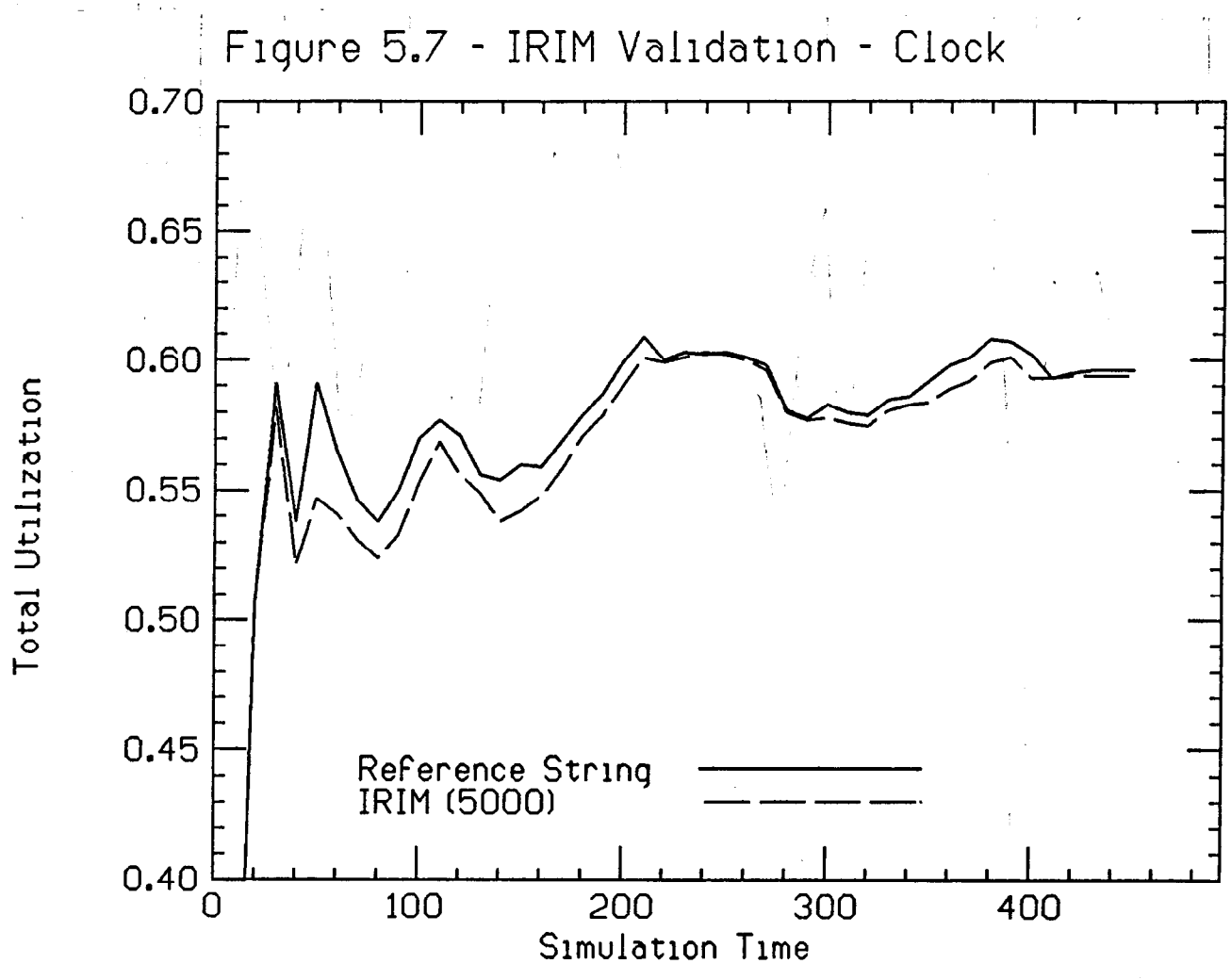
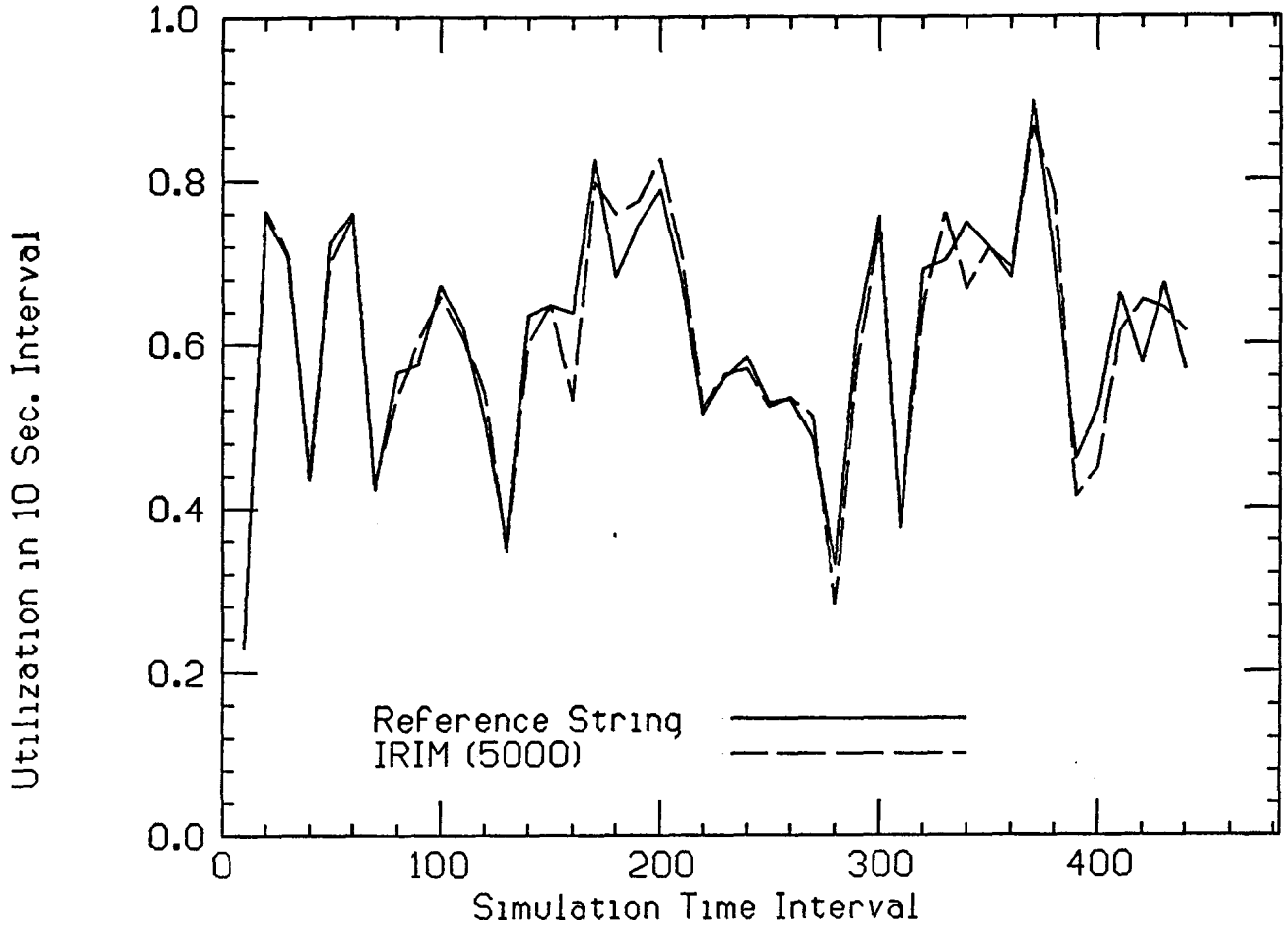


Figure 5.7 - IRIM Validation - Clock
Total Utilization
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0
100
200
300
400
Simulation Time
Reference String
IRIM (5000)

Figure 5.8 - IRIM Short-Term Validation - Working Set



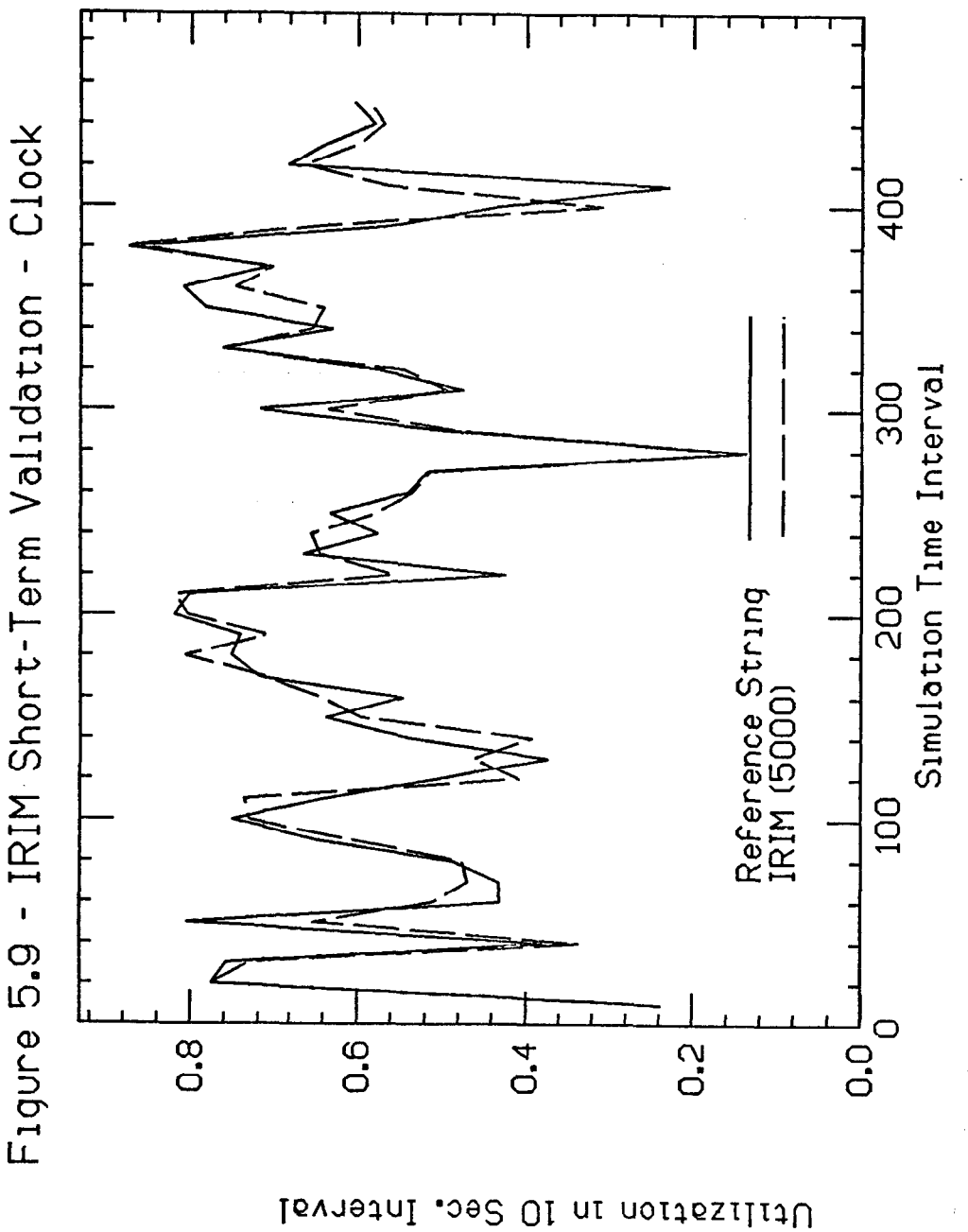


Figure 5.9 - IRIM Short-Term Validation - Clock
Utilization in 10 Sec. Interval
0.0
0.2
0.4
0.6
0.8
0
100
200
300
400
Simulation Time Interval
Reference String
IRIM (5000)

5.2.3 Simulation Efficiency

Measurements of simulation efficiency are based on the IRIM validation runs of the previous section. We consider raw simulation efficiency, and the relative efficiency of the reference string and IRIM string models. It is assumed that the IBM 370/168 (used to make the simulation runs) is nominally capable of processing 4,000,000 references per second of processor time.

Table 5.9 shows that, with the IRIM, simulation time proceeds about 10 times as fast as real time, while the reference string model reverses that ratio.

Table 5.9 - Simulation Efficiency

Replacement Algorithm/Model	Simulated Time (10^6 references)	Run Time (CPU Seconds)	Simulation Cost (10^6 references)	Sim. Time/ Sim. Cost Ratio
WSEXACT/IRI(5000)	444	10.68	42.72	10.39/1
CLOCK/IRI(5000)	459	12.16	48.64	9.44/1
WSEXACT/Ref. String	271	1700.	6800.	1/25.1
CLOCK/Ref. String	459	774.	2996.	1/6.5

Reference String Efficiency

We claim that the simulator, as constructed, represents a "best-effort" attempt to use reference strings as efficiently as possible. When the simulator was first written, it was presumed that reference strings would be the primary method of driving the task model, despite the anticipated high cost. Considerable effort was expended to minimize the cost of using reference strings before it was realized that some other method would be necessary.

The simulator is executed under a monitoring tool which samples an executing program and estimates the amount of processor time used in each section of the program. When reference strings are used to drive the simulator, over 99% of processor time is used in fetching the strings and simulating memory references. Table 5.10 differs from the previous table because it is based on the number of successful references simulated instead of total simulated time.

Table 5.10 - Reference String Efficiency

Replacement Algorithm	References Simulated(10^6)	Run Time (CPU Sec)	Cost (10^6 refs)	Real Reference Times/ Simulated Reference
WSEXACT	163	1700.	6800.	41.7
CLOCK	274	774.	2996.	10.9

Under CLOCK, about 11 reference times are required to simulate each reference. This cost is very low; the simulator must access the reference string record, separate the dirty indicator from the page number, access the proper page model data structure, verify that the page is resident (signaling a fault if it is not), access the proper frame model data structure, set the frame's use-bit, test the dirty-reference indicator and possibly set the frame's dirty bit, update the task's virtual time, check for a simulated interrupt event, and, finally, move to the next reference record while checking for an end-of-buffer condition. Although some techniques were used to perform several of the above operations simultaneously, it is estimated that about 20 to 25 memory references are required.

To explain this discrepancy, we postulate that an unusually large proportion of memory references were to the 370/168 cache memory, making the effective memory reference rate much greater than 4,000,000 per second. The reference simulation loop is small and executed many times in succession so all instructions should occupy the cache almost all the time. The linear access of the reference string, one byte for each reference, implies that a non-cache memory access is required a small fraction of the time. Most references can be simulated without updating any memory location (e.g., it is not necessary to set a use-bit which is already set). Since the IBM architecture uses store-through cache policy, store references have a higher average cost than fetch references; thus, average programs that have a reasonable proportion of stores will achieve a lower references-per-second rate than a program that stores infrequently.

Although we are not prepared to suggest a practical alternative, this discrepancy is a good example of one of the weaknesses of the reference string model. Many circumstances can cause references to require widely varying amounts of time to be processed.

Simulation of WSEXACT, using reference strings, is considerably more expensive than the other algorithms, including the other WS algorithms. The WSEXACT algorithm must update a table containing the page *LastReferenceTime*, requiring at least one store access per simulated reference and a slower memory-referencing rate.

IRIM String Efficiency

When the simulator was monitored when processing IRIM strings, approximately 40 to 50% of processor time is consumed simulating memory references. In Table 5.11 we assume the 50% figure in estimating the cost of reference simulation.

Table 5.11 - IRIM String Efficiency

Replacement Algorithm	References Simulated(10^6)	Run Time (CPU Sec)	Cost (10^6 refs)	Simulated References/ Real Reference Time
WSEXACT	273	10.68	21.36	12.7
CLOCK	274	12.16	24.32	11.2

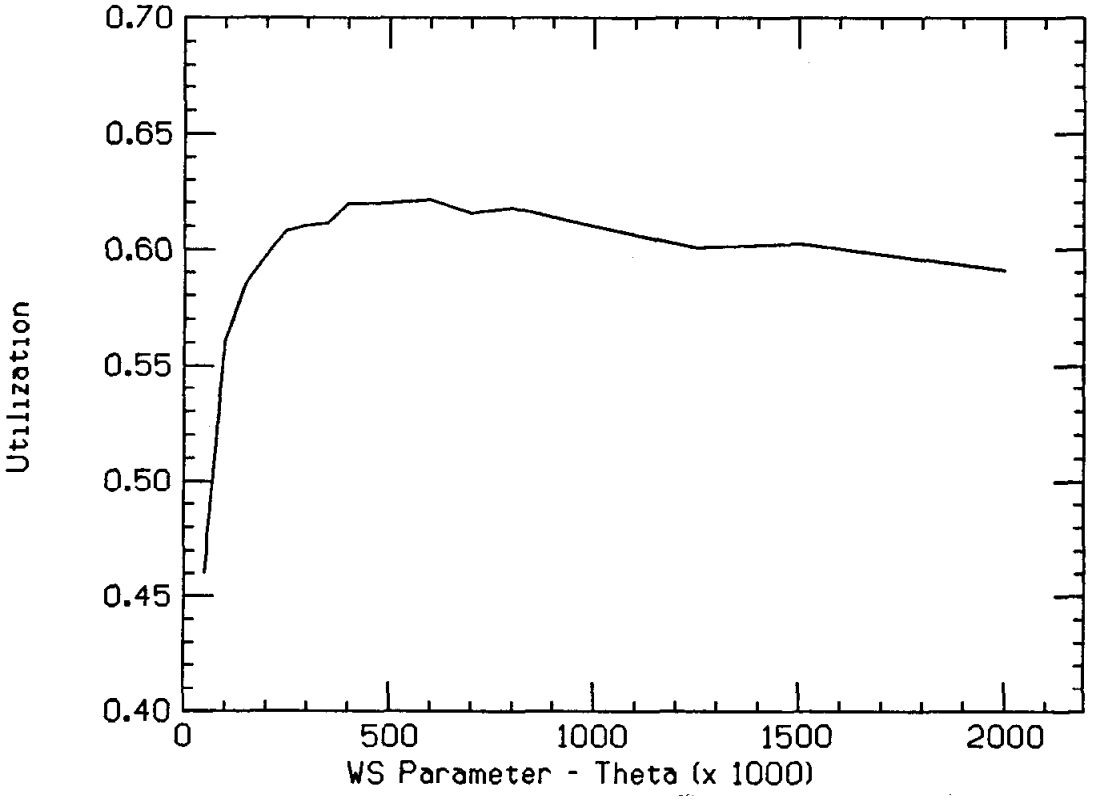
The efficiency of IRIM itself (i.e., separated from the full simulation model) is about 10 simulated references per real reference time, with only a small penalty for simulating the WSEXACT replacement. Compared to using reference strings, IRIM strings are about 100 times as efficient for practical replacement algorithms and about 500 times as efficient for WSEXACT.

5.3 Tuning the WSEXACT Replacement Algorithm

The following sections describe a number of experiments to measure the effect of various parameters and policies on a system with the WSEXACT replacement algorithm. Our dual purpose is to understand the nature and extent of each effect and to tune the WSEXACT for best performance before it is compared to global replacement. Each experiment consists of simulating the system with the standard configuration and workload described in the previous section. For each set of parameters to be tested, the system is simulated for a range of values of the working

set parameter, θ. The resulting performance measures are plotted, producing a curve such as the one in Figure 5.10.

Figure 5.10 - Sample WSEXACT Performance Curve



In preliminary tests, we have found that the value of θ that optimizes performance will vary considerably from one set of parameters to another. Furthermore, since real workloads vary considerably, it is not practical to determine the optimum value of θ for a sample workload and expect it to be optimum for the entire workload. Thus, it is necessary to consider the sensitivity of performance to θ.

An example of the problem we try to avoid appears in Figure 5.11. In Example 1, the two curves are practically the same, except for one point at peak performance. A simple comparison of peak performance would lead us to an unjustified conclusion that one model is better than the other. Another effect that we wish to examine is the stability or robustness of a particular model. In Example 2, both models have the same peak performance but model A is preferable to model B.

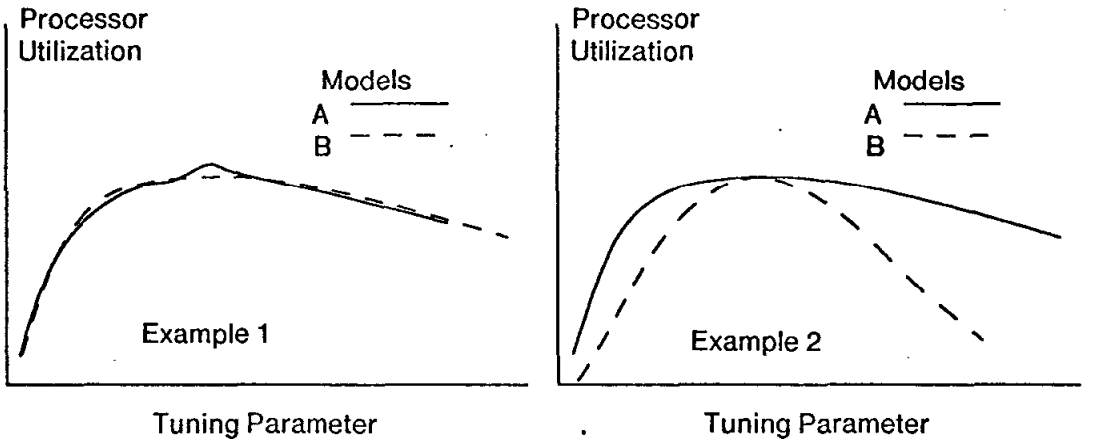


Figure 5.11 - Comparing Models

5.3.1 Task Demotion Policy

We consider the six task demotion policies described in Section 2.3.6. As shown in Figure 5.12, there is little significant difference between choosing (1) the most recently activated task, (2) the task with the smallest resident set, or (3) the task with the largest remaining quantum. This is not surprising, since each of these policies will often result in demoting the same task; the most recently activated task is also the one that has had the least opportunity to load a large resident set or consume its quantum.

Somewhat less effective is the policy of demoting the faulting task. Apparently, the elimination of a single replacement and paging-I/O operation for each demotion does not result in a significant saving. This policy is still better than random selection, probably because the last activated task (or smallest task) also has a greater chance of faulting. Demoting the task with the largest resident set is a worse policy than random choice.

In Figure 5.13, we examine the relationship between the demotion policy and the frequency of overcommitment and demotion. Surprisingly, the three superior policies have significantly *more* demotions (60 to 80% more in the peak performance range) than the poorer policies. Apparently, the tasks demoted by these policies are considerably smaller than the average task, indicating that overcommitment must occur shortly after a task is newly activated and before it has had a chance to gather a large resident set.

Figure 5.12 - Demotion Task Choice vs. Utilization - WSEXACT

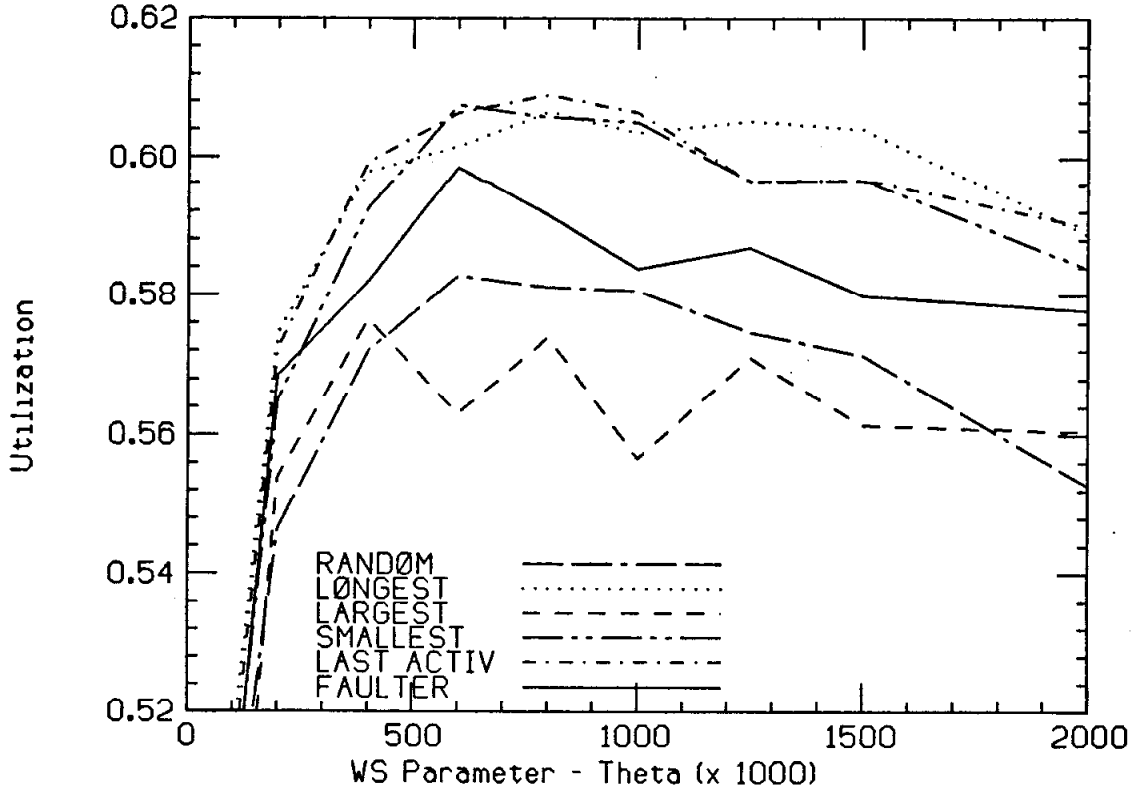
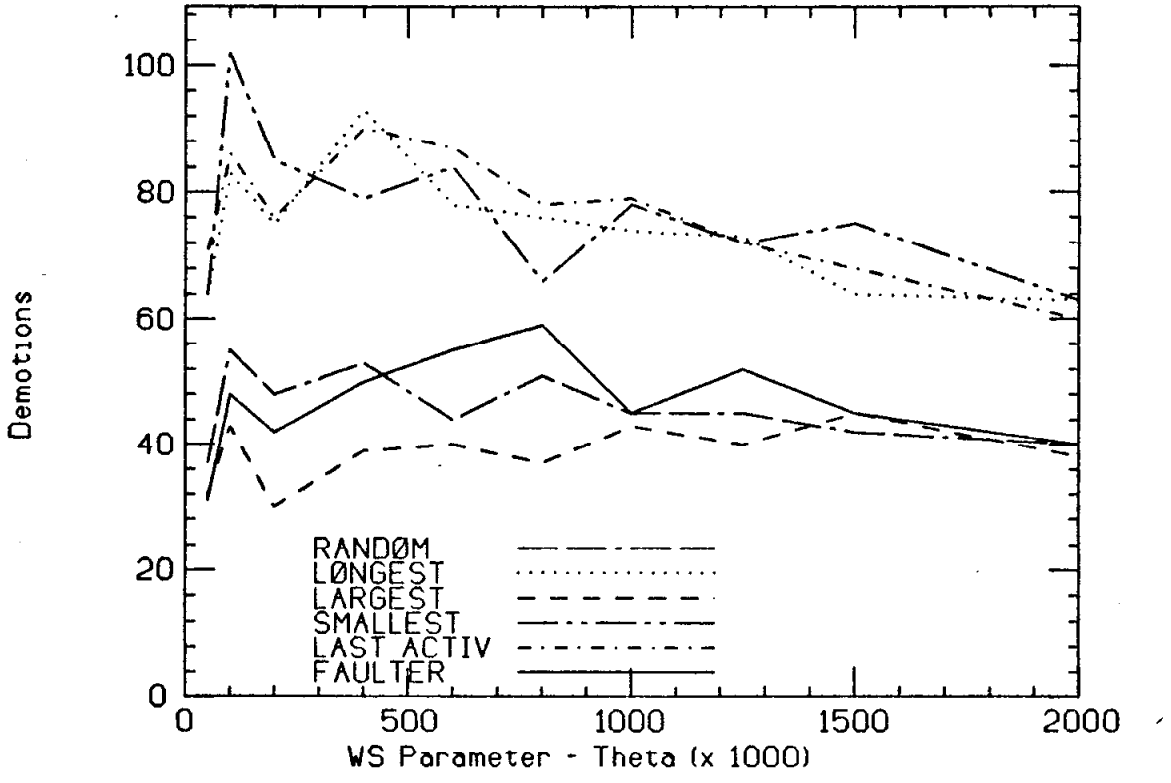


Figure 5.13 - Demotion Task Choice vs. Demotions - WSEXACT



5.3.2 *LT/RT* Control

The *LT/RT* mechanism to limit the number of loading tasks is described in Section 2.3.2. The control has two tuning parameters: (1) a newly activated task is considered to be a loading task until it has executed τ references or has made an I/O request; (2) the system never activates a task if there are L loading tasks in the active queue.

Figure 5.14 shows system performance with various values of L with the standard model configuration of one paging device. Performance degrades with increasing L, supporting the hypothesis that the number of loading tasks should not exceed the number of page faults that can be serviced simultaneously. In Figure 5.15 a model configuration with two paging devices is measured; the resulting performance curves for $L=1$ and $L=2$ are comparable, and both are better than larger values of L.

Figure 5.14 - Loading Task Limit - WSEXACT

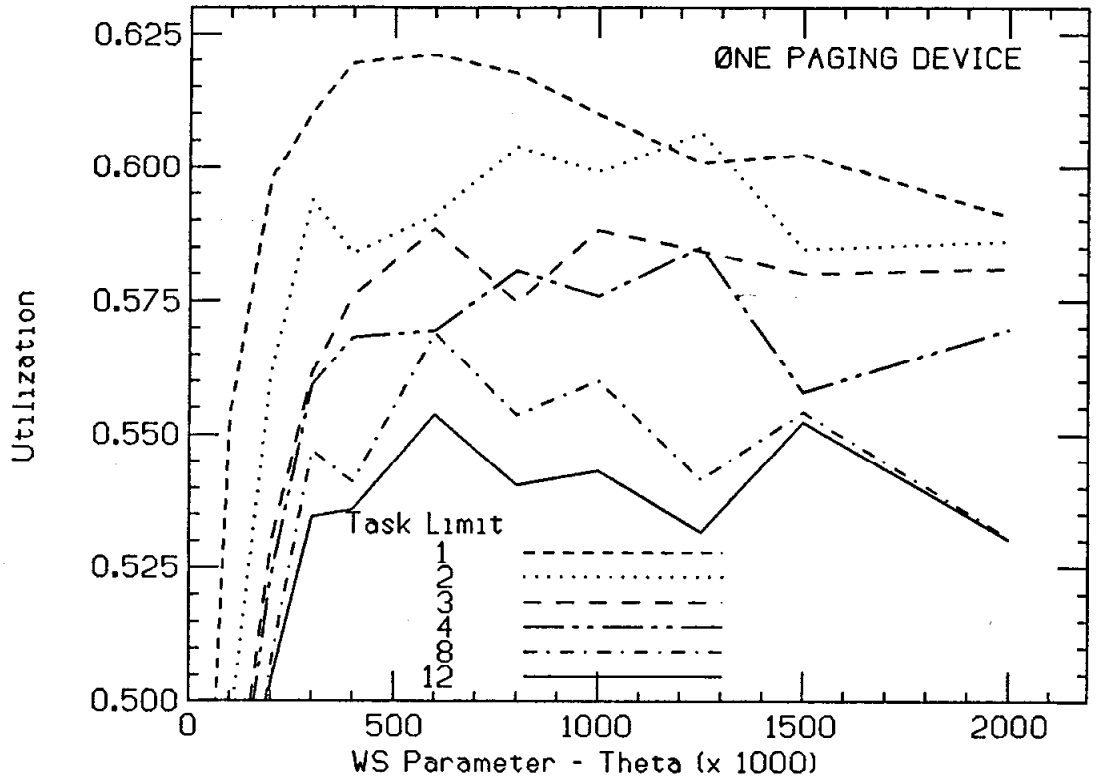


Figure 5.15 - Loading Task Limit - WSEXACT

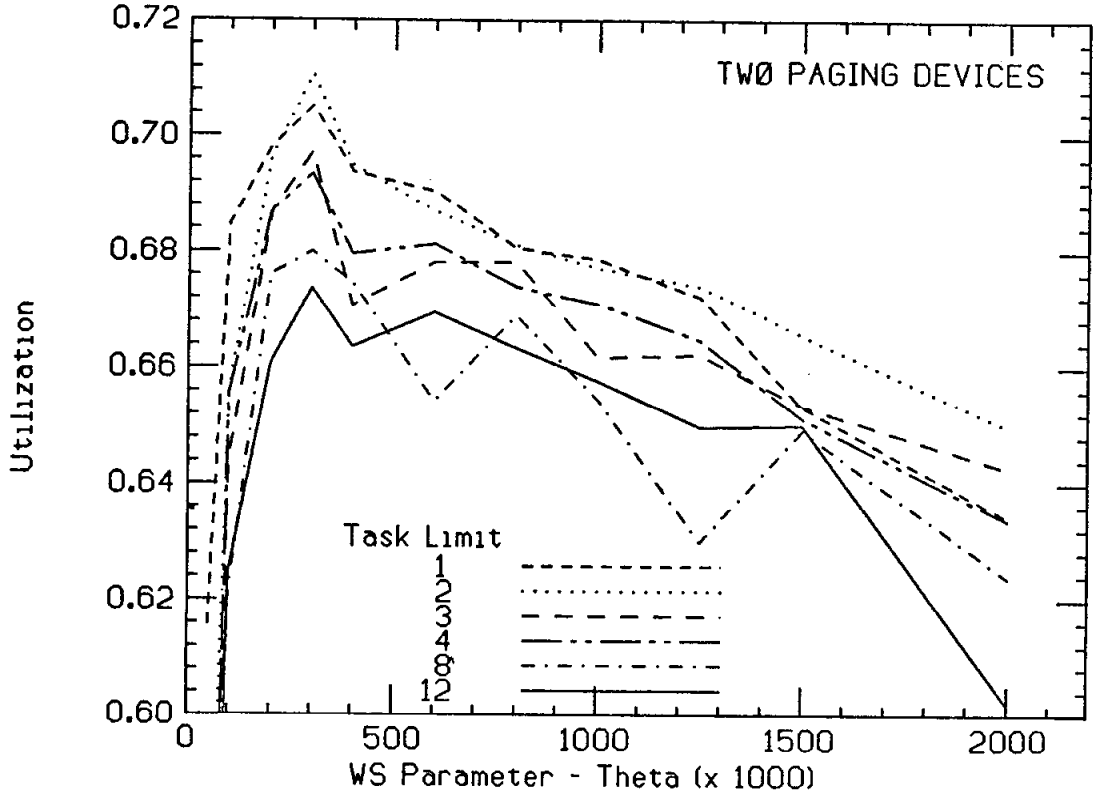


Figure 5.16 - Loading Task Lifetime- WSEXACT

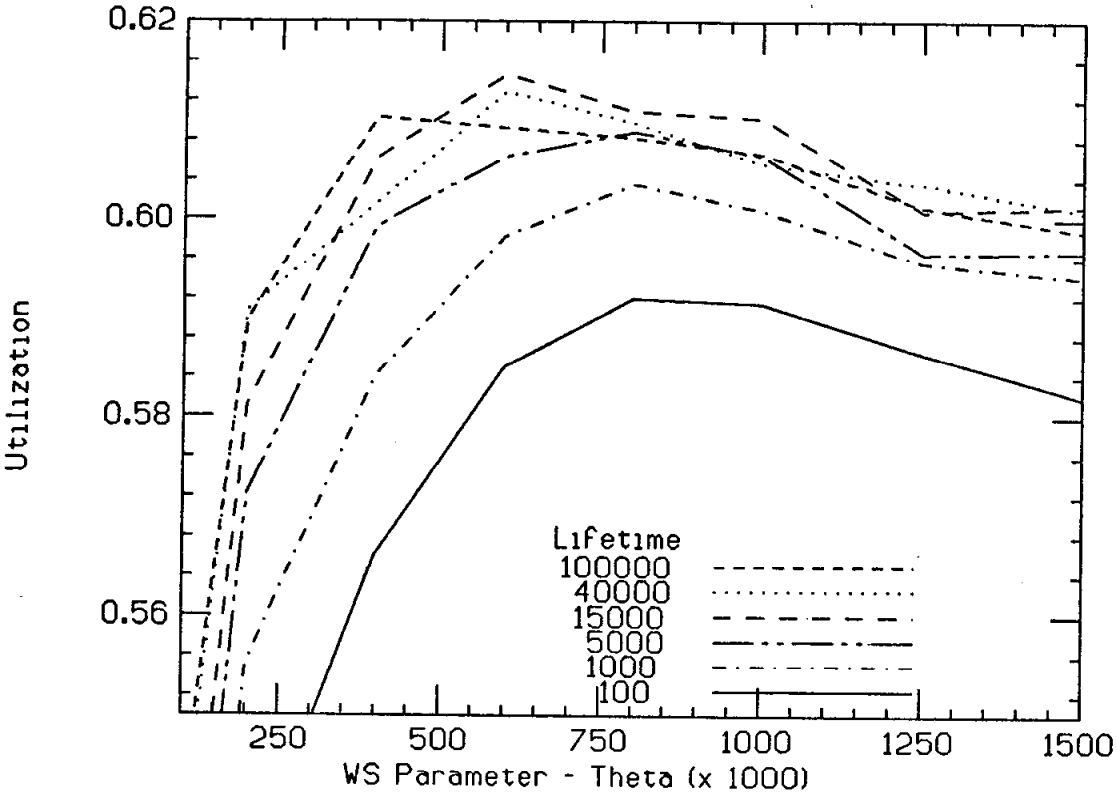
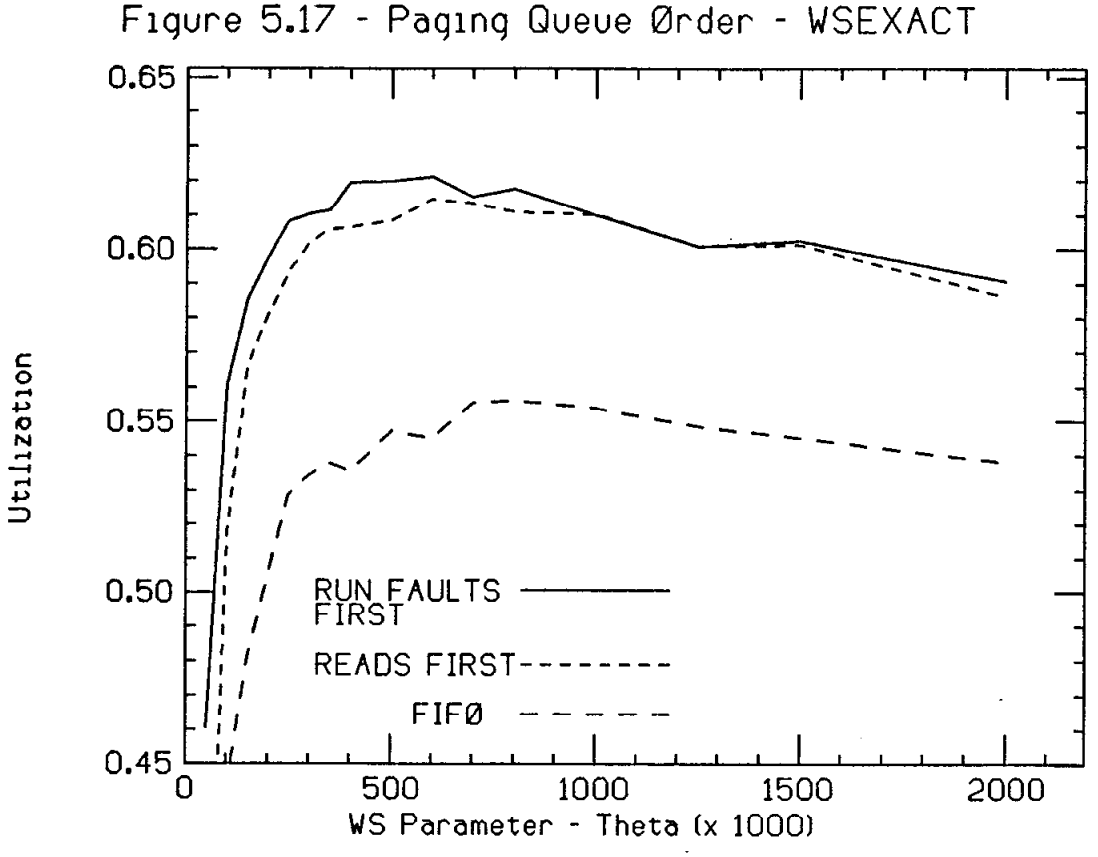


Figure 5.16 shows the effect of τ. The six performance curves are representative of a much larger number of tested values. As τ is increased to 15000, there is a steady, logarithmic improvement in performance. After this point, performance declines very slowly. These results demonstrate the robustness of the contol mechanism if τ is larger than the optimum. If τ is too small, some programs will be faulting frequently when the system no longer considers them to be loading their working sets; this error should be avoided.

5.3.3 Paging Queue Order

Figure 5.17 shows the performance of the model system under three paging I/O queueing policies: (1) page-ins and page-outs are processed in the order received; (2) all page-ins are processed before page-outs; and (3) page-ins for running programs are processed before page-ins for loading programs, which are processed before page-outs. The effect of processing page-in I/O operations before page-outs is clear.



5.3.4 WS Free Page Pool Size

The effect of varying the WS free page pool size K_0 is unclear, but we can still tune WSEXACT satisfactorily. We measure the model system for values of K_0 between 0 and 25, in steps of 5, (Figure 5.18(a)) and between 0 and 125 in steps of 25 (Figure 5.18(b)). While the best performance results for $K_0=0$, the amount of degradation when K_0 is 5, 10, 15, and 20 follows no obvious pattern. The range of performance between $K_0=0$ and $K_0=25$ is small: the maximum difference is approximately .7%. For larger values of K_0, there is a clear decline in performance as the pool size increases.

The purpose of the free page pool is to reduce the frequency of memory overcommitment and task demotion. It also has a negative effect of depressing the multiprogramming level and, thus, processor utilization. In Figures 5.19(a) and (b), K_0 has the expected effect on the number of demotions. It is interesting that the number of demotions is approximately constant across the range of θ; the significance of this observation is unclear.

In the next pair of Figures, 5.20(a) and (b), the effect of K_0 on the multiprogramming level is shown. Here we see little discernible difference when K_0 is 5, 10, or 15, which corresponds to the anomalous performance effect. As K_0 is increased, the positive effect of reducing overcommitment and demotions dominates the negative effect of reducing the multiprogramming level. In general, the need for a free page pool is questionable.

Figure 5.18(a) - Free Page Pool Size - WSEXACT

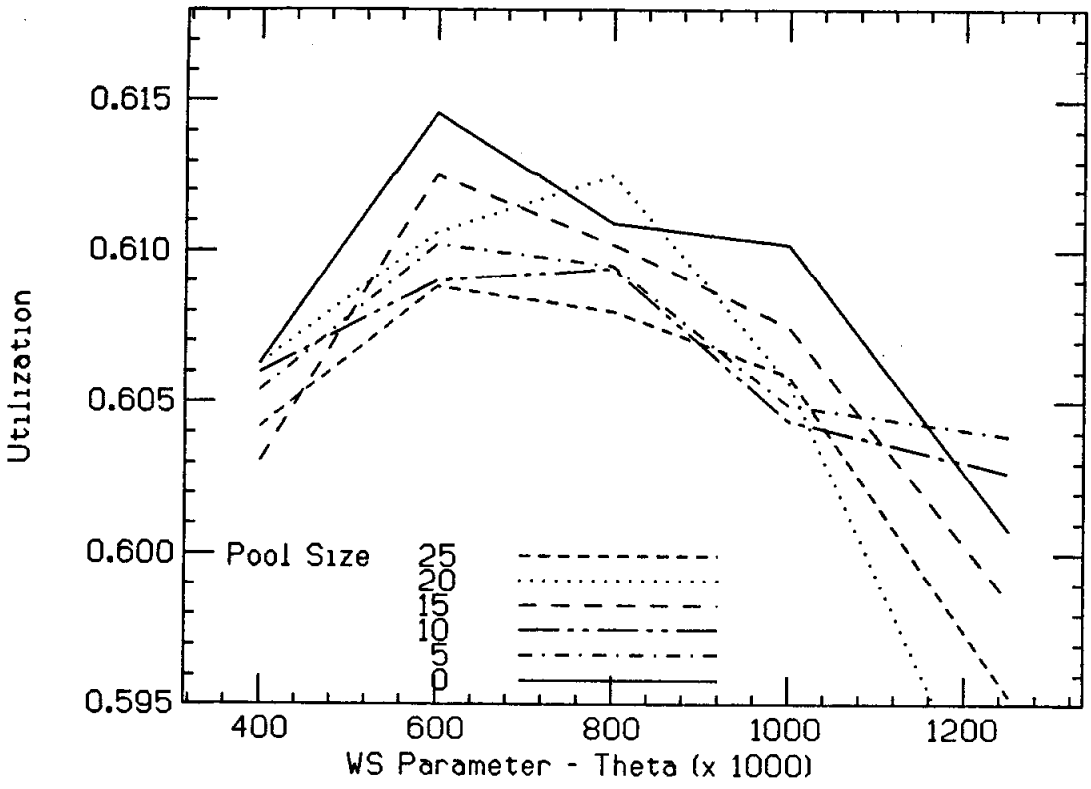


Figure 5.18(b) - Free Page Pool Size - WSEXACT

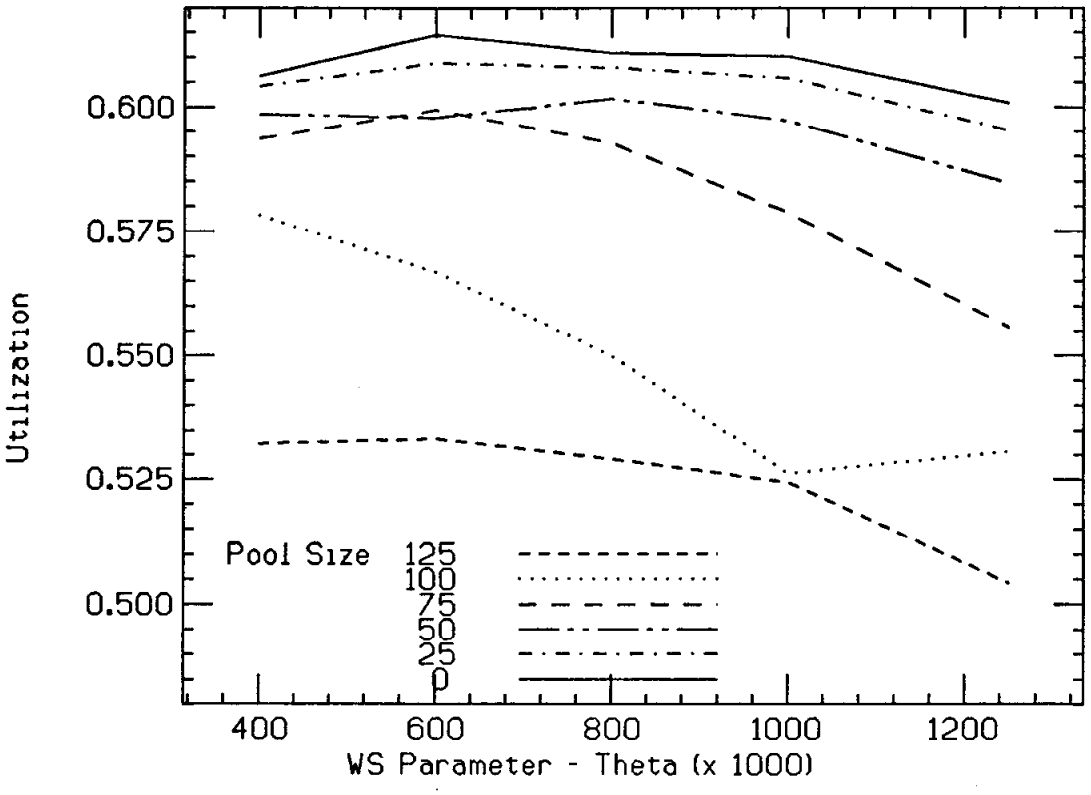


Figure 5.19(a) - Page Pool vs. Demotions - WSEXACT

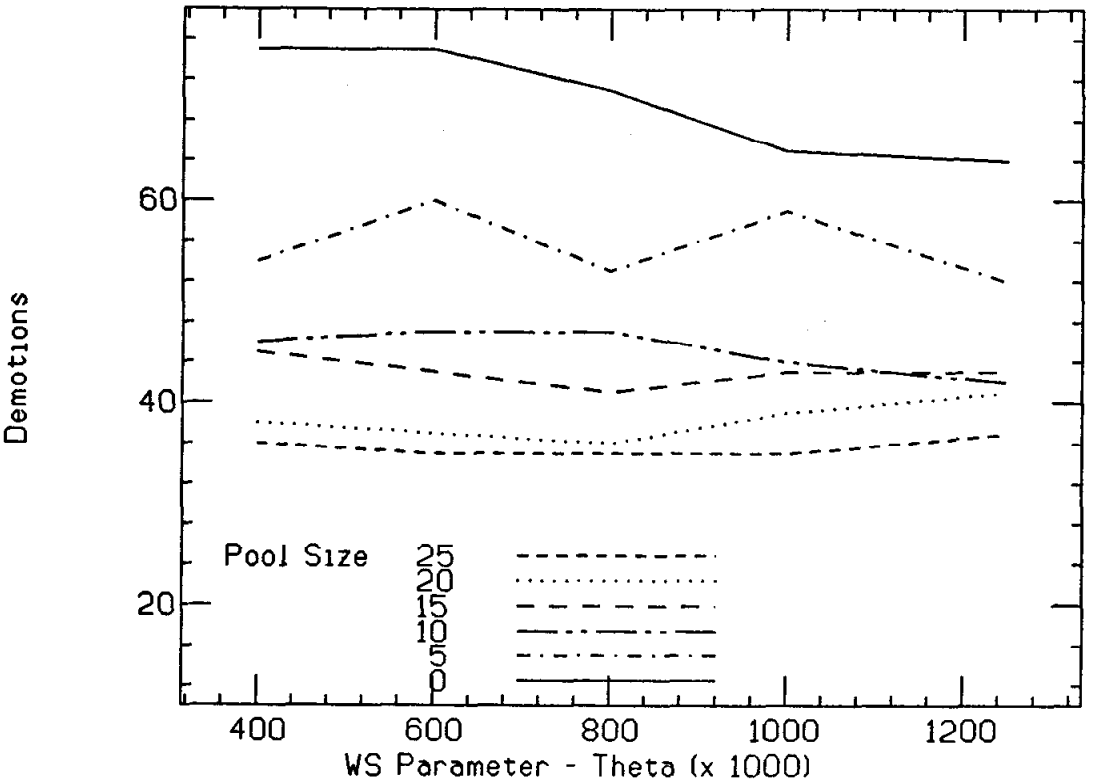


Figure 5.19(b) - Page Pool vs. Demotions - WSEXACT

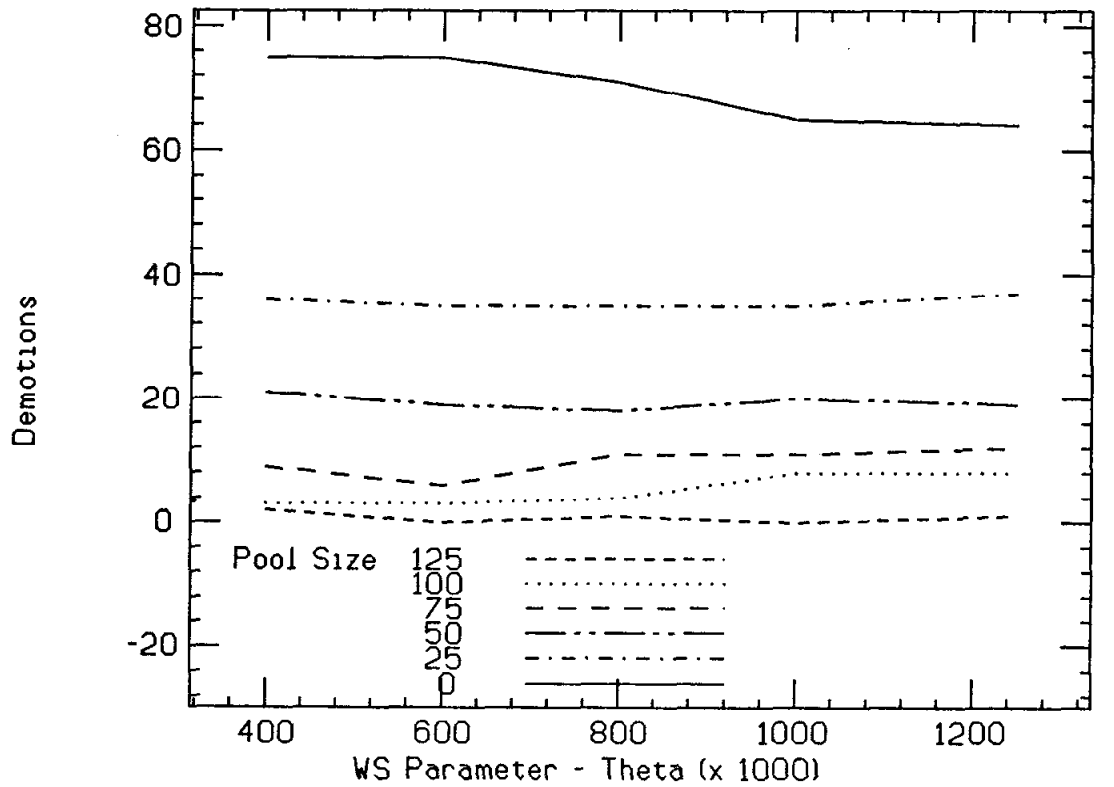


Figure 5.20(a) - Page Pool vs. MPL - WSEXACT

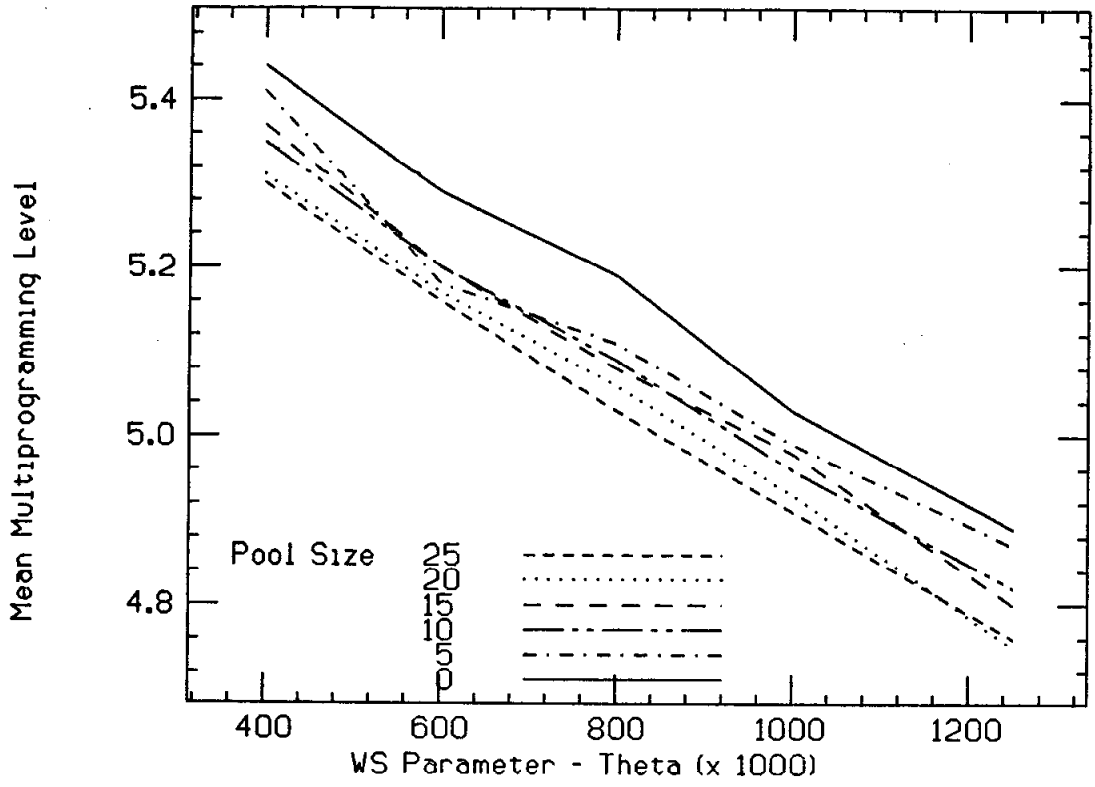
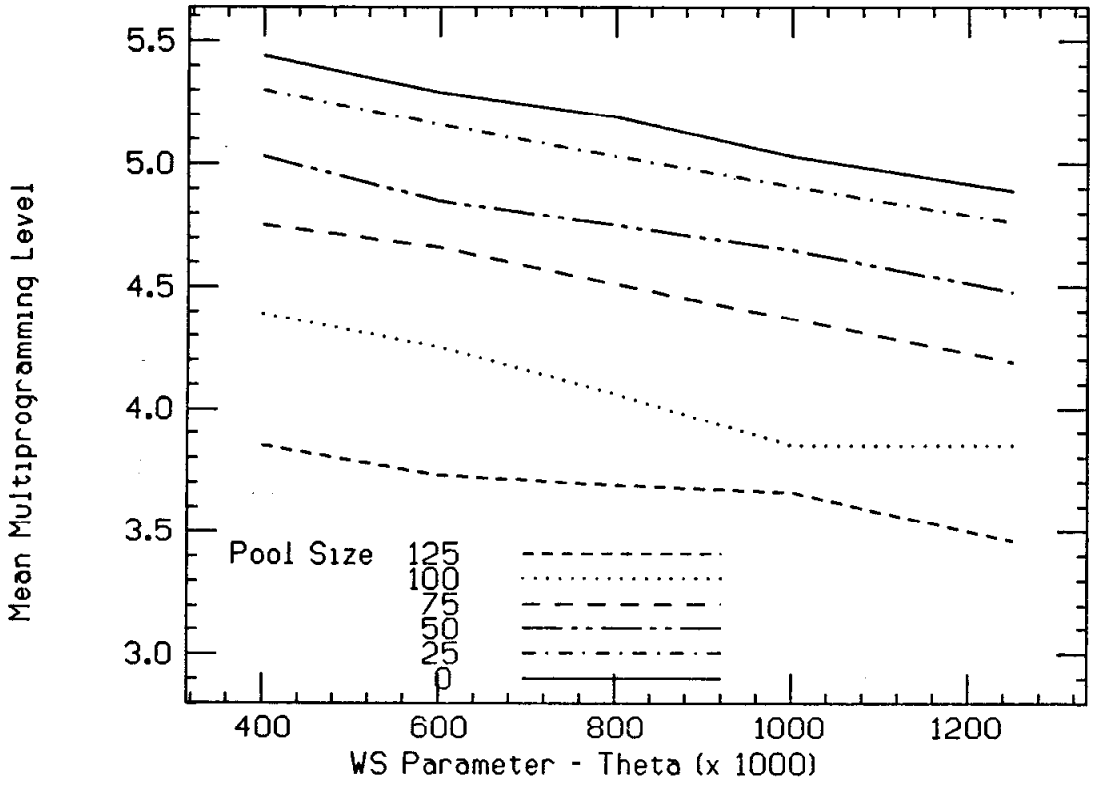


Figure 5.20(b) - Page Pool vs. MPL - WSEXACT



5.3.5 Evaluation of WS Approximations and VMIN

The practical WS approximations, WSFAULT and WSINTERVAL, are measured using the same configuration and load. The tuning parameters determined in the previous section are set to the values that optimized the performance of WSEXACT; although it is possible that different values would optimize each of the approximations, our experience with tuning WSEXACT indicates that the improvements would be minimal.

In Figure 5.21, WSEXACT, WSFAULT and WSINTERVAL, achieve about the same peak performance. Although the practical algorithms appear to be more sensitive to θ, the maximum observed difference among the algorithms is about 1.5%.

Finally, Figure 5.22 shows that the lookahead algorithm VMIN is considerably better than WSEXACT. The maximum difference in processor utilization is 6%, which represents a throughput improvement of 10%.

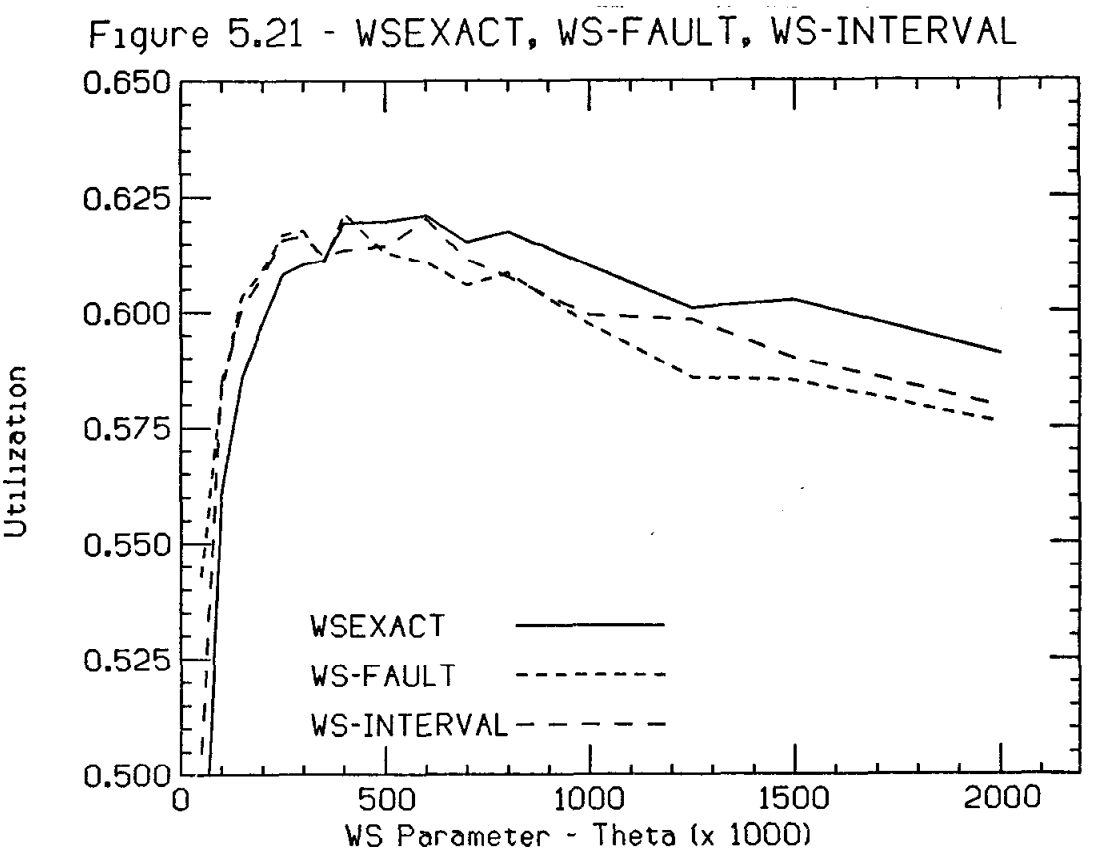
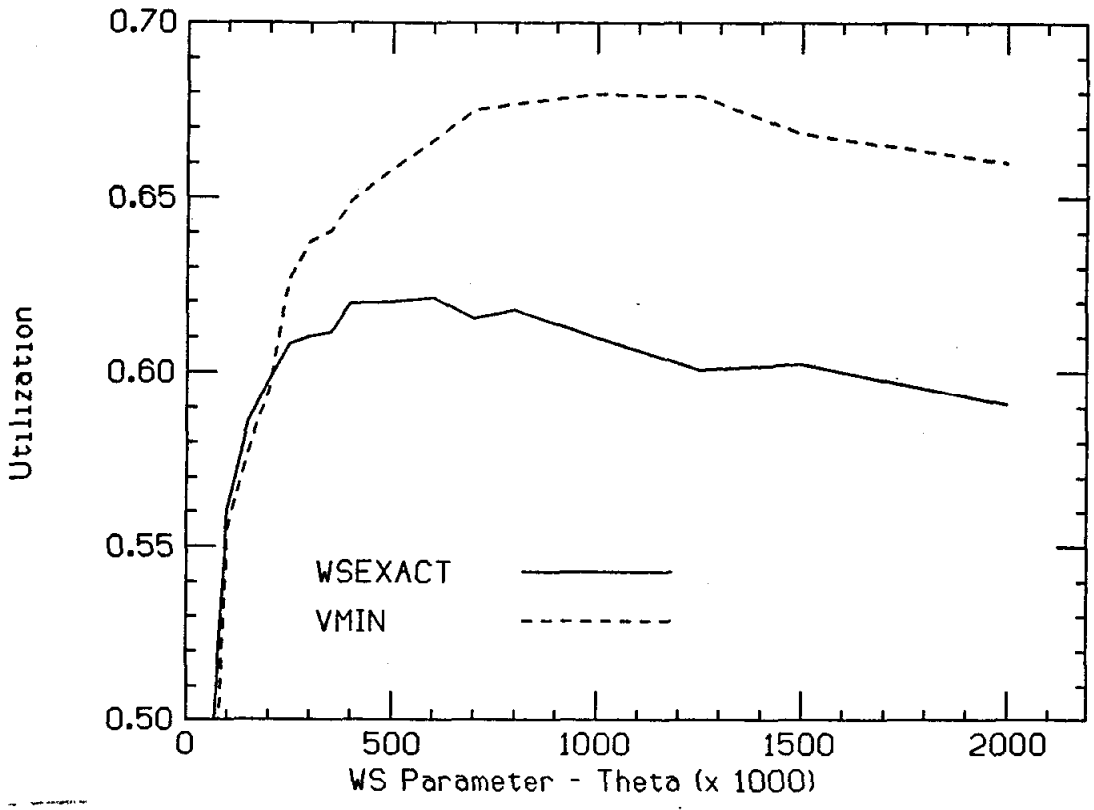


Figure 5.22 - WSEXACT, VMIN



5.4 Tuning the WSCLOCK Replacement Algorithm

We have verified that the the demotion-task policy, the *LT/RT* control, the paging queue policy and the WS free pool have the same effect in WSCLOCK as WSEXACT. Since the *LT/RT* control is a basic element of WSCLOCK, its effect is shown in Figures 5.23 and 5.24.

Figure 5.23 - Loading Task Limit - WSCLØCK

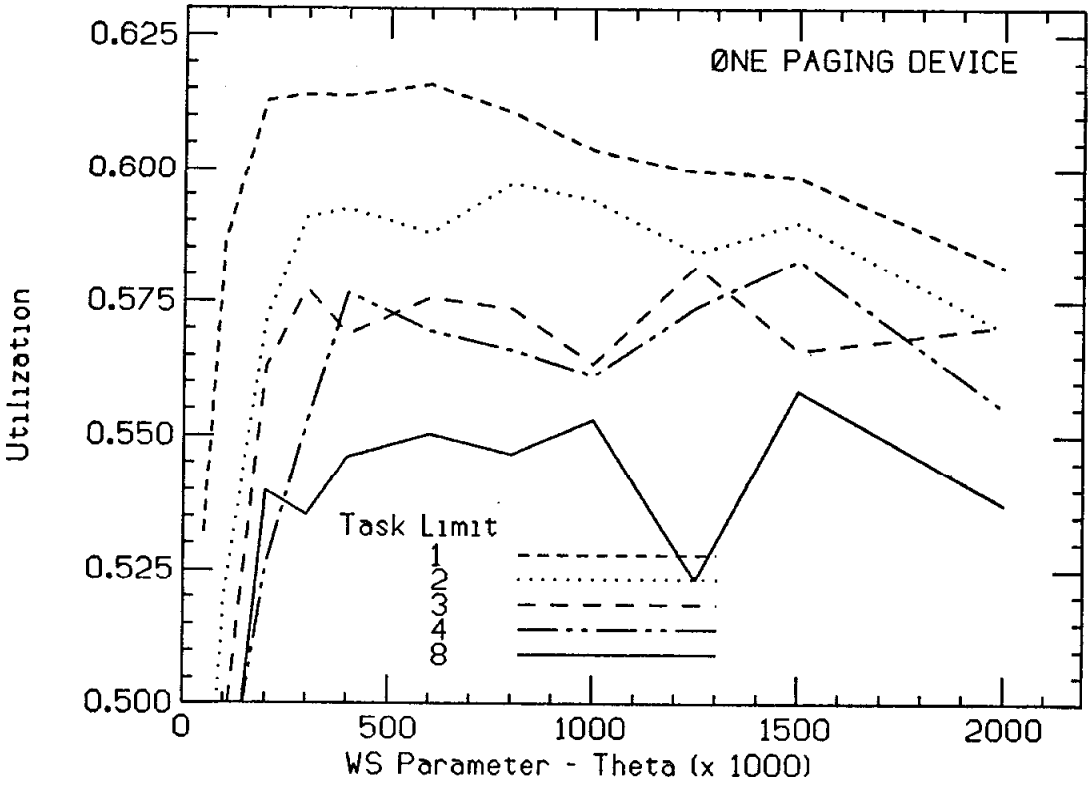
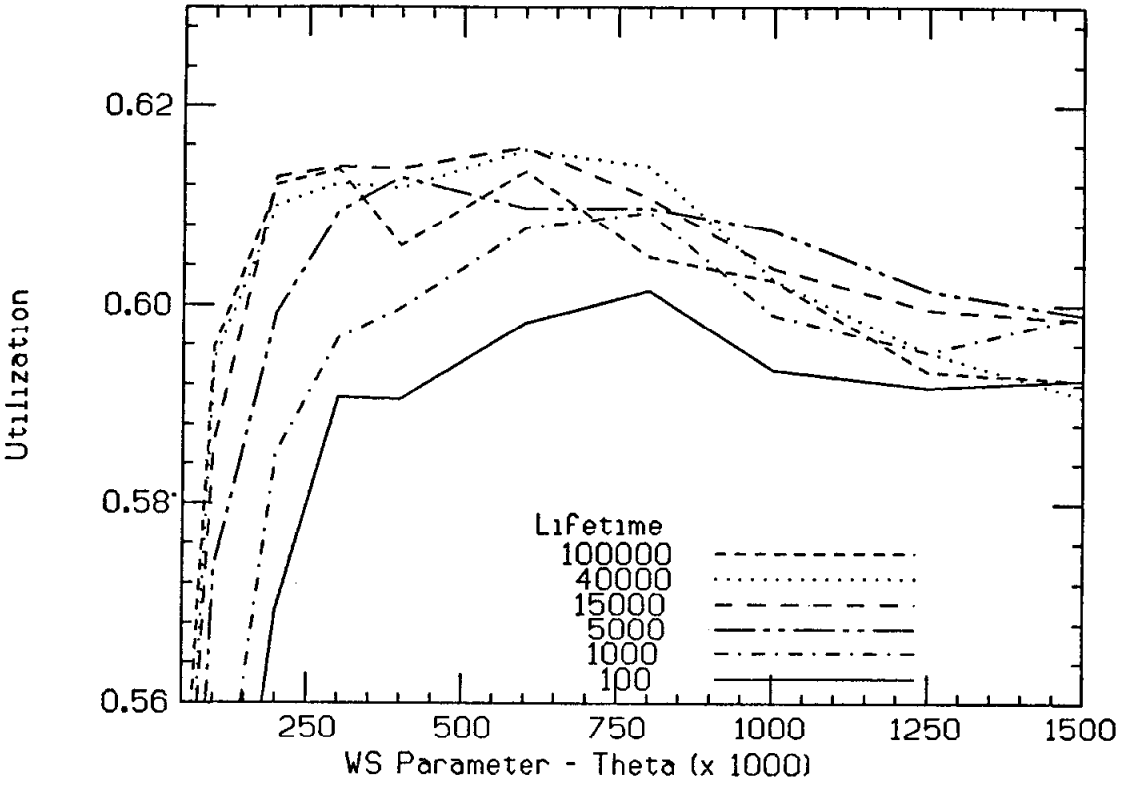


Figure 5.24 - Loading Task Lifetime - WSCLØCK

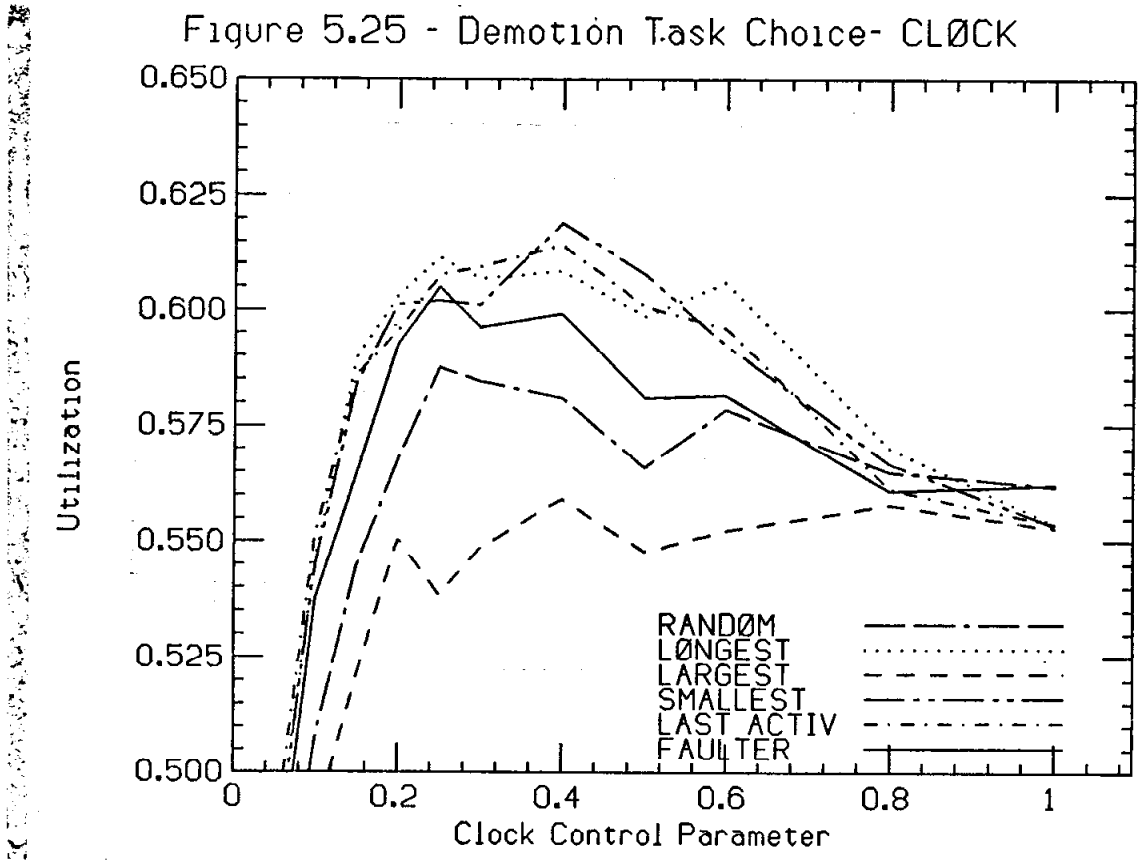


5.5 Tuning the CLOCK Replacement Algorithm

The procedures of this section follow those of Section 5.3, except that the performance of CLOCK is observed over a range of the control parameter C_0.

5.5.1 Task Demotion

Measurements of the six task demotion policies under CLOCK confirm the observations made under WSEXACT (Figure 5.25). Demoting (1) the task with the smallest resident set, (2) the task with the longest remaining quantum, or (3) the last activated task all result in comparable performance. Demoting the faulting task is poorer that these three but better than random selection. Demoting the task with the largest resident set is worse than a random selection.

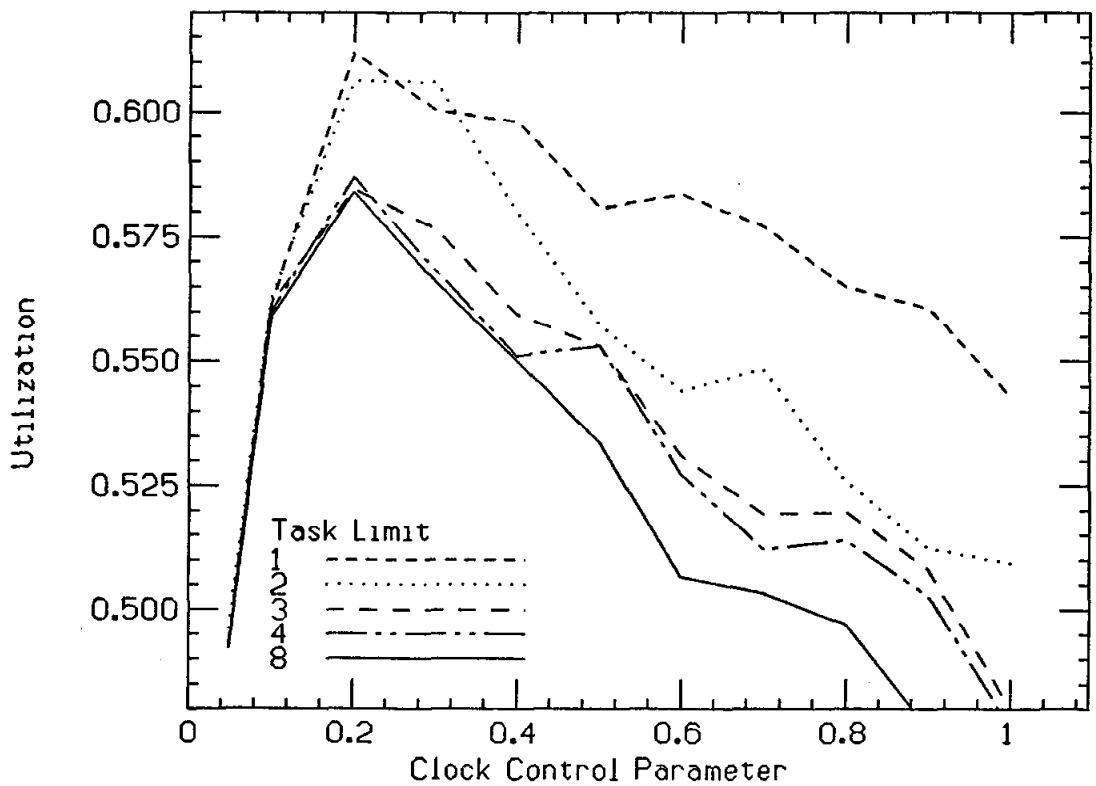


5.5.2 *LT/RT* Control

Measurement of the *LT/RT* control shows it to be useful under CLOCK (Figure 5.26) but having a somewhat different effect than under WSEXACT (see Figure 5.14). Under WSEXACT a loading-task limit $L=1$ is significantly superior to $L>1$, while under CLOCK both $L=1$ and $L=2$ are similar and are significantly better than $L>2$.

A clue to explaining this unpredicted result is the behavior to the right of the peak performance, where $L=1$ is superior to $L=2$. In this range of C_0, the multiprogramming level is higher than optimal and, thus, there is more opportunity for more than one loading task to be activated at the same time. We surmise that, at near-peak performance, the CLOCK load control naturally limits the number of loading tasks to no more than 1 and, in cases where two loading tasks are activated, competition between them has no ill effects.

Figure 5.26 - Loading Task Limit - CLØCK



The effect of the loading-task lifetime τ is quite small in the peak-performance range and grows in importance as C_0, and the multiprogramming level, become higher than optimal (Figure 5.27). As shown in Figure 5.28, larger values of τ depress the multiprogramming level and counterbalance a larger-than-optimal C_0, increasing the robustness of CLOCK.

We choose $L=1$ and $\tau=15000$ as our tuned system parameters; these represent a favorable compromise between peak performance and robustness.

Figure 5.27 - Loading Task Lifetime- CLØCK

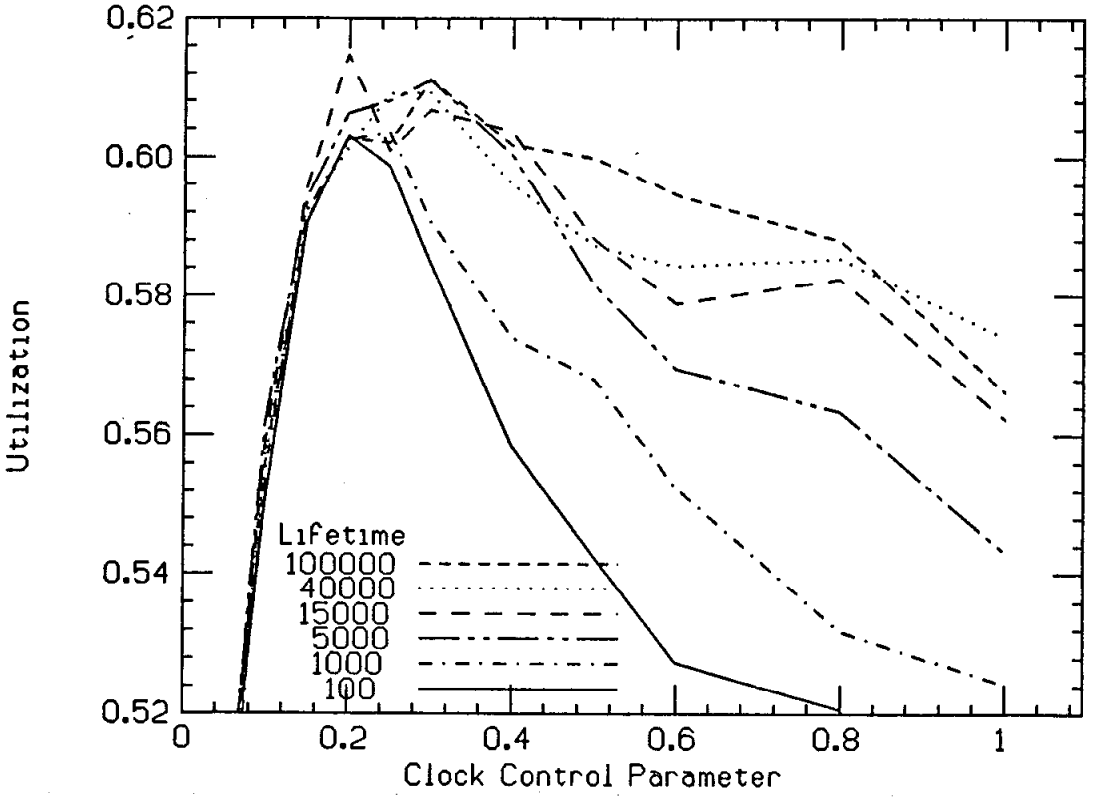
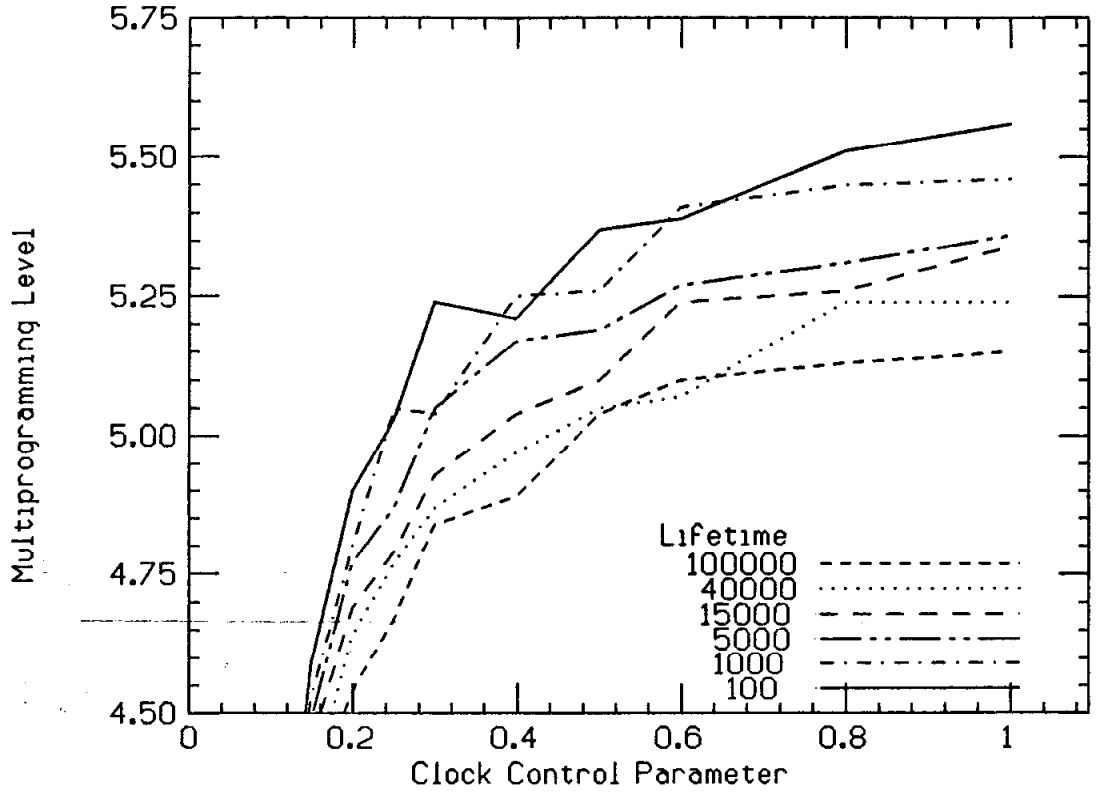
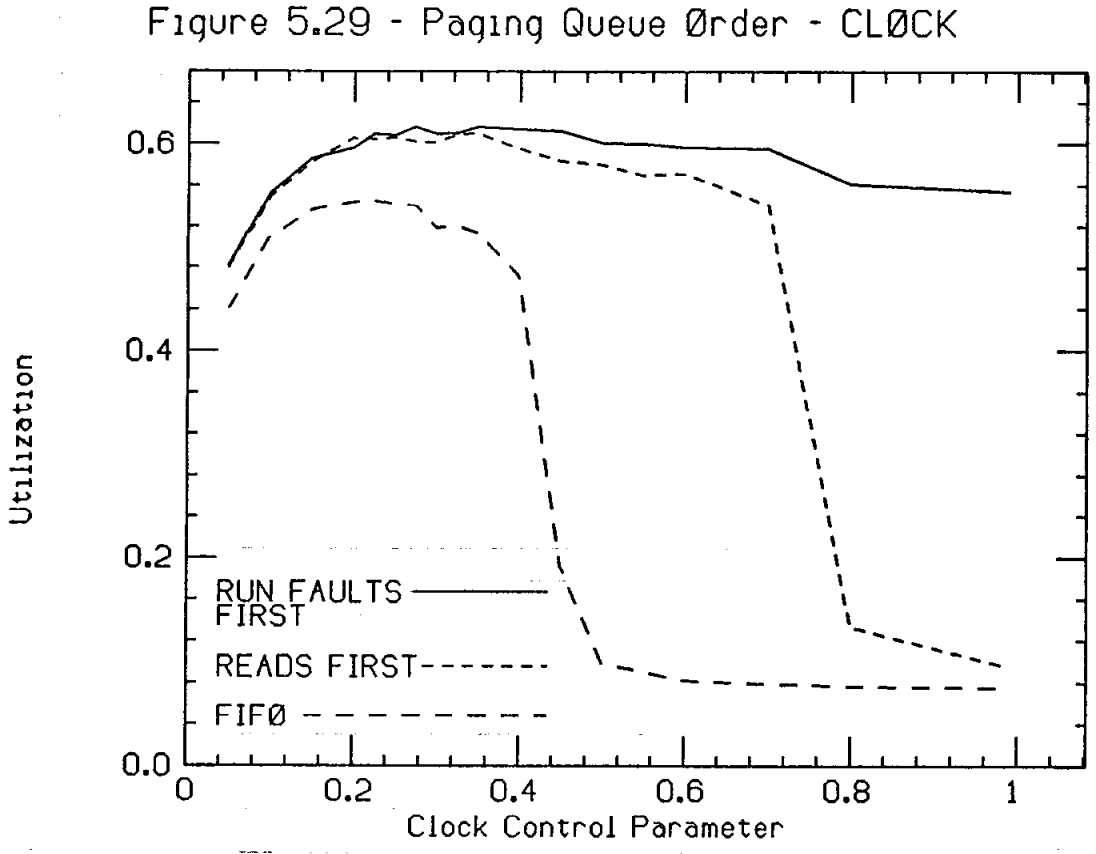


Figure 5.28 - Loading Task Lifetime vs. MPL - CLØCK



5.5.3 Paging I/O Queue Order

As observed under WSEXACT, ordering the paging I/O queue to process page-ins before page-outs improves CLOCK performance significantly, while ordering page-ins for running programs before page-ins for loading programs results in only a minor improvement in peak performance (Figure 5.29). Both of these policies, however, contribute to the robustness of CLOCK. With a FIFO paging I/O queue, peak performance is observed for a narrow range of the control parameter C_0, while the page-ins-first policy has a wider range and the running-task-page-ins-first policy has an even wider peak-performance range.



5.5.4 CLOCK Load Control Parameters

Our final tuning experiment is to measure the effect of the three CLOCK load control parameters δ, α, and φ. We perform a preliminary search for the optimal parameters using a coarse 3×3×3 design. For each parameter, we select 3 widely-spaced values as shown in Table 5.12.

Table 5.12 - Coarse Search-Design Parameters

Parameter	Values		
δ (observation interval)	.2 sec.	.5 sec.	.8 sec.
α (smoothing weight)	.2	.5	.8
φ (confidence level)	.67	.90	.98

As with the previous tuning experiments, each of the 27 parameter combinations is examined at each of the 11 values of C_0. Each simulation run models about 500 seconds of simulated time; this single experiment represents about 40 hours of simulated time. The results of this experiment are not shown graphically because it is difficult to display all 27 performance curves in meaningful combinations without overwhelming the reader.

In Table 5.13, we show the peak performance utilization observed with each parameter combination. The difference between the highest and lowest values is only 1%, and any simple relationship between the parameters and performance is indistinct.

Table 5.13 — Peak Performance - Coarse Search Design

		φ		
	δ	.67	.90	.98
	20	.6091	.6101	.6063
α=.2	50	.6071	.6142	.6137
	80	.6114	.6102	.6148
	20	.6151	.6117	.6173
α=.5	50	.6151	.6153	.6130
	80	.6134	.6116	.6100
	20	.6107	.6168	.6141
α=.8	50	.6136	.6098	.6158
	80	.6139	.6155	.6129

In order to detect any subtle patterns in these results, we construct Tables 5.14-5.16. In Table 5.14, we enter the value of α which gave the best performance for each combination of the other two parameters. If performance differed by less than .1%, we entered both or all values of α. We can see that $\alpha = .5$ has a majority, but no relationship to the other parameters is evident.

Table 5.14 – Peak Performance α's

	φ		
δ	.67	..90	.98
.2 sec	.5	.8	.5
.5 sec	.5	.2, .5	.8
.8 sec	.5, .8	.8	.2

Table 5.15 analyzes δ in a similar fashion: the value $\delta = .5$ has a majority; δ has no distinct relationship to the other parameters.

Table 5.15 – Peak Performance δ's

	φ		
α	.67	.90	.98
.2	.8	.5	.5, .8
.5	.2, .5	.5	.2
.8	.5, .8	.2, .8	.5

Table 5.16 analyzes φ in a similar fashion: no value of φ has a clear majority; no relationship to the other parameters is evident.

Table 5.16 – Peak Performance φ's

	α		
δ	.2	.5	.8
.2 sec	.67, .90	.98	.90
.5 sec	.90, .98	.67 .90	.98
.8 sec	.98	.67	.90

Normally, we would proceed from searching with this coarse design to searching with a refined design in the regions of higher performance. The results of this experiment give us little motivation to do so. The lack of any distinct difference or trend in each of the three parameters makes it highly unlikely that there exist parameters that are significantly better.

The preceding results might lead one to suspect that the clock load control is ineffective. This is not the case. In each of the experiments above, there are numerous (typically 50 to 100) times that the load control detects overcommitment and demotes a task. However, it appears that the *LT/RT* control plays a large part in preventing extreme overcommitment in the first place.

To show this effect, we measure the system with the CLOCK replacement algorithm but without an adaptive load control. Instead, we substituted a fixed multiprogramming-level (MPL) limit and the *LT/RT* control. We ran the model for MPL values from 2 to 10 (Figure 5.30), and found that performance leveled off at about 10% below the peak for MPL greater than 8. Without the *LT/RT* control, thrashing and very low performance would have resulted from increasing MPL. Figure 5.31 shows the mean number of active tasks for each value of MPL. At any given time the number of active tasks is limited by both the MPL limit and the *LT/RT* control. At large values of the MPL limit, the *LT/RT* control plays a more active role and effectively prevents extreme overcommitment.

Figure 5.30 - Fixed MPL Load Control - CLØCK

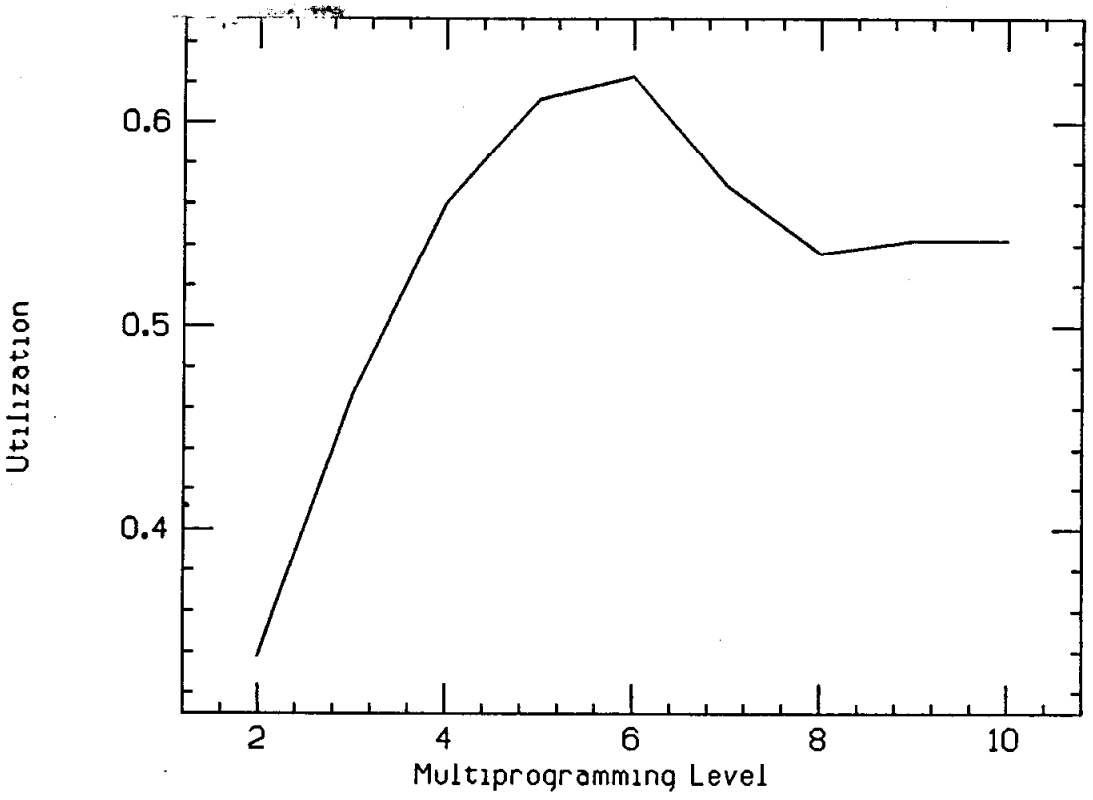
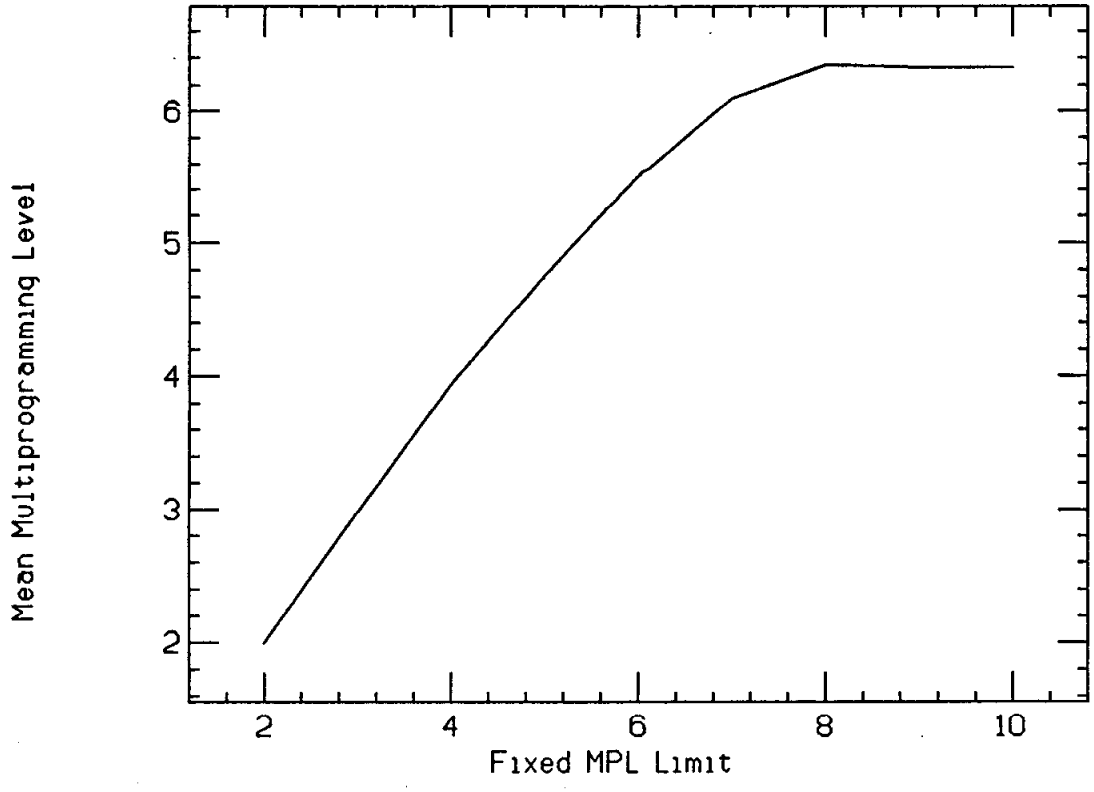


Figure 5.31 - Fixed MPL VS. Mean Multiprogramming Level

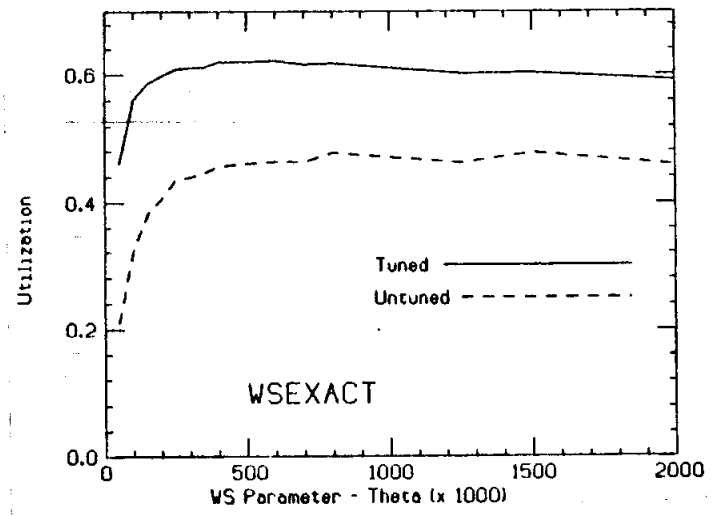


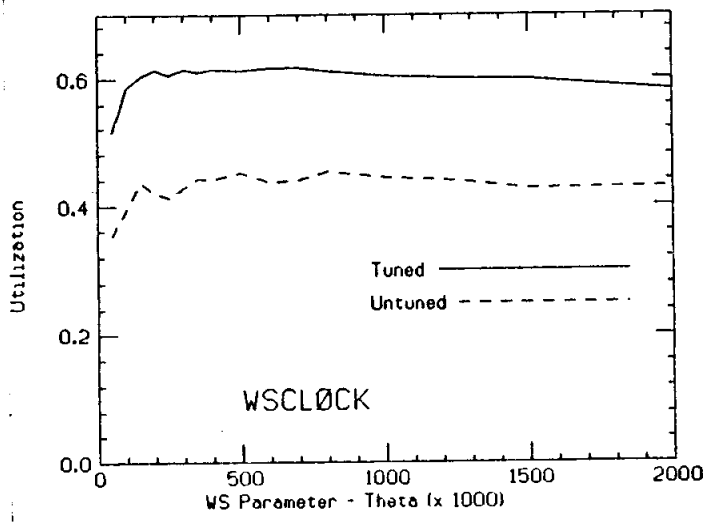
5.6 Comparison of WSEXACT, WSCLOCK, and CLOCK Policies

5.6.1 Performance Comparison

The performance curves of the untuned and the fully-tuned WSEXACT, WSCLOCK, and CLOCK policies appear in Figure 5.32. The untuned policies have a random demotion-task policy, no *LT/RT* control, and a FIFO paging queue. Since the algorithms have different control parameters, we do not compare these performance curves directly. All three policies show sufficient robustness. Although CLOCK appears to have more sensitivity to its control parameter C_0, it has a wide range of stablility; a more "judicious" choice of parameter range (e.g., only those between .2 and .6) would have resulted in a performance curve that has a flatter appearance. In Figure 5.33, we show that all three policies reach their peak performance at the same mean multiprogramming level, even though their replacement strategies and load controls are significantly different.

Figure 5.32 - WSEXACT, WSCLØCK, and CLØCK





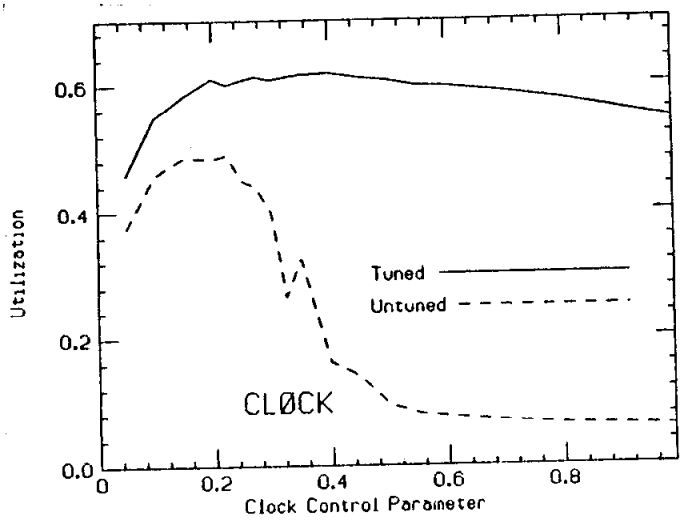
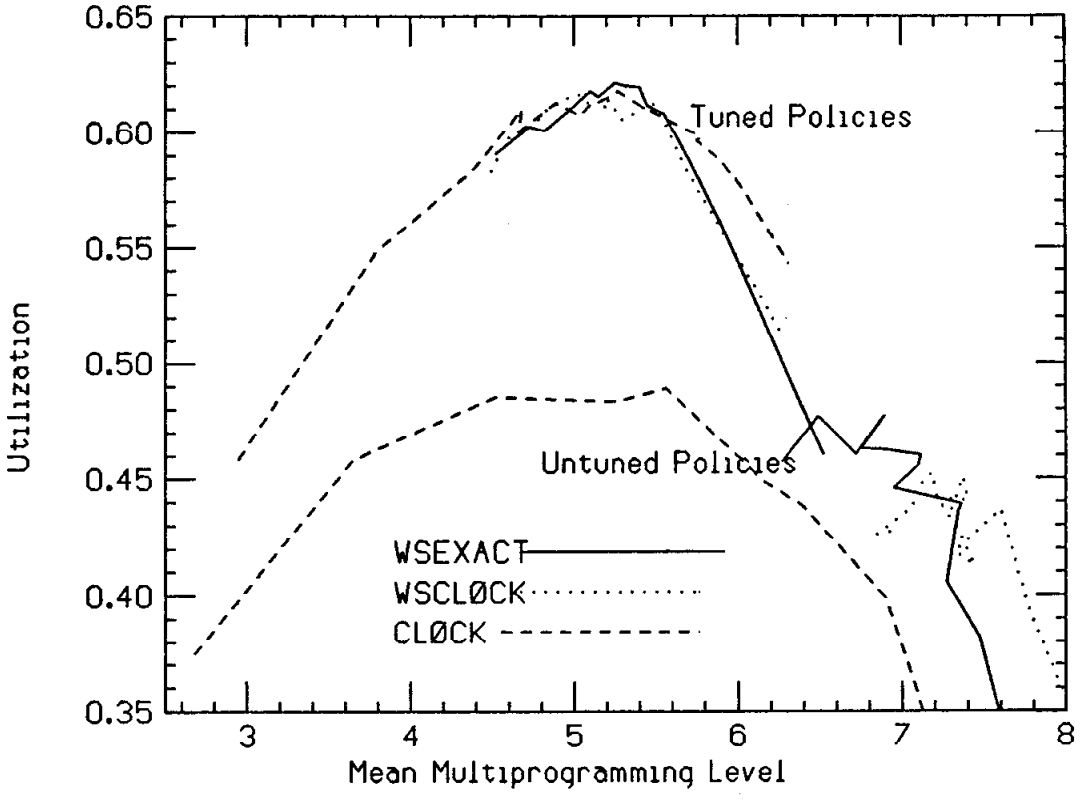


Figure 5.33 - Utilization vs Mean Multiprogramming Level



Although we dislike characterizing algorithms by their peak performance, we have little alternative. Table 5.17 gives the statistics for the peak performance observed for WSEXACT, WSCLOCK, and CLOCK for the same configuration, load, and simulation run length.

Table 5.17 - Peak Performance - WSEXACT, WSCLOCK, CLOCK

Algorithm	Utilization	Mean MPL	Parameter
WSEXACT	.6211	5.25	$\theta = 600{,}000$
WSCLOCK	.6164	5.01	$\theta = 700{,}000$
CLOCK	.6177	5.27	$C0 = .40$

The maximum difference in performance is .47%. Considering the accuracy of the model and non-exhaustive methods of tuning each algorithm, we find this difference to be insignificant.

We experiment with the algorithms using the tuning parameters that produced peak performance by running the simulation for a much longer time. Figure 5.34(a) shows cumulative utilization at each point in the simulation run time. This graph, in which the utilization scale is based at zero, illustrates the insignificance of the difference among the algorithms. In Figure 5.34(b), the scale is reduced to magnify the difference. It can be readily seen that the load-produced variations dwarf the measured differences between the algorithms.

In these experiments, utilization was measured for each five-second interval of simulated time. We compare the distributions of these intervals in Figure 5.35. This figure shows that the three algorithms have an equivalent distribution of high-utilization intervals, but that in low-utilization intervals, CLOCK is slightly worse than the WS algorithms. A follow-up study might determine whether the distributional difference was due to longer periods of undercommitment or overcommitment, and determine if the load control could be improved.

Figure 5.34(a) - Cumulative Utilization

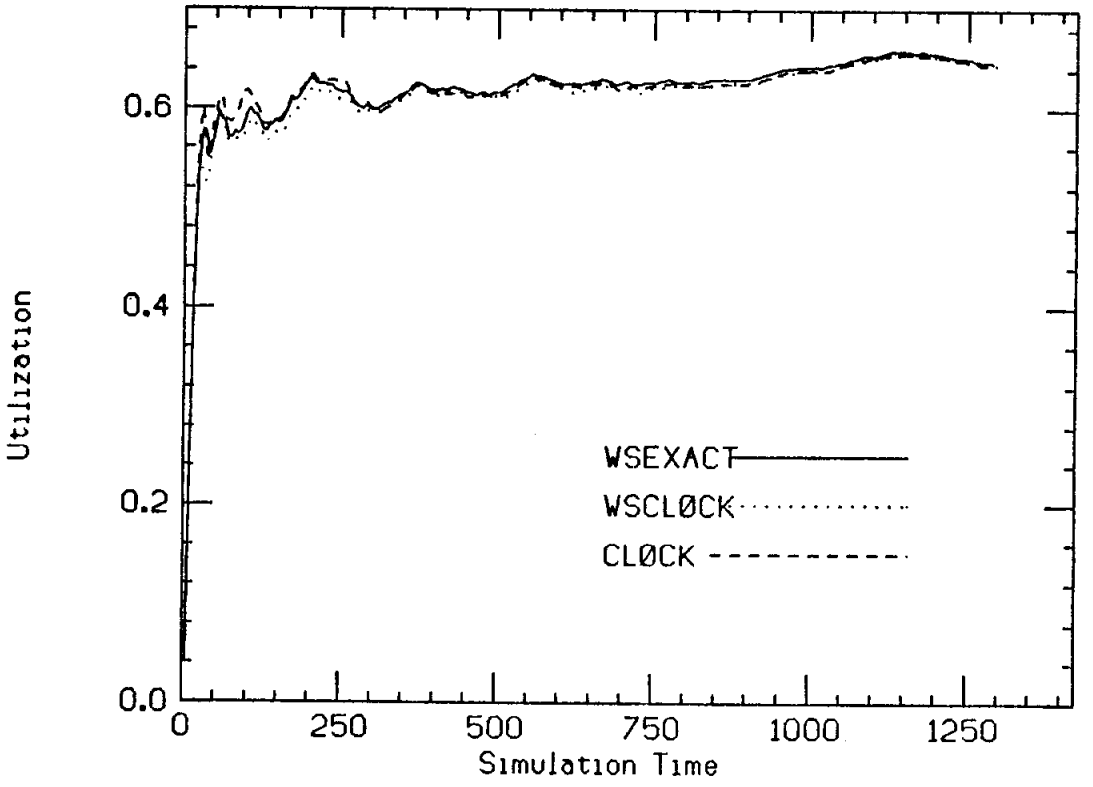


Figure 5.34(b) - Cumulative Utilization

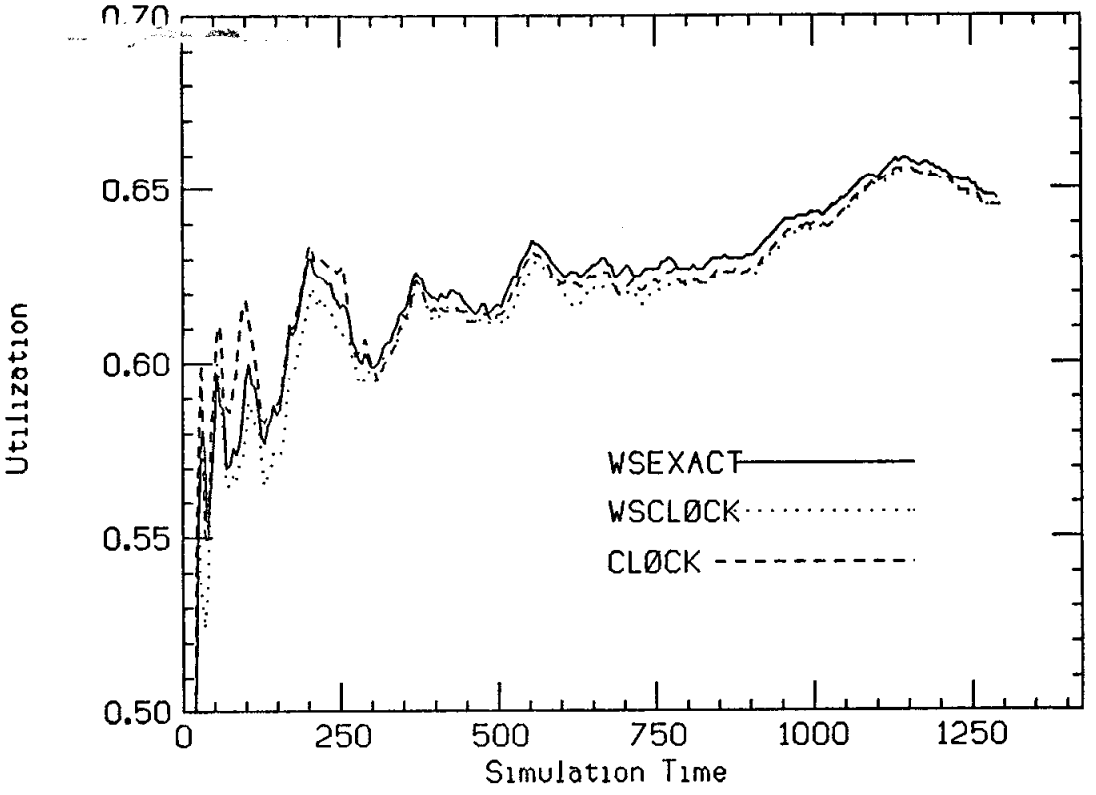
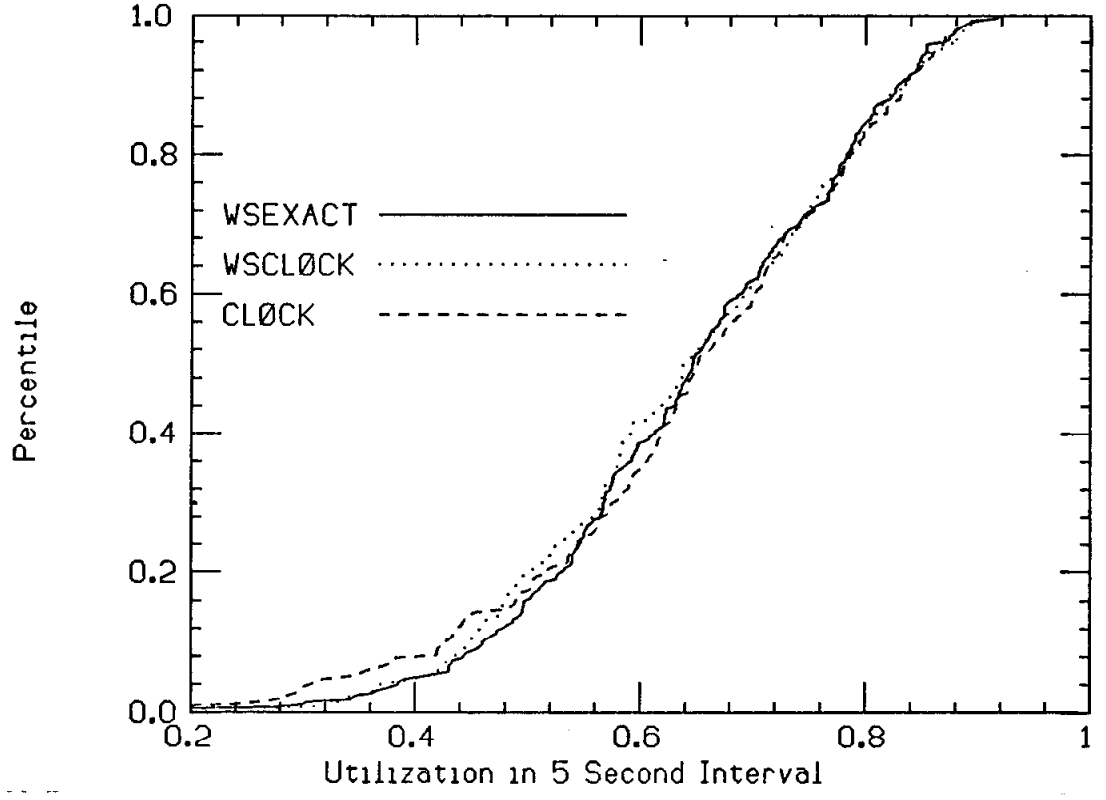


Figure 5.35 - Cumulative Distribution of Utilization



5.6.2 Operating System Overhead Comparison

This research has been inspired by certain conjectures about different memory management policies. [DENN80, SIMO79] These conjectures hold that one policy is inherently superior to another, without regard for the cost of implementation or overhead. Thus, this work has examined the relative performance of these policies in a model that did not include operating system overhead.

Although the incorporation of operating system overhead in the simulation model would have presented little difficulty, this was not done. If the model included overhead, it would have been necessary to assign costs to each of the operating system functions. In real systems these costs can vary considerably, depending upon the processor, the system implementation, the complexity of the scheduler or the memory hierarchy, and the availability of special aids such as microcode space for implementing operating system functions in firmware.

In lieu of modeling the effect of overhead, we shall present some external measures of the cost of computing the policies studied above. This measurement depends on two factors:

1) The simulator program is written in a systems implementation language, and the programming used to model the operating system is very close to the programming that might appear in a real operating system. Thus the execution of the simulator program models the execution of the model operating system.

2) There exists a method for measuring the execution of the simulator and estimating the total time spent in each of the functional areas of the operating system model.

We have describe the programming of the operating system model in Chapter 4. The measurement of the simulator is performed by a program PROGLOOK described in [JOHN76]. PROGLOOK monitors the execution of any other program and samples the program counter at regular intervals. A histogram, such as the one in Figure 5.36, tabulates these samples and provides an accurate estimate of the time spent in each small section of the program. With the

aid of a program map, the time spent in each functional part of the operating system model is easily determined.

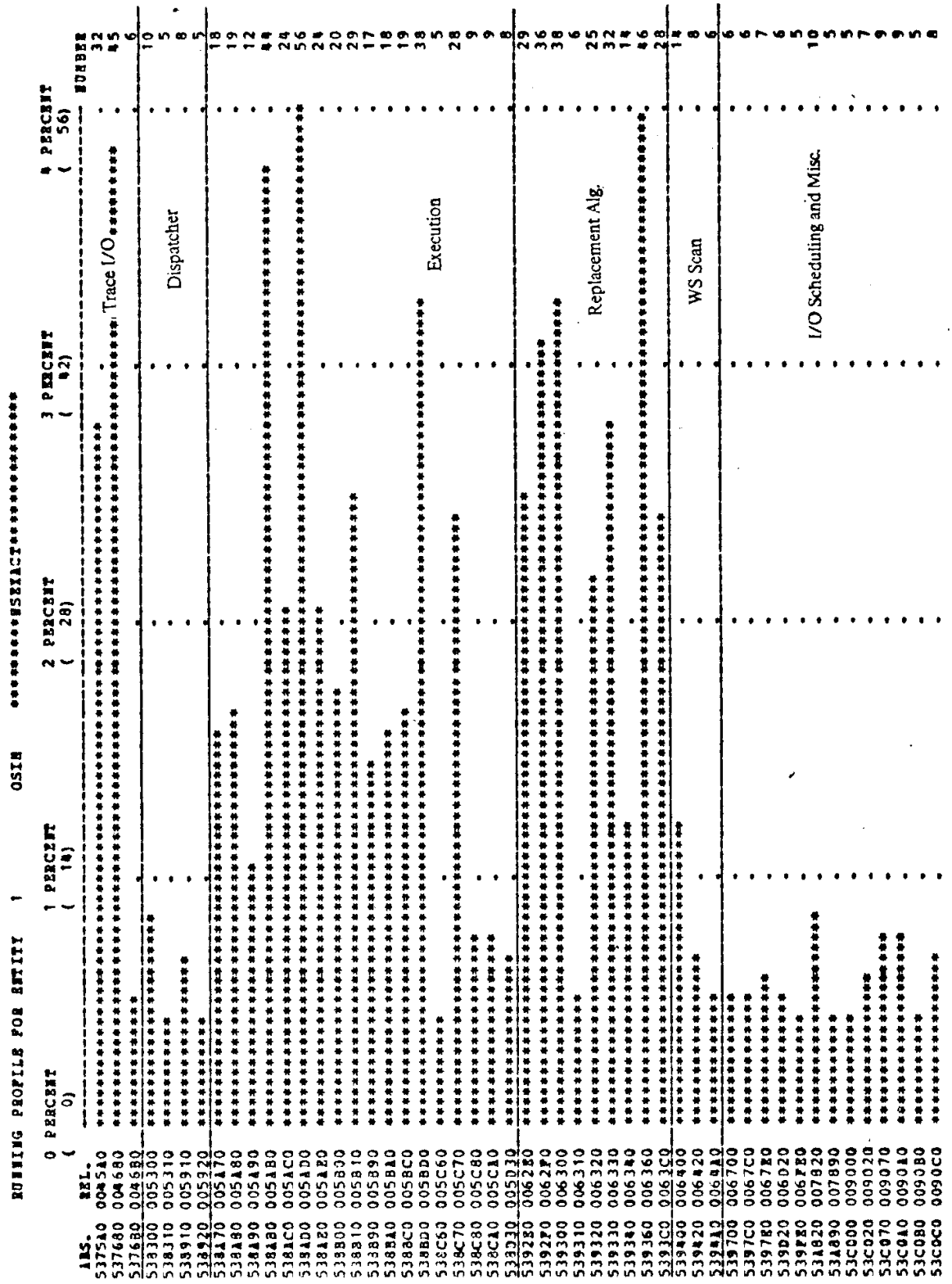


Figure 5.36 – PROGLOOK Report

The peak performance simulation results of the previous subsection were measured by this method and are presented in Table 5.18. Each of the simulations processed the same number of tasks and simulated references in the same amount of simulated time (within .5%).

Table 5.18 - Policy Cost Comparison

	Policy		
	WSEXACT	WSCLOCK	CLOCK
Simulation Measures			
Utilization	.6211	.6164	.6177
Page Reads (Faults)	5437	5041	5640
Page Writes	3556	3313	3802
Mean Clock Scan	15.92	14.07	6.92
Simulation Costs (CPU Secs.)			
Total	16.1	18.9	14.7
Task Model			
IRIM Trace I/O	1.6	1.6	1.5
Task Execution	5.3	8.1	7.9
Total	6.9	9.7	9.4
Operating System Model			
OS Scheduler	0.5	0.7	0.5
Replacement Alg.	4.1	4.5	0.7
Working-set Scan	0.6	—	—
I/O Scheduling	3.7	4.1	4.0
Total	8.9	9.3	5.2

The first four rows of Table 5.18 describe the net processor utilization, the number of simulated pages reads and writes, and the average number of pages examined in the clock scan before a replaceable page was found. The following lines represent the time spent executing various functional parts of the simulator.

Although each policy had a comparable performance, we note that CLOCK had about 4% more page faults than WSEXACT, which had about 8% more faults than WSCLOCK. On each fault, CLOCK examined fewer than half as many pages to find a replaceable page than did either WSEXACT or WSCLOCK. This behavior is expected, since the WS algorithms have stricter rules preventing the replacement of recently referenced pages.

Examining the relative costs, we observe that WSCLOCK and CLOCK used considerably more computer time to simulate the same task models as WSEXACT. This is due to the procedure to ensure that the use-bit of each busy page is set whenever the task is executed, while the simulation of WSEXACT does not require the setting of the use-bits. (See the description of *Swapin* in Section 4.3.4.) In any case. the cost of processing the task model is simply a modeling cost, and does not represent the cost of processing tasks in a real system.

The cost of processing the operating system model, on the other hand, is directly related to corresponding costs in a real system. Since each modeling run simulates a real time equivalent to about 400,000,000 reference times, and since the IBM 370/168 computer processes about 4,000,000 references per second, each second of execution in the operating system model represents about 1% of total simulated processor capacity.

Of primary interest in this section is that the WS policies spend about 6 times as much time in the replacement algorithm and, for WSEXACT, the working-set scan. Thus, in these measurements, the WS paging algorithms consume about 4.5% of capacity, while the CLOCK algorithm consumes less than 1%. The cost of computing the adaptive feedback control for CLOCK is negligible.

While these measurements are dependent upon the implementation of the model, we believe they are indicative of costs in a real system.

5.7 Summary

We have described the preparation of a simulation model to study virtual memory management policies. The model is shown to be both efficient and accurate. Numerous experiments have compared WSEXACT to variant WS replacement algorithms and a CLOCK replacement algorithm. It has been shown that approximations to WSEXACT have performance comparable to WSEXACT; in particular, the simpler WSCLOCK algorithm is shown to perform quite well. This study has been unable to find significant performance differences between the local WSEXACT algorithm and the CLOCK algorithm.

Although we claim this comparison of local and global memory management policies to be as accurate as any reported, we do not consider it definitive. Virtual memory is used in many different systems with widely varying configurations and loads; there may be significant advantages to one policy or the other in environments not explored by this study. In particular, interactive systems make very different use of the replacement algorithm and load control mechanisms and may favor one policy over the other. Our basic conclusion is that there is little evidence for claiming a theoretical superiority for any of the non-lookahead policies studied.

On the practical level, it appears that the CLOCK algorithm is preferable to the WS algorithms because it has less operating system overhead while achieving approximately the same performance in the processing of user tasks. We also show that the WSCLOCK algorithm is easier to implement tha the other WS algorithms, but performs as well.

Chapter 6

Summary and Conclusions

6.1 Summary

A pragmatic approach was pursued to extend our understanding of virtual memory management. New methods of managing a virtual system were developed and incorporated in a general, but functional, description of practical system management methods.

To evaluate virtual memory management methods, we investigated the use of analytical and simulation models. Analytical models, as well as simulation models that depend on analytical sub-models, were found to be too restrictive and inaccurate for this particular application. The major problem of detailed simulation models—their computation cost—was addressed and solved without sacrificing accuracy or detail.

Numerous virtual memory computer system policies have been studied and evaluated. In particular, a direct comparison of local and global memory management policies was made.

6.2 Contributions

The contributions of this thesis fall under three headings. First, we have described a wide spectrum of modern scheduling, memory-management, and load-control methods for large-scale, multiprogrammed, virtual-memory computer systems. We have developed new algorithms for scheduling interactive systems, for page replacement, and for adaptive load control. Second, we have developed a number of models that are well suited for evaluating and comparing these methods. Third, we have used the models to show the usefulness of the methods we have developed, and to provide new insight into long-standing questions about virtual memory management.

Operating System Methods

In Section 2.1.3, we introduced the multi-level load-balancing queue. This scheduling discipline solves a number of problems of maintaining good interactive response time while achieving high utilization of system resources. In particular, it addresses problems of servicing distributed processors from a large central processor. In Section 2.2.5, we described the CLOCK algorithm for global page replacement and extended it to be a universal mechanism for both local (WS) and global replacement. We also devised a new algorithm, WSCLOCK, that combines the efficiency and simplicity of global memory management policies with the load control capability of local WS policies. WSCLOCK is actually simpler than either the standard WS or global policies, but it performs about as well. In Section 2.3.4, we describe a new adaptive CLOCK load-control mechanism for global policies. This mechanism is based on the CLOCK replacement algorithm and uses exponentially-smoothed confidence intervals, which are also developed in Section 2.3.4. In Section 2.3.5, we describe the *LT/RT* control, which is a simply-implemented method to attenuate thrashing under both local and global memory management policies.

Models

Chapter 4 described an discrete-event trace-driven simulation model of a virtual memory, multiprogrammed computer system. The model incorporates a detailed model of memory management, including the modeling of individual pages and page frames, but eliminates the need to represent and transfer page contents. In Section 4.3.3, we described a deterministic, trace-driven, memory-referencing model of program behavior, the Inter-Reference Interval Model. The IRIM is an economical method of analyzing real programs and modeling their memory-referencing behavior accurately and efficiently. The IRIM is a key development that permits the construction of a virtual memory system model that is both practical and has a precise representation of virtual memory management. In Section 4.4, we developed an operating system model that incorporates the scheduling, memory management, and load control methods described in Chapter 2. This model is specifically designed to eliminate implementation artifacts that could obscure the direct comparison of memory management and load control methods.

Experimental Studies

In Section 5.3, we studied the WSEXACT policy and evaluated the usefulness of various other controls and policies. The demotion-task policy, which selects a task to deactivate when overcommitment is detected, should choose the last-activated task, or the task with the longest-remaining quantum, rather than the faulting task, the task with the largest resident set, or a randomly-chosen task. The choice of the task with the smallest resident set will result in good performance, but is not a good policy since it would penalize programmers who reduce their programs' localities.

Under WSEXACT, the *LT/RT* control improves performance significantly over the standard WS controls. In the representative model system, the *LT/RT* control improved throughput by about 10%. The model also confirms the usefulness of the policy of performing page reads before page writes and shows a small additional improvement by performing page-ins for running tasks before page-ins for loading tasks. Finally, the experimental evidence indicates that the WS free page pool, designed to reduce overcommitment, has a negative effect on performance and should not be used.

The WSFAULT and WSINTERVAL algorithms to approximate WSEXACT are quite satisfactory: there is little need for special hardware to implement the standard WS policy. The lookahead VMIN algorithm does result in a significant gain in performance.

The new algorithm, WSCLOCK, also performs as well as WSEXACT, even though it is much simpler than any of the other WS algorithms.

In Section 5.5, it is shown that (1) the demotion-task policy, (2) the *LT/RT* control, and (3) the paging-I/O queuing policy have similar effects under GLOBAL as they do under WSEXACT. The clock-based adaptive load control is shown to be effective, although the usefulness of exponentially-smoothed confidence intervals is inconclusive and deserves further study.

In Section 5.6, we are able to conclude that we can find no substantial evidence that either WSEXACT or GLOBAL is inherently superior to the other. The three policies cited in the previous paragraph all have substantially greater effects on performance than the choice of replacement algorithm. The WSCLOCK algorithm also performs as well as either WSEXACT or GLOBAL. An analysis of operating system overhead confirms that CLOCK has less cost than the WS algorithms and, on this basis, is preferable to the WS algorithms.

6.3 Conclusions

It has been generally assumed that detailed simulation models of virtual memory computer systems are very expensive to create and use. This situation has led those who would study such systems to favor either analytical queueing network models or simulation models that use simple analytical models of program behavior. While these models may be economical to use, they fail to capture the dynamics of memory allocation and sharing, and do not permit the clear comparison of policies such as local and global memory management. This work has demonstrated the feasibility of constructing a simulation model that overcomes all of these problems.

We have made a number of comparative studies and shown the usefulness of several new virtual system management methods, but the primary motivation for this work has been the comparison of local and global memory management policies. Our general conclusion is that little significant difference between them has been observed in a representative system.

The reader, especially one who has the widely-held opinion that local policies are inherently better than global policies, should note that WS has been implemented and tuned using all known techniques, while GLOBAL has a load control which is not represented as the "best" in any sense. The WSEXACT algorithm was implemented precisely, requiring the modeling of uncommon hardware, although this did not result in a significant improvement over the approximate WS algorithms. Finally, this model does not incorporate the overhead of executing

each algorithm. Futher research in load control mechanisms and the effect of overhead is clearly worthwhile, and may show that a GLOBAL algorithm performs significantly better than WSEXACT. On the other hand, further refinement of the WS algorithm may improve its performance.

We have presented a new algorithm, WSCLOCK, that uses the commonly-available frame use bit and does not require special working-set measuring hardware to implement the WS policy or require the working set scan routines to estimate the working set. WSCLOCK's load control is simple and based on the standard WS load control. The performance of WSCLOCK is equivalent to WSEXACT performance.

6.4 Directions for Future Work

Our study is not intended to be a definitive comparison of local and global memory management policies. The policies are measured in a single model computer system configuration with a single model workload; a study of another configuration and workload, particularly of an interactive system, may show different results. The simulation model is easily changed to model new hardware configurations and workloads; it can also be adapted readily to model new policies and techniques.

The model of GLOBAL uses a simple load-control mechanism of our own invention. Given the encouraging nature of its performance – compared to WSEXACT – GLOBAL with an improved load control may be the best practical memory management policy.

The IRIM is well suited to model the local lookahead VMIN algorithm. The development and evaluation of a global lookahead algorithm is a practical next step in the study of optimal page replacement algorithms, and could be evaluated in a similar manner. A global version of OPT would replace the resident page with the largest task virtual time until the next reference, but this is not the optimal real-time, multi-program, variable-space algorithm.

This work assumed that task scheduling and load control are functionally partitioned; composite mechanisms are more complex and harder to analyze, but should result in increased performance.

This work provides both the tools and the incentive to make a more intensive investigation of virtual memory computer systems. We hope to see, and contribute to, an awakened interest in the practical methodology of scheduling, virtual memory management and load control.

Appendix A

Simulation Specification Language

General Specification Format

The description of the simulation language is somewhat informal, but a few rules are generally adhered to. Input parameters which are described exactly (i.e., terminals) are shown in SMALL CAPITALS; non-terminal symbols are in ordinary lower case; the symbol "⇒" means "is replaced by"; a vertical bar, "|" separates a series of alternative choices. Items in <brackets> are conceptual elements whose definitions should be clear from the context or narrative description. Commonly understood notions such as integer or string are not rigorously defined.

Each simulation run is specified by a sequential list of statements in a non-hierarchical free-form specification language. Statements are entered on a set of records (i.e., card images) and are delimited by blank characters or by commas. Each record can contain any number of statements.

The language contains numerous *simple statements* and a few *compound statements.* Simple statements have two basic forms. The simple:

keyword

selects a basic simulation option which has no parameters. For example, PRINT simply specifies the printing of the simulation output. The form:

keyword = value

specifies a single-valued simulation parameter, such as FRAMES = 200. The equal sign is optional and merely improves the readability of the input. A few parameters, such as SLICE and QUOTA, have two values following the keyword. Each simple statement must appear on a single record.

Compound statements contain a variable number of parameters delimited by a keyword and "END". They have the general form:

keyword value <simple statements> END

Compound statements may span several input records but each included simple statement must appear on a single record.

The input for each simulation has two sections: the simulation specification statements and the simulation output statements; these sections are delimited by the keyword START. Parameters following the START keyword on the same input record are processed after the initialization phase

and before the simulation execution; they control output during the simulation.

Several simulation runs may be included in a single input record set; all simulation variables are reset to default values between runs and no specifications are carried over from one run to the next. The input for a multi-simulation run has the following form:

<simulation input>
RESET
<simulation input>
RESET
<simulation input>
STOP

To simplify the specification of parameters that do not vary from one run to the next, they can be stored in a file and inserted in the main simulation input by the statement:

READFILE file-name {NOPRINT}

This facility extends only to a single level. READFILE, START, RESET, or STOP statements are illegal in a READFILE file. The optional NOPRINT parameter suspends printing of the file statements if the main input file is being printed. Any other parameters following a READFILE statement, on the same input record, are ignored.

Configuration

Processor

The definition of the processor is implicit. Processor speed is assumed to be fixed and equal to one memory reference per microsecond. Thus, in simulation specification statements, time can be specified in seconds or fractions of seconds as follows:

time ⇒ integer | integerM | integerC | integerS ,

where the appended letter denotes "M"illiseconds, "C"entiseconds, or "S"econds. A plain integer specifies time in references or microseconds.

Main Memory

The parameter:

FRAMES = integer,

specifies the size of main memory. Main memory required to support the operating system is in addition to this specification.

I/O Devices

Both task and paging I/O devices are defined in a common manner. In fact, the same device can be accessed by both tasks and the paging supervisor. The general form of the device specification is:

DEVICE device-name device-definition END

where the device-name is a 1-to-8 character string. The device access time distribution is defined by a set of parameters:

MIN = time
MAX = time
MEAN = time

whose use is described in the section on random variate generation in chapter 7. The SERIAL option specifies the I/O requests for the device must be processed sequentially; otherwise, all requests are processed concurrently. The LONG option indicates the scheduler should consider the device to have an indeterminate access time common to interactive devices; this option has no effect on the actual access time of the device. The parameter:

LIKE = device-name

causes the device definition of a previously defined device to be copied. Subsequent input parameters will override the copied parameters.

Tasks and System Load

The system load is determined by specifying a number of task prototypes. Each time the system decides to admit a new task, it chooses one prototype randomly and generates the task. If its desired to have more tasks of one type than another, the task prototypes should be specified repetitively until the desired ratios are achieved. This operation is made simpler by use of the LIKE parameter below.

The task generation process (i.e., choosing task prototypes when initiating a task) is controlled by an independent pseudorandom number generator whose seed is generated randomly. If is desired to produce identical task initiation sequences in simulation runs with different seed *generators*, the parameter:

LOADSEED integer

overrides the generated seed with a fixed seed and produces identical sequences of tasks when the seeds are identical.

Each task has an associated reference string. The format of each string must be the same in a single simulation run. The default format is the simple (but expensive) reduced reference string. If an alternative format is used, one of the following options should be specified:

SNAPSHOT – snapshot strings
REFIRI – IRIM model reference strings

Task Prototypes

Each task prototype is defined by the compound statement:

TASK task-name task-definition END

where the task-name is a 1-to-6 character string; when a task is initiated, a 2-digit number is appended to the task type name to create a unique task name. The file containing the task reference string is specified by the parameter:

FILE <file-name>

where file-name is operating system dependent (In IBM OS/VS, the file-name specifies the job control DDNAME.) If the task makes I/O requests, the definition must include the inter-request time distribution by a subset of the parameters:

MIN = time
MAX = time
MEAN = time

and a collection (one or more) of device selections, each of which have the form:

device-name = integer

where the value is the *relative* use of the device compared to the other devices. For example, the specification:

DISK1 3 DISK2 1 DISK3 1

will cause the task to access device DISK1 with probability .6 . If it is desired that all tasks of the same type have identical I/O request sequences, the the seed of the pseudorandom number

generation process is specified by the parameter:

SEED = integer

Normally, a task is terminated when its reference string is exhausted, but the parameter:

REFLIMIT = time

will terminate the task after a specified number of references. The working set parameter, q, is normally a system wide parameter, but it can be set for individual tasks witht the parameter:

THETA = time

Similarly, the task can have individual minimum and maximum working set size estimates by secifying:

WSMINWS = integer
WSMAXWS = integer

To copy all of the above task definition parameters from a previously defined task, specify:

LIKE = task-name

The copied parameters can be overriden only by subsequent parameters; previously supplied parameters are overriden by the LIKE statement..

Memory Management

General Parameters

There are a number of parameters for managing virtual memory which are independent of the specific replacement and associated load control strategies. Four basic options are:

CREATE
NOPAGOUT
NORECLAM
CLEANQ

The CREATE option specifies that the initial reference to each page does not require a page-in; it is assumed that a page has no previously defined contents and can be created by simple allocation of a page frame. The NOPAGOUT option eliminates the need to write dirty pages before replacing them; this option is used to emulate analytical models which have no provision for page cleaning. NORECLAM prevents page reclamation; once a page is determined to be replaceable (i.e., has left

the task working set), it *must* be replaced and subsequent references to the page will require a page-in operation. The CLEANQ option causes the paging supervisor to maintain a queue of pages which have been cleanedand to replace them before any others; the default option is to replace cleaned pages when they are found by the normal clock-search process.

Auxiliary memory devices are specified by the following compound statement:

PAGEFILE device-name END

Each paging device requires a separate statement; the use of a compound statement is for future enhancement to simulate multiple page transfers in a single I/O operation.

The order in which queued paging I/O requests are performed is specified by the parameter:

PAGQUEUE = FIFO | READS1ST | RUNFS1ST

where FIFO (the default) processes requests in the order the are received, READS1ST processes page-ins before page-outs, and RUNFS1ST process page-ins for running tasks before page-ins for loading tasks and all page-ins before page-outs.

When page-outs are queued after page-in requests, they may be delayed indefinitely, forcing the replacement of more useful, but clean, pages. The parameter:

FORCECNT = integer

forces a dirty page to be written when the paging supervisor has made the specified number of attempts to replace the page.

Working Set Page Replacement Algorithms

There are five variants of the working set replacement algorithm which can be selected:

WSEXACT
WSINTVL
WSFAULT
WSCLOCK
VMIN

The WSEXACT algorithm makes precise calculation of working sets; it causes the model to simulate a hardware mechanism which automatically records the time of last reference for each page. The WSINTVL and WSFAULT algorithms are approximations of WS which periodically scan the task pages and estimate the times of last reference by using the simulated hardware used bits. Both

WSINTVL and WSFAULT scan the pages at fixed intervals (see WSSCANS) and the WSFAULT algorithm scans the pages after each fault. The WSCLOCK algorithm approximates WS using the simulated used bits but only examines resident pages during the replacement clock scan. The VMIN algorithm is the lookahead analogue to WS; it requires use of the IRI model of task memory referencing.

The basic WS replacement parameter, θ, is specified by:

THETA = time

The maximum interval between scans in WSINTVL and WSFAULT is specified using the parameter:

WSSCANS = integer

which specifies the number of scans to be made during each interval of θ references. Thus, the interval between scans is $\lceil$THETA/WSSCANS$\rceil$.

Working Set Load Control

WS load control is implied when a WS replacement algorithm is selected. The parameter:

WSPOOL = integer

specifies the number of uncommitted frames (the free pool) *following* a task activation. The number of committed frames is the sum of active task's working set size estimates. These estimates can be bounded by the parameters:

WSMINWS = integer
WSMAXWS = integer

Global Page Replacement and Load Control

The only global page replacement algorithm supported is:

GLOBAL

Global load control also uses the WS parameter, THETA, but in a much more restricted fashion; its default is 5000 references.

There are three load control mechanisms for use with global replacement. First, there is a simple fixed multiprogramming level control, specified as:

MPL = integer

Second, load control based on page file busy-ness is specified as:

LDCPFUSE = real

where the real value is between 0 and 1 and is the desired utilization of the page files. Third, load control based on the rate at which the clock-scan pointer moves is by specifying:

LDCREVPS = real

where the real value is the number of pointer revolutions per second.

The heuristic feedback load controls (PAGFLUSE and REVSPSEC) maintain an exponentially weighted mean and variance of the control variable. Three parameters control the sensitivity of the feedback control:

LDCTIME = time
LDCALPHA = real
LDCCONF = 80 | 90 | 95 | 975 | 98 | 99 | 995 | 998 | 999

LDCTIME specifies the time interval between observations of the control variable; the default is .2 seconds (20C). LDCALPHA is the exponential weight given to all observations before the current observation; the default is .95 . As LDCALPHA approaches 1, all observations are given equal weight; as it approaches 0, the current observation becomes the exclusive estimate of the control variable. Using the variance of the control variable, the load control build a confidence interval about the mean; when the target value is outside the confidence interval, the appropraiate change to the multiprogramming level is made. LDCCONF specifies the level of confidence (default is 95 = 95% confidence); as LDCCONF approaches 1, the confidence interval becomes larger and changes to the multiprogramming level are less frequent.

The multiprogramming level can be bounded by the parameters:

MPLMIN = integer
MPLMAX = integer

Task Scheduling

The maximum number of tasks admitted to the system simulaneously is specified by:

MAXTASKS = integer

The default is 16 tasks. Task activation is performed primarily by load control, but an additional,

optional, limit on the number of loading tasks is specified by:

LOADTIME = time
LOADRMAX = integer

Loading tasks are defined to be those who have not executed more than LOADTIME references or made a task I/O request since activation. The default LOADTIME is 5000 references. LOADRMAX is the limit on the number of simultaneously active loading tasks. Note that LOADTIME is also used to define loading tasks for paging I/O queue ordering.

The task ready queue is a multi-level load balancing queue. Each level of the queue is defined a statement:

SLICE integer time

where the first value is the priority level (zero being the lowest priority). The second value is the time-slice received by a task at that level. In general, the time-slices should decrease with increasing priority. If only a single level (priority zero) is specified, the queue discipline reduces to round-robin. A task's priority is decreased whenever it consumes a time-slice. Priority is increased whenever the task waits for a sufficiently long time. The wait times are specified by the statement:

QUOTA integer time

for each priority level except zero. Thus, the statement QUOTA 2 1S indicates that a task is raised to priority 2 (at least) whenever it performs an I/O request that requires 1 second or more to perform.

Once activated, tasks are ordered in a round-robin queue; each task receives service for a quantum before being moved to the end of the active queue. The basic quantum is defined by:

QUANTUM = time

When memory overcommitment is detected, the task chosen to be demoted is specified by the parameter:

DEMOTE = LASTACTV | FAULTER | SMALLEST | LARGEST | LONGEST | RANDOM

where: LASTACTV is the last task activated; FAULTER is the task whose page fault signalled the overcommitment; SMALLEST is the task with the smallest memory allocation; LARGEST is the task with the largest memory allocation; LONGEST is the task with the most amount of processing time remaining in the current time-slice; and RANDOM is a randomly selected task.

General Simulation Control

Stochastic Processes

All stochastic processes have a independent pseudo-random number generator with independently generated pseudo-random seeds. The seed generator has a default seed of 987564321; this can be changed to provide an entirely different set of seeds with the statement:

SEED integer

Some stochastic processes, such as load generation, can have seeds specified in the simulation input. The simulator always generates a seed for each process, even one is already specified, in the same order; given the same seed, the unspecified processes will be identical from one run to the next.

Simulation Measurement

When a simulation run begins, the initial phase may not be re[presentative of steady-state behavior and can be deleted from the simulation measurement. The parameter:

DELETIME = time

causes all measurement to be reset to initial values when the specified amount of time has been simulated. Alternatively, the parameter:

DELETASK = integer

resets the measurement after the specified number of tasks have terminated. If both controls are specified, measurement starts after *both* the specified time and number of task terminations have been simulated. The run length is determined by the parameters:

MEASTIME = time

MEASTASK = integer

which terminates the run after the specified time or number of task completions after measurement has been started; if both are specified, the run is terminated when the *first* limit is reached. The run can also be terminated if it is requiring an excessive amount of computer time by specifying:

MAXCOST = time

The basic measurement of system performance in processor utilization. In addition to producing

the mean utilization, the system performance can be sampled periodically to derive the variance and to calculate an exponentially weighted mean and variance of utilization. The sampling interval is specified by:

SAMPLE time

The variance is not reported directly, but is used to develop a normal confidence interval about the mean utilization. The size of the interval depends on the level of confidence specified by the statement:

CONFDNCE = 80 | 90 | 95 | 975 | 98 | 99 | 995 | 998 | 999

The exponentially weighted mean has the parameter:

ESTALPHA = real

which is the weight applied to all past samples; as ESTALPHA approaches 1, the weighted estimate approaches the unweighted estimate.

Execution Tracing and Debugging

A detailed trace of simulation operation is produced by specifying the parameter:

TRACE

which will produce a record of every event-list insertion, processor idle period, task activation/deactivation, time-slice or quantum allocation, global load control operation, task execution start, page fault, page replacement, working set scan, task and paging I/O request, and interactive response time. To reduce the size of the trace by eliminating unwanted records, any collection of the following parameters can be specified instead of TRACE:

PAGTRACE – traces paging related operations
EXETRACE – traces execution related operations
LDCTRACE – traces load control operations
RESTRACE – trace interactive response time

Since the event which is to be analyzed by tracing may not occur until the simulation has run for a time the parameter:

TRACEON time

delays generating the trace until the specified time; similarly, the parameter:

TRACEOFF time

turns the trace off at the specified time.

Debugging can be performed at a low level, using an interactive program debugging capability. To facilitate running the simulation until a time that a bug is expected to appear the following statements are provided:

PAUSE time
PAUSEVRY time

When run interactively, the simulation will halt to permit debugging at the time specified in the first statement. If the second statement is also supplied, the simulation will halt whenever the second quantity of time has been simulated.

Output Control

The output of the simulation is controlled very simply by the use of the statements:

PRINT
NOPRINT

which are interspersed in the simulation input. During the simulation parameter input, a PRINT/NOPRINT controls printing of the input parameters themselves. On the START statement, preceding the START parameter, a PRINT/NOPRINT controls the printing of the pre-simulation report. Following the START parameter, on the same input statement, a PRINT/NOPRINT controls output during the simulation run (i.e., tracing) and the post-simulation report. Following the START statement, a PRINT/NOPRINT controls the output of OUTPUT statements before the next RESET statement.

When the simulator is run interactively, a TYPE/NOTYPE statement is analogous to the PRINT/NOPRINT statement, but controls output to the terminal. The printing status defaults to PRINT/NOTYPE each time the simultor is re-initialized. The current setting remains in effect for each output phase.

A page title can be specified to identify the output of each simulation run:

TITLE 'any text'

The date and time of the run is appended to the title.

There are two types of traces that are not part of the trace facility described above since they are

generally considered as a normal part of the simulator measurement report. The first type produces a detailed description of each task executed when it terminates. This trace is selected by specifying:

TSKTRACE

The second type of trace can be invoked periodically to produce a time-series of processor utilization based on the mean sampling process described above; this trace is selected by specifying:

SAMTRACE

Following the simulation run, various measurements can be output in a form which can be stored and used for later analysis. Statements of the form:

OUTPUT output-list END

appear after the START statement and before the RESET or STOP statement. The elements of the output list are either the variable names listed below or any other string which is reproduced verbatim. The string can be enclosed in quotes or not; a misspelled variable name will simply appear in the output as a string.

Output Variables

COST	–	computer time for simulation run
DATE	–	date of simulation run
SIMTIME	–	measured simulation time
IDLETIME	–	processor idle time
UTIL	–	processor utilization during measured period
CONFDNCE	–	width of utilization confidence interval
EWUTIL	–	exponentially weighted processor utilization
EWCONF	–	exponentially weighted confidence interval
ACTIVEQ	–	mean tasks in active queue
LOADQ	–	mean loading tasks in active queue
PROMOTES	–	count of task activations
DEMOTES	–	count of load control deactivations
MAXPRIO	–	maximum priority in multi-level queue
QUANTUM	–	processor quantum
LOADTIME	–	virtual time limit definition of loading task
LOADRMAX	–	maximum number of loading tasks permitted

FRAMES – system page frames
PAGDEVS – count of page file devices
PAGREADS – count of page-in operations
PAGWRITS – count of page-out operations
PAGFORCE – count of forced page-out operations
FORCETIM – idle time due to forced page-outs
PFLRUNF – page file utiliztaion for running task faults
PFLLOAD – page file utilization for loading task faults
PFLWRIT – page file utilization for page-outs
PAGQ – mean length of paging I/O queue

REPLALGM – page replacement algorithm
REPSCANS – number of page replacement clock-scans
CLSMEAN – mean number of pages scanned for each replacement
LOADFLTS – count of page faults by loading tasks
RUNFLTS – count of page faults by running tasks
FRAMWAIT – count of replacement algorithm failures

LDCTYPE – load control type (GLOBAL)
LDCVALUE – load control nominal value (GLOBAL)
LDCTIME – load control sampling interval (GLOBAL)
LDCONFID – load control confidence interval (GLOBAL)
LDCALPHA – load control exponential weight (GLOBAL)
MPLMAX – upper bound on multiprogramming level (GLOBAL)
MPLMIN – lower bound on multiprogramming level (GLOBAL)
MPLMEAN – mean value of load control MPL (GLOBAL)
MPLCHNGS – count of changes to load control MPL (GLOBAL)

THETA – working set parameter (WS)
WSSCANS – working set scans in THETA interval (WS)
WSMINWS – minimum estimated task working set (WS)
WSMAXWS – maximum estimated task working set (WS)
WSFSCAN – count of working set scans at faults (WS)
WSFISCAN – count of working set scans at intervals (WS)
WSPOOL – size or uncommitted working set pool (WS)

Appendix B

IS THE WORKING SET POLICY NEARLY OPTIMAL?

Richard W. Carr

October 1980

Computer Science Department, Stanford University
and
Stanford Linear Accelerator Center

This work was supported by the Department of Energy under contract number DE-AC03-76SF00515.

Introduction

The capacity of a virtual memory computer system is highly dependent on the methods used to manage the virtual memory hierarchy. In particular, the *interaction* between processor scheduling and memory management is among the most important and least understood factors affecting capacity. Recently, Denning has claimed that the working set (WS) replacement policy and dispatcher solves the problem of virtual memory management [DENN80]. In that article, he states:

> Simon compared the optimum throughput from the tuned WS policy to the optimum from the VMIN policy. He found that VMIN improved the optimum throughput from 5 percent to 30 percent depending on the workload, the average improvement being about 10 percent. ... This is the most compelling evidence available that no one is likely to find a policy that improves significantly over the performance of the tuned WS policy.

We have read Simon's thesis [SIMO79], and while it may be the best evidence for the superiority of WS, we have been forced to conclude that the evidence is far from conclusive. Our examination discloses serious flaws, not only in the modeling process, but also in the interpretation of the results.

This paper should not be construed as a general criticism of Simon's thesis. Of 160 pages, the comparison of WS and VMIN occupied only three pages of text, two graphs, and one figure (pp. 65-70). No mention of the results appeared in the summary chapter. Simon was mainly concerned with techniques of model construction and not in comparing page replacement algorithms.

Summary

Prieve and Fabry describe a lookahead page replacement algorithm, VMIN, which is characterized as optimal [PRIE76]. Simon develops an analytic queuing network model of a virtual memory computer system in which both WS and VMIN replacement can be modeled. The model incorporates WS and VMIN lifetime curves, phase-transition (P-T) program behavior models, and a multi-class central server queuing network model.

In this paper, we claim:

- The lifetime curves used in the model are badly skewed, particularly in the regions critical to the modeling results.

- The P-T model is not used in a logically consistent manner.

- The queuing network model has fundamental limitations which prevent it from imitating the behavior of real virtual systems.

- The model solutions depend on parameters which are unrealistic and minimize the difference between all page replacement algorithms.

- The characterization of VMIN as *the* optimal paging algorithm is incorrect.

Lifetime Curves

Simon's model incorporates Kahn's P-T model of program behavior [KAHN76]. Kahn measures 7 programs (PAS, RUM, SNO, PAG, ASP, INT, and LIF) to calculate lifetime curves, *L(m)*, and phase-transition parameters. The lifetime values, which are reproduced in Table 1, are collected by simulating the WS or VMIN replacement algorithm on a program trace, as described by Denning in [DENN75b].

Maximum lifetimes range between 8099 and 22727, mostly nearer the lower figure. These values are lower, by a factor of 5 to 20, than any we have ever observed in our own measurements of typical programs. When lifetimes are maximized no page is ever replaced; all faults are caused by initial page references which are unavoidable under a demand paging policy. Apparently, the programs were not measured long enough to reduce the effect of the startup transient, which, especially for large *m*, skews the curves.

Lifetime curves are commonly used to evaluate replacement algorithms [BARD73, PRIE76, DENN75a]. Kahn's measurements of WS and VMIN lifetimes confirm the findings of Prieve and Fabry: for common values of *m*, VMIN lifetimes are 2 to 3 times longer than WS lifetimes. At large values of *m*, however, the WS and VMIN curves converge because they are dominated by the startup transient.

Table 1. (Source: [SIMO79])

WS Lifetime Measurements

m (pages)	PAS	RUM	SNO	PAG	ASP	INT	LIF
33.3	102	169	158	16949	45	599	1695
35.7	109	177	167	16949	48	740	1939
38.5	117	203	179	17241	53	1052	2188
41.7	127	244	205	17241	56	1160	2320
45.5	144	311	227	17857	66	1287	2494
50.0	162	475	243	18519	79	1443	2660
55.6	182	806	273	18868	156	1567	2890
62.5	214	1297	341	"	1157	1946	2950
71.4	300	2174	493	"	9804	3268	3274
83.3	532	2976	854	"	22222	6135	3876
100.0	1238	4831	2577	"	22722	7519	5495
125.0	3049	8850	4785	"	"	9524	10417
166.7	5376	15385	7752	"	"	10753	13158
250.0	7813	20000	9901	"	"	11364	17544
500.0	8099	"	11111	"	"	"	"

VMIN Lifetime Measurements

m (pages)	PAS	RUM	SNO	PAG	ASP	INT	LIF
33.3	215	448	312	19231	105	1387	2950
35.7	234	519	344	"	116	1590	3125
38.5	257	630	380	"	136	1792	3401
41.7	287	783	426	"	168	2049	3597
45.5	329	1027	498	"	227	2433	4016
50.0	386	1422	587	"	345	3021	4444
55.6	475	2114	735	"	760	3953	4975
62.5	626	3049	975	"	3021	5587	5495
71.4	927	4348	1460	"	20000	7752	7407
83.3	1558	6579	2817	"	22727	10000	9901
100.0	2976	10870	5181	"	"	10753	13158
125.0	5376	15385	8264	"	"	11364	17544
166.7	7813	16667	9901	"	"	"	"
250.0	"	20000	11111	"	"	"	"
500.0	8099	"	"	"	"	"	"

When *m* equals or exceeds the programs total size, WS and VMIN lifetimes converge to the same maximum. Thus, we can infer the program size of the 7 programs:

Table 2.

Program Size

Program pages	Programs
50-55	PAG
83-100	ASP
167-250	RUM, INT, LIF
250-500	PAS, SNO

In our own experiments with program behavior, we have observed an average program size of about 50,000 to 80,000 words, or 50 to 80 1k-word (4k-byte) pages. Either the measured programs were very large or the measurements used a small page size, apparently the latter. A small page size increases the number of pages and intensifies P-T behavior, but it aggravates the effect of the startup transient.

The Phase-Transition Model

Kahn's phase-transition model is a 2-state semi-Markov chain with four model parameters:

- L_{phase} – mean lifetime in phases
- $L_{transition}$ – mean lifetime in transitions
- F_{phase} – mean faults in a phase
- $F_{transition}$ – mean faults in a transition

L_{phase} and $L_{transition}$ determine the holding time in a state, while the branching probabilities are derived from F_{phase} and $F_{transition}$. Lifetimes are assumed to be i.i.d exponentially distributed and faults are assumed to be geometrically distributed. Kahn devised a filter which distinguished transition faults from phase faults and calculated values for $L_{transition}$, F_{phase}, and $F_{transition}$. These parameters and the overall $L(m)$ determine L_{phase} as follows: the fraction of faults in each state are:

$$S_{phase} = \frac{F_{phase}}{F_{phase} + F_{transition}} \qquad S_{transition} = \frac{F_{transition}}{F_{phase} + F_{transition}} = 1 - S_{phase} .$$

$L(m)$ is decomposed into phase and transition lifetimes,

$$L(m) = S_{phase}\, L_{phase} + S_{transition}\, L_{transition},$$

so,

$$L_{phase} = (L(m) - S_{transition}\, L_{transition}) \,/\, S_{phase} .$$

The P-T parameters measured by Kahn are reproduced in Table 3. Denning reports that 40% to 50% of WS faults occur in transitions, but five of the seven programs used in the model have 90% or more of their faults in phases. In these cases the P-T model has marginal value and can be replaced with a simpler lifetime model.

Table 3. (Source: [SIMO79])

	Phase-Transition Parameters				
Program	*Ftransition*	*Fphase*	*Ltransition*	*Stransition*	*Sphase*
PAS	15.0	170.9	21.5	.08	.92
RUM	11.0	231.9	29.0	.04	.96
SNO	11.5	33.6	45.8	.25	.75
PAG	12.5	278.5	35.5	.04	.96
ASM	14.5	131.0	17.1	.10	.90
INT	20.7	317.0	12.6	.06	.94
LIF	8.0	7.8	36.3	.51	.49

A fundamental problem with the P-T model is the assumption that $L_{transition}$, F_{phase}, and $F_{transition}$ are independent of both m and the page replacement algorithm. Denning remarks that "the same phases were observed by the WS policy over wide ranges of its control parameter (θ)" [DENN80], but m and $L(m)$ also change slowly over wide ranges of θ. When $L(m)$ changes significantly, it is difficult to believe that the phases and transitions remain the same. WS and VMIN have identical phases and transitions at equal values of θ, but at significantly different values of m. Conversely, at equal values of m, WS and VMIN can not be expected to have the same P-T parameters.

The Phase-Transition-Swapping Model

Simon incorporated task swapping into the P-T model to compare various loading policies in an analytic queuing network model. The P-T model is expanded to a 3-state Phase-Transition-Swapping (P-T-S) model, in which a program is swapped out after *each* file request and every t units of virtual time. P-T-S model parameters are derived from the P-T model with special rules to characterize the difference between pre-loading and demand paging after each swapout. When WS and VMIN are compared, the model uses only demand paging rules.

The Queuing Network Model

The queuing network has a source/sink and three servers: CPU, DISK, and DRUM. By characterizing each state of the P-T-S model as a separate job class, the model is solved using the Baskett, Chandy, Muntz and Palacios theorem [BASK75]. The queuing network model has two fundamental limitations: all programs in the model are identical and the model fails to represent dynamic space-sharing adequately.

The use of identical programs may simplify the model, but it obscures the effect of tuning WS with a single value of θ. In each model solution, peak WS throughput will be observed when the optimum θ for each program is located. If different programs are processed with a single θ, some of the programs will be detuned and the composite WS peak throughput will be less.

The model assumes that N identical programs are sharing M pages of memory. Unlike a real working set dispatcher, the model has a fixed multiprogramming level and presumes that the sum

of resident sets, not just their means, is always equal to M. The penalty of variable-space replacement is omitted in the model: WS replacement forces the removal, and later reloading, of a program when the sum of working sets exceeds M. This not only adds to system overhead but it also reduces $L(m)$ for the programs that are removed. To ignore this effect creates a significant bias favoring WS.

Model Solutions

The model solutions used the following system parameters:

M – 500 pages

Drum latency – 5000 references

Drum transfer – 1000 references

Disk service – 30000 references

These parameters are unrealistic in today's computer systems. For simplicity, Simon assumed a processing rate of 10^6 *references* per second, or about .5 MIPS. To couple such a slow processor with a very fast secondary storage (5 msec. latency) would be economic folly. The parameters reduce the cost of a page fault to an unrealistic level and conceal differences between paging algorithms. The parameters controlling the frequency of swapouts and file requests can have a similar biasing effect but were not described.

Simon presents solutions of models which used the programs SNO and ASP. The results of the other models are summarized:

> The largest difference [between WS and VMIN] was observed for program SNO (about 14%) and the smallest for programs ASP and PAG (less than 5% for ASP; the curves for PAG differed even less). The other programs showed differences of about 10%.

Tables 4 and 5 reproduce the model solutions; the values were taken from graphs and may have small errors. They give the WS and VMIN throughput for each multiprogramming level, N. The mean resident set size, m, is $500/N$. WS and VMIN lifetimes, which are the only difference between the WS and VMIN models, are reproduced from Table 1. Peak throughput values are marked with an asterisk.

Table 4. (Source: [SIMO79])
Model Solution: Program = ASP, $M = 500$ pages

		Throughput		$L(m)$	
N	m	WS	VMIN	WS	VMIN
1	500	45%	45%	22727	22727
2	250	75%	75%	22727	22727
3	167	90%	90%	22727	22727
4	125	95%	95%	22727	22727
5	100	98%	98%	22727	22727
6	83	99%*	99%	22222*	22727
7	71	95%	100%*	9804	20000*

Of the two models, ASP is the simplest to analyze. Although the difference is described as 5%, the actual difference is only 1%. The reason is clear: when ASP has a mean resident set size of 83 pages or more, the entire program is loaded, paging is minimized, and the choice of paging algorithm has a negligible effect. With $M = 500$, 6 full copies of ASP can be loaded simultaneously to achieve maximum throughput. This model merely shows that an abundance of main memory eliminates the need to replace pages and makes the choice of replacement algorithm irrelevant.

Table 5. (Source: [SIMO79])
Model Solution: Program = SNO, $M = 500$ pages

		Throughput		$L(m)$	
N	m	WS	VMIN	WS	VMIN
1	500	40%	40%	11111	11111
2	250	64%	66%	9901	11111
3	167	68%*	78%	7752*	9901
4	125	62%	82%*	4785	8264*
5	100	40%	65%	2577	5181
6	83	15%	45%	852	2817
7	71	10%	25%	493	1460

In the model with program SNO, peak WS throughput (68%) occurs with $N = 3$ and $L(m) = 7752$. Peak VMIN throughput (82%) occurrs with $N = 4$ and $L(m) = 8264$. VMIN processes 1/3 *more* programs with *less* paging due to normal faults (i.e., not caused by swapouts). It is important to remember that the fundamental objective of paging is to lower the mean resident set size and *raise the multiprogramming level.* VMIN is considerably more effective than WS in doing this. Other system parameters, such as the swapping and file request rates, limits the improvement to 14%. The assumption that a program is swapped out at every file request would be distastrous in a real system. The model has a single disk server which may be a bottleneck when N is increased

from 3 to 4.

These results were obtained with skewed lifetime curves minimizes the difference between WS and VMIN because much of the paging is unrelated to the replacement algorithm. Models comparing WS and VMIN should also model fixed-space LRU replacement to show the model's sensitivity to significant changes in the replacement algorithm.

The Optimality of VMIN

Even if WS is shown to be close to VMIN, it does not follow that WS is near-optimal. First, Prieve and Fabry's proof of VMIN's optimality assumes that page faults are uniformly distributed in virtual time. This is not consistent either with the P-T model, which has a high fault rate during the short-lived transitions, or with real programs. Second, VMIN is a *local* replacement algorithm and attempts to find a local optimum for each program. It has not been shown that optimal local replacement is as good or better than optimal global replacement.

Since it is difficult to analyze replacement algorithms, we are usually content with circumstantial evidence that one algorithm is better than another. However, a claim that an algorithm is *optimal* is a fundamental assertion which requires a rigorous argument.

Conclusions

Denning's contention that no one is likely to find a policy that improves significantly over WS is not supported by Simon's thesis. Numerous improvements are required to make a conclusive comparison of WS and VMIN with a queuing network model of the type described above. Even if this is done, the question of VMIN's optimality must still be addressed.

Bibliography

[BARD73] Y. Bard,
"Characterization of Program Paging in a Time-sharing Environment,"
IBM J. Res. Develop., Vol 17(5), p 387-393, September 1973

[BASK75] F. Baskett, K. Chandy, R. Muntz, and F. Palacios,
"Open, Closed, and Mixed Networks of Queues with Different Classes of Customers,"
JACM, Vol 22(10), p 248-260, October 1975

[DENN75a] P.J. Denning and K.C. Kahn,
"A Study of Program Locality and Lifetime Functions,"
Proc. 5th Symp. Operating Systems Principles, ACM SIGOPS, p 207-216, Nov 1975

[DENN75b] P.J. Denning,

"The Computation and Use of Optimal Paging Curves,"
Report CSD TR 154, Computer Science Dept, Purdue U., 1975

[DENN80] P.J. Denning,
"Working Sets Past and Present,"
IEEE Trans. on Software Engineering, Vol SE-6(1), p 64-84, January 1980

[KAHN76] K. C. Kahn,
"Program Behavior and Load Dependent System Performance,"
Ph.D. dissertation, Dept. Comp. Sci., Purdue U., 1976

[PRIE76] B. Prieve and R.S. Fabry,
"VMIN - An Optimal Variable-Space Page Replacement Algorithm,"
CACM, Vol 19(5), p 295-297, May 1976

[SIMO79] R. Simon,
"The Modeling of Virtual Memory Systems,"
Ph.D. dissertation, Dept. Comp. Sci., Purdue U., 1979

Bibliography

Cited references are followed by the chapter number(s), in braces, in which the citation occurs. The remaining references all bear some relevance to this work, but are not specifically cited.

[ADIR71] I. ADIRI,
"A Dynamic Time-Sharing Priority Queue,"
J. ACM **18**, 4 (Oct. 1971), 603-610 {2}

[AHO71] A.V. AHO; P.J. DENNING; and J.D. ULLMAN,
"Principles of Optimal Page Replacement,"
J. ACM **18**, 1 (Jan. 1971), 80-93 {3}

[ALEX70] M.T. ALEXANDER,
Time-sharing Supervisor Programs.
Univ. of Michigan Computing Center, Ann Arbor, Mich., 1970

[ALDE71] A. ALDERSON; W.C. LYNCH; and B. RANDELL,
"Thrashing in a Multiprogrammed System,"
in *Operating Systems Techniques*, Academic Press, London, 1972 {3}

[ANDE74] H.A. ANDERSON; M. REIRER; and G.L. GALATI,
Techniques for Tuning the Performance of a Virtual Storage System,
RC-4938, IBM Research, Yorktown Heights, N.Y., July 1974

[ARVI73] ARVIND,
Models for the Comparison of Memory Management Algorithms,
Ph.D. Dissertation, University of Minnesota, 1973

[BADE75] M. BADEL; E. GELENBE; J. LEROUDIER; and D. POTTIER,
"Adaptive Optimization of a Time-Sharing System's Performance,"
Proc. IEEE 63, 6 (June 1975), 958-965 {2, 3}

[BARD73a] Y. BARD,
"Characterization of Program Paging in a Time-sharing Environment,"
IBM J. Res. Dev. **17**, 5 (Sept. 1973), 387-393 {3}

[BARD73b] Y. BARD,
"Experimental Evaluation of System Performance,"
IBM Syst. J. **12**, (1973), 302-314

[BARD75] Y. BARD,
"Application of the Page Survival Index to Virtual Memory System Performance"
IBM J. Res. Dev. **19**, 3 (May 1975), 212-220

[BARD78] Y. BARD,
"An Analytic Model of the VM/370 System"
IBM J. Res. Dev. **19**, 3 (May 1975), 212-220

[BASK75] F. BASKETT; K. M. CHANDY; R.R. MUNTZ; and F.G. PALACIOS-GOMEZ,
"Open, Closed, and Mixed Networks of Queues with Different Classes of Customers,"
J. ACM 22, 2 (April 1975), 248-260 {3}

[BELA66] L.A. BELADY,
"A Study of Replacement Algorithms for a Virtual Storage Computer,"
IBM Syst. J. 5, 2 (1966), 78-101

[BELA69] L.A. BELADY; and C.J. KUENER,
"Dynamic Space Sharing in Computer Systems,"
Commun. ACM 12, 5 (May 1969), 282-288 {3}

[BELA73] L.A. BELADY; and R.F. TSAO,
"Memory Allocation and Program Behavior under Multiprogramming,"
Proc. Computer Science and Statistics: 7th Ann. Symp Interface, pp. 72-78, 1973

[BENS72] A. BENSOSSAN; C.T. CLINGEN; and R.C. DALEY,
"The Multics Virtual Memory,"
Commun. ACM 15, 5 (May 1972), 308-318

[BOBR72] D.G. BOBROW; J.D. BURCHFIEL; D.L. MURPHY; and R.S. TOMLINSON
"TENEX, a Paged Time Sharing System for the PDP-10,"
Commun. ACM 15, 3 (March 1972), 135-172

[BOGO75] R.P. BOGOTT; and M.A. FRANKLIN,
"Evaluation of Markov Program Models in Virtual Memory Systems,"
Software – Practice and Experience 5, (1975), 337-346

[BOKS73] C. BOKSENBAUM; S. GREENBERG; C. TILLMAN,
Simulation of CP-67,
IBM Cambridge Scientific Center Tech. Report 2093, Cambridge, Mass., 1973 {3}

[BOOT71] W.P. BOOTE; S.R. CLARK; and T.A. ROURKE,
"Simulation of a Paging Computer System,"
The Computer Journal 15, 1 (Feb. 1971), 51-57

[BRYA75] P. BRYANT,
"Predicting Working Set Sizes,"
IBM J. Res. Dev. 19, 3 (May 1975), 221-229

[BUZE71] J. BUZEN,
Queuing Network Models of Multiprogramming,
Ph.D. Dissertation, Harvard Univ., Cambridge, Mass., 1971

[CANO80] R.D. CANON; D.H.FRITZ; J.H. HOWARD; T.D. HOWELL; M.F. MITOMA; and J. RODRIGUEZ-ROSELL,
"A Virtual Machine Emulator for Performance Evaluation,"
Commun. ACM 23, 2 (Feb. 1980), 71-80 {3,4}

[CARR81] R.W. CARR; and J.L. HENNESSY,
"WSCLOCK – A Simple and Efficient Algorithm for Virtual Memory Management,"
Proc. 8th Symp. Operating Systems Principles, ACM SIGOPS, Dec. 1981 {4}

[CHAM72] D.D. CHAMBERLAIN,
A Scheduling Mechanism for Interactive Systems with Virtual Memory,
RC-3911, IBM Research, Yorktown Heights, N.Y., 1972

[CHAM73] D.D. CHAMBERLAIN; S.H. FULLER; and L.Y. LIU,
"An Analysis of Page Allocation Strategies for Multiprogramming Systems with Virtual Memory,"
IBM J. Res. Dev. 17 (1973), 404-412 {3}

[CHAN77] S.T. CHANSON; and C.D. BISHOP,
"A Simulation Study of Adaptive Scheduling Policies in Interactive Computer Systems,"
Proc. Winter Simulation Conference, 1977 {2, 3}

[COFF72] E.G. COFFMAN; and T.A. RYAN,
"A Study of Storage Partitioning using a Mathematical Model of Locality,"
Commun. ACM 15, 3 (March 1972), 185-190 {3}

[COFF73] E.G. COFFMAN; and P.J. DENNING,
Operating Systems Theory,
Prentice-Hall, Englewood Cliffs, N.J., 1973 {2}

[CORB62] F.J. CORBATO,
"An Experimental Time Sharing System,"
Proc. 1962 AFIPS Fall Jt. Computer Conf., Vol 21, Spartan Books, Washington, D.C., pp. 335-343 {2}

[CORB68] F.J. CORBATO,
A Paging Experiment with the Multics System,
MIT Project MAC Report MAC-M-384, May 1968 {2}

[DALE68] R.C. DALEY; and J.B. DENNIS,
"Virtual Memory, Processes, and Sharing in Multics,"
Commun. ACM 11, 5 (May 1968), 306-312

[DEC78] DIGITAL EQUIPMENT CORPORATION,
VAX 11/780 Technical Summary,
Maynard Mass., 1978 {1}

[DENN68a] P.J. DENNING,
"Thrashing : Its Causes and Prevention,"
Proc. 1968 AFIPS Spring Jt..Computer Conf., Vol 33, Spartan Books, Washington, D.C., pp. 915-922

[DENN68b] P.J. DENNING,
"The Working Set Model for Program Behavior,"
Commun. ACM 11, 5 (May 1968), 323-333

[DENN70] P.J. DENNING,
"Virtual Memory,"
Comput. Surv. 2, 3 (Sept. 1970), 153-189 {1,2}

[DENN72] P.J. DENNING; and S.C. SCHWARTZ,
"Properties of the Working Set Model,"
Commun. ACM 15, 3 (March 1972), 191-198

[DENN72a] P.J. DENNING,
"On modeling Program Behavior,"
Proc. 1972 AFIPS Spring Jt. Computer Conf., Vol 40, Spartan Books, Washington, D.C., pp. 937-944

[DENN74] P.J. DENNING,
"Comment on a Linear Paging Model,"
2nd Annual Sigmetrics Symposium on Measurement and Evaluation, 1974, pp. 34-48 {3}

[DENN75a] P.J. DENNING; and K.C. KAHN,
"A Study of Program Locality and Lifetime Functions,"
Proc. 5th Symp. Operating Systems Principles, ACM SIGOPS, pp. 207-216, Nov. 1975 {3}

[DENN75b] P.J. DENNING,
The Computation and Use of Optimal Paging Curves,
Report CSD-TR 154, Computer Science Dept, Purdue U., 1975

[DENN75c] P.J. DENNING; and G.S. GRAHAM,
"Multiprogrammed Memory Management,"
Proc. IEEE 63, 6 (June 1975), 924-939 {2}

[DENN76] P.J. DENNING; et al.,
"Optimal Multiprogramming,"
Acta Informatica 7 (1976), 197-216 {2, 3}

[DENN78] P.J. DENNING,
"Optimal Multiprogrammed Memory Management,"
in *Advances in Programming Methodology III*, Prentice-Hall, 1978

[DENN80] P.J. DENNING,
"Working Sets Past and Present,"
IEEE Trans. on Software Engineering 6, 1 (Jan. 1980), 64-84 {2, 3, 4}

[EAST76] M.C. EASTON; and P.A. FRANASZEK,
Use Bit Scanning in Replacement Decisions,
RC-6192, IBM Research, Yorktown Heights, N.Y., Sept. 1976 {2}

[FISH73] G.S. FISHMAN,
Concepts and Methods in Discrete Event Digital Simulation,
Wiley, New York, 1973 {4}

[FISH76] G.S. FISHMAN,
"Some Tests of the Simscript and Simpl/1 Pseudorandom Number Generators,"
Simuletter 8, 1 (Oct. 1976), 79-84 {4}

[FOGE74] M. FOGEL,
"The VMOS Paging Algorithm,"
Operating Systems Review 8, 1 (Jan. 1974), 8-17

[FRAN74] P.A. FRANASZEK; and T.J. WAGNER,
"Some Distribution-free Aspects of Paging Algorithm Performance,"
J. ACM **21**, 1 (Jan. 1974), 31-39

[GEHE76] E.F. GEHRINGER; and H.D. SCHWETMAN,
"Run-time Characteristics of a Simulation Model,"
Proc. Symposium on the Simulation of Computer Systems, 1976

[GELE73] E. GELENBE,
"A Unified Approach to the Evaluation of a Class of Replacement Algorithms,"
IEEE Trans. on Electronic Computers **22**, 6 (June 1973), 611-618

[GHAN74] M.Z. GHANEM,
On the Optimal Memory Allocation Problem,
RC-4443, IBM Research, Yorktown Heights, N.Y., 1974

[GHAN75] M.Z. GHANEM,
"Dynamic Partitioning of the Main Memory Using the Working Set Concept,"
IBM J. Res. Dev. **19**, 5 (Sept. 1975), 445-450

[GHAN75a] M.Z. GHANEM,
"Study of Memory Partitioning for Multiprogramming Systems with Virtual Memory,"
IBM J. Res. Dev. **19**, 5 (Sept. 1975), 451-457

[GHAN75b] M.Z. GHANEM,
Experimental Study of the Behavior of Programs,
RC-5427, IBM Research, Yorktown Heights, N.Y., 1975

[GOMA79] H. GOMAA,
"A Simulation Based Model of a Virtual Storage System,"
Proc. 12th Annual Simulation Symposium, 1979 {3}

[GRAH74] G.S. GRAHAM; and P.J. DENNING,
Multiprogramming and Program Behavior,
Report CSD-TR 122, Computer Science Dept, Purdue U., 1974

[GRAH76] G.S. GRAHAM,
A Study of Program and Memory Policy Behavior
Ph.D. disseration, Dept. Comp. Sci., Purdue U., 1976 {3}

[GRIT77] D.H. GRIT,
"Global LRU Page Replacement in a Multiprogrammed Environment,"
Proc. 1977 Sigmetrics/CMG VIII Conf., pp. 265-270, Dec. 1977 {3}

[HABE76] A.N. HABERMAN,
Introduction to Operating System Design,
Science Research Associates, Chicago, 1976 {2}

[HATF72a] D.J. HATFIELD,
"Experiments on Page Size, Program Access Patterns, and Virtual Memory Performance,"
IBM J. Res. Dev. **16**, 1 (Jan. 1972), 58-66

[HEND74] G. HENDERSON; and J. RODRIGUEZ-ROSELL
"The Optimal Choice of Window Sizes for Working Set Dispatching,"
2nd Annual Sigmetrics Symposium on Measurement and Evaluation, 1974, 10-33

[HORO66] L.P. HOROWOTZ; R.M. KARP; R.E. MILLER; and S. WINOGRAD,
"Index Register Allocation,"
J. ACM **13**, (Jan. 1966), 43-61 {4}

[INGA74] G. INGARGIOLA; and J.F. KOSH,
"Finding Optimal Demand Paging Algorithms,"
J. ACM **21**, 1 (Jan. 1974), 40-53

[JOHN76] R. JOHNSON; and T. JOHNSTON,
PROGLOOK Users Guide,
User Note 33, SLAC Computing Services, Stanford Linear Accelerator Center, 1976 {5}

[KAHN76] K.C. KAHN,
Program Behavior and Load Dependent System Performance,
Ph.D. disseration, Dept. Comp. Sci., Purdue U., 1976 {3}

[KAIN75] R.Y. KAIN,
"How to Evaluate Page Replacement Algorithms,"
Proc. 5th Symp. Operating Systems Principles, ACM SIGOPS, pp. 1-5, Nov. 1975

[KLEI75] J.P. KLEIJNEN,
Statistical Techniques in Simulation,
Dekker, New York, 1975 {4}

[KNUT69] D.E. KNUTH,
The Art or Computer Programming: Seminumerical Algorithms, Vol. 2
Addison-Wesley, Reading Mass., 1969 {4}

[KNUT73] D.E. KNUTH,
The Art or Computer Programming: Sorting and Searching, Vol. 3
Addison-Wesley, Reading Mass., 1973 {1, 2}

[LEWI69] P.A.W. LEWIS; A.S. GOODMAN; and J.M. MILLER,
"A Pseudo-Random Number Generator for the System/360,"
IBM Syst. J. **8**, 2 (1969), 136-145 {4}

[LEWI73] P.A.W. LEWIS; and G.S. SHEDLER,
"Empirically Derived Micromodels for Sequences of Page Exceptions,"
IBM J. Res. Dev. **17**, 2 (March 1973), 86-99 {3}

[MARS69] B.S. MARSHALL,
"Dynamic Calculation of Dispatching Priorities under OS/360 MVT,"
Datamation (Aug 1969), 93-97

[MARS79] W. MARSHALL; and C.T. NUTE,
"Analytic Modeling of 'Working-Set Like' Replacement Algorithms,
Conf. on Simulation, Measurement, and Modeling of Computer Systems, 1979 {2}

[MASU77] T. MASUDA,
"Effect of Program Localities on Memory Management Studies,"
Proc. 6th Symp. Operating Systems Principles, ACM SIGOPS, pp. 117-124, 1977 {3}

[MATT70] R.L. MATTSON; J. GECSEI; D.R. SLUTZ; and I.L. TRAIGER,
"Evaluation Techniques for Storage Hierarchies,"
IBM Syst. J. **9**, 2 (1970), 78-117

[MINI76] V. MINETTI,
"Performance Evaluation of a Batch-Time Sharing Computer System using a Trace Driven Model.,"
in *Modeling and Performance Evaluation of Computer Systems*, North-Holland, 1976

[MORR72] J.B. MORRIS,
"Demand Paging Through Utilization of Working Sets on the MANIAC II,"
Commun. ACM **15**, 10 (Oct. 1972), 867-872

[MUNT75] R.R. MUNTZ,
"Analytic Modeling of Interactive Systems,"
Proc. IEEE **63**, 6 (June 1975), 946-953

[NEIL66] N.R. NIELSEN,
The Analysis of General Purpose Computer Time-Sharing Systems,
Ph. D. Thesis, Stanford Univ., Stanford, Ca., Dec. 1966 {3}

[NIEL67] N.R. NIELSEN,
"The Simulation of Time Sharing Systems,"
Commun. ACM **10**, 7 (July 1967), 397-412 {3}

[ODEN72] P.H. ODEN; and G.S. SHEDLER,
"A Model of Memory Contention in a Paging Machine,"
Commun. ACM **15**, 8 (August 1978), 761-771

[OLIV74] N.A. OLIVER,
"Experimental Data on Page Replacement Algorithms,"
Proc. 1974 AFIPS Nat. Computer Conf., Vol 44, Spartan Books, Washington, D.C., pp. 179-184

[POTI77] D. POTIER,
"Analysis of Demand Paging Policies with Swapped Working Sets,"
Proc. 6th Symp. Operating Systems Principles, ACM SIGOPS, pp. 125-131, 1977

[PRIE73] B. PRIEVE,
A Page Partition Replacement Algorithm,
Ph. D. Thesis, Univ. of California, Berkeley, Dec. 1973 {2}

[PRIE76] B. PRIEVE; and R.S. FABRY,
"VMIN - An Optimal Variable-Space Page Replacement Algorithm,"
Commun. ACM **19**, 5 (May 1976), 295-297 {2, 3}

[RAFI76] A. RAFII,
Empirical and Analytical Studies of Program Reference Behavior,
Ph. D. Thesis, Stanford University, July 1976

[RODR71] J. RODRIGUEZ-ROSELL,
"Experimental Data on How Program Behavior Affects the Choice of Scheduler Parameters,"
Proc. 3rd Symp. Operating Systems Principles, ACM SIGOPS, pp. 247-253, 1971

[RODR72] J. RODRIGUEZ-ROSELL; and J.P. DUPUY,
"The Evaluation of a Time-Sharing Page Demand System,"
Proc. 1972 AFIPS Spring Jt. Computer Conf., Vol 40, Spartan Books, Washington, D.C., pp. 759-765

[RODR73] J. RODRIGUEZ-ROSELL; and J.P. DUPUY,
"The Design, Implementation, and Evaluation of a Working Set Dispatcher,"
Commun. ACM 16, 4 (April 1973), 247-253

[RODR73a] J. RODRIGUEZ-ROSELL,
"Empirical Working Set Behavior,"
Commun. ACM 16, 9 (Sept. 1973), 556-560

[RYDE70] K.D. RYDER,
"A Heuristic Approach to Task Dispatching,"
IBM Syst. J. 8, 3 (1970), 189-198 {2}

[SCHE67] A.L. SCHERR,
An Analysis of Time-Shared Computer Systems,
Ph.D. Thesis, MIT, Cambridge, Mass., 1967

[SALT74] J.H. SALTZER,
"A Simple Linear Model of Demand Paging Performance,"
Commun. ACM 17, 4 (April 1974), 181-186 {3}

[SEKI72] A. SEKINO,
Performance Evaluation of Multiprogrammed Time-shared Computer Systems,
Ph. D. Thesis, M.I.T., Sept. 1972 {3}

[SHAN75] R. SHANNON,
Systems Simulation: the Art and Science,
Prentice-Hall, Englewood Cliffs, N.J., 1975 {4}

[SHAW74] A.C. SHAW,
The Logical Design of Operating Systems,
Prentice-Hall, Englewood Cliffs, N.J., 1974 {2}

[SHEM66] J.E. SHEMER; and G.A. SHIPPEY,
"Statistical Analysis of Paged and Segmented Computer Systems,"
IEEE Trans. on Electronic Computers 15, 6 (Dec. 1966), 855-863

[SHED72] G.S. SCHEDLER; and C. TUNG,
"Locality in Page Reference Strings,"
SIAM J. Computing 1, 3 (Sept. 1972), 218-241 {3}

[SHER72] S. SHERMAN; F. BASKETT; and J.C. BROWNE
"Trace-Driven Modeling and Analysis in a Multiprogramming System,"
Commun. ACM 15, 2 (Dec. 1972), 1063-1069 {2}

[SHIL68] A.J. SHILS,
The Load Leveler,
RC-2233, IBM Research, Yorktown Heights, N.Y., 1968

[SHUS78] L. SHUSTEK,
Analysis and Performance of Computer Instruction Sets,
Ph.D. Thesis, Stanford Univ., 1978 {3,5}

[SIMO79] R. SIMON,
The Modeling of Virtual Memory Systems,
Ph.D. disseration, Dept. Comp. Sci., Purdue U., 1979 {3,4}

[SLUT74] D.R. SLUTZ; and I.L. TRAIGER,
"A Note on the Calculation of the Average Working Set Size,"
Commun. ACM 17, 10 (Oct. 1974), 563-565

[SLUT75] D.R. SLUTZ,
A Relation Between Working Set and Optimal Algorithms for Segment Reference Strings,
RJ-1623, IBM Research, San Jose, Ca., July 1975

[SMIT76] A.J. SMITH,
"A Modified Working Set Paging Algorithm,"
IEEE Trans. on Computers 25, 9 (Sept. 1976), 907-914 {2}

[SMIT77] A.J. SMITH,
"Two Methods for the Efficient Analysis of Memory Address Trace Data,"
IEEE Trans. on Software Engineering 26, 1 (Jan. 1977), 94-101 {3}

[SMIT80] A.J. SMITH,
"Multiprogramming and Memory Contention,"
Software – Practice and Experience 10 (1980), 531-552

[SMIT67] J.L. SMITH,
"Multiprogramming Under a Page on Demand Strategy,"
Commun. ACM 10, 10 (Oct. 1967), 636-646

[SPIR76] J.R. SPIRN,
"Distance String Models for Program Behavior,"
Computer 9, 11 (Nov. 1976), 14-20 {3}

[SREE74] K. SREENIVASAN; and A.J. KLEINMAN,
"On the Construction of a Representative Synthetic Workload,"
Commun. ACM 17, 3 (March 1974), 127-132

[STEV68] D.F. STEVENS,
"On Overcoming High-Priority Paralysis in Multiprogramming Systems: A Case History,"
Commun. ACM 11, 8 (Aug. 1968), 539-541

[STIM69] S. STIMLER,
"Some Criteria for Time-Sharing System Performance,"
Commun. ACM 12, 1 (Jan. 1969), 47-53

[TRIP77] S. TRIPATHI; and K.C. SEVCIK,
"The Influence of Multiprogramming Limit on Interactive Response Time in a Virtual Memory System,"
Proc. Sigmetrics/CMG VIII Conference, Wash. D.C., pp. 121-130, Nov. 1977 {3}

[WINO71] J. WINOGRAD; S.J. MORGANSTEIN; and R. HERMAN,
"Simulation Studies of a Virtual Memory, Time-shared, Demand Paging Operating System"
Proc. 3rd Symp. Operating Systems Principles, ACM SIGOPS, pp. 149-155, Oct 1971

[WULF69] W.A. WULF,
"Performance Monitors for Multiprogramming Systems,"
Proc 2nd Symposium on Operating System Principles, 1969, pp. 175-185 {2}

[YU76] F. YU,
Modeling the Write Behavior of Programs,
Ph.D. Thesis, Stanford Univ., 1976 {4}