1978

Working Sets Past and Present

Peter J. Denning

Report Number:
78-276

WORKING SETS PAST AND PRESENT*

Peter J. Denning

Computer Science Department
Purdue University
W. Lafayette, IN 47907

CSD-TR-276

July 1978
(revised May 1979)

Abstract

A program's working set is the collection of segments (or pages) recently referenced. This concept has led to efficient methods for measuring a program's intrinsic memory demand; it has assisted in understanding and in modeling program behavior; and it has been used as the basis of optimal multiprogrammed memory management. The total cost of a working set dispatcher is no larger than the total cost of other common dispatchers. This paper outlines the argument why it is unlikely that anyone will find a cheaper nonlookahead memory policy that delivers significantly better performance.

Index Terms

Working sets, memory management, virtual memory, multiprogramming, optimal multiprogramming, lifetime curves, program measurement, program behavior, stochastic program models, phase/transition behavior, program locality, multiprogrammed load controllers, dispatchers, working set dispatchers, memory space-time product.

---

*Work reported herein was supported in part by NSF Grants GJ-41289 and MCS78-01729 at Purdue University. A condensed, preliminary draft of this paper was presented as an invited lecture at the International Symposium on Operating Systems, IRIA Laboria, Rocquencourt, France, October 2-4, 1978 [DENN78d].

THE BEGINNING

In the summer of 1965 Project MAC at MIT tingled with the excitement of MULTICS. The basic specifications were complete. Papers for a special session at the Fall Joint Computer Conference had been written. Having read all available literature on "one-level stores", on "page-turning algorithms", on "automatic folding", and on "overlays", and having just completed a master's thesis on the performance of drum memory systems, I was eager to contribute to the design of the multiprogrammed memory manager of MULTICS.

Jerry Saltzer characterized the ultimate objective of a multiprogrammed memory manager as an adaptive control that would allocate memory and schedule the central processor (CPU) in order to maximize performance. The resulting system could have a knob by which the operator could occasionally tune it. (See Figure 1.)

Such a delightfully simple problem statement! Of course we had no idea how to do this. In 1965, experience with paging algorithms was almost nil. No one knew which of the contenders -- first-in-first-out (FIFO), random, eldest unused (as LRU was then called), or the Ferranti Atlas Computer's Loop Detector [KILB62] -- was the best. No one knew how to manage paging in a multiprogrammed memory. Few yet suspected that strong coupling between memory and CPU scheduling is essential -- the prevailing view was that the successful multilevel feedback queue of the Compatible Time Sharing System (CTSS) would be used to feed jobs into the multiprogramming mix, where they would then neatly be managed by an appropriate

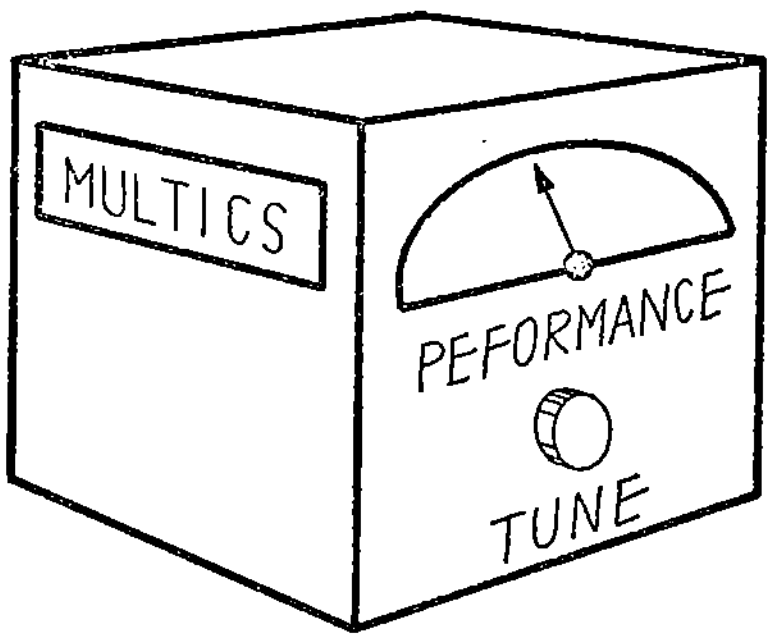


FIGURE 1. Abstract mathematical representation of Saltzer's Problem.

page-turning algorithm.

By mid 1967 I saw a solution of Saltzer's Problem -- using a balance-policy scheduler with working-set memory management. (See DENN68a,b and DENN69.) But by that time the conventional optimism had changed to circumspection; no one wanted to risk my unconventional proposal which, in the standards of the day, was elaborate.

The circumspection had several sources. Fine, Jackson, and McIssac had shaken the early enthusiasm with a pessimistic study of virtual memory when applied to existing programs [FINE66]. Belady's famous study of programs on the M44/44X computer showed no clear "winner" among the leading contenders for page replacement policies [BELA66]. Saltzer knew from preliminary studies of MULTICS that performance could collapse on attempted overcommitment of the main memory; he used the term "thrashing" to describe this unexpected behavior. Before they would risk building it, the designers of MULTICS thus wanted hard evidence that my proposal would be a "winner" and would not thrash.

But there was scant hope that I could collect enough data and develop enough theory in time to influence MULTICS. Recording and analyzing program address traces was tedious and expensive: the "Stack algorithms" [MATT70] for simplifying the data reductions had not yet been discovered. Moreover, it was important to test programs developed specifically for the virtual memory's environment: Brawn, Gustavson, Mankin, and Sayre had found that significant improvements in program behavior would result if programmers attempted even simple schemes to enhance "locality" [BRAW68, BRAW70, SAYR69]. Few such programs existed in 1967.

Testing programs designed when locality does not matter can lead to unduly pessimistic conclusions -- e.g., the Fine et al. study [FINE66].

However convincing my arguments might have been, there were many who believed that usage bits were all the hardware support for memory management that could be afforded. My proposal was, for the time, out of the question.

The working set is usually defined as a collection of recently referenced segments (or pages) of a program's virtual address space. Because it is specified in the program's virtual time, the working set provides an intrinsic measurement of the program's memory demand -- a measurement that is unperturbed by any other program in the system or by the measurement procedure itself. Data collected from independent measurements of programs can be recombined within a system model in order to estimate the overall performance of the system subjected to a given program load. Queueing networks are widely used for this purpose owing to their ability to estimate throughputs and utilizations well [DENN78c]. It was not until 1976 that the collective results of many researchers contained the data (on program behavior for various memory policies) and the theory (on combining these data with queueing network models of systems) to allow a convincing argument that the working set principle is indeed a cost-effective basis for managing multiprogrammed memory to within a few per cent of optimum throughput -- a solution of Saltzer's Problem.

Following the next section, which defines the terminology used throughout the paper, are four main sections. The first

describes the working set as an efficient tool for measuring the memory demands of programs; the second describes a progression of program behavior models culminating in the phase/transition model; the third describes the experimental evidence demonstrating that a working set policy can operate a system to within a few per cent of optimum; and the fourth describes an inexpensive implementation of a working set dispatcher. A concluding section assesses the state of the art.

TERMINOLOGY

Segmentation and Paging

A segment is a named block of contiguous locations in a (logical) address space. A segment can be small, as a single-entry-single-exit instruction sequence (detected by a compiler); medium, as a data file (declared by a programmer); or large, as an entire address space. Normally the biggest segments are several orders of magnitude larger than the smallest; this complicates memory managers that try to store segments contiguously. Paging simplifies segment management by allocating space in blocks all of the same size; a segment can be divided into equal size "pages", any one of which can be stored in any "page frame" of main memory. One or more small segments can be fitted into a single page. A large segment can be partitioned into a sequence of pages of which the last is only partly filled; the common scheme of paging a large, linear address space is an example of this use of segmentation.

Segmentation is an important hardware tool for implementing programming-language features -- for example, access controls, scope rules, controlled sharing, encapsulation of subsystems, error confinement, or storage objects whose sizes change. Paging is an important tool for implementing efficient storage managers. Some systems try to obtain both sets of advantages by combining aspects of both; for example, MULTICS pages each segment independently with 1024-word pages (see ORGA72). The compilers on the Burroughs B6700 enforce a maximum segment size but treat word 0 of segment i+1 as the logical successor of the last word of segment i; thus a large file can span several large fixed-size segments and a smaller one (see ORGA73).

In the following discussion, I shall use the term "segments" to include the possibility of "pages", except when discussing matters pertaining specifically to paging.

Memory Policies

A reference string is a sequence of T references, r(1)...r(t) ...r(t), in which r(t) is the segment that contains the t^{th} virtual address generated by a given program. Time is measured in memory references; thus t = 1,2,3,... measures the program's internal "virtual time" or "process time".

A resident set is the subset of all the program's segments present in the main memory at a given time. If the reference r(t) is not in the resident set established at time t-1, a segment (or page) fault occurs at time t. This fault interrupts the program until the missing segment can be loaded in the resident set. Segments made resident by the fault mechanism are "loaded on demand" (others are "preloaded").

The memory policies of interest here determine the content of the resident set by loading segments on demand and then deciding when to remove them. To save initial segment faults, some memory policies also swap an initial resident set just prior to starting a program. (Easton and Fagin refer to the case of an empty initial resident set as a "cold start", and an initially nonempty resident set as a "warm start" [EAST78b].)

The memory policy's control parameter, denoted θ, is used to trade paging load against resident set size. For the working set policy, but not necessarily for others, larger values of θ usually

produce larger mean resident set sizes in return for longer mean interfault times. (See FRAN78.) In principle, θ could be generalized to a set of parameters -- e.g., a separate parameter for each segment -- but no one has found a mutliple parameter policy that improves significantly over all single parameter policies.

The performance of a memory policy can be expressed through its swapping curve, which is a function f relating the rate of segment faults to the size of the resident set. A fixed-space memory policy, a concept usually restricted to paging, interprets the control parameter θ as the size of the resident set; in this case the swapping curve $f(\theta)$ specifies the corresponding rate of page faults. A variable-space memory policy uses the control parameter θ to determine a bound on the residence times of segments. Thus a value of θ implicitly determines a mean resident set size x, and also a rate of segment faults y; the swapping curve, $y = f(x)$, is determined parametrically from the set of (x,y) points generated for the various θ. (See DENN78b.)

One of the parameters needed in a queueing network model of a multiprogramming system is the paging rate [DENN75, DENN78a]. This parameter is easily determined from the lifetime curve, which is the function $g(x) = 1/f(x)$ giving the mean number of references between segment faults when the mean resident set size is x. Lifetime curves for individual programs under given memory policies are easy to measure. A knee of the lifetime curve is a point at which $g(x)/x$ is locally maximum, and the primary knee is the global maximum of $g(x)/x$. (See Figure 2.)

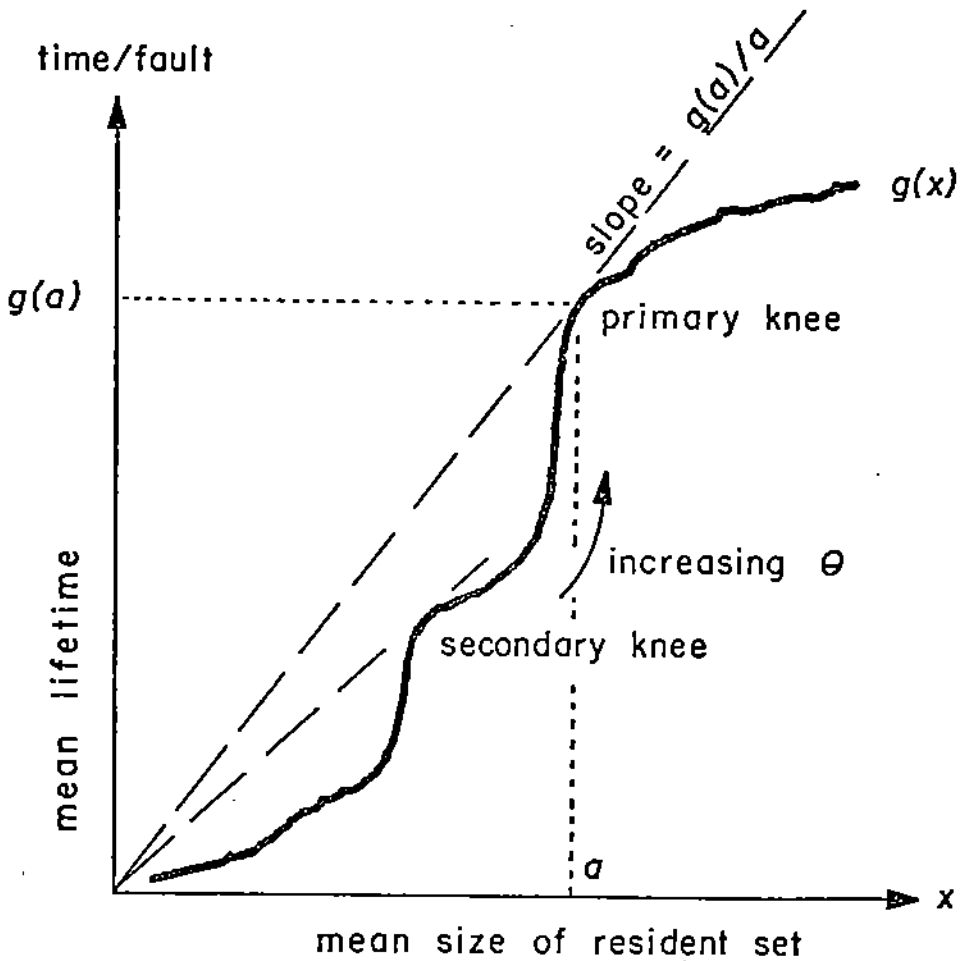


FIGURE 2. A lifetime curve.

A memory policy satisfies the inclusion property if $R(t, \theta) \subseteq R(t, \theta+a)$ for $a > 0$. This means that, for increasing θ, the mean resident set size never decreases and the rate of segment faults never increases. In Figure 2, this means that the lifetime curve increases uniformly as θ increases. (See DENN78a,b and FRAN78.)

Several empirical models of the lifetime curve have been proposed. One is the Belady Model [BELA69]:

$$g(x) = a \cdot x^k,$$

where x is the mean resident set size, a is a constant, and k is normally between 1.5 and 3 (a and k depend on the program). This model is often a reasonable approximation of the portion of the lifetime curve below the primary knee, but it is otherwise poor [DENN75c, SPIR77].* A second model is the Chamberlin Model [CHAM73]:

$$g(x) = \frac{T/2}{1 + (d/x)^2},$$

where T is the program execution time and d is the resident set size at which lifetime is T/2. Though this function has a knee, it is a poor match for real programs. The recent empirical studies by Burgevin, Lenfant, and Leroudier contain many interesting observations

*Easton and Fagin have found that the quality of the Belady model improves on changing from an assumption of "cold start" (resident set initially empty) to "warm start" [EAST78b]; however, the "warm start" merely increases the height of the primary knee without significantly changing the knee's resident set size. (See also GRAH76, KAHN76, SPIR77.) Parent and Potier observed that the overhead of swapping can cause programs conforming to the Belady model to exhibit lifetime curves, measured while the system is in operation, with flattening beyond the primary knee [PARE77, POTI77]; however, real programs exhibit flattening beyond the primary knee even if all the faults normally caused by initial references are ignored. (See GRAH76, KAHN76, SPIR73, SPIR77.)

about, and refinements of, these models [LENF75, LERO76a]. Since it is quite easy to measure lifetime curves [DENN78a,b, EAST77] I have greater confidence in results when the model parameters are derived from real data rather than estimated from the models. Since optimal performance is associated with the knees of lifetime functions [DENN76b, GRAH76, GRAH77] I am hesitant to use lifetime curve models that have no knees.

It is well to remember that a lifetime (or swapping) curve is an average for an interval of program execution. If the program's behavior during a subinterval can differ significantly from the average, conclusions based on its lifetime function may be inaccurate. For example, a temporary overload of the swapping device may be caused by a burst of segment faults -- an event that might not be predicted if the mean lifetime is long.

Space-Time Product

A program's space-time product is the integral of its resident set size over the time it is running or waiting for a missing segment to be swapped into main memory. If $s(t)$ is the size of the resident set at time t, t_i is the time of the i^{th} segment fault ($i = 1,\ldots,K$) and D is the mean swapping delay, the space-time product is

$$ST = \sum_{t=1}^{T} s(t) + D \cdot \sum_{i=1}^{K} s(t_i).$$

Note that the first sum is $x \cdot T$, where x is the mean resident set size. If we approximate the second sum by $x \cdot K$ and note that $x \cdot K = x \cdot (K/T) \cdot T = x \cdot f(x) \cdot T$, where $f(x)$ is the missing segment rate,

the space-time product is approximated by

$$ST(x) = T \cdot x \cdot (1 + D \cdot f(x)).$$

Though simple to calculate, this approximation is not very reliable -- it is not consistently high or low and can be in error by as much as 20% [GRAH76].

Note that $x \cdot f(x) = x/g(x)$ is minimum at the primary knee of the lifetime curve. If D is large, choosing x at this knee will (approximately) minimize the space-time. However, since D is usually also a function of f(x), finding a formula for the minimum in ST(x) is not easy [GELE73b].

THE WORKING SET: MEASURER OF MEMORY DEMAND

My first concept of the working set was a set of recently referenced pages that estimated a program's memory demand in the immediate future. I regarded the working set as a model of program behavior -- that is, a set of hypotheses about how a program demands memory space. However, once I saw that working set statistics could be calculated for arbitrary assumptions about times between repeated references and correlations among references, I realized that the working set is not a model for programs, but for a class of procedures that measure the memory demands of programs. In the following paragraphs I will trace the development of the working set concept, as a measurement tool, from its inception in 1965 through its present form, the "generalized working set".

The Early Working Set

In the fall of 1965 I undertook analytic studies to compare three page replacement policies: FIFO (first-in-first-out), LRU (least recently used), and Atlas Loop Detection (ALD) [DENN66]. FIFO is best suited for programs that reference segments in sequence, LRU for programs that reference subsets of segments repetitively, and ALD for programs that reference segments in loops. Because all three kinds of behavior are encountered in practice, it was not clear how to prove the superiority of any one of the three policies. Moreover, it is easy to find examples of programs for which any given policy excels while the others falter.

It occurred to me that a scheme based on sampling usage bits every θ units of virtual time should work for all three kinds of behavior -- thereby avoiding the problem of choosing among FIFO, LRU,

and ALD. I borrowed the Algol term working set to refer to the set of segments used during the most recent sample interval. (See DENN66.) It seemed as if θ could be chosen to effect a good compromise among the three behaviors; segments of clusters and loops would appear in the working set, and in no case would any segment stay resident for longer than θ units of time after it fell out of use.

It was also apparent that the operating system should swap in the working sets as units prior to starting a program on the CPU. This would reduce the overhead required to bring each working-set segment in separately on demand. (See DENN66, DENN68a,b, and also POTI77, SIMO79).

By the end of 1966 I was using the concept of moving window as an abstraction of the sampling process: the working set at time t is defined as the distinct segments among $r(t-\theta+1)...r(t)$. The mean number of segments in the working set (over the reference string) is denoted $s(\theta)$ and the rate of segment faults as $m(\theta)$. The working set (WS) policy is the memory policy whose resident sets are always the program's working sets. The working set policy satisfies the inclusion property for each program.

Then I noticed that a reference to segment i could cause a fault if and only if the time since the prior reference to segment i exceeds the window size θ. Thus $m(\theta)$ depends on the interreference distribution, $h_i(k)$, which gives the probability that two successive references to segment i are k time units apart. By assuming that the successive references to each segment i are the recurrent events of a renewal process with recurrence distribution $h_i(k)$, I was able to derive formulae for $s(\theta)$ and $m(\theta)$ as follows. Let n_i denote the number of references to segment i in the reference string of length T. Then,

as T gets large, the quantity n_i/T tends to the long-term probability a_i of referencing segment i, and T/n_i tends to the mean interreference interval M_i; this implies that $a_i = 1/M_i$. The overall interreference distribution is

$$(1) \qquad h(k) = \sum_i a_i \cdot h_i(k).$$

I showed that

$$(2) \qquad m(\theta) = \sum_{k>\theta} h(k) \qquad \text{[WS miss rate]}$$

$$(3) \qquad s(\theta) = \sum_{k=0}^{\theta-1} m(k) \qquad \text{[WS mean size]}$$

(These results were reported in DENN68b, and improved in DENN72a and COFF73. The formula for $s(\theta)$ is the solution of the backward recurrence problem for the joint renewal processes. Opderbeck and Chu rediscovered these results a few years later [OPDE75].)

These formulae show that calculating swapping or lifetime curves is straightforward once the easily-measured* overall interreference distribution, h(k), has been determined.

Operational Analysis of Working Sets

The renewal theory analysis leaves ample room for skepticism. Are the successive interreference intervals of a given segment independent samples from a common distribution? How accurate are the results when applied to finite reference strings? (The formulae for $m(\theta)$ and $s(\theta)$ were obtained by taking a limit as T becomes

*The basis of an efficient procedure for measuring h(k) is an array TIME that records the most recent reference time for each segment. At time t, let i = r(t), set k = t - TIME[i], add 1 to a counter c(k), and set TIME[i] = t. After the last reference (at time T) set h h(k) = c(k)/T. (See DENN78a,b, and EAST77.)

infinite.)

Slutz and Traiger started a significant shift in my thinking in 1973, when they showed me how to derive formulae (2) and (3) for finite reference strings in terms of the empirical distribution of interreference times. (See SLUZ74.) They viewed their analysis as the time analog of the "stack algorithm" analysis to which they had contributed earlier (see MATT70, COFF73). They had derived the working set formulae without recourse to any stochastic assumptions whatsoever.

The Slutz-Traiger discovery caused me to abandon stochastic analyses of working sets. It convinced me that working sets are tools for measuring memory demand, but not models of program behavior. It set me to wondering if there might not be a unified model of all memory policies satisfying the inclusion property, of which the stack algorithms and the moving-window working sets would be special cases.

In the spring of 1975, Prieve and Fabry told me of their algorithm VMIN, which is the optimal variable-space memory policy -- it generates the least possible fault rate for each value of mean resident set size (see PRIE76).* At each reference r(t) = i, VMIN looks ahead: if the next reference to segment i occurs in the interval (t, t+θ], VMIN keeps i in the resident set until that reference; otherwise, VMIN removes i immediately. In this case θ serves as a window for lookahead, analogous to its use by WS as a window for lookbehind. I noticed that VMIN generates segment

* I understand that Don Slutz knew the principle of this policy in 1971 and that Alan Smith also discovered it for himself in 1974.

faults exactly when WS does (for the same θ) and I modified the procedure for collecting working set statistics to also collect VMIN statistics [DENN75b]. The formula for the mean number of segments in VMIN's resident set is

$$v(\theta) = \frac{1}{T} \sum_{k>\theta} k \cdot h(k) \tag{4}$$

(see also SMIT76b). The mean WS resident set size is, approximately,

$$s(\theta) = v(\theta) + \theta \cdot m(\theta). \tag{5}$$

What intrigued me about this was that we could, in effect calculate VMIN statistics cheaply in real time -- even more cheaply than WS statistics! -- even though VMIN itself cannot be implemented in real time. (For empirical VMIN lifetime curves see GRAH76, PRIE76, SMITH76b.)

The Generalized Working Set

In summer 1975 I discovered that Don Slutz had independently worked out results similar to (4) and (5) for segment reference strings [SLUZ75]. To deal with segments, he had introduced a space-time working set: the set of all segments, each of which has accumulated since prior reference space-time not exceeding ϕ. This is analogous to the moving-window working set, which comprises each segment whose time since prior reference does not exceed θ.

Slutz and I then collaborated on the generalized working set (GWS) [DENN78b]. The GWS comprises each segment whose "retention cost" accumulated since prior reference is not more than ϕ; the retention cost is any function that does not decrease with time since

prior reference and is reset to zero just after a reference. The GWS is a model for all memory policies satisfying the inclusion property. Special cases include the stack algorithms [MATT70], VMIN [PRIE76], and the time and space-time working sets. The swapping curves for GWS policies for selected values of ϕ can be calculated efficiently from formulae similar to (2)-(5). The calculations replace the interreference distribution h(k) with the "retention cost" distribution, which can be measured efficiently on a single pass of the reference string. (DENN78a,b and also EAST77.)

Summary

The moving-window working set, and its descendant, the generalized working set, are best viewed as models of memory policies satisfying the inclusion property. Highly efficient procedures for calculating memory demand statistics for programs operating under these policies have been developed. The derivations of these statistics are purely operational, requiring no stochastic assumptions or any other assumptions about program behavior. Because they are easy to compute, working set statistics are often used as approximations to the statistics of the various relatives to the WS policy, such as global LRU or global FIFO with usage bits [EAST77, EAST78a] However, these approximations are not always good [GRAH76].

The working set also serves as a dynamic estimator of the segments currently needed by a program. The working set is defined in a program's virtual time, independently of other programs; thus, there is no danger that the load on the system can influence the measurement, as can happen with any memory policy applied "globally" to the entire contents of the main memory [DENN75a]. This is why we

refer to the working set as a measurer of a program's intrinsic memory demand.

PROGRAM BEHAVIOR

The previous section describes theoretical studies of the working set in its role as a measurement tool. This section describes how this tool has contributed to the separate question of program behavior.

Most studies of program behavior begin with the hypothesis that a program's reference string is the realization of a (tractable) stochastic process; the subsequent analysis seeks to calculate the swapping curve for particular memory policies and to specify the optimal memory policy. Stochastic models of program behavior usually have a Markov Renewal Structure -- that is, some events or states are assumed to recur, and the results are expressed in terms of interstate transition probabilities and recurrence distributions.

My interest in program models has been to substantiate my long-term intuition that the working set memory policy generates near-optimum space-time product for programs of good locality. I believe that this goal has been realized.

Since the middle 1960s a series of increasingly complex stochastic models of programs has been studied. Each model led to predictions about the behaviors of memory policies. When experiments on real programs failed to support the predictions, the models were revised or discarded. The evolution of program models is a superb example of successful interplay between model development on the one hand, and experimental testing of hypotheses on the other. I

will emphasize this iterative process in the paragraphs following.

The Phase Behavior of Programs

From the middle 1950s computer engineers have been interested in automatic solutions to the "overlay problem" -- i.e., the problem of running large programs in small memories. The early skeptics of virtual memory believed that programmers and compilers, not hardware usage bits and interval timers, were the most reliable sources of information about memory demand.

Proponents of manual overlays drew phase diagrams to help plan a good overlay sequence. A phase diagram depicts program time as a sequence of phases, and address space as a sequence of segments. The segments needed in a given phase can be indicated by check-marks in the diagram. (See Figure 3.) Early descriptions of this concept can be found in ACM61 and DENS65.

The concept that a program favors a subset of its segments during extended intervals (phases) is called locality. The set of segments needed in a given phase is called the locality set of that phase.

Experiments have confirmed the existence of locality in real programs even when programmers do not consciously plan for it. A common method of taking the measurement in motivated by the phase diagram. It displays segment use with a reference map, which is a matrix whose rows correspond to successive Δ - intervals of virtual time; a mark is put in position (i,j) whenever segment i is referenced in the j^{th} time interval. The reference map is displaying the working set $W(j\Delta,\Delta)$ for $j = 1,2,3,\ldots$. (See HATF71, CHU72, CHU76b, and KAHN76 for examples of page reference maps.)

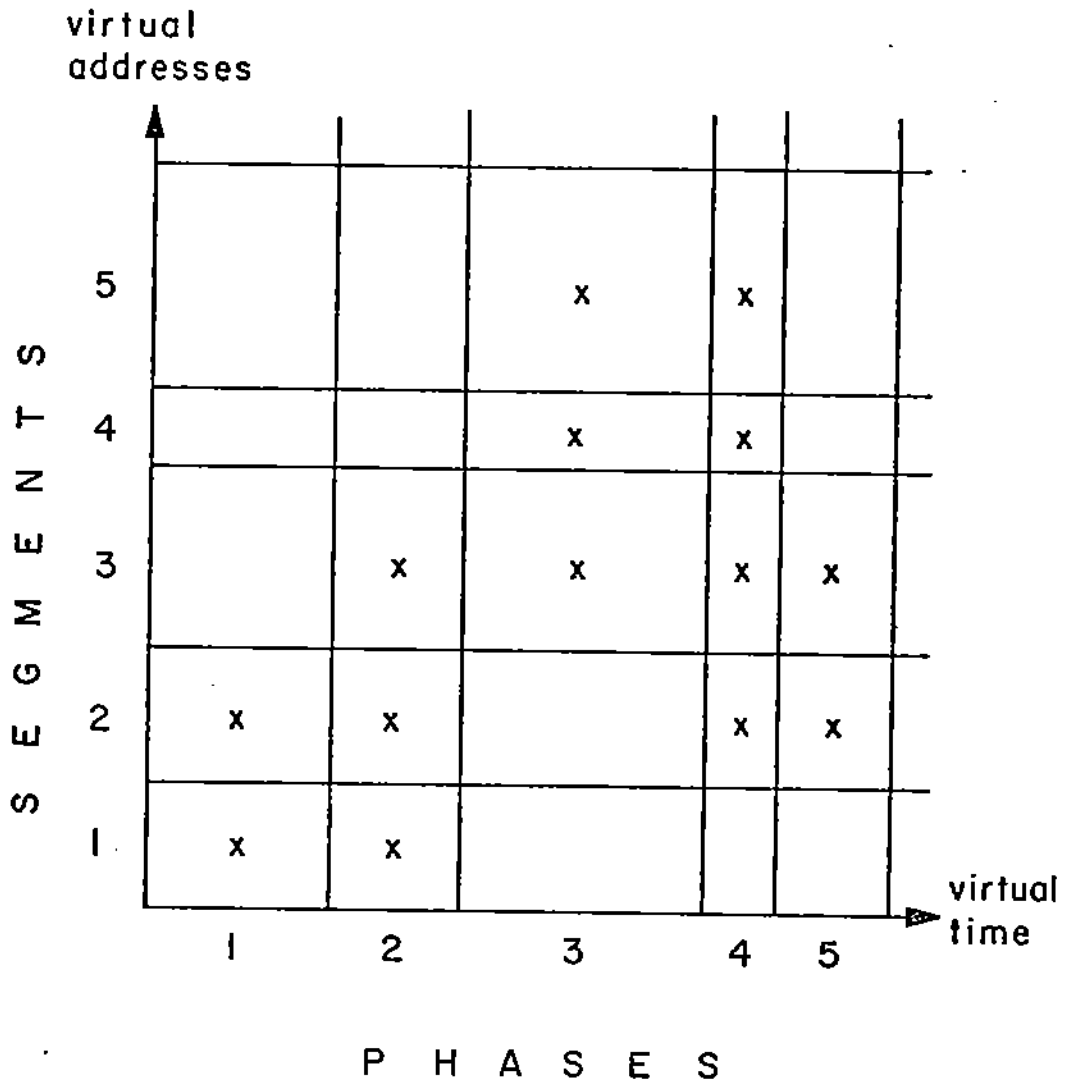


FIGURE 3. Example of a program's phase diagram.

Despite early recognition of the locality of programs, the phase diagram long remained merely a descriptive tool -- a pictorial method of explaining why virtual memory should work. In 1972 I suggested that this description should be the basis of a stochastic program model [DENN72b, SPIR72]. But none of us explored this suggestion very seriously until 1974, when we began experimental studies of phase behavior in real programs. (I will discuss these studies shortly.)

It is difficult to explain why the phase concept took so long to become explicitly a part of program models. Part of the explanation may be that many of us spent the years 1969-1974 studying simple models, moving to the more complex only when convinced that the simple omitted essential features of real programs. The more important part of the explanation may be that many of us had great faith in the slow-drift concept of locality, which holds that phases are long and that changes in the locality set from one phase to another are mild. By calling attention to the phases -- rather than to the changes -- this view ignored the possibility that transitions might be disruptive.

I began using the slow-drift concept of locality in 1968 [DENN68a,b] and persisted with it through 1973 [DENN72a,b, COFF68, SPIR72]. Although I realized that disruptive changes in the locality set could occur at the transitions between phases, I worked with the hypothesis that the phase behavior was so strong that disruptions could safely be ignored. This hypothesis was consistent with such experimental studies as FINE66, BELA66, and COFF68. Moreover, the work of Brawn, Gustavson, Mankin, and Sayre showed very clearly that

high degrees of locality could be instilled into programs at only a minor cost of a programmer's effort, and that such programs would typically produce less total swapping with paging than programs that used manual overlays instead of paging [BRAW68, BRAW70, SAYR69]. These data led me to believe that the disruptive effects of transitions could be "tuned out" by good programming.

The trouble with the slow-drift concept of locality is that it is wrong.

A proper program model should account for the disruptions of the transitions -- which we know today are every bit as significant as the tranquility of phases. After digressing to discuss techniques for automatically improving observed locality in paged programs, I will discuss how the experimental studies of stochastic models led us away from the slow-drift concept to the phase/transition concept.

Program Reorganization to Improve Observed Locality

The Brawn et al. studies showed not only that programmers could help themselves by striving for slow-drift locality as an ideal, but also that programmers could take advantage of knowledge of how the compiler assigned program segments to pages. A study by Comeau in 1967 showed that the order in which the card-decks of subroutines were presented to a loader has a most significant effect on the amount of paging generated [COME67]. It is easy to construct examples in which the paging overhead by the working set policy on a program with good locality (long phases, mild transitions) is less than the paging produced by the optimal policy on another version of the same program with poor locality (short phases, disruptive transitions).

These facts have inspired much interest in program reorganization, which studies how the compiler (or loader) can assign segments of a program to the pages of the address space in order to preserve, in the page reference patterns, as much of the original locality as possible. It is important to note that "segments" in this context are small logical blocks of a program detectable by a compiler; they might be array rows or single-entry-single-exit instruction sequences. Some machines, such as the Burrough B6700 series, implement such segments directly in the virtual memory [ORGA73]; no program reorganization is needed. But other machines, such as IBM Virtual Storage systems or MULTICS, were designed on the concept of paging whole address spaces or segments; program reorganization is potentially useful in these contexts. (See also DENN71, DENS65.)

The recent experimental studies by Batson and Madison confirm that there is a good deal of locality present in the symbolic segment reference patterns of programs; hence there can be a big payoff in program reorganization. (See MADI76, BATS76a,b.)

The essence of program reorganization is straightforward. Through measurement or analysis one obtains a matrix $[a_{ij}]$ in which a_{ij} measures the maximal swapping cost that would be caused by putting segments i and j on different pages. (Thus $\Sigma a_{ij}/2$ would be the swapping cost of running the program in a one-page memory if each segment occupied its own page.) One then uses a "clustering algorithm" to group segments onto pages to minimize the quantity

$$\sum_{I,J}\left(\sum_{i\in I}\ \sum_{j\in J} a_{ij}\right) ,$$

where I and J are distinct pages. In the scheme of Hatfield and Gerald, a_{ij} counts the number of times segment j is referenced next after i [HATF71]. In the critical working set scheme of Ferrari, a_{ij} measures the number of times segment j was referenced and found missing from a logical working set of segments, of which segment i was a member; putting i and j on the same page would then remove a potential working-set page fault [FERR74, FERR75, FERR76]. Ferrari reports that his method could reduce paging by as much as 1/3 relative to Hatfield's method, which could in turn improve by as much as 1/3 relative to unreorganized compiler output. Other authors have reported similar findings [BABO77, MASU74].

Despite its dramatic effects,* program reorganization is expensive. It is cost-effective only for oft-run production programs. It is well to remember that this technique has been motivated by the need to compensate for virtual memory hardware designed with page sizes that are large compared to logical program blocks -- e.g., 1024-word pages versus a median sugment size of less than 50 words [BATS70]. Large page size is motivated by the need to mitigate the high latency time of mechanical secondary storage devices [DENN70, GELE73a, GELE74]. If the page size could be, say, 64 words, or if small segments could be assigned in the virtual memory, there would be little danger that the compiler's unreorganized output would mask

*A statistical study by Tsao, Comeau, and Margolin [TSAO72] seemed to show main memory size and job-type have more significant influences on system performance than program organization. The dramatic improvements observed by Ferrari, Hatfield, and others result from the reductions in working set sizes and paging rates, which allow higher levels of multiprogramming and resource utilization. Tsao et al. did not study the possibility that program reorganization could reduce main memory requirements and change the job-types.

off a program's intrinsic locality. The new low-latency circulating-store secondary memories -- e.g., charge-coupled devices and bubble memories -- can be used as a new, intermediate level of the memory hierarchy [as envisaged in DENN68c]. It could efficiently handle transfers of small segments across the interface with the high-speed main store; it could swap the contents of its circulating storage rings, in the manner of pages, across the interface with the slow-speed bulk store. Ferrari's studies can be used to infer that such a system would be inherently more efficient than paging systems.

Simple Stochastic Models for Program Locality

The models studied most actively from 1965-1975 were the simple renewal model, the independent reference model, and the LRU stack model.

The simple renewal model (SRM) treats the successive references to each segment as the recurrent events of a renewal process that is asymptotically uncorrelated with the renewal processes of the other segments. This model has been used for calculating working set statistics [DENN68b, DENN72a, COFF73, OPDE75]. It has not yielded any useful insights into optimal memory management. In 1974 Slutz and Traiger showed that operational assumptions could replace stochastic assumptions in the derivations of formulae for working set statistics [SLUZ74]; this showed that renewal theory is not essential to study working sets as a measurement tool.

In 1971 Schwartz and I showed that, if programs conform to the assumption that reference substrings at large separations tend to be uncorrelated, the working set size will tend to be normally distributed (See DENN72a). At about the same time, Rodriguez-Rosell presented experimental results showing several programs with multimodal working set size distributions -- a direct contradiction of the "asymptotic uncorrelation" assumption for these programs [RODR71]. Subsequent experimental studies by Bryant, Burgevin, Ghanem, Kobayashi, Lenfant, Leroudier, and others have all confirmed that some programs have normal working-set size distributions, others do not [BRYA75, GHAN74, LENF74, LENF75, LERO76a]. Bryant's autocorrelation functions for

working set sizes directly demonstrated long-term correlations in several programs [BRYA75]. These data tend to cast doubt on the simple renewal model as a general description of behavior. (However, Spirn believes that the normal distribution is valid within program phases; see SPIR77.)

The independent reference model (IRM) regards the reference string as a sequence of independent random variables with a common stationary reference distribution:

$$\Pr[r(t) = i] = a_i \qquad \text{for all } t.$$

(This model was used informally in DENN66 and introduced formally in AHO71.) This model predicts a geometric interreference distribution,

$$h_i(k) = (1-a_i)^{k-1}a_i \qquad \text{for } k = 1,2,3,\ldots$$

The optimal memory policy for the IRM (denoted A_0) replaces the segment with smallest value of a_i among the segments present in the resident set [AHO71]. Formulae for the swapping curves of various major memory policies applied to IRM reference strings were derived by King [KING71] and by Gelenbe [GELE73b]; these formulae have been collected in COFF73 and SMIT76b.

The IRM is the simplest way of accounting for nonlinearities observed in swapping curves of real programs -- an assumption of completely random references would imply linear swapping curves [BELA69, DENN68b, FINE66]. In 1972 Spirn and I reported that the IRM overestimates real working set sizes by factors of 2 or 3 when the a_i are the observed reference densities of the program's pages [SPIR72]; this has been corroborated by Lenfant and Burgevin [LENF75] and by Arvind, Kain, and Sadeh [ARVI73]. The conclusion is that,

at best, different sets of a_i hold at different times -- i.e., programs have multiple phases. The IRM is not a good model of overall program behavior.

The LRU stack model (LRUSM) is motivated by the LRU memory policy [SHED72, SPIR76, SPIR77]. The "LRU stack" just after reference r(t) is a vector ordering the segments by decreasing recency of reference; r(t) is at the first position. The stack distance d(t) associated with reference r(t) is the position of r(t) in the stack defined just after r(t-1). The LRU stack has the property that the LRU policy's resident set of capacity θ segments always contains the first θ elements of the stack, and the missing-segment rate is the frequency of occurrences of the event $d(t) > \theta$. The LRUSM assumes that the distances are independent random variables with a common stationary distribution:

$$\Pr[d(t) = i] = b_i \qquad \text{for all } t.$$

If $b_1 \geq b_2 \geq \ldots \geq b_i \geq \ldots$, the LRU policy is optimal both in variable space and in fixed space [COFF73, SMIT76b, SPIR77]. In 1975 Chu and Opderbeck [CHU76a] and Sadeh [SADE75] independently developed a technique for constructing a semi-Markov model for the resident set size and page-fault rate of a memory policy when applied to an LRUSM reference string; however the semi-Markov model is equivalent to the LRUSM itself. Spirn developed an algorithm for computing WS swapping curves in the LRUSM [SPIR73, SPIR77]. Coffman and Ryan established that the probability distribution of WS size in the LRUSM is approximately normal [COFF72] and Lenfant developed an exact formula for this distribution [LENF74].

The LRUSM has fared only slightly better under testing than the IRM. In 1971 Lewis and Yue reported that most programs exhibit strong correlations among stack distances [LEWI71]. In 1972 Spirn and I reported that the LRUSM estimates the working set size quite well (within 10%) when the b_i are the observed stack-distance frequencies of the program [SPIR72]. However, the LRUSM estimates the swapping curve poorly, with maximum errors around 40%. These results have been corroborated by ARVI73, LENF74, and LENF75. Moreover, as sketched in Figure 4, the LRUSM predicts that WS will perform worse than LRU -- even though WS almost always performs considerable better than LRU, especially in the region near the primary knee of the WS lifetime curve [GRAH76, SPIR77]. Finally, the LRUSM predicts that the long-term page reference densities are equal, contradicting observations of real programs [COFF73, SPIR73, SPIR77].

In 1976 Baskett and Rafii [BASK76] reported the curious result that, if the IRM's a_i are chosen so that the swapping curve of the optimal IRM policy (A_0) matches that of the MIN policy [BELA66] on the real program, the IRM formulae for other policies (LRU, FIFO, WS, etc.) will estimate the actual swapping curves surprisingly well. (The errors are of the same order as the LRUSM's errors.) Unfortunately there is no physical interpretation of the a_i thus determined.

Another defect of both the IRM and the LRUSM is that neither includes a concept of changing locality set size. In LRUSM, for example, the locality set comprising the top θ stack positions is always referenced with the fixed probability $b_1+\ldots+b_\theta$ -- θ does not have to vary to keep the locality-set reference probability constant. In 1972 Chu and Opderbeck observed that WS generates

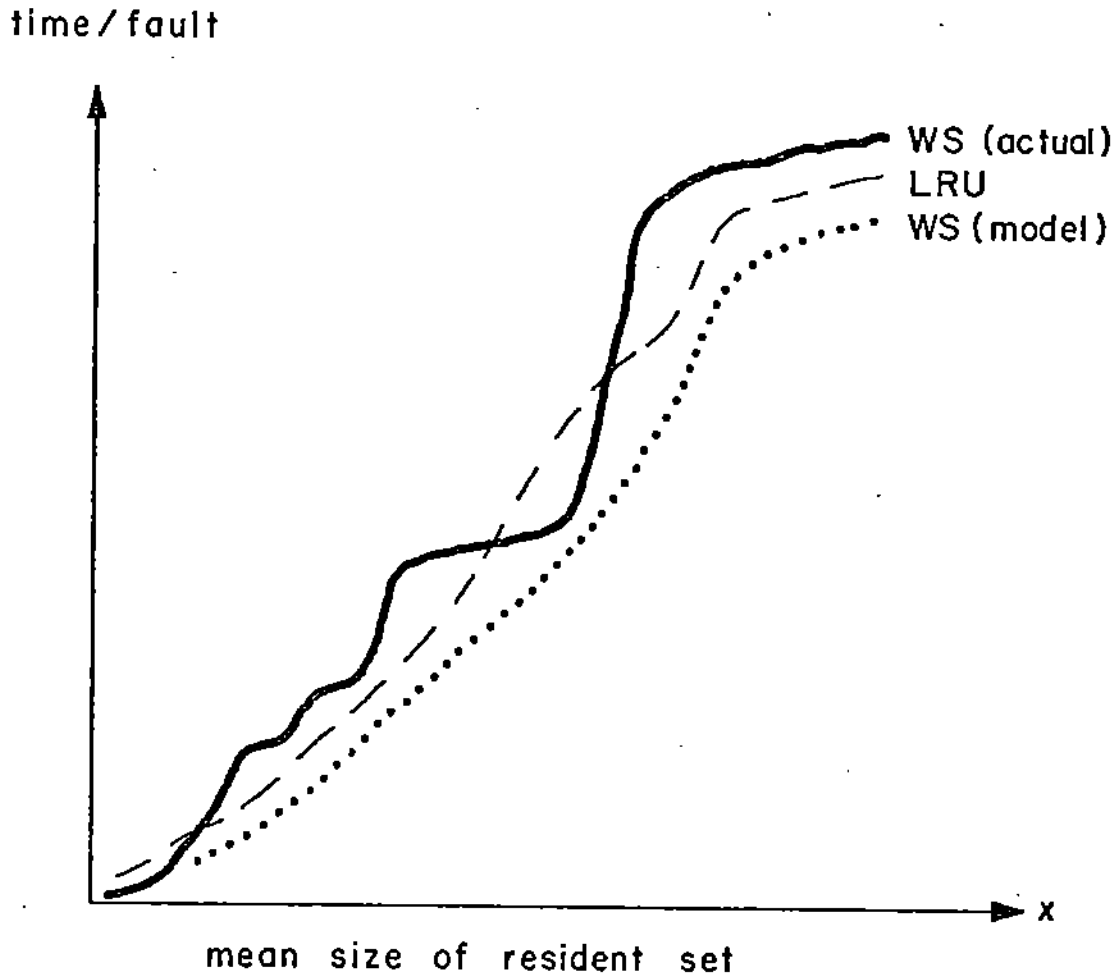


FIGURE 4. Comparison of lifetime curves.

lower space-time than the least space-time generable on the LRU policy; this could be explained only by supposing that the locality set size changes (see CHU72 and also GRAH76). In 1975 Graham and I presented examples of page reference strings over locality sets of different sizes; even though we chose these strings so that LRU would be the optimal policy for fixed memory space, WS produced less paging for some mean resident set sizes [DENN75a].

Despite these setbacks, IRM and LRUSM have not been written off; they may still be of some use for modeling program behavior within phases. (See DENN72b, DENN75a, DENN78a, GRAH76, SPIR72.)

Based on all these studies, I had, by the middle of 1974, reached this conclusion about program models: A realistic model must account for multiple program phases over locality sets of significantly different sizes and must not rule out strong correlations between distant phases. Not only had the SRM, IRM and LRUSM failed to tell much about whether working set memory management is optimal for real programs, but they failed to capture the essence of program behavior, the changing need for memory from one phase to another.

Phase-Transition Models

In the fall of 1974, Kahn and I undertook an experimental study to test the importance of transitions between locality sets of different sizes. We used a program model to generate reference strings, for which we measured LRU and WS lifetime curves. The program model comprised a macromodel and a micromodel. The macromodel was a semi-Markov chain whose "states" were mutually disjoint locality sets and "holding times" were phases. The macromodel was used to generate a sequence of locality-set/holding-

time pairs (S,T), the micromodel was used to generate a reference substring of length T over the pages of the locality set S; choices for the micromodel included the IRM, the LRUSM, and cyclic referencing.

Although we made no attempt to choose states, holding times, and other model parameters from real programs, we found that this model was better able to reproduce features of real programs than the IRM or the LRUSM. This model could be made to exhibit working set size distributions similar to those observed in practice. It was able to reproduce the behavior always observed for real programs, the dominance of WS lifetime over LRU in the vicinity of the knees (Figure 4); changing the micromodel for a given macromodel did not significantly affect this pattern. If the locality set sizes had a sufficiently small coefficient of variation, the WS dominance disappeared. (See DENN75c.)

At about the same time, Courtois and Vantilborgh applied the concepts of decomposition to a program model that treated pages as the states of a Markov chain [COUR76, COUR77]. By assuming that pages could be aggregated into weakly-interacting, mutually disjoint sets, they showed how to compute mean locality set size, paging rate, and an estimate of the distribution of working set size. Their calculations also revealed that this model was capable of reproducing the multimodal distributions of working set size already observed in practice.

Also at about the same time Batson and Madison undertook experimental studies of phases and transitions in the symbolic reference strings of real programs. (See MADI76, BATS76a,b.) They defined a phase as a maximal interval during which a given

set of segments, each referenced at least once, stayed on top of the LRU stack.* Their data revealed that:

1. Programs have marked phase behavior, that smaller phases over locality subsets are nested inside larger ones, and that there are significant disruptive transition-periods between major phases.
2. 90% of virtual time was covered by phases lasting 10^5 references or more; over 90% of the phases were fleeting and embedded within transition-periods between the long phases.
3. There is little correlation between the locality set size before and after a transition.

These three studies came independently to the same conclusions: phases and transitions are of equal importance in program behavior -- long phases dominate virtual time, as anticipated by the earliest virtual memory engineers, and transitions, being unpredictable, account for a substantial part of the missing segment faults. Moreover, decomposition is the appropriate analytic tool for program models.

As part of his doctoral research, Kahn [KAHN76] devised a filter that would classify the page faults generated by WS for real programs as "transition faults" or "phase faults"; a series of faults in close succession was treated as a sequence of transition faults. He found:

---

*Batson and Madison used the term "bounded locality interval" to refer to the combination of a phase and its locality set, and the term "activity set" where I would use "locality set". Locality sets are not the same as working sets [LENF78].

1. Phases covered at least 98% of virtual time.
2. 40% to 50% of the WS page faults occurred in transition periods; thus about half the paging occurred in about 2% of the virtual time.
3. The same phases were observed by the WS policy over wide ranges of its control parameter (Q).
4. Fault rates in transitions were 100 to 1000 times higher than fault rates in phases.
5. Successive interfault intervals in phases had strong serial correlations. In contrast, the number of faults in a transition was (approximately) geometrically distributed, and the lengths of interfault intervals within transitions were (approximately) exponentially distributed (with mean about 25 references).

These findings are corroborated by those of Jamp and Spirn, who used abrupt changes in VMIN resident set sizes as a criterion for detecting transitions [JAMP79]. The last finding corroborates Batson's [BATS76a], that transitions tend to be random in behavior. A similar effect has been observed in data base reference strings during the intervals between "reference clusters" [EAST78].

Kahn also suggested that decomposition of program data into phases and transitions can be used to simplify queueing network models of computer systems. Rather than treat a set of N active programs as N job-classes with different lifetime curves, the analyst can use just two job-classes: the jobs in phases, and the jobs in transitions. Jobs can change between these two classes. Kahn derived the parameters of the model from phase/transition data taken from real programs and used it to confirm that the optimal multiprogramming level tends to be the highest load at which two or more jobs are rarely observed in the transition class at the same time.

Tsur [TSUR78] and Simon [SIMO79], who also used multiclass queueing network models, came to the same conclusion: the performance of the model depends significantly on the assumptions one makes about the phases and transitions of programs. It appears necessary to obtain the parameters by decomposing the reference string into phases and transitions, using methods such as devised by Madison and Batson [MADI76] or by Kahn [KAHN76]. Graham reports an unsatisfactory attempt to generate artificial reference strings by using an LRUSM micromodel with a WS macromodel [GRAH76]; Spirn likewise reports an unsatisfactory attempt to disrupt an LRUSM by occasionally switching to a different set of distance frequencies [SPIR72].

I note in passing that several authors have used the Belady model or the Chamberlin model of the lifetime curve to derive parameters for queueing network models of multiprogramming (e.g., BRAN74, BRAN77, GELE73a GELE78, PARE77, POIT73, TSUR78). Inasmuch as optimal operation seems to be correlated with operating a program at the primary knee of the lifetime function [GRAH77], and inasmuch as neither of these life time models accurately represents actual knees, some skepticism is in order until someone shows that these models lead to exactly the same conclusions as when parameters are derived from the real data.

Despite the omission of transition behavior from early models of program behavior, many conjectures about the relative merits of the several memory policies have proved to be substantially correct. The reason is that transition periods, which cannot be anticipated by nonlookahead memory policies, affect all these policies in the same way. Differences among the memory policies therefore result from their behavior during phases, to which the slow-drift concept of locality does apply.

OPTIMAL MEMORY MANAGEMENT

A goal of constant interest to me since 1968 has been substantiating my conjecture that working set memory management can be "tuned" for near-optimum performance. This goal was achieved only recently. Because it requires both queueing network models of systems and phase-transition models of programs, it could not have been achieved sooner.

For a long time I was enamored of convexity arguments and probabilistic inequalities as an approach to showing the superiority of working set memory management. Inspired by Belady [BELA67] I worked out a marginally convincing argument that variable-space policies are more efficient than fixed-space policies [DENN68b]; this argument was based on the convexity of the working set size function $s(\theta)$. Spirn and I extended this line of argument [DENN73, SPIR77] and Spirn later pushed it to its limit [SPIR79], but even so the conditions under which the analysis applies are difficult to verify in practice.

In 1968 I also showed that, to achieve the same overflow probability, a memory containing N programs under fixed partitioning need be at least $N^{1/2}$ times larger than a variably partitioned memory holding the same programs [DENN68b, DENN69]. In 1972 Coffman and Ryan used the assumption of normally distributed working set size to prove much tighter bounds on overflow probability and to compute the mean amount by which demand exceeds available memory space (see COFF72 and also COFF73).

These arguments focus only fuzzily on the system's performance, leading to qualitative conclusions like working set memory management gives "higher" CPU utilization [DENN73] or "better" space utilization [COFF72] than fixed partition policies. Wanting

more precision, I abondoned this line of investigation in 1973. I turned to queueing network models, which can focus sharply on the relation between program behavior and a system's performance. In the following discussion I will emphasize the important role queueing network models have played in the theory of memory management. (An operational overview of these models is in DENN78c.)

Buzen was among the first to show how to use queueing network models to study optimal degrees of multiprogramming; this was a departure from the traditional use of these models because some of the parameters, such as the paging rates, could depend on the size of the load [BUZE71]. Courtois combined the principle of decomposition with queueing networks to develop the first rigorous analysis of instability and of thrashing (See COUR72, COUR75, and COUR77). The first explicit attempts to study optimal controls on the multiprogramming level were made by Brandwajn [BRAN74] and by Badel, Gelenbe, Lenfant, and Potier [BADE75].

Queueing Network Models of Multiprogramming

A queueing network model of a computer system specifies the configuration of a set of devices, each representing the queueing for a particular type of resource such as CPU, I/O, file storage, or page swapping. The parameters of simplest models are

N - The multiprogramming level (MPL), or load on the system;

D_i - The demand per job for the i^{th} device -- i.e., the mean total time required by each job for the device.

The demand per job (D_i) for a device is the product of the mean number of requests per job for that device and the mean time to

service one request. For devices such as CPU, I/O, and file storage the demand per job does not depend on N. But for the paging device, the demand per job grows with N because higher MPLs imply smaller resident sets and higher rates of paging.

The demand for the CPU is the mean execution time, E, of a job. The mean number of page faults per job is E/L(N), where L(N) denotes the lifetime, or mean-time-between-faults, for MPL N. The demand for the paging device is D_i = ES/L(N), where S is the mean time to service one page swap (exclusive of queueing delays).

If the curve L(N) is not available from a direct measurement of the system it can be estimated from the lifetime curve of a typical program. The most common method when P pages of main memory are available is to set L(N) = g(P/N), where g(x) is the mean time between faults measured for a typical program when the given memory policy produces mean resident set size of x pages. If the memory policy maintains a pool of unallocated page frames, the available memory is, approximately, this fraction of the actual memory:

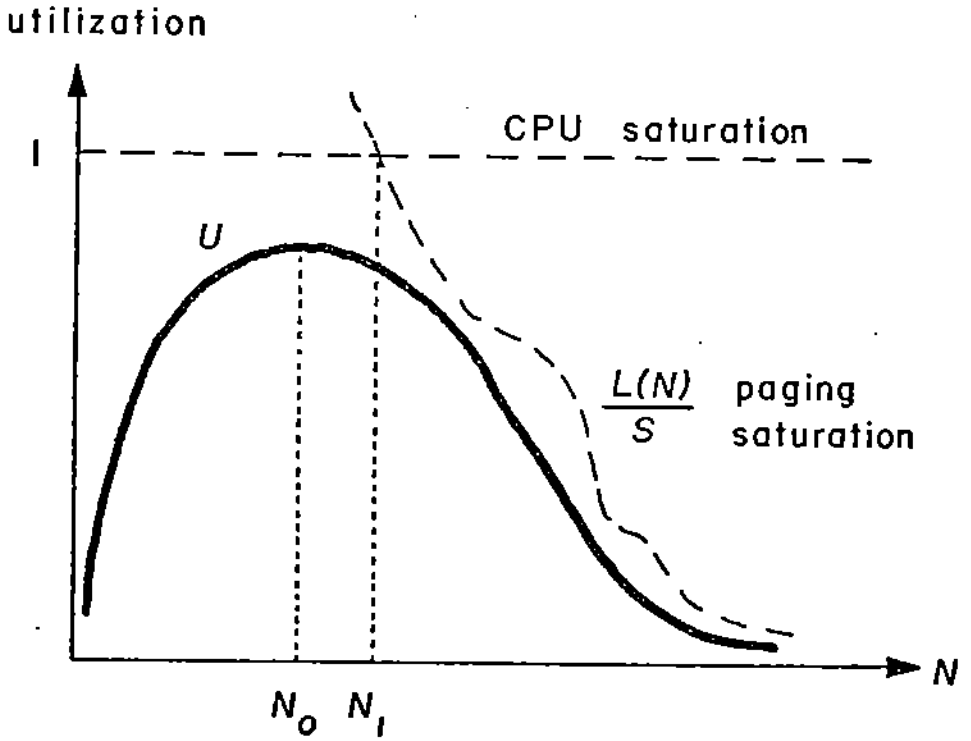
$$\frac{N}{N + (C^2+1)/2} ,$$

where C is the coefficient of variation of the resident set size of a program over time.* (For working sets, C is less than 0.3 [RODR73b].) Simon validated this formula by comparing queueing network and simula-

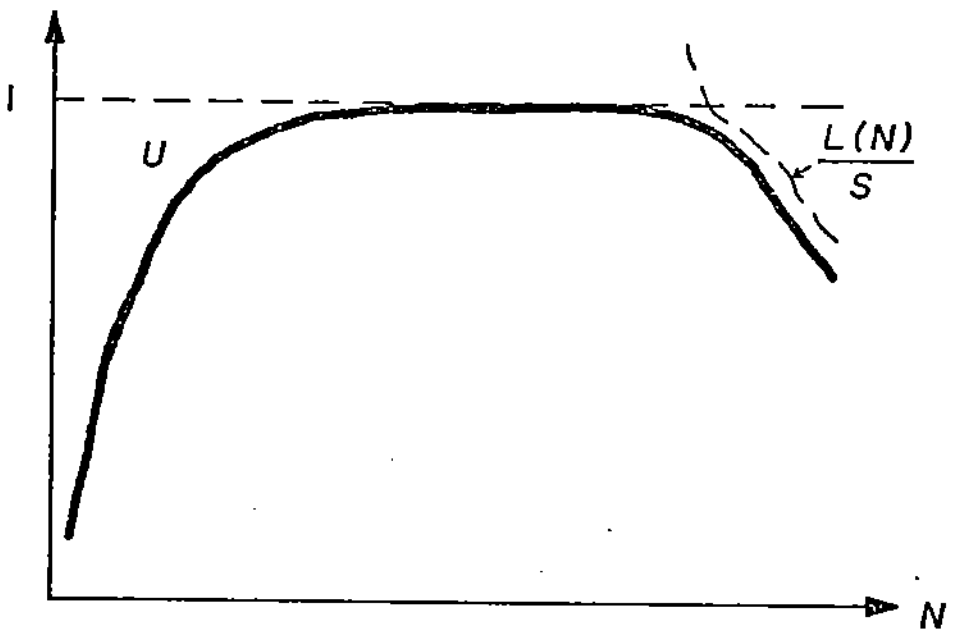
*C is the ratio of standard deviation to the mean. For a program whose resident set size at time t is x(t), the mean m is the average of x(t) over all t and C^2 is the average of $(x(t)-m)^2/m^2$ over all t. The small, time-weighted variations of working set size typically observed within one program [RODR73b] should not be confused with the large variations among the working set sizes observed among different programs.

tion models for various workloads [SIMO79].

Queueing network models estimate the system's throughput, X_0, the number of jobs per second being completed. The throughput is proportional to the utilization of the CPU, U. (In fact, $U = X_0E$; see BRAN74, COUR77, DENN75a, DENN76b, or Denn78a.) Figure 5(a) illustrates a typical CPU utilization curve as a function of the MPL, N, for a fixed size of main memory. The curve rises toward CPU saturation but is eventually depressed by the ratio L(N)/S, the utilization of the saturated paging device. The curve of Figure 5(a) is valid under general conditions that apply to almost all real multiprogramming systmes [BADE75, COUR72, DENN75a, DENN76a, DENN78c].



(a) Small main memory



(b) Large main memory

FIGURE 5. Optimum multiprogramming levels.

The existence of an optimum MPL was known long before queueing network models were used to characterize it precisely. I argued in 1968 that an optimum MPL would exist [DENN68b,c]. Simulations of the RCA Spectra/70 confirmed this [WEIZ69]. An extensive study of a CP-67 system demonstrated that a working set dispatcher could control the MPL for maximum CPU utilization [RODR73a], a finding reconfirmed on the Edinburgh Multi Access System (EMAS) [ADAM75].

Figure 5(a) suggests that N_1, the MPL at which $L = S$, is slightly larger than the optimum N_0. Using this as a starting point, Kahn and I carried out an experimental study which revealed that, indeed, this "L=S criterion" could be used as an adaptive load control [DENN76a].

The intuition underlying the "L=S criterion" also illuminates an interesting tradeoff resulting from the size of the main memory. A very large main memory buffers against instabilities in memory policies and overheads created when the resident sets attempt to overflow the space available. By staving off the effect of swapping overhead, the very large main memory transforms the CPU utilization bounds to the form shown in Figure 5(b). Once the main memory is large enough to allow the CPU utilization to be near 100% for some N, further increases of memory cannot increase the system throughput or decrease response time. In effect, the very large main memory serves as a job queue -- holding waiting jobs in secondary memory may be cheaper.

Working independently, Leroudier and Potier discovered that CPU utilization tends to be maximum when the utilization of the paging device is approximately 50% -- which will occur when the mean queue

there is approximately 1.0, the onset of thrashing [LERO76b].* Suri observed that the "L=S criterion" and this "50% criterion" are closely related. We pooled our findings in a joint paper and concluded that adaptive load controls can be both simple and effective [DENN76b]. Simon has since found that the target value of utilization for the paging device should actually be (50+A)%, where A is the utilization due to preloading working sets as jobs are (re)activated; A in the order of 25% may be typical [SIMO79]. (The previous studies had not considered preloading, which does not affect the "L=S criterion".) Gelenbe, Kurinckx, and Mitrani have studied controllers that add load at a rate proportional to the CPU utilization; this will reduce the load as the CPU utilization drops at the onset of thrashing [GELE78a,b].

Optimal Load Control

The intuition of Figure 5 -- that the optimum MPL is characterized by the relation L = aS for some constant a -- is crude. It fails when the system is I/O bound or when the maximum lifetime L does not exceed the segment swapping time S [DENN76b]. The optimum MPL is actually associated with running each job at its minimum space-time product, which is more difficult to achieve than L = aS.

If the system's throughput is X_0 jobs per second over an observation period of T seconds, then X_0T jobs are completed. If the main memory has capacity P words, there are PT word-seconds of main memory space-time available. Therefore the memory space-time per job, ST,

*This intuition has been confirmed by measurements in real systems, notably MULTICS and EMAS (see ADAM75, DENN76b, LERO76b, and POTI77). It has also been confirmed by Kahn's two class queueing network model (one class for programs in phases, the other for programs in transitions) [KAHN76]; Kahn observed that at the optimum load the probability of finding two or more jobs together in the transition class is small and the mean queue at the swapping device was near 1.

is

$$ST = PT/X_0T = P/X_0 \quad \text{word-seconds.}$$

It follows that the optimum MPL, N_0, maximizes throughput and minimizes memory space-time per job. (See also BUZE76.)

As noted earlier, if the total delay per segment fault (queueing time plus swap time S) is large, the space-time will be minimized approximately at the primary knee of the lifetime curve. Graham confirmed this intuition: direct measurements of 8 real programs showed the resident set size of the primary knee of the WS policy to be within 2% of the resident set size that minimizes space-time; see Figure 6 [GRAH76, GRAH77]. (This led to the "knee criterion", a basis for load control which is more robust than either the "L=S criterion" or the "50% criterion" [DENN76b, GRAH77].)

To limit the drop of CPU utilization under an excessive MPL (thrashing), most operating systems partition the submitted jobs into the active and inactive jobs. Only the active jobs may hold space in main memory and use the CPU or I/O devices. (See Figure 7.) There is a maximum limit, M, on the size of the MPL. If the number of submitted jobs at a given time does not exceed M, all are active; otherwise, the excess jobs are held, inactive, in a memory queue. The limiting effect of the memory queue is sketched in Figure 8. (See COUR75.) Evidently, if M were set to N_0, thrashing could not occur at all and the system would operate at optimum throughput whenever a sufficient number of jobs were submitted. In practice, the optimum MPL varies with the workload; hence an adaptive control is needed to adjust M. Fixing M at the smallest possible value of N_0 is usually unsuitable, for this will cause the system to be underloaded

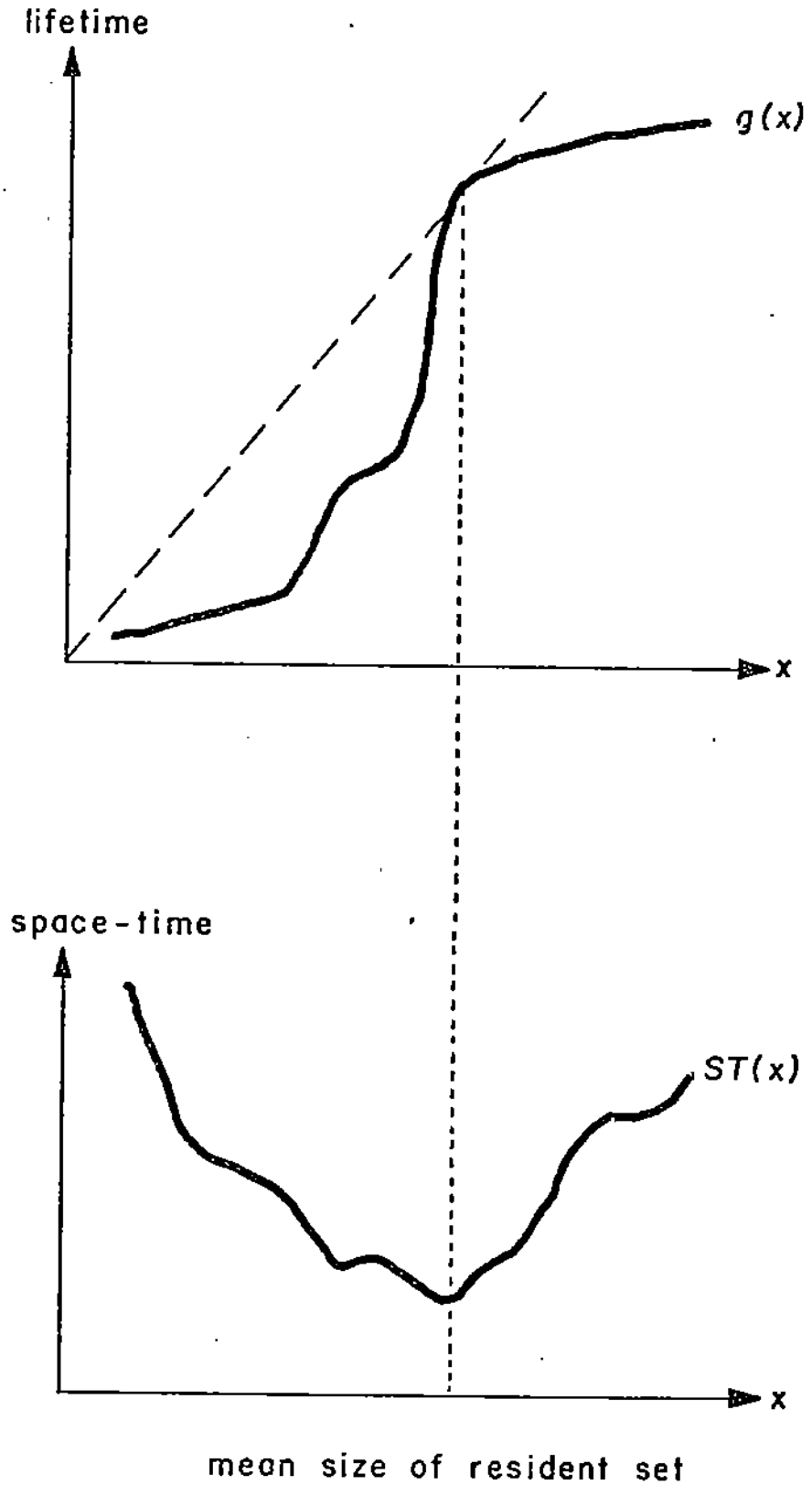


FIGURE 6. Lifetime knee and space-time minimum.

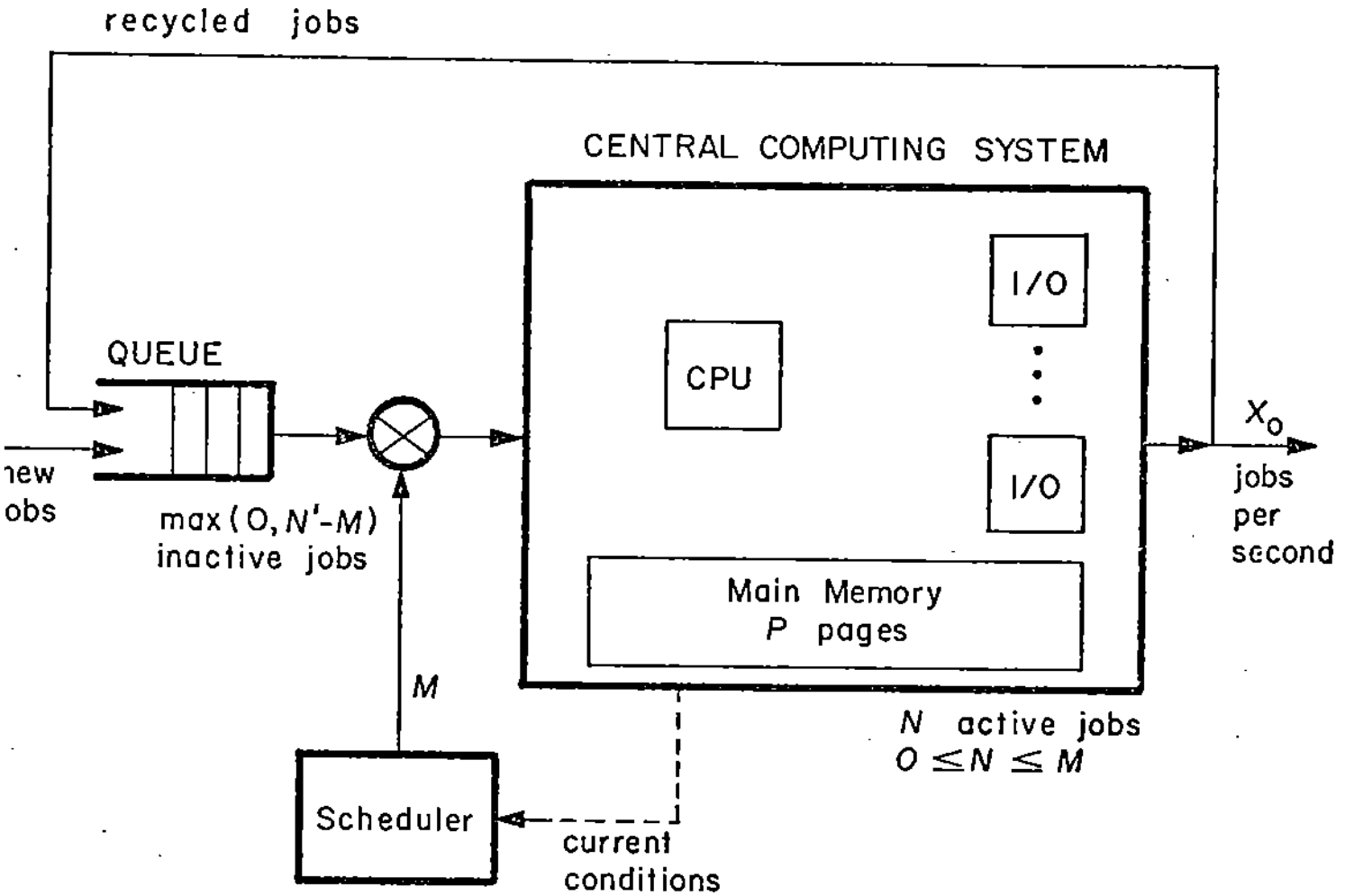
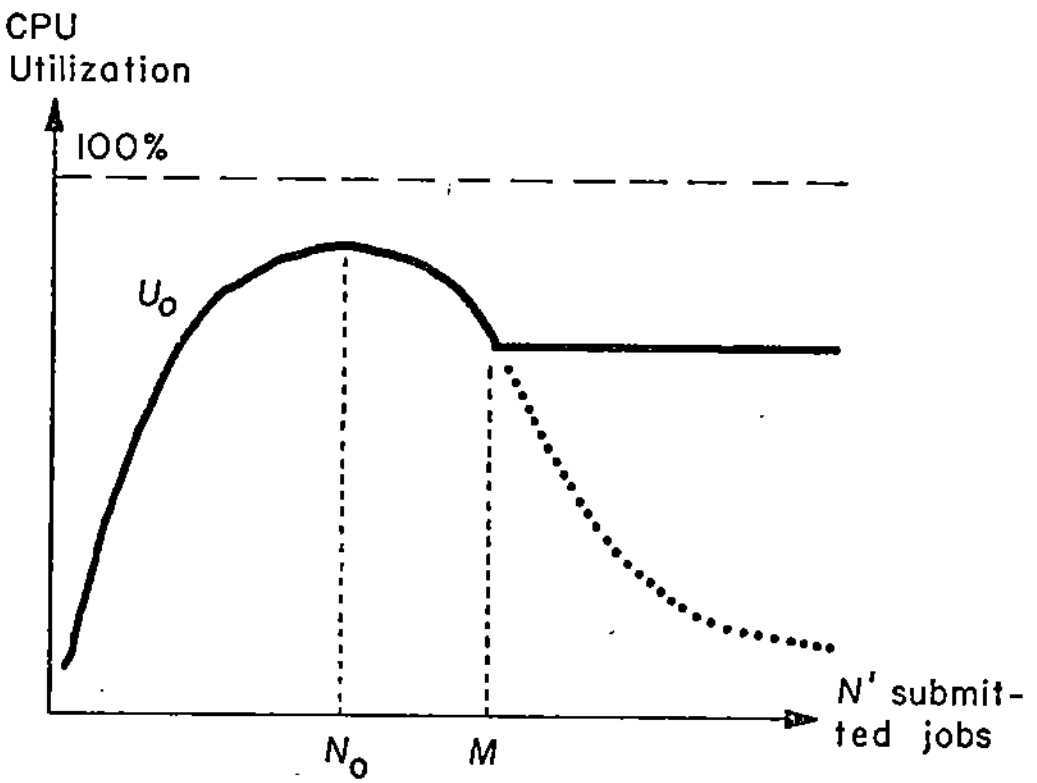


FIGURE 7. Limiting the load on the central computing system by enqueueing excess submitted jobs.



FUGURE 8. Effect of the load control on CPU utilization.

most of the time.

To summarize: the load controller seeks to set the maximum MPL, M, near the current optimum. The optimum MPL is achieved by minimizing the space-time product per program, which in most cases is equivalent to operating each program at the primary knee of the program's lifetime curve for the given memory policy.

Dispatchers for Multiprogrammed Computer Systems

The purpose of the dispatcher is to control the scheduling of jobs and allocation of main memory so that the throughput for each workload (MVS "performance group" [BUZE78]) is maximum. The dispatcher contains three components: the scheduler, the memory policy, and the load controller.

The scheduler determines the composition of the active set of jobs. It does this by activating jobs (moving them from the memory queue into the active set -- see Figure 7) and setting a limit on the time a job may stay active. Normally the next job to be activated is the one with highest priority among those waiting. When there are multiple workloads, part of the memory is reserved for each and there is a separate scheduler for each workload.

The memory policy determines a resident set for each active job. Two broad classes of memory policies are in use. The global policies partition the memory among the active programs according to procedures that depend on the aggregate behavior of all active programs; the local policies determine a separate resident set for each program by observing that program in its own virtual time independently of the other programs. Examples of global and local policies will be given in the next section.

All memory policies manage a pool of unused space in main memory. The pool contains the pages of resident sets of recently deactivated jobs; under a local policy, the pool also contains pages which have recently left the resident sets of active jobs. In most VM/370 systems, which use a global policy (described below), the pool is empty less than 20% of the time. In systems with a working-set dispatcher, which is a local policy, the pool is likely to be empty less than 5% of the time.

The load controller adjusts the limit M on the multiprogramming level. The ideal value of M is the current optimum N_0. This limit is a function of "current conditions" in the active computing subsystem (see Figure 7). If a global policy is in use, the "current conditions" will be an aggregate measure of the total workload's demand for swapping; for example, this measure can be the current aggregate value of lifetime (for the "L=S criterion") or the utilization of the swapping device (for the "50% criterion"). If a local policy is in use, the "current conditions" will be the size of the pool.

Sometimes a static load control is proposed. To avoid thrashing, the fixed limit M must be set near the smallest value N_0 is likely to take. As a result, the chronically underloaded system will deliver unduly low throughput. The VM/370 system, for example, uses a dynamic limit M determined from estimates of active jobs' working sets; this system is more efficient than its predecessor, Release 2 of CP-67, which used a fixed limit M.

By comparing the measured memory demand of a job with the pool's size, the scheduler avoids activating a job if the activation would overload the system. Therefore, a dispatcher based on a local policy

actually employs a "feedforward control," rather than the "feedback control" suggested in Figure 7. Feedforward controls are inherently more stable than feedback controls. This is so because a dispatcher based on a local memory policy can prevent overload, whereas a dispatcher based on a global policy can only react, after the fact, to an overload.

Memory Policies

This section describes four common memory policies -- two of the global type and two of the local type. They will be described for paging systems, the context in which they have been analyzed, measured, and compared.

One global policy is LRU (least recently used). All the resident pages of all active jobs are listed in an LRU stack in order of decreasing recency of use. On a page fault, the resident page farthest down the stack is chosen for replacement. The CDC STAR-100 computer uses this scheme. MULTICS also uses it to control page migration between the drum and the disk [SALT74]. Systems using such policies are difficult to analyze [SMIT76c, d].

A widely-used global policy is called the CLOCK algorithm. On a page fault, a pointer resumes a cyclic scan through the page frames of main memory, skipping used frames and resetting their usage bits, selecting for replacement the page in the first unused frame. (The term "CLOCK" comes from the image of the pointer as the hand of a clock on whose circumference are the page frames.) This algorithm attempts to approximate LRU within the simple implementation of FIFO (first in first out). An early version was studied by Belady [BELA66]. It was being considered for MULTICS in 1967, under the working name "first in not used first out" (FINUFO) [DENN68b], and it has been

operational in MULTICS since 1969 [ORGA72]. It was used in an experimental version of CP-67 [BARD75] and is now in all released versions of VM/370. It is difficult to analyze [BARD75, EAST79].

Both global CLOCK and global LRU tend to favor the pages of the job using the CPU most recently and the job having the smallest locality set [DENN68a, b]. Therefore, a job's resident set depends on many factors resides its own locality -- these policies thrash easily and analyze poorly.

There is, unfortunately, little published performance data on the CLOCK and global LRU obtained from real systems in operation. Bard reported some data on CLOCK in a CP-67 [BARD75] but did not compare with other policies. An early study in MULTICS suggested that global CLOCK might be somewhat better than global LRU [CORB69]. Belady's data, however, suggest that CLOCK and LRU give similar results when applied to single programs [BELA66]. Graham's data shows that LRU is normally significantly worse than WS when applied to single programs [GRAH76]. Experience with Release 2 of CP-67 [RODR73a] and the EMAS [ADAM75, POTI77] suggests further that replacing a global policy with a WS policy can improve performance significantly.

The evidence available thus suggests that CLOCK and LRU do not perform as well as WS. This is because these global policies cannot ensure that the block of memory allocated to a program minimizes that program's space-time [DENN75a]. The main attraction of CLOCK is its apparently simple mechanism; but, as described below, its poorer performance and the additional mechanism for feedback control cancel this advantage.

The working set (WS) policy, which assigns each program a resident set identical to its working set, is an example of a local policy.

In 1972, Chu and Opderbeck proposed the page fault frequency (PFF) policy, which was to be an easily-implemented alternative to WS [CHU72]. PFF is designed to rely only on hardware usage bits and an interval timer, and it is invoked only at page fault times; thus it is easily incorporated into most existing operating systems built on conventional hardware. Let t' and t (t >t') denote two successive (virtual) times at which a page fault occurs in a given program; let R(t,θ) denote the PFF resident set just after time t, given that the control parameter of PFF has the value θ. Then

$$R(t,\theta) = \begin{cases} W(t,\ t-t'), & \text{if } t-t' > \theta \\ R(t',\theta) + r(t), & \text{otherwise} \end{cases}$$

where r(t) is the page referenced at time t (and found missing from the resident set). The idea is to use the interfault interval as a working-set window. The parameter θ acts a threshold to guard against underestimating the working set in case of a short interfault interval: if the interval is too short, the resident set is augmented by adding the faulting page r(t).* The usage bits, which are reset at each page fault, are used to determine the resident set if the timer reveals that the interfault interval exceeds the threshold. Note that 1/θ can be interpreted as the maximum tolerable frequency of page faults.

---

*In programs with strong phase behavior, PFF can have considerably higher space-time than WS. This is because bursts of short interfault intervals occurring at transitions will be followed by a long interfault interval spanning all (or part of) a phase; in the worst case, the PFF resident set will contain both the current and prior locality sets. For the same program, WS will remove all the old locality set's pages within θ time units after the transition completes. That PFF is less able than WS to track changing locality has been corroborated by Graham's experiments [GRAH76].

Various experimental studies have revealed that WS and PFF, when properly "tuned" by good choices of their control parameters, perform nearly the same and considerably better than LRU; WS has a slight tendency to produce lower space-time minima than PFF, but the differences are within 10% [CHU72, CHU76b, GRAH76, GRAH77]. However, PFF may display anomalies for certain programs -- i.e., the lifetime or mean resident set size (or both) may decrease for increasing θ [GRAH76, FRAN78]. This is because PFF does not satisfy the inclusion property. Moreover, the performance of PFF is much more sensitive to the choice of control parameter than is the performance of WS [GRAH76, GUPT78].

Controllability of Memory Policies

Since global memory policies make no distinctions among programs, their load controls (e.g., according to the "L=S criterion" or the "50% criterion") have no dynamically adjustable parameters; but these controls cannot ensure that each active program is allocated a space-time minimizing resident set. Local memory policies, such as WS and PFF, offer a much finer level of control and are capable of much better performance than global policies. However, these policies also present the problem of selecting a proper value of the control parameter θ, for each active program. The question of sensitivity to the control parameter setting is of central importance.

Since both WS and PFF space-time functions typically have flat minimal regions as functions of θ [CHU72, GRAH76], there is little point in considering policies that dynamically vary θ. The main problem is to associate a proper value of θ with a program as soon as it is activated.

At one extreme, we can design the policy so that each program is assigned a value of θ that minimizes its resident set's space-time product. We call this the (fully) tuned policy [SIMO79]. A tuned policy may have a high overhead in the mechanism that monitors each program and assigns the proper θ At the other extreme, we can design the policy to use one global θ for all programs. We call this the (fully) detuned policy. A detuned policy has no overhead in θ-detection -- but this may be at the cost of operating some programs far from their space-time minima and, hence at the risk of thrashing. As a compromise we can design a p% detuned policy that assigns each program a θ for operation with p% of its minimum space-time.

Graham experimented with 8 programs in order to determine the sensitivity of WS and PFF to detuning the control parameter [GRAH77]. Two questions were asked:

1. For the given programs and a given value of p, what is a minimal set of θ-values for p% detuned operation? (The size of this minimal set represents the least number of choices that a θ-detector must make for a given program to achieve system throughput no worse than p% from optimum.)

2. If one best global θ-value is used for all programs in the sample, what is the largest difference from minimum space-time that must be tolerated? Specifically, what is the smallest p such that the p% detuned policy is fully detuned?

To answer these questions, Graham constructed diagrams like Figure 9, in which the horizontal bars represent ranges of θ-values in which space-time is within p% of minimum. He visually located minimal sets of θ-values by finding sets of vertical bars that cut all the horizontal bars. He found these sizes of the minimal sets of θ-values:

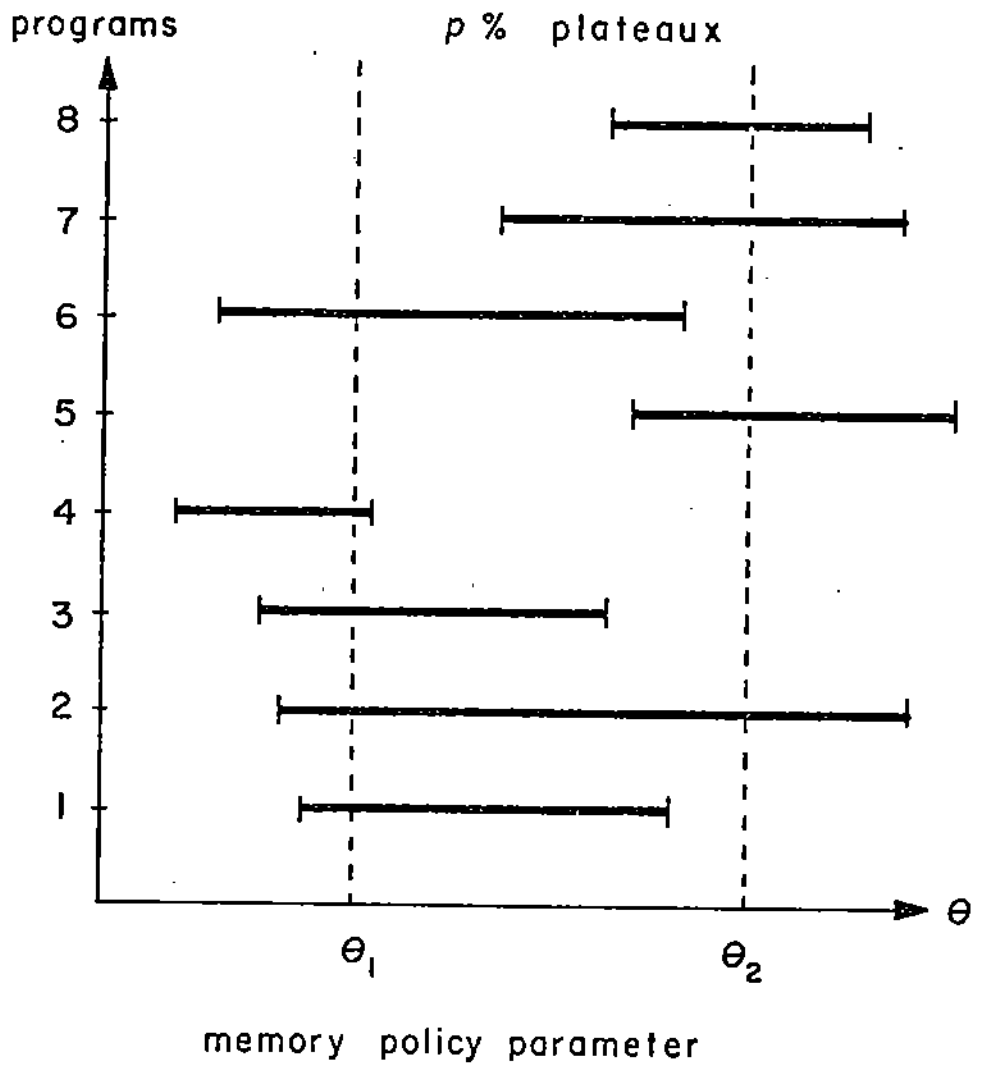


FIGURE 9. Minimal covering set for a given value of p.

clusion has been reached by Gupta and Franklin [GUPT78].)

Assuming that these characteristics are typical, the Q-detector that endows PFF with performance similar to a single-Q WS makes a multiprogrammed PFF at least as expensive as a multiprogrammed WS. Where it has been substituted for CLOCK or LRU, a detuned WS has performed significantly better than the original global policy.

Are There Better Policies?

Do there exist memory policies that perform significantly better than properly tuned WS without costing significantly more? No one as found such a policy. The operation of the VMIN optimal policy on programs with marked phase behavior suggests that it is unlikely that anyone will ever find such a policy.

Recall that the VMIN policy uses its parameter Q to select one of two choices for each reference r(t): if the forward interval to the next reference or segment r(t) exceeds Q, r(t) is removed immediately after time t, to be reclaimed later when needed by a fault; otherwise r(t) is kept resident until its next reference [PRIE76, DENN78b]. For each mean resident set size, VMIN produces the smallest possible fault rate.

As suggested in Figure 10, VMIN anticipates a transition into a new phase by removing each old segment from residence after its last reference prior to the transition; in contrast, WS retains each segment for as long as Q time units after the transition. This behavior has been confirmed experimentally by Jamp and Spirn [JAMP79]. Since VMIN and WS generate exactly the same sequence of segment faults [DENN78a,b], the suboptimality of WS results from resident set "overshoot" at interphase transitions.

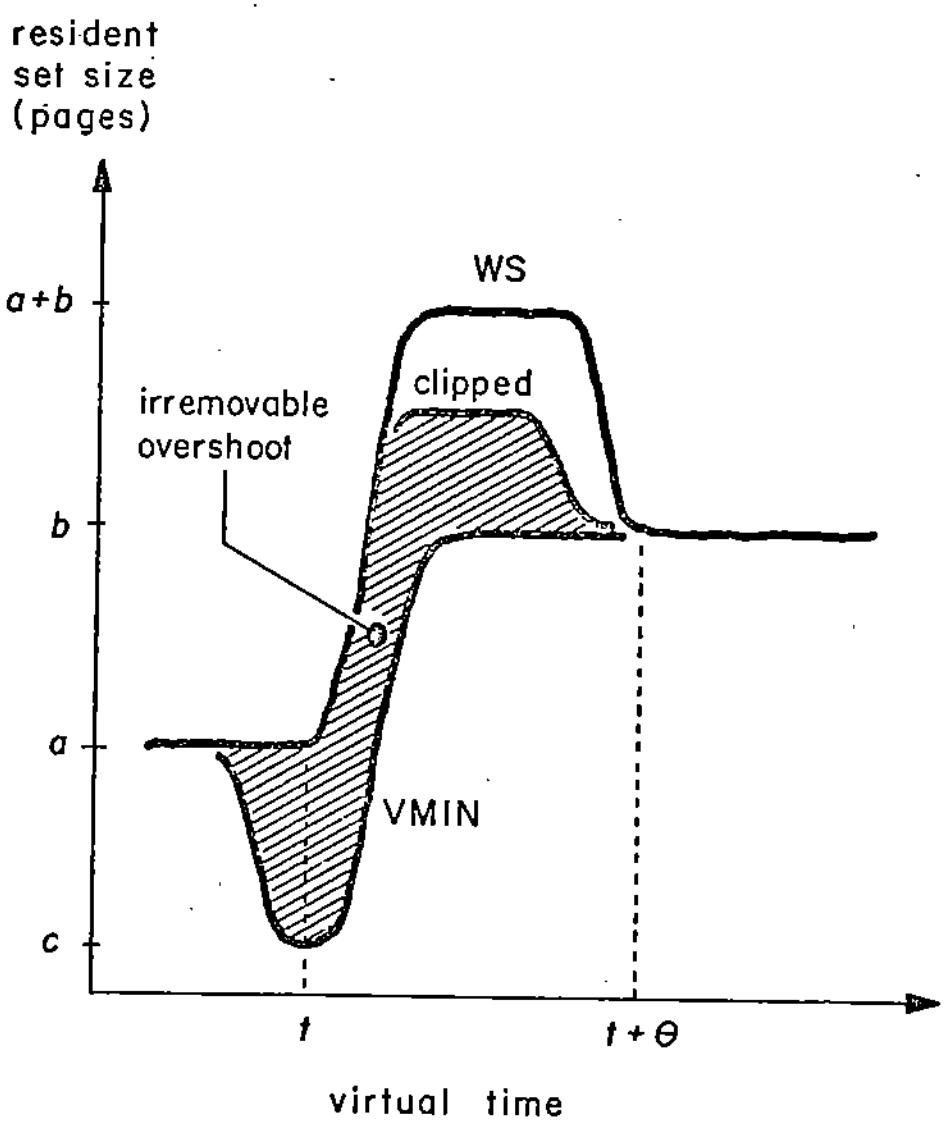


FIGURE 10. Behavior of policies near a transition between phases.

The only way to make WS more like VMIN is to "clip off" the overshoot. Smith's method [SMIT76a] typically reduces space-time by less than 5% [GRAH76]. No method is likely to do better because it is impossible without lookahead to tell that a transition is in progress until it has generated a few stoccato faults; by the time a "clipping action" is begun, a good deal of overshoot will already have occurred. Smith's suggested application of clipping is not space-time reduction, but controlling the overhead caused by temporary memory overflow.

Simon compared the optimum throughput from the tuned WS policy to the optimum from the VMIN policy [SIMO79]. He found that VMIN improved the optimum throughput from 5% to 30% depending on the workload, the average improvement being about 10%. He estimated that improvements from the best possible clipped WS policy would average less than 5%. 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.

Batson has suggested that analysis of cycles* in programs may reveal the program's locality sets and phases [BATS77]. If this is so, it may be possible for a compiler to implant instructions that advise the memory policy when a given segment has been referenced for the last time in a phase, thereby allowing the memory policy to behave more like VMIN as in Figure 10.

---

*In this context a "cycle" over a set of segments is a minimal reference substring mentioning each member of the set at least once. Easton has used a similar definition ("cluster") to analyze use-bit scanning policies [EAST76] and locality [EAST78b].

A somewhat more subtle argument for the optimality of WS is a property of ideal phase-transition programs called "space-time dominance" [DENN78a]; this property states that a memory policy capable of tracking the locality set exactly will generate the least space-time product among all nonlookahead policies. If the mean holding time in a phase is long compared to the working set parameter θ, and if the mean time between two references to a locality-set segment is short compared to θ, WS will also be dominant in space-time. However, it is not known how many real programs satisfy these properties.

This evidence has convinced me that it is unlikely that anyone will discover a nonlookahead policy that consistently produces

signficantly lower space-time that the WS policy on real programs. A working set policy will generate near-maximum throughput among all possible nonlookahead policies.

IMPLEMENTING A WORKING SET DISPATCHER

Among the more interesting practical implementations of WS memory management are the special hardware designed by Morris for the MANIAC II [MORR72], the dispatcher for the Edinburgh Multi Access System (EMAS) [ADAM75], and the dispatcher designed by Rodriguez and Dupuy for a CP-67 system [RODR73]. Variants of the WS policy are used in Univac's VMOS [FOGE74] and IBM's MVS [BUZE78, CHIU78]. The CP-67 dispatcher [RODR73] showed that a WS policy can be implemented easily and cheaply in the context of a traditional operating system, even though the only "memory management hardware" is usage bits.

Recent technological advances make working-set detecting hardware, such as Morris proposed [MORR72], even more attractive. Such hardware would simplify the operating system and reduce the overheads of job scheduling and memory management; it would do this by replacing a considerable amount of mechanism that would otherwise be in the operating system software. Following is a description of a working set dispatcher for paging; it combines ideas from the CP-67 and the MANIAC II implementations.

Overview of the Dispatcher

The working set dispatcher, which comprises the scheduler and the working set detector, controls the transitions of processes among five states. (See Figure 11.) A process is entitled to use

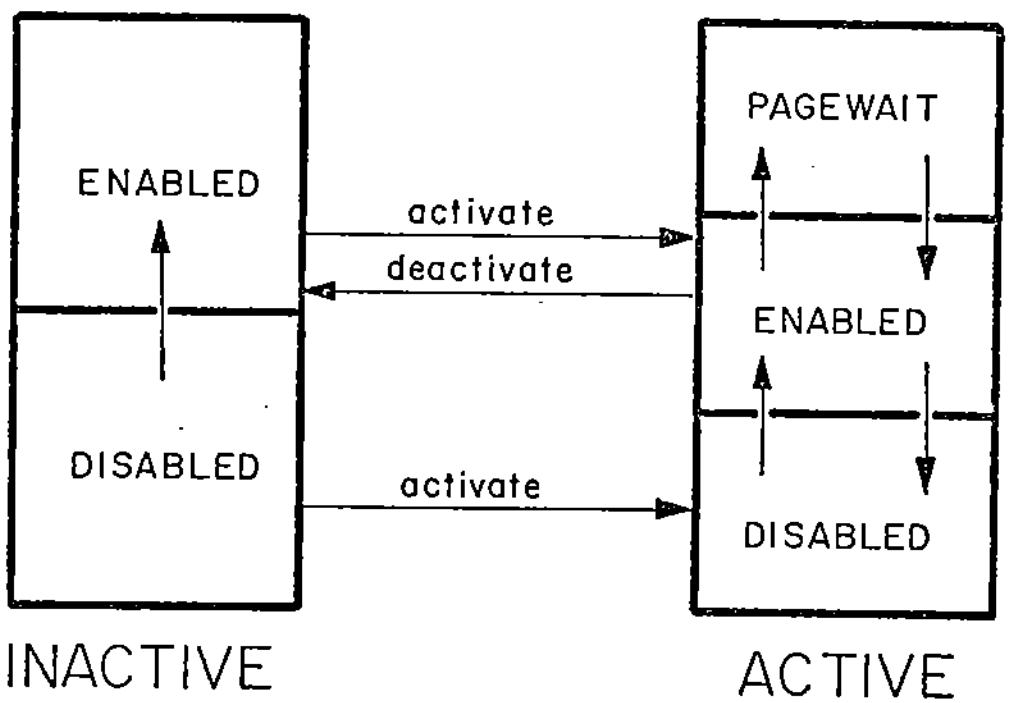


FIGURE 11. Process states in working set dispatcher.

CPU, I/O devices, and main memory only when in the ACTIVE state. A process is entitled to execute its next instruction only when in the ENABLED state; when DISABLED, a process is waiting for another process to signal it via a semaphore. An ACTIVE-ENABLED process enters the ACTIVE-DISABLED state only when it executes a wait operation on a semaphore.

This state description actually encompasses a three-level hierarchy. At the highest level are the "superstates" ACTIVE and INACTIVE. At the intermediate level are the components of these "superstates" -- ENABLED, DISABLED, and PAGEWAIT. At the lowest level (not shown in Figure 11) are components of DISABLED, substates indicating precisely which semaphore is delaying a process.

Many systems allow for multiple job classes to which guaranteed resources are available. In this case there would be one ACTIVE "superstate" for each job class, and a separate working set dispatcher for each job class. (The "domain" structure of MVS illustrates this; see BUZE78 and CHIU78.)

The working set detector maintains the pool, which is a list of available page frames, and a count K of the pool's (nonnegative) size. The scheduler may active the highest priority INACTIVE-ENABLED process only if that process'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 process is made ACTIVE. Chu and Opderbeck, who assumed that memory overflow triggers the deactivation of a job, found that K_0 near 10 pages was sufficient [OPDE74]; Simon,

who assumed that memory overflow triggered the preemption of a page from the largest resident set, found that $K_0 = 4$ was sufficient [SIM079]. Note that $K < K_0$ may occur because working sets may expand after loading.

The page fault handler program puts a faulting process in the ACTIVE-PAGEWAIT state until the missing page is loaded in main memory. Before requesting that the auxiliary memory device load the missing page, the page fault handler obtains a free page from the pool and subtracts 1 from the count K. If K is already 0 the page fault handler will first cause the working set detector to preempt a page from the lowest priority ACTIVE working set; this implies that the lowest priority ACTIVE process may not have its working set fully resident. (See WILK73, SPIR73, and SIMP79.)

A deactivate decision may be issued by the time-quantum exception handler program (if a process uses up its time in the ACTIVE state), by the wait-semaphore operation (if a process stops to wait on a semaphore on which the delay is indefinite), or by the page fault handler (if the lowest priority ACTIVE process has its resident set reduced to naught). In the first two cases, the process's working set pages are released and returned to the pool. (Releasing pages entails swapping out those modified during their residences in main memory.)

The working set detector hardware can be patterned after Morris's [MORR72]. With each page frame of memory is associated an identifier register and a counter. The identifier register contains the index number of the protection domain in which the page was most recently referenced, or a zero if the page frame is free. (Note that shared pages may, from the viewpoint of the working set detector, appear to change domains. This does not affect access control, which is enforced separately in the virtual addressing mechanism.) At regular

intervals, a broadcast clock pulse increments the counter of each page frame whose identifier register matches the domain of the process presently running on the CPU. (This causes the counters to operate in virtual time.) When a counter overflows, the corresponding page frame is no longer a member of a working set. When the running process refers to a page, the counter of that page's frame is automatically reset, and the identifier register of that page's frame is set to match the "current domain" register in the CPU.

If clock pulses are generated every H seconds and the counters contain k bits, this scheme implements the working set with $\theta = H \cdot 2^k$. The value of H, which can be stored in a register, can be "tuned" so that the WS policy exhibits best performance. The previous sections argue that one global value of H is sufficient for this purpose.

Cost of Implementation

Morris reported that the circuitry of the working set detector could be built for about $20 per page frame in the technology of 1972 [MORR72]. The same circuits would be considerably cheaper today, especially if incorporated in the initial design of the memory hardware. Microcomputers could be used to initiate swapouts of pages released from working sets and to maintain the pool. The cost of these microcomputers should be less than the cost of the part of the operating system's software that they replace.

Because the parameter θ does not need readjustment for each new program, the clock pulse time H does not need to be updated each time the CPU is switched between processes. There is no need for a mechanism to measure a suitable clock pulse time

for each process.

This dispatcher is cost-effective because it is known to optimize performance. Its simple, cheap hardware replaces a considerable amount of mechanism for scheduling and load control which otherwise would be present in the operating system's software.

Comparison with Other Dispatchers

Three parts of a dispatcher depend on the underlying memory policy: 1) a mechanism for determining a value of the policy's control parameter (θ) for each process, 2) a mechanism for determining a process's resident set for the given θ, and 3) a load controller. The scheduler and the manager of the pool of free space have the same complexity in all dispatchers.

There is no formal "proof" that local memory policies are inherently more efficient than global ones. However, two real systems give compelling empirical demonstrations -- the Edinburgh Multi Access System (EMAS) [ADAM75] and the CP-67 at Grenoble [RODR73a]. Both these systems replaced a dispatcher based on a global memory policy (similar to "CLOCK") with a working set dispatcher and observed significant improvements in performance. This is direct evidence that the total system with a working set dispatcher, despite its apparent overheads, is nonetheless more efficient than the total system with the apparently simpler global memory policy.

Although the resident-set detecting hardware is simpler for PFF than for WS, the great parameter sensitivity of PFF necessitates a mechanism not needed in the WS dispatcher -- one for determining a θ-value for each program. The cost of the θ-detector cancels the apparent advantage of the PFF dispatcher. The total cost of the PFF dispatcher is higher than the total cost of a WS dispatcher

capable of the same performance.

It is sometimes argued that the CLOCK dispatcher is more efficient than WS for "data-stream programs" -- database programs that scan linearly through large amounts of data. This argument arises from the mental picture of a WS policy with a large window (Q) that acts as a long pipeline filled with useless pages from the data stream. In fact, the minimum WS space-time for data-stream programs occurs at a small window [DENN68b]. A properly tuned WS policy does not retain useless pages any longer than a CLOCK policy.

It is sometimes argued that, when software maintenance is taken into account, the simple CLOCK dispatcher is actually cheaper than a WS dispatcher. This was not considered a problem in either EMAS or CP-67 [ADAM75, RODR73a]. In fact, the strong hardware support of the WS dispatcher reduces the complexity of the operating system, thereby simplifying maintenance.

CONCLUSION

The working set dispatcher solves Saltzer's Problem.

This conclusion is not speculation. Experiments with real programs have revealed that the working set policy is the most likely, among (nonlookahead) policies, to generate minimum space-time for any given program; and that one properly chosen control parameter value is normally sufficient to cause any program's working-set space-time to be within 10% of the minimum possible for that program. Working set dispatchers automatically control the level of multiprogramming while maintaining near-minimum space-time for each program. Working set detecting hardware can be built cheaply.

Working set dispatchers have been built in real operating systems where they have been cost effective even without much hardware support. Rodriguez-Rosell reported a successful implementation for a CP-67 system [RODR73a]. Potier reports that in EMAS a working set dispatcher increased the time the machine spent in user state by 10%, decreased supervisor overhead, and increased the utilization of the swapping channel [POTI77].

Non-working-set dispatchers required additional mechanism, either for selecting a memory policy parameter suitable for each program, or for a load control with a global policy. It is a false economy to limit the hardware support for memory management to usage bits and interval timers, for the savings in hardware are cancelled by performance losses (relative to the working set dispatcher) or by additional mechanism elsewhere in the operating system.

What of the Future?

New memory and computer-network technologies are changing the computing milieu. Has the solution for the memory management problem "arrived" just as the problem has receded? I think not.

This paper has presented a detailed view of the "life cycle" of an important problem area, optimal multiprogrammed memory management. Well over two hundred individuals from many countries have participated in this research since 1965. It is quite rare to see so many involved in the solution of a difficult problem. The most important results of this research are:

1. Basic instabilities exist in computer systems. These instabilities are closely related to the behavior of the programs being run.
2. Queueing network models, which are robust and amenable to hierarchical analysis, can be used to characterize these instabilities precisely.
3. Optimal or near-optimal policies of memory management can be designed with the help of queueing network models.
4. Adaptive procedures which are practical approximations to the controls suggested by the models can be built.

This larger perspective on the accomplishments of memory management research shows that the primary results transcend the specific solution of Saltzer's Problem. Many of the techniques can be extended to deal with instabilities in networks of computers, automatic telephone and communication systems, or distributed data management systems.

What problems face us in the near term? Certainly some form of working set dispatcher will be needed in any system that multiprograms a main memory. Conventional multiprogramming systems will be with us for some time to come. This will be true whether virtual memories are based on segmentation or on paging. It will also be true despite larger amounts of cheaper main memory.

The new technology enables new solutions to old problems even as it provides solutions to new problems. For example, a form of multiprogramming is likely to appear in dynamically reconfigurable memories, wherein each process has assigned to it a working set of memory modules. A stored object can be transmitted from one processor to another by disconnecting the module containing the object from the sender's working set and attaching it to the receiver's working set. Memory policies will free modules by moving their contents to secondary storage.

Another example: the technology will soon permit each user to have a private computer system (either personally owned or rented from a central house); these will be connected via networks to central data bases and long-term storage systems. Programs and data will be transferred to the local computer for processing, and results returned to the central storage system for safekeeping. What little

memory management there is in the local computers is mostly concerned with swapping and buffering of entire programs or files. Such networks are essentially elaborate transaction-processing systems; we have known for a long time that sophisticated memory policies are of marginal value in such systems. In this case, the technology is not giving us a new problem, but rather an old problem in new guise. Sophisticated memory systems will still exist in the central facilities accessed by the local computers.

Data management systems are forcing us to reevaluate the fundamental concept of a working set. A basic question is whether the system's working set should be measured from the merged reference patterns of the processes actively reading and updating the data base, or whether it should be the union of the working sets measured from the individual reference patterns of the active processes. The record reference string representing the joint activity of many processes is likely to have much less apparent locality [RODR76] than the individual record reference strings of active processes [EAST78a]. A moment's reflection shows that these questions are fundamentally the same as were asked about page reference strings some years ago. This appears not to be a new problem, but rather an old problem in a new guise. The answer may turn out to be, as before, that it is better to measure the working set of each active process and store the union of these working sets in the high-speed memory.

When combined with paged virtual address spaces, data management systems have given rise to new problems that are sometimes regarded as worthy research problems. An example is the so-called "double paging problem", which can arise when the data base manager uses a buffer area in the virtual address space. The data base manager may use the buffer area with such poor locality that the virtual memory manager is constantly removing the most needed pages of the buffer: a reference to a missing record causes both a data manager fault and a page fault. This problem originated in the IBM IMS (Information Management System), where a linear search was used to

locate a record in the virtual buffer. It has been fixed in IMS by using an index table for the records in the virtual buffer.

The stopgap solution for the double paging problem is usually to purchase enough real memory to hold the entire buffer. The real solution is, of course, to use a segmented virtual memory: each record would already be a part of the address space and the data base manager would not be forced to solve an unneeded instance of the overlay problem in the address space. (See also DENN71.) The double paging problem is the consequence of a flaw in the architecture of the computer; it is not an interesting subject of memory management research.

What is not an old problem in new guise is closing the semantic gap -- making the hardware capable of directly supporting the concepts used in programming languages. Curiously, we have over the years expended herculean efforts on the massive mechanisms of operating systems; by comparison, pixie effort has been devoted to understanding the nature of the programs that drive the mechanisms. The result has been one computer after another that presents an inhospitable environment to its users. The art of tailoring the machine's design to allow highly efficient execution of the programs likely to be run on it -- long practiced by the Burroughs Corporation [ORGA73] -- is beginning (grudgingly) to find favor among the general computer architecture community. For example, Tannenbaum has recently shown that properly designed machines can run "structured programs" several times faster than conventional machines and with memory several times smaller [TANN78]. Myers has shown that a tagged architecture to support self-identifying data objects significantly shortens programs

and increases software reliability [MYER78].

The same principle applies to memory management. The system as a whole will be simpler and more reliable if there is no gap between the concepts of program behavior and the concepts supported by the memory management hardware. (Rau's data shows that virtually all manifestations of locality are masked off at the level of address interpretation in conventional memory architectures [RAU79]. This would not be so in Myers' machine [MYER78].) The Batson and Madison studies, and the Ferrari studies, show that working set memory management on machines capable of supporting small segments in their virtual memories would be much more efficient than is possible on any conventional machine. (See BATS76a,b, MADI76, FERR74, FERR75, and FERR76.) Batson has recently suggested that cyclic structures in the program text can be exploited to reduce resident sets near the transitions between program phases.[BATS77]; this would require coordination between the design of a compiler and the design of the working set detector. Cyclic structures have also been found useful in analyzing data base reference patterns [EAST78a].

Because of the great potential for improving computer architecture, and because of the strong influence program behavior has on the stable operation of computer networks, characterizing workloads and tailoring the machine's design thereto is perhaps the most important, intriguing, and fruitful direction of memory management research in the next period.

ACKNOWLEDGEMENTS

My special gratitude goes to Erol Gelenbe, who invited me to write this paper and then patiently pored through the lengthy draft. Alan Smith, G. Scott Graham, Alan Batson, Yonathan Bard, James B. Morris, James Bouhana, and Jeffrey Spirn were most helpful in their comments. Yonathan Bard graciously provided some performance data on CP-67 and VM/370.

REFERENCES

ADAM75 M. C. Adams, G. E. Millard, "Performance measurements on the Edinburgh Multi Access System (EMAS)," Proc. ICS 75, Antibes (June 1975).

AHO71 A. V. Aho, P. J. Denning, J. D. Ullman, "Principles of optimal page replacement," J. ACM 18, 1 (January 1971), 80-93.

ACM61 ACM, "Proceedings of a symposium on storage allocation," Comm. ACM 4, 10 (October 1961).

ARVI73 Arvind, R. Y. Kain, E. Sadeh, "On reference string generation processes," Proc. 4th ACM Symposium on Operating Systems Principles (October 1973), 80-87.

BABO77 J. Y. Babonneau, M. S. Achard, G. Morisset, M. B. Mounajjed, "Automatic and general solution to the adaptation of programs in a paging environment," Proc. 6th ACM Symposium on Operating Systems Principles (November 1977), 109-116.

BADE75 M. Badel, E. Gelenbe, J. Lenfant, D. Potier, "Adaptive optimization of a time sharing system's performance," Proc. IEEE 63, 6 (June 1975), 958-965.

BARD75 Y. Bard, "Application of the page survival index (PSI) to virtual memory system performance," IBM J. R & D 19, 3 (May 1975), 212-220.

BASK76 F. Baskett, A. Rafii, "The AO inversion model of program paging behavior," Stanford University, Computer Science Dept., Report STAN-CS-76-579 (November 1976).

BATS70 A. P. Batson, S. Ju, D. Wood, "Measurements of segment size," Comm. ACM 13, 3 (March 1970), 155-159.

BATS76a A. P. Batson, W. Madison, "Measurements of major locality phases in symbolic reference strings," Proc. Int'l Symposium on Computer Performance Modeling, Measurement, and Evaluation, ACM SIGMETRICS and IFIP WG7.3 (March 1976), 75-84.

BATS76b A. P. Batson, "Program behavior at the symbolic level," IEEE Computer 9, 11 (November 1976), 21-28.

BATS77 A. P. Batson, W. E. Blatt, J. P. Kearns, "Structure within locality intervals," Proc. Symposium on Modeling and Performance Evaluation of Computer Systems (H. Beilner and E. Gelenbe, Eds.) North-Holland Publishing Co. (October 1977), 221-232.

BELA66 L. A. Belady, "A study of replacement algorithms for virtual storage computers," IBM Syst. J. 5, 2 (1966), 78-101.

BELA67 L. A. Belady, "Biased replacement algorithms for multiprogramming," IBM T. J. Watson Research Center, Research Note NC697 (March 1967).

BELA69 L. A. Belady, C. J. Kuehner, "Dynamic space sharing in computer systems," Comm. ACM 12, 5 (May 1969), 282-288.

BRAN74 A. Brandwajn, "A model of a time sharing virtual memory system solved using equivalence and decomposition methods," Acta Informatica 4 (1974), 11-47.

BRAN77 A. Brandwajn, "A queueing model of multiprogrammed computer systems under full load conditions," J. ACM 24, 2 (April 1977), 222-240.

BRAW68 B. Brawn, F. G. Gustavson, "Program behavior in a paging environment," AFIPS Conf. Proc. 33 (1968 FJCC), 1019-1032.

BRAW70 B. Brawn, F. G. Gustavson, E. Mankin, "Sorting in a paged environment," Comm. ACM 13, 8 (August 1970), 483-494.

BRYA75 P. Bryant, "Predicting working set sizes," IBM J R & D 19, 3 (May 1975), 221-229.

BUZE71 J. P. Buzen, "Optimizing the degree of multiprogramming in demand paging systems," Proc. IEEE Compcon (September 1971), 139-140.

BUZE76 J. P. Buzen, "Fundamental operational laws of computer system performance," Acta Informatica 7, 2 (1976), 167-182.

BUZE78 J. P. Buzen, "A queueing network model of MVS," Computing Surveys 10, 3 (September 1978), 319-332.

CHIU78 W. W. Chiu, W-M Chow, "A hybrid hierarchical model of a multiple virtual storage (MVS) operating system," IBM T. J. Watson Research Center Report RC6947 (January 1978).

CHU72 W. W. Chu, H. Opderbeck, "The page fault frequency replacement algorithm," AFIPS Conf. Proc. 41 (1972 FJCC), 597-609.

CHU76a W. W. Chu, H. Opderbeck, "Analysis of the PFF algorithm using a semi-Markov model," Comm. ACM 19, 5 (May 1976), 298-304.

CHU76b W. W. Chu, H. Opderbeck, "Program behavior and the page fault frequency replacement algorithm," IEEE Computer 9, 11 (November 1976), 29-38.

CHAM73 D. D. Chamberlin, S. H. Fuller, L. Liu, "An analysis of page alllocation strategies for virtual memory systems," IBM J. R & D 17 (1973), 404-412.

COFF73 E. G. Coffman, Jr., P. J. Denning, Operating Systems Theory, Prentice-Hall (1973).

COFF72 E. G. Coffman, Jr., T. A. Ryan, Jr., "A study of storage partitioning using a mathematical model of locality," Comm. ACM 15, 3 (March 1972), 183-190.

COFF68 E. G. Coffman, Jr., L. C. Varian, "Further experimental data on the behavior of programs in a paging environment," Comm. ACM 11, 7 (July 1968), 471-474.

COME76 L. W. Comeau, "A study of the effect of user program optimization in a paging system," Proc. ACM Symposium on Operating Systems Principles (October 1967).

COUR72 P. J. Courtois, "On the near complete decomposability of networks of queues and of stochastic models of multiprogramming," Research Report CMU-CS-72-11 (1972), CS Dept., Carnegie-Mellon University.

COUR75 P. J. Courtois, "Decomposability, instabilities, and saturation in multiprogramming systems," Comm. ACM 18, 7 (July 1975), 371-377.

COUR77 P. J. Courtois, Decomposability, Academic Press (1977).

COUR76 P. J. Courtois, H. Vantilborgh, "A decomposable model of program paging behavior," Acta Informatica 6, 3 (1976), 251-276.

DENN66 P. J. Denning, "Memory allocation in multiprogrammed computer systems," MIT Project MAC Computation Structures Group Memo No. 24 (March 1966).

DENN68a P. J. Denning, "The working set model for program behavior," Comm. ACM 11, 5 (May 1968), 323-333.

DENN68b P. J. Denning, "Resource allocation in multiprocess computer systems," Ph.D. Thesis, Report MAC-TR-50, MIT Project MAC (May 1968).

DENN68c P. J. Denning, "Thrashing: Its causes and prevention," AFIPS Conf. Proc. 33 (1968 FJCC), 915-922.

DENN69 P. J. Denning, "Equipment configuration in balanced computer systems," IEEE Trans. Computers C-18, 11 (November 1969), 1008-1012.

DENN70 P. J. Denning, "Virtual memory," Computing Surveys 2, 3 (September 1970), 153-189.

DENN71 P. J. Denning, "Third generation computer systems," Computing Surveys 3, 4 (December 1971), 175-216.

DENN72a P. J. Denning, S. C. Schwartz, "Properties of the working set model," Comm. ACM 15, 3 (March 1972), 191-198. Corrigendum: Comm. ACM 16, 2 (February 1973), 122.

DENN72b P. J. Denning, "On modeling program behavior," AFIPS Conf. Proc. 40 (1972 SJCC), 937-944.

DENN73 P. J. Denning, J. R. Spirn, "Dynamic storage partitioning," Proc. 4th ACM Symposium on Operating Systems Principles (October 1973), 74-79.

DENN75a P. J. Denning, G. S. Graham, "Multiprogrammed memory management," IEEE Proc. 63, 6 (June 1975), 924-939.

DENN75b P. J. Denning, "The computation and use of optimal paging curves," Report CSD-TR-154 (June 1975), Computer Sciences Dept., Purdue University, W. Lafayette, IN 47907.

DENN75c P. J. Denning, K. C. Kahn, "A study of program locality and lifetime functions," Proc. 5th ACM Symposium on Operating Systems Principles (November 1975), 207-216.

DENN76a P. J. Denning, K. C. Kahn, "An L=S criterion for optimal multiprogramming," Proc. Int'l Symposium on Computer Performance Modeling, Measurement, and Evaluation, ACM SIGMETRICS and IFIP WG7.3 (March 1976), 219-229.

DENN76b P. J. Denning, K. C. Kahn, J. Leroudier, D. Potier, and R. Suri, "Optimal multiprogramming," Acta Informatica 7, 2 (1976), 197-216.

DENN78a P. J. Denning, "Optimal multiprogrammed memory management," in Current Trends in Programming Methodology III (K. M. Chandy and R. Yeh, Eds.) Prentice-Hall (1978), 298-322.

DENN78b P. J. Denning, D. R. Slutz, "Generalized working sets for segment reference strings," Comm. ACM 21, 9 (September 1978), 750-759.

DENN78c P. J. Denning, J. P. Buzen, "The operational analysis of queueing network models," Computing Surveys 10, 3 (September 1978), 225-262.

DENN78d P. J. Denning, "Working sets then and now," Proc. Int'l Symp. on Operating Systems (D. Lanciaux, Ed.) IRIA Laboria, Rocquencourt, France (October 1978). Published by North-Holland, Amsterdam (1979).

DENS65 J. B. Dennis, "Segmentation and the design of multiprogrammed computer systems," J. ACM 12, 4 (October 1965), 589-602.

EAST76 M. C. Easton, P. A. Franaszek, "Use-bit scanning in replacement decisions," Proc. 5th Texas Conf. on Computing Systems, Computer Science Dept., Univ. Texas, Austin, TX 78712 (October 1976), 160ff.

EAST77 M. C. Easton, B. T. Bennett, "Transient free working set statistics," Comm. ACM 20, 2 (February 1977), 93-99.

EAST78a M. C. Easton, "Model for database reference strings based on behavior of reference clusters," IBM J. R & D 22, 2 (March 1978), 197-202.

EAST78b M. C. Easton, R. Fagin, "Cold-start v. warm-start miss ratios," Comm. ACM 21, 10 (October 1978), 866-872.

FERR74 D. Ferrari, "Improving locality by crticial working sets," Comm. ACM 17, 11 (November 1974), 614-620.

FERR75 D. Ferrari, "Tailoring programs to models of program behavior," IBM J. R & D 19, 3 (May 1975), 244-251.

FERR76 D. Ferrari, "The improvement of program behavior," IEEE Computer 9, 11 (November 1976), 39-47.

FINE66 G. H. Fine, C. W. Jackson, P. V. McIssac, "Dynamic program behavior under paging," Proc. ACM Annual Conf. (1966), 223-228.

FOGE74 M. H. Fogel, "The VMOS paging algorithm," ACM SIGOPS Operating Systems Review 8, 1 (January 1974), 8-17.

FRAN78 M. A. Franklin, G. S. Graham, R. K. Gupta, "Anomalies with variable partition paging algorithms," Comm. ACM 21, 3 (March 1978), 232-236.

GELE73a E. Gelenbe, P. Tiberio, J. Boekhorst, "Page size in demand paging systems," Acta Informatica 3, 1 (1973), 1-24.

GELE73b E. Gelenbe, "A unified approach to the evaluation of a class of replacement algorithms," IEEE Trans. Computers C-22, 6 (June 1973), 611-618.

GELE74 E. Gelenbe, J. Lenfant, D. Potier, "Anaylse d'un algorithme de gestion de mémoire centrale et d'un disque de pagination," Acta Informatica 3 (1974), 321-345.

GELE78a E. Gelenbe, A. Kurinckx, "Random injection control of multiprogramming in virtual memory," IEEE Trans. Software Engrg SE-4, 1 (January 1978), 2-17.

GELE78b E. Gelenbe, A. Kurinckx, I. Mitrani, "The rate control policy for virtual memory management," Proc. 2nd Int'l Symposium on Operating Systems, IRIA Laboria, Rocquencourt, France (October 1978).

GHAN74 M. Z. Ghanem, H. Kobayashi, "A parametric representation of program behavior in a virtual memory system," Proc. 8th Princeton Conference on Information Sciences and Systems, Dept EECS, Princeton University (March 1974), 327-330.

GRAH76 G. S. Graham, "A study of program and memory policy behaviour," Ph.D. Thesis, Dept. Computer Sciences, Purdue University, W. Lafayette, IN 47907 (December 1976).

GRAH77 G. S. Graham, P. J. Denning, "On the relative controllability of memory policies," Computer Performance (K. M. Chandy and M. Reiser, Eds.) North-Holland Publishing Co. (August 1977), 411-428.

GUPT78 R. K. Gupta, M. A. Franklin, "Working set and page fault frequency replacement algorithms: A performance comparison," IEEE Trans. Computers C-27, 8 (August 1978), 706-712.

HATF71 D. Hatfield, J. Gerald, "Program restructuring for virtual memory," IBM Syst. J. 10 (1971), 168-192.

KAHN76 K. C. Kahn, "Program behavior and load dependent system performance," Ph.D Thesis, Dept. Computer Sciences, Purdue University, W. Lafayette, IN 47907 (August 1976).

KILB62 T. Kilburn, D. B. G. Edwards, M. J. Lanigan, F. H. Sumner, "One level storage system," IRE Trans. EC-11, 2 (April 1962), 223-235.

LENF74 J. Lenfant, "Comportment des programmes dans leur espace d'addressage," Thesis, University Rennes (November 1974).

LENF75 J. Lenfant, P. Burgevin, "Empirical data on program behavior," Proc. ACM Int'l Symposium (E. Gelenbe and D. Potier, Eds.) North-Holland Publishing Co. (1975), 163-170.

LENF78 J. Lenfant, "Ensembles de travail et intervalles bornés de localité," RAIRO-Informatique (AFCET) 12, 1 (1978), 15-35.

LERO76a J. Leroudier, P. Burgevin, "Characteristics and models of program behavior," Proc. ACM Annual Conf. (1976), 344-350.

LERO76b J. Leroudier, D. Potier, "Principles of optimality for multiprogramming," Proc. Int'l Symposium on Computer Performance Modeling, Measurement, and Evaluation, ACM SIGMETRICS and IFIP WG7.3 (March 1976), 211-218.

LEWI71 P. A. W. Lewis, P. C. Yue, "Statistical analysis of program reference patterns in a paging environment," Digest IEEE Conf. (1971).

MADI76 A. W. Madison, A. P. Batson, "Characteristics of program localities," Comm. ACM 19, 5 (May 1976), 285-294.

MASU74 T. Masuda, H. Shiota, K. Noguchi, T. Ohki, "Optimization by cluster analysis," Proc. IFIP Congress (1974), 261-265.

MATT70 R. L. Mattson, J. Gecsei, D. R. Slutz, I. L. Traiger, "Evaluation techniques for storage hierarchies," IBM Syst. J. 9, 2 (1970), 78-117.

MORR72 J. B. Morris, "Demand paging through the use of working sets on the MANIAC II," Comm. ACM 15, 10 (October 1972), 867-872.

MYER78 Myers, G.J., Advances in Computer Architecture, New York, Wiley (1978).

OPDE74 H. Opderbeck, W. W. Chu, "Performance of the page fault frequency algorithm in a multiprogramming environment," Proc. IFIP Congress (1974), 235-241.

OPDE75 H. Opderbeck, W. W. Chu, "The renewal model for program behavior," SIAM J. Computing 4, 3 (September 1975), 356-374.

ORGA72 E. I. Organick, The MULTICS System: An Examination of Its Structure, MIT Press (1972).

ORGA73 E. I. Organick, Computer System Organization: The B5700/B6700 Series, Academic Press (1973).

PARE77 M. Parent, D. Potier, "A note on the influence of program loading on the page fault rate," Acta Informatica 8, 4 (1977), 359-370.

PRIE76 B. G. Prieve, R. S. Fabry, "VMIN -- an optimal variable space page replacement algorithm," Comm. ACM 19, 5 (May 1976), 295-297.

POTI77 D. Potier, "Analysis of demand paging policies with swapped working sets," Proc. 6th ACM Symposium on Operating Systems Principles (November 1977), 125-131.

RAU79 Rau, B.R., "Program behavior and the performance of interleaved memories," IEEE Trans. Computer C-28, 3 (March 1979), 191-199.

RODR71 J. Rodriguez-Rosell, "Experimental data on how program behavior affects the choice of scheduler parameters," Proc. 3rd ACM Symposium on Operating Systems Principles (October 1971), 156-163.

RODR73a J. Rodriguez-Rosell, J. P. Dupuy, "The design, implementation, and evaluation of a working set dispatcher," Comm. ACM 16, 4 (April 1973), 247-253.

RODR73b J. Rodriguez-Rosell, "Empirical working set behavior," Comm. ACM 16, 9 (September 1973), 556-560.

RODR76 J. Rodriguez-Rosell, "Empirical data reference behavior in data base systems," IEEE Computer 9, 11 (November 1976), 9-13.

SADE75 E. Sadeh, "An analysis of the performance of the page fault frequency (PFF) replacement algorithm," Proc. 5th ACM Symposium on Operating Systems Principles (November 1975), 6-13.

SALT74 J. H. Saltzer, "A simple linear model of demand paging performance," Comm. ACM 17, 4 (April 1974), 181-186.

SAYE69 D. Sayre, "Is automatic folding of programs efficient enough to displace manual?" Comm. ACM 13, 12 (December 1969), 656-660.

SHED72 G. S. Shedler, C. Tung, "Locality in page reference strings," SIAM J. Computing 1, 3 (September 1972), 218-241.

SIMO79 Simon, R., "The modeling of virtual memory systems," Ph.D. Thesis, Computer Science Dept., Purdue University, W. Lafayette, IN 47907 (May 1979).

SLUZ74 D. R. Slutz, I. L. Traiger, "A note on the calculation of average working set size," Comm. ACM 17, 10 (October 1974), 563-565.

SLUZ75 D. R. Slutz, "A relation between working set and optimal algorithms for segment reference strings," IBM Research Report TJ1623 (July 1975).

SMIT76a A. J. Smith, "A modified working set paging algorithm," IEEE Trans. Computers C-25 (September 1976), 907-914.

SMIT76b A. J. Smith, "Analysis of the optimal, lookahead, demand paging algorithms," SIAM J. Computing 5, 4 (December 1976).

SMIT76c A. J. Smith, "Multiprogramming and internal scheduling," Dept EECS, DS Division, University of California, Berkeley, CA 94720 Technical Report (November 1976).

SMIT76d A. J. Smith, "Mutliprogramming and memory contention," Dept EECS, CS Division, University of California, Berkeley, CA 94720, Technical Report (November 1976).

SPIR72 J. R. Spirn, P. J. Denning, "Experiments with program locality," AFIPS Conf. Proc. 41 (1972 FJCC), 611-621.

SPIR73 J. R. Spirn, "Program locality and dynamic memory management," PhD thesis, Dept EE, Princeton University, Princeton, NJ 08540 (March 1973).

SPIR76 J. R. Spirn, "Distance string models for program behavior," IEEE Computer 9, 11 (November 1976), 14-20.

SPIR77 J. R. Spirn, Program Behavior: Models and Measurement, Elsevier/ North-Holland Publishing Co. (1977).

SPIR79 Spirn, J.R., "Biased memory partitioning with swapping delays," Tech. Rpt. CS-79-28, Computer Science Dept., Penn State Univ., Univ. Park, PA 16802 (January 1979).

TSAO72 R. F. Tsao, L. W. Comeau, B. H. Margolin, "A multifactor paging experiment I: The experiment and the conclusions," in Statistical Computer Performance Evaluation (W. Freiberger, Ed.), Academic Press (1972), 103-134.

TSUR78 Tsur, S., "Analysis of queueing networks in which processes exhibit locality-transition behavior," Information Processing Letters 7,1 (January 1978), 20-23.

WEIZ69 N. Weizer, G. Oppenheimer, "Virtual memory management in a paging environment," AFIPS Conf. Proc. 34 (1969 SJCC), 234ff.

WILK73 M. V. Wilkes, "The dynamics of paging," Computer J. 16 (February 1973), 4-9.

Insert on page 67 after HATF71:

JAMP79 Jamp, R., and J. R. Spirn, "VMIN based determination of program macro-behavior," Report CS-79-34, Computer Science Department, Penn State University, University Park, PA 16802 (May 1979).