Department of Computer Science Technical Reports

Department of Computer Science

1978

Working Sets Today

Peter J. Denning

Report Number:

78-287

WORKING SETS TODAY

Peter J. Denning

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

September 1978

CSD-TR-287

Preprint of invited paper to appear in Proc. IEEE COMPSAC, November 13-16, 1978.

WORKING SETS TODAY[1]

Peter J. Denning

Computer Sciences Department
Purdue University
West Lafayette, IN 47907

Abstract: A program's working set is the collection of pages (or segments) recently referenced. This concept has led to efficient methods for measuring a program's intrinsic memory demand; it has assisted in understanding program behavior; and it has been used as the basis of optimal multiprogrammed memory management. This paper outlines the argument why it is unlikely that anyone will find a cheaper nonlookahead memory policy that delivers significantly better performance.

This paper is based on a longer paper that presents the arguments in greater detail [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 -- 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 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.) But by that time the conventional optimism had changed to circumspection; no one wanted to risk my unconventional proposal which, by 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 main memory; he used the term "thrashing"

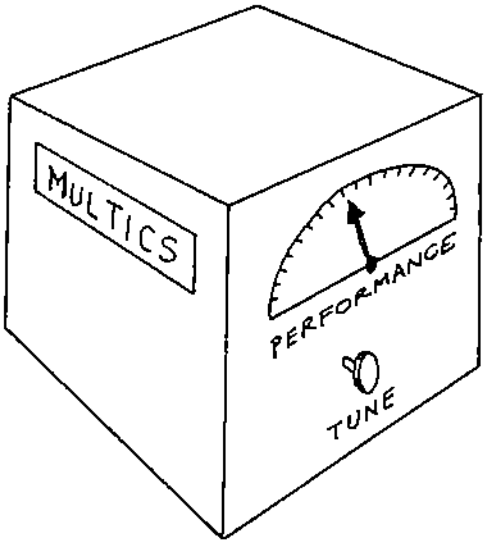

FIGURE 1. Abstract representation of Saltzer's Problem.

[1] This work supported in part by NSF Grants GJ-41289 and MCS78-01729 at Purdue University.

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: Sayre and his colleagues had found that significant differences in program behavior would result if programmers attempted even simple schemes to improve "locality" [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 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 -- i.e., 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. 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.

This paper outlines the history of the working set concept and the lessons it has taught about designing a dispatcher for a multiprogrammed virtual memory system. (See DENN78d for the details.) The conclusion is that the working-set dispatcher is the most cost-effective dispatcher known; it is unlikely that someone will discover a nonlookahead memory policy whose cost is significantly lower and performance significantly better.

Basic Performance Measures

Queueing network models are widely used as analytic tools for obtaining accurate estimates of utilizations and throughputs of multiple resource computer systems [DENN78c]. 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 gives the mean virtual time between page faults (the reciprocal of the paging rate) as a function of the mean size of the resident set (the pages loaded in main memory); see Figure 2. Lifetime curves for individual programs under given memory policies are easy to measure.

The memory policies of interest here have a control parameter θ which 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.)

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_0. Figure 3 illustrates a typical CPU utilization curve as a function of N, the multiprogramming level (MPL), for a fixed size main memory comprising P pages. The curve rises

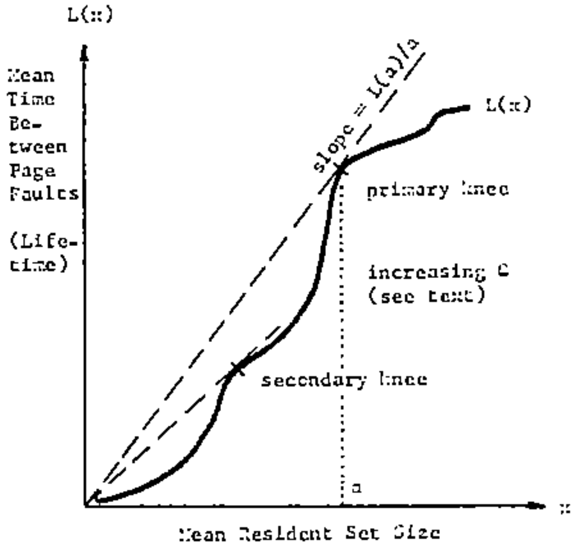

FIGURE 2. A typical lifetime curve.

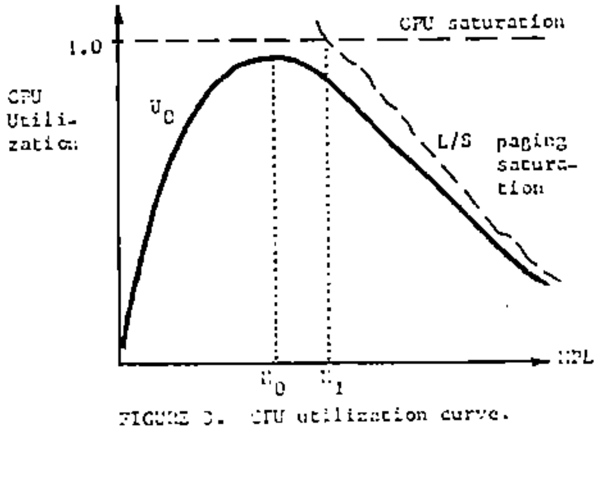

FIGURE 3. CPU utilization curve.

toward CPU saturation, but is eventually depressed by saturation of the paging device. The paging device bound is given by the ratio L/S, where L is the mean CPU execution time between page faults and S is the mean page swap time. Note that L is a decreasing function of N, because larger MPLs imply less space for each resident set; S is usually independent of N. In many cases the MPL N_1 at which L=S is just slightly larger than the optimum MPL, N_0, which suggests that monitoring L could be a basis for a load controller.

Characteristics of Optimum MPL

The intuition of Figure 3 -- 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 page swap time S [DENN76]. 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 pages, there are PT page-seconds of main memory space-time available. Therefore the memory space-time per job is

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

It follows that the optimum MPL, N_0, maximizes throughput and minimizes memory space-time per job.

An approximation for the space-time of a job whose mean resident set size is x pages is

$$ST(x) = E\cdot x\cdot(1 + S'/L(x)),$$

where E is the job's execution time, S' is the mean delay per page fault (S' includes the queueing delay and the page swap time S), and L(x) is the program's lifetime value. (This approximation is not always very accurate; it is neither consistently high nor low and may be in error by as much as 20% [GRAH76].) If S' is large, ST(x) is minimized approximately when the ratio x/L(x) is minimized, which occurs near the primary knee of the lifetime curve (Fig. 2).

To limit the sharp 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 4.) 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 are held, inactive, in a memory queue. The limiting effect of the memory queue is sketched in Figure 5. (See COUR75.) Evidently, if M were set to N_0, thrashing could not occur and the system would operate at optimum throughput whenever a sufficient number of jobs is submitted. In practice, the optimum load varies with the workload; hence an adaptive control is needed to adjust M. Setting M to the smallest possible value of N_0 is usually unsuitable: the system will be underloaded.

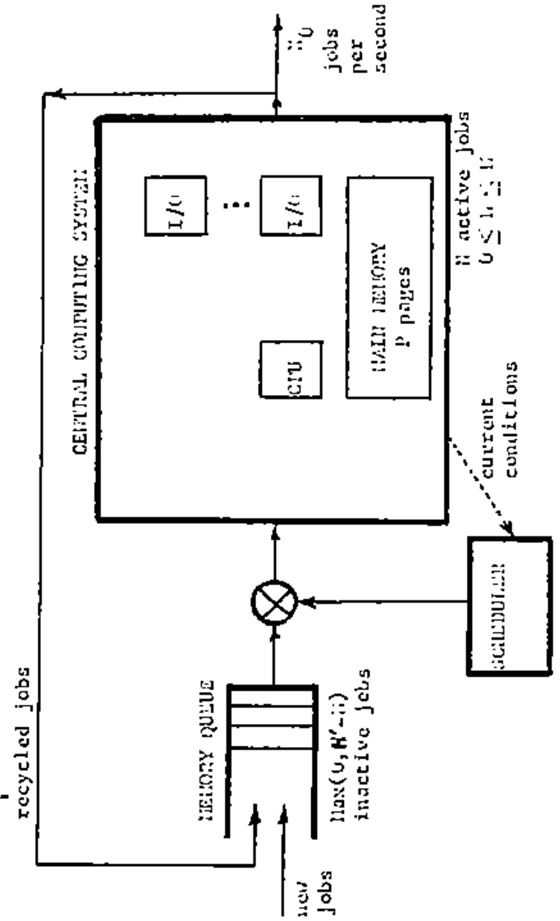

FIGURE 4. Limiting the load on the central computing system by a load control and a memory queue.

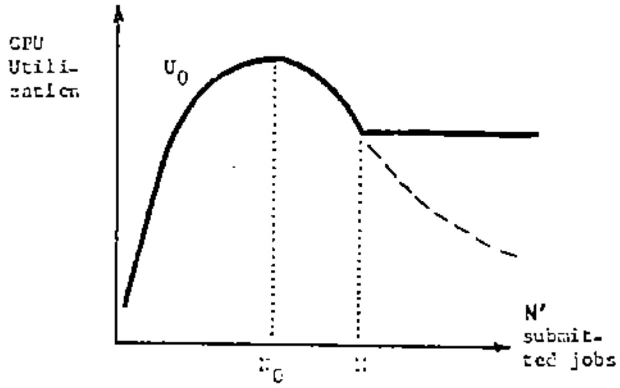

FIGURE 5. Utilization under load control.

To summarize: the objective of the load controller is setting the maximum MPL, M, near the current optimum. The optimum MPL is achieved by minimizing the space-time per program, which is strongly correlated with the primary knee of the program's lifetime curve for the given memory policy.

The Working Set Concept

The term "working set of information" originated in the early implementations of ALGOL (ca. 1960) where it denoted the smallest set of instruction and data words that should be in the main store in order to keep the CPU efficiency acceptable. Successive refinements of this intuition have helped us understand how to set a memory policy's control parameter to achieve minimum space-time for each active job, the basis for optimal memory management.

In 1966 I suggested that a working set could be measured by sampling (and resetting) hardware usage bits of pages every θ virtual time units [DENN66]. The resident set at a particular (virtual) time would be the most recent sampled working set plus any pages added by page faults. In case of interruption, the resident set would be swapped out in toto and reloaded as a unit prior to resumption of the program. Such a policy has been implemented on a CP-67 system [RODR73] and on the Edinburgh Multi Access System (EMAS) [ADAM75] where it performed well.

In 1967 I suggested the *moving window working set* as an abstraction of the sampling process. The working set $W(t,\theta)$ is the set of pages (or segments) referenced in the virtual time interval $[t-\theta+1,t]$, looking backward from (virtual) time t [DENN68a]. For t an integer multiple of θ, this working set is the same as the sampled working set. Under the *working set memory policy* (WS) a program's resident set at time t is just $W(t,\theta)$.

In 1972 Morris reported the construction of hardware for the MANIAC II computer to implement the moving window working set [MORR72]. This hardware requires a timer register and indentification register for each page frame of main memory. A page frame's timer is enabled only when the running job's index-number, stored in a CPU register, matches the identification register. This scheme ensures that the measurement is taken in the job's virtual time. Expired timers mark pages which have left the working set. Morris reported that this hardware cost less than $20 per page frame in 1971.

Notice that the page referenced at time t is absent from the working set, thereby causing a page fault, if and only if the time since the prior reference to that page exceeds the window size θ. This property has been exploited to define a highly efficient procedure which, in one pass over a program's address trace, measures the mean resident set size $s(\theta)$ and the missing page rate $m(\theta)$. A table is kept of the time of most recent reference to each page; on a new reference to that page the interval, say k, since prior reference is calculated before the table is updated for that page; then a counter $c(k)$ is incremented. After all the program's references are observed, the counters are normalized, thereby defining the *interreference frequency distribution* $h(k)$. Then, as shown in COFF73, DENN68b, DENN72, EAST77, or SLUZ74,

$$m(\theta) \;=\; \sum_{k>\theta} h(k) \qquad \text{[missing page rate]}$$

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

Then $(s(\theta), 1/m(\theta))$ is a point on the lifetime curve of the WS policy for the measured program.

In 1975, Slutz and I generalized the working set concept by introducing a "retention cost" function that measures the (accumulating) cost of nonreference for a page (or segment) kept resident; this cost is reset to 0 just after a reference. The "generalized working set" (GWS) contains all pages (or segments) whose retention cost at time t does not exceed θ. Special cases of the GWS (for proper choices of the retention cost) are the "stack algorithms" for paging in fixed resident-set size [MATT70, COFF73], the moving window working set, and the optimal policy VMIN [PRIE76]. Any memory policy whose resident sets satisfy an inclusion property under increasing values of the control parameter (θ) is an instance of the GWS. All instances of the BWS have a simple procedure for calculating points on the lifetime curve -- similar to the one noted above but with $h(k)$ replaced by the frequency distribution of retention costs. (See DENN78a,b for the details.)

The moving window working set was first envisaged as a model of program behavior -- i.e., as an abstract description of the mechanisms by which programs demand main memory space and create page swapping [DENN68a]. However, the working set measurement procedure does not depend on assumptions about the interreference distribution $h(k)$. For this reason, working sets are regarded as models of a large class of memory policies; program models are treated as a separate issue. Program behavior models are discussed in DENN75, DENN78a,d.

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 job-class (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 4) 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.

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 by observing the aggregated behavior of them all; 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. More details about memory policies will be given in the next section. Notice that a local policy will necessarily maintain a *pool* of available page frames -- i.e., those not used by any active job's resident set.

The load controller sets the limit M on the multiprogramming level (MPL); ideally, M should be the optimum N_0 (see Figure 4). There are two kinds of load-controller corresponding to the two kinds of memory policy. The global-feedback controller employs some aggregated measure of the swapping demand to adjust M; two successful methods are [DENN76]:

> L=S control. Allow M to rise as high as demand warrants so long as the observed mean CPU time between page faults in the system (L) is never smaller than the mean page swap time (S).
>
> 50% control. Allow M to rise as high as demand warrants so long as the observed utilization of the paging device does not exceed 50%.

The motivation for the L=S control was discussed in connection with Figure 3. The motivation for the 50% control is that 50% utilization corresponds to mean queue length of 1 paging request, the onset of thrashing [DENN76]. The second type of controller, the local-feedback controller, operates according to the size of the pool of available page frames. The highest priority job in the memory queue is activated as soon as the pool is sufficient to contain the job's working set.

Memory Policies

The purpose of this section is to describe four common memory policies -- two of the global type and two of the local type.

One of the most common global policies is called the CLOCK algorithm. On a page fault, a pointer is cycled through the page frames of main memory, skipping frames whose usage bit is set (and resetting them) and selecting for replacement the first page whose usage bit is not set. (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 is a variant of FIFO (first in first out); it was under consideration for MULTICS in 1967 [DENN68b], and it is used in at least one version of CP-67 (now VM/370) [BARD75]. (See also EAST76.)

Another global policy is LRU (least recently used). All the resident pages of all active jobs are ordered as an LRU stack by decreasing recency of use. At a page fault time, the resident page farthest down the stack is chosen for replacement. The CDC STAR-100 computer uses this scheme.

There is, unfortunately, little published performance data on 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 CLOCK might be somewhat better than global LRU [CORB69]. From Belady's data on single programs, one may deduce that CLOCK and LRU give similar performance [BELA66]. From Graham's data on single programs, one may deduce that LRU is significantly worse than WS [GRAH76]. Experience with a CP-67 [RODR73] and the EMAS [ADAM75, POTI77] suggests further than replacing a global policy with a WS policy can improve performance significantly.

The available evidence thus suggests that CLOCK and LRU cannot perform as well as WS. This is because these global policies cannot ensure that the block of memory allocated to a particular program minimizes that program's space-time [DENN75]. The main attraction of CLOCK is its simple implementation; but this may not be justified owing to its poorer performance.

The WS policy is an example of a local policy. In 1972, Chu and Opderbeck proposed another, 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 an existing operating system 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 $R(t,\theta)$ denote the PFF resident set just after time t, given that the control parameter of PFF has value θ. Then

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

where p(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 as 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 p(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 $1/\theta$ can be interpreted as the maximum tolerable frequency of page faults.

Graham's experimental study showed that WS and PFF are general comparable in performance and considerably better than LRU [GRAH76, GRAH77]. WS has a slight tendency to produce lower space-time minima than PFF, but the differences are within 10%. 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 impossible with WS. Moreover, the PFF performance is much more sensitive to the proper choice of parameter than is WS performance [GRAH76, GUPT78].

Controllability of Memory Policies

Since global memory policies make no distinctions among programs, their load controls (e.g., the "L=S control" or the "50% control") 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.

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 -- but this may be at the cost of 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, thereby eliminating the overhead of 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.

Graham performed some experiments with a sample of 8 programs. These experiments aimed to determine the sensitivity of WS and PFF to the control parameter [GRAH77]. Two questions were asked:

1. For the given sample and a given value of p, what is a minimal set of θ-values so that each program's space-time is within p% of minimum for at least one of these θ-values? (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?

Graham found these sizes of the minimal sets of θ-values:

	p	
	10%	5%
WS	1	2
PFF	3	4

(8 programs in the sample.)

He also found these as the maximum tolerances that must be tolerated when one θ-value is used:

	minimum p
WS	10%
PFF	50%

The conclusion from this study is that, for the given sample of programs, the WS policy could be run with a single, global θ-value (θ = 73,000 references) and would deliver throughput no worse than 10% from optimum. For comparable performance, PFF would need a dynamic θ-detector capable of distinguishing among 3 candidate values of θ. The performance of PFF is therefore much more sensitive to θ than is the performance of WS. (A similar conclusion has been reached by Gupta and Franklin [GUPT78].)

Assuming that similar characteristics are reproducible for other typical workloads, it appears that the θ-detector needed to run PFF with performance similar to a single-θ WS makes a multiprogrammed PFF at least as expensive to implement as a multiprogrammed WS. It also appears that a properly tuned WS or PFF policy will perform significantly better than either CLOCK or global LRU.

A final question is: Do there exist memory policies that perform significantly better than properly tuned WS or PFF without costing significantly more? No one has found such a policy. If one compares the behavior of WS and the optimal policy, VMIN, one finds that a) WS and VMIN produce the same page fault sequence for given θ, and b) the lower VMIN resident set size is caused by VMIN's ability to anticipate the end of a current program phase and remove unneeded pages from residence. A careful analysis of these facts, which is beyond the scope of this paper, leads to the conclusion that no one is likely to find a nonlookahead memory policy significantly better than WS. (See DENN78a,d.)

Conclusion

The working set dispatcher is the solution of 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 sufficient to cause any program's working-set space-time to be within 10% of the minimum possible space-time 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 [RODR73]. 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 require additional mechanism, either for selecting a memory policy parameter suitable for each program, or for a global-feedback load control. 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.

Acknowledgements

My special gratitude goes to Erol Gelenbe, who invited me to write the original version of this paper [DENN78d], and then patiently pored though the length draft. Alan J. Smith and G. Scott Graham were most generous in their comments on the draft.

References

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

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.

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

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

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

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

CORB69. F. J. Corbato, "A paging experiment with the MULTICS system," in In Honor of P. M. Morse (K. U. Ingard, Ed.) MIT Press (1969), 217-228.

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

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," PhD Thesis, Report MAC-TR-50, MIT Project MAC (May 1968).

DENN72. P. J. Denning and 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.

DENN75. P. J. Denning and G. S. Graham, "Multiprogrammed memory management," Proc. IEEE 63, 6 (June 1975), 924-939.

DENN76. P. J. Denning, K. C. Kahn, J. Leroudier, D. Potier, 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).

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

DENN78d. P. J. Denning, "Working sets then and now," Proc. 2nd Int'l Symposium on Operating Systems, IRIA Laboria, Rocquencourt, 78150 Le Chesnay, France (October 1978).

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 78712 (October 1976), 160ff.

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

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

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

GRAH76. G. S. Graham, "A study of program and memory policy behaviour," PhD Thesis, Computer Sciences, Purdue Univ. (December 1976).

GRAH77. G. S. Graham, P. J. Denning, "On the relative controllability of memory policies," Computer Performance (Chandy & 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," IEEETC C-27 (August 1978), 706-712.

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.

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 Symp. on O. S. Princs. (Nov 1977), 125-131.

RODR73. J. Rodriguez-Rosell, J. P. Dupuy, "The design, implementation, and evaluation of a working set dispatcher," Comm. ACM 16, 4 (April 1973), 247-253.

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

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.