On modeling program behavior

by PETER J. DENNING

Princeton University
Princeton, New Jersey

INTRODUCTION

This is a paper about the history of the working set model for program behavior. It traces briefly the origins and bases of the idea and some of the results subsequently obtained. The physical context is a hierarchical memory system consisting of a severely limited quantity of main (directly-addressable) storage and an essentially unlimited quantity of secondary (backup) storage. In this context, the intuitive notion of "working information" as the set of words which are (or should be) loaded in main memory at any given time in order that a program may operate efficiently is as old as programming itself.[1] The sharply increased interest in program models since the mid-1960s is a direct consequence of the widening use of virtual memory and multiprogramming techniques, which have shifted the responsibility of memory management from programmers to machines. I am assuming here that the purpose of memory management is ensuring that an active program's working information is present in main memory, and the purpose of a program model is providing a basis for determining a program's working information at a given time and predicting what it will be at a future time.

The intuitive notion calls for "working information" to be defined as machine-independent as possible, so that there can be an "absolute" measure of a program's memory demand. Moreover, the definition should be dynamic since we wish ultimately to derive adaptive memory management policies from it. In my own efforts to formalize the intuitive notion, I adopted the spirit of an approach suggested as early as 1965, according to which a program's "working set" at a given time consists of those objects which must be loaded in main memory in order to guarantee a given level of processing efficiency.[19] It seemed to me that an ideal definition should, either explicitly or implicitly, specify a priority listing on a program's information objects at each moment of time; then, given the desired level of processing efficiency and the physical characteristics of the system (e.g., the ratios of speeds between various levels of memory, or the capacities of various levels of memory), one should be able to compute, at any given time, an integer k such that the k objects of highest priority constitute the program's working set at that time. Because a direct formalization of this idea did not appear immediately amenable to analysis, I settled instead on a definition of "working set" as a collection of recently referenced objects.[11,12] As will be seen, this alternative definition is equivalent to the former one under realistic assumptions about programs.

A useful definition of working set should, in addition to the above, satisfy two properties. First, it should be based on easily observable properties of program behavior. As suggested in Figure 1, the stream of addresses (the *address trace*) generated by a processor is perhaps the most easily observed aspect of dynamic program behavior. Now, a given program's instruction and data code is ordinarily divided into blocks of various sizes, each block being identified by a number and consisting of words with contiguous addresses. Since memory allocation policies deal with blocks, not words, as the units of allocation, there is no loss of generality in supposing that a string of block numbers (the *reference string*) represents observable program behavior. Specifically, if $a_1a_2 \ldots a_t \ldots$ is an address trace in which a_t is the address referenced at time t in the time frame of the program (program time is measured discretely, $t=1, 2, 3, \ldots$), and r_t is the number of the block containing address a_t, the reference string is $r_1r_2 \ldots r_t \ldots$. In the discussion to follow, I shall assume that the blocks of a program are all of the same size (i.e., paging is used); this is purely a matter of convenience, as generalizations to variable block size are straightforward. Under this assumption, a reference string is a sequence of page numbers.

Second, the definition of working set should be based

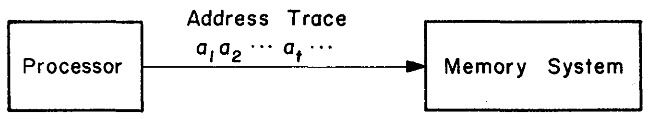

Figure 1—Address trace

on information about a program's observed past and not on information about its future—i.e., the working set at time t should depend only on $r_1 r_2 \ldots r_t$ and not on $r_{t+1} r_{t+2} \ldots$. I imposed this requirement because I wanted a program model which would guide a choice of memory management policy in the absence of prior knowledge of program behavior (e.g., programmer or compiler advice or predictions[1]), since experience suggests that prior knowledge is either inordinately expensive to obtain or is unreliable. I was not attempting to deny that, if reliable prior information is present, it can and should be used to improve an estimate of a program's working set; I wanted only to avoid making prior information a fundamental assumption of the model.

A DEFINITION OF WORKING SET

With these general notions about how "working set" should be defined, one can proceed to develop the definition itself. As will be seen, the viability of the "most recently referenced pages" definition depends on the property of *locality* (known also as "locality of reference") which is exhibited to varying degrees by all practical programs.[2,3,5–8,12,13,15,17,24,26] The property of locality can be summarized as three statements:

L1. A program distributes its references non-uniformly over its pages, some pages being favored over others.
L2. The density of references to a given page tends to change slowly in time.
L3. Two reference string segments are highly correlated when the interval between them is small, and tend to become uncorrelated as the interval between them becomes large.

These properties are abstractions of commonly observed phenomena: programs tend to reference pages unequally, they tend to cluster references to certain pages in short time intervals, they can be run efficiently in memory spaces considerably smaller than the program size.[2,21,26] Locality derives from the behavior and style of programmers: they tend to use sequential and looping control structures frequently, they tend to group data into content-related blocks, and they tend to concentrate on small parts of large problems for moderately long intervals.[5–7,15,24,26] The degree to which locality is exhibited is strongly a function of programming style—i.e., algorithm strategy and data organization;[5–7,26] it is reported that high degrees of locality can often be achieved at a cost of under 5 percent of total programming costs.[26] As will be seen below, the definition of working set as a collection of recently referenced pages improves as an estimator of a program's working information as the degree of locality increases.

Based on the foregoing ideas, one can formulate a model for locality. Suppose P is the set of pages of a given program. Associated with the given program is a collection of *localities*, each a distinct subset of P; we denote them by $A, B, C, \ldots$. Two localities may overlap. As suggested in Figure 2, one may imagine that the executing program traces a path of operation through its localities; the path defines a sequence of localities, e.g., in Fig. 2 it is

$$L_1 L_2 L_3 \ldots L_i \ldots = A\ B\ D\ C\ B\ E\ C \ldots$$

Now, let t_i denote the length of time (the *residence time*) the program spends in the ith locality L_i of the locality sequence; while the path is in locality L_i, the program generates a reference substring of length t_i, consisting of pages from L_i only. As an example, consider a program with pages $P = \{1, 2, \ldots, 6\}$ and three localities

$$A = \{1, 2, 3\} \qquad B = \{4, 5, 6\} \qquad C = \{1, 2, 6\}$$

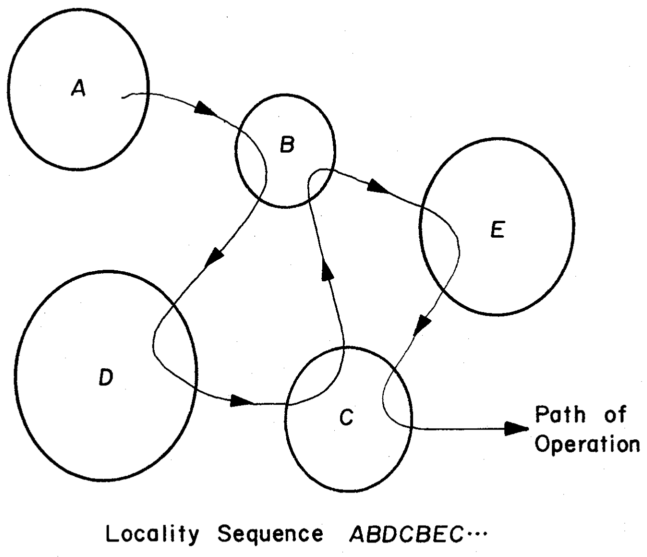

Figure 2—Locality

Suppose the locality sequence is $L_1L_2L_3 = ABC$ and the residence times are $t_1t_2t_3 = 656$. A reference string consistent with these conditions is

$$\underbrace{1\,3\,2\,1\,2\,3}_{A}\quad\underbrace{6\,4\,5\,6\,4}_{B}\quad\underbrace{6\,1\,2\,6\,2\,1}_{C} \qquad (1)$$

Based on the property of locality, one would expect that the residence times tend to be long and that neighboring localities (on the path of operation) tend to overlap.

Ideally, a program's working set at any given time should comprise exactly the pages from the current locality at that time. In practice (at least without the services of a Delphic Oracle) the working set will be an estimate of locality based on observations of the recent past reference string. Accordingly, if $r_1r_2 \ldots r_t \ldots$ is a reference string, we define the *working set* $W(t, T)$ at time t to be the set of distinct pages referenced among the T most recently referenced pages, i.e., among $r_{t-T+1} \ldots r_t$. If $t<T$, $W(t, T)$ consists of the distinct pages among $r_1r_2 \ldots r_t$ and if $t<0$, $W(t, T)$ is empty. The parameter T is called the *window size,* since $W(t, T)$ can be regarded as the contents of a window looking backward at the reference string. The working sets for $T=4$ on the reference string (1) given above are shown in Figure 3.

The working set is *correct* when it comprises exactly the pages of the current locality. The correctness depends on the choice of window size T; in general, two conditions would have to be satisfied:

1. T is sufficiently large that every page of a locality is in the working set with high probability.
2. T is comparable to or much less than the average locality residence time, i.e., the probability of the window's containing more than one interlocality transition is small.

	Refs.	1	3	2	1	2	3	6	4	5	6	4	6	1	2	6	2	1
Locality	Page																	
AC	1	x	x	x	x	x	x	x						x	x	x	x	x
AC	2			x	x	x	x	x	x						x	x	x	x
A	3		x	x	x	x	x	x	x	x								
B	4								x	x	x	x	x	x	x			
B	5									x	x	x	x					
BC	6							x	x	x	x	x	x	x	x	x	x	x
Correctness -				x	x	x	x				x	x				x	x	x
Size -		1	2	3	3	3	3	4	4	4	3	3	3	3	4	3	3	3

Figure 3—Working sets for $T = 4$

The higher the degree of locality exhibited by the program, the more likely it is that T can be chosen to satisfy the two conditions, and consequently the larger the fraction of time during which the working set is a correct estimate of current locality. Additional factors influencing the choice of T will be mentioned shortly.

The use of a moving window to estimate locality as discussed above is more a matter of convenience than necessity. In a practical implementation it would be sufficient to associate "use-bits" with pages; a page's use-bit is set when the page is referenced, and is read and reset at the end of each interval of T references. The use-bits found to be set at the end of a T-interval would define a working set, and the working set so measured would be used as an estimator of locality until updated at the end of the subsequent T-interval. Indeed, my original proposal for working sets was based on this idea.[10] The T-intervals can coincide with the time slices used to schedule programs.[9,11,12,22,23,27]

Suppose one postulates a "working set paging algorithm"—i.e., a page replacement algorithm that replaces only the non-working set pages of a program. It can be demonstrated that this algorithm will implement the "principle of optimality" for page replacement (replace the page not expected to be used again for the longest time) provided locality is good.[15] In other words, the use of locality as a basis for memory management is close to being optimal.

THE WORKING SET PRINCIPLE AND THRASHING

The *working set principle* for memory management states that a program may be active (i.e., eligible to receive time slices on a processor) only if its working set is loaded in main memory.[11,12,17] It follows from this statement that the number of active programs is limited by the main memory capacity and the sizes of working sets; indeed, if there are n active programs, the main memory capacity is M pages, and the ith working set $W_i(t, T_i)$ contains w_i pages, the working set principle states that the relation

$$w_1 + \cdots + w_n \leq M \qquad (2)$$

must be true with high probability at all times. It follows from this statement that only the pages not in the working set of any active program are subject to removal from main memory; in particular, if every page in memory were a member of a working set, it would be necessary to deactivate some program in order to create candidate pages for removal.

If the working set principle is not followed, it is possible for attempted overcommitment of memory to precipitate *thrashing,* a collapse of system performance, especially when the ratio of speeds between auxiliary and main memory is high.[9,12,13,15,23,27] It is of course possible for the working set principle to be violated without having thrashing; the point is that the principle defines a sufficient condition for avoiding thrashing.[13,15]

An efficient implementation of the principle would employ a compromise between the ideas of demand paging and swapping. Demand paging would be used to acquire pages during a program's active intervals, and swapping would be used to move a working set as a unit at the beginnings and ends of active intervals. Such strategy can be remarkably effective when properly designed.[20]

WORKING SET SIZE AND PAGING RATE

The working set size $w(t, T)$ is the number of pages in the working set $W(t, T)$. The average size for the first k references is

$$s_k(T) = \frac{1}{k}\sum_{t=1}^{k} w(t, T) \tag{3}$$

and the limiting *average size* is

$$s(T) = \lim_{k\to\infty} s_k(T) \tag{4}$$

(The error introduced by using $s(T)$ when in practice one would use $s_k(T)$ for some k is not significant under the assumptions of locality.[17]) Consider a reference string segment $r_1r_2 \ldots r_k$; for this segment, let $q(k)$ denote the number of times r_{t+1} was not in $W(t, T)$, $0 \leq t < k$. The *missing-page rate* is

$$m(T) = \lim_{k\to\infty} \frac{q(k)}{k} \tag{5}$$

so that $m(T)$ measures the average rate at which pages are entering the working set. Note that $m(T)$ is an upper bound on the page-fault rate of a single program in a system using the working set principle of memory management, for a page may leave the working set and subsequently return without leaving main memory in the meantime.

The important properties of $s(T)$ are summarized in Figure 4. The curve $s(T)$ is upper-bounded by the smaller of T and N, where N is the size of the program; $s(T)$ is nondecreasing in T, i.e., $s(T) \leq s(T+1)$; $s(T)$ is concave downward; and the "slope" of $s(T)$ is the missing-page rate:

$$s(T+1) - s(T) = m(T) \tag{6}$$

Finally, $m(T)$ is nonincreasing, i.e., $m(T) \geq m(T+1)$. The proof of (6) is straightforward: note that

$$w(t+1, T+1) = w(t, T) + \Delta w \tag{7}$$

where Δw is 1 if the next reference r_{t+1} is not in $W(t, T)$ and 0 otherwise, i.e., Δw is 1 if and only if the next page referenced is missing. Observe that the expected value of Δw is $\overline{\Delta w} = m(T)$. Taking expectations on both sides of (7), one finds

$$s(T+1) = s(T) + \overline{\Delta w} = s(T) + m(T) \tag{8}$$

which is equivalent to (6). A complete discussion of these properties, together with proofs, can be found in Reference 17. Experimental verification is presented in Reference 25. Further relations between $s(T)$ and $m(T)$ and page "interreference-interval distributions" can be found in Reference 17.

PRACTICAL IMPLEMENTATIONS OF WORKING SET POLICIES

There is a close kinship between the working set model and LRU (least-recently-used) paging, a kinship which can be used to approximate the working set model in systems where implementing a window T is impractical. The LRU paging algorithm is a demand-paging algorithm for managing a main memory space held fixed at k pages; at each page fault the least-recently-used page is replaced to make way for the page entering memory, so that the memory always contains the k most recently used pages. By comparison, a working set

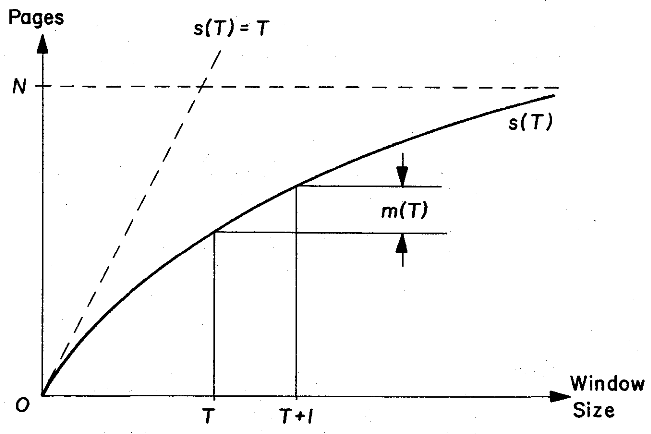

Figure 4—Mean working set size

$W(t, T)$ always contains $w(t, T)$ most recently used pages, where $w(t, T)$ is variable. It follows immediately that the $s(T)$ and $m(T)$ curves of a program can be used to estimate the page-fault rate $F(k)$ of the LRU algorithm in a k-page memory; see Figure. 5. A more detailed argument[17] shows that, under general conditions, $F(k) \geq m(T_k)$.

Now, one can imagine implementing a working set policy according to which the window size T is varied dynamically so that the missing-page rate $m(T)$ is controlled to be within a range m_1 to m_2, where $m_1 > m_2$. The importance of such a policy is twofold: not only does it imply each program operates in a known range of processing efficiency, but it allows a system designer to determine bounds on the paging traffic generated by a given statistical mix of active programs. The policy determines a sequence of times $t_0 t_1 t_2 \ldots t_i \ldots$ such that $T_i = t_i - t_{i-1}$ is the ith value of window size and (t_{i-1}, t_i) is the ith "sampling interval." During the ith sampling interval the program is run under a working set policy with window size T_i, the number p_i of pages entering the working set during that interval is counted, and p_i/T_i serves as an estimate for the missing page rate $m(T_i)$ during that interval. If p_i/T_i exceeds m_1 then T_i was too small and T_{i+1} is set to a value larger than T_i; if p_i/T_i is in the range m_1 to m_2, T_{i+1} is the same as T_i; and if p_i/T_i is smaller than m_2 then T_i is too large and T_{i+1} is set to a value smaller than T_i.

By virtue of the relation between the working set model and LRU paging, one can approximate the foregoing procedure using LRU paging in a memory space whose size is varied from time to time. In this case, T is taken as a fixed sampling interval (e.g., a time slice), p the number of page faults in the previous sampling interval, and p/T the estimate of page fault rate. If p/T exceeds m_1, the program is allocated one additional page of memory during the next sampling interval (this page being filled on demand at the next fault); if p/T is in the range m_1 to m_2, the memory allocation is unchanged; and if p/T is smaller than m_2, the memory allocation is reduced one page during the next sampling interval (the least-recently-used page being removed). This process is repeated for each sampling interval. During each interval the memory space allocation is fixed and LRU paging is employed. At any time the contents of memory serve as an estimate of the working set.

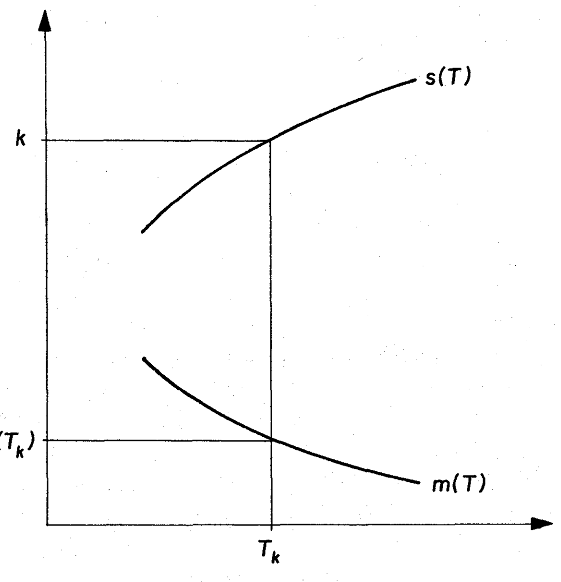

Figure 5—Estimating LRU paging rate

It is important to note that, if the LRU approximation to the working set principle is used in a multiprogrammed memory, a page fault generated by a program causes a page to be replaced from that program's own working set but never from another program's working set. Put another way, the LRU algorithm is applied on a "local" basis but not on a "global" one. Thus a program's missing-page rate depends only on its own behavior and never on that of another program: its working set survives intact despite the paging activities of other programs with which it shares main memory.

DISTRIBUTION OF WORKING SET SIZE

It can be shown[17] that whenever the third property L3 of locality holds—reference substrings tend to become uncorrelated as the interval between them becomes large—the working set size is approximately normally distributed about its mean $s(T)$. Suppose $w(t, T)$ is a working set size process with mean s and variance σ^2. The normal approximation for the distribution of $w(t, T)$ asserts that, for large t,

$$\Pr[w(t, T) \leq x] = \Phi\left(\frac{x-s}{\sigma}\right) \tag{9}$$

where

$$\Phi(x) = \frac{1}{(2\pi)^{1/2}} \int_{-\infty}^{x} e^{-u^2/2}\, du \tag{10}$$

is the cumulative distribution of a normally distributed random variable with zero mean and unit variance. The ability to consider $w(t, T)$ as a normally distributed random process is of key importance, in view of the enormous literature and theory extant about such processes.[8]

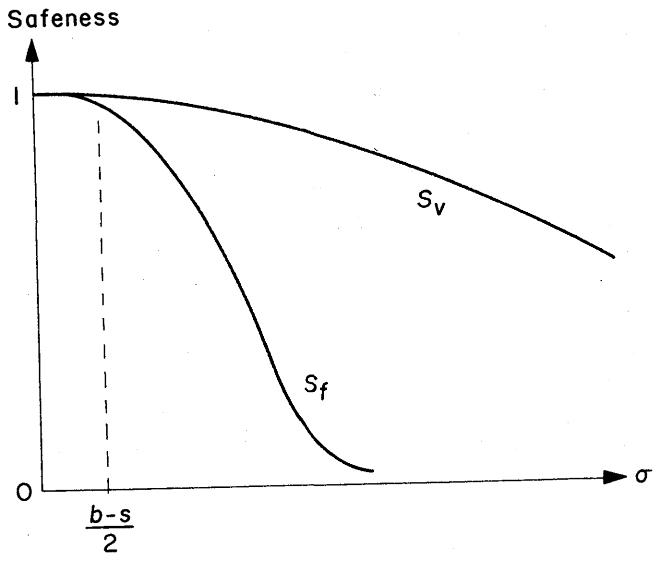

Figure 6—Safeness of fixed and variable partitioning

Two deviations are inherent in the use of the normal approximation to the distribution of working set size: the normal distribution is continuous whereas working set size is discrete, and the normal distribution is defined for negative values. The first deviation is not significant for programs whose working set size varies over a wide enough range of values, and the second is not significant for most parameter values (s and σ) of interest.

The normal approximation has been found remarkably accurate when the correlation property (L3) of locality holds.[8] If this property fails to hold (e.g., the program contains a global loop), it will be possible to predict arbitrarily distant future reference patterns on the basis of present ones and significant deviations from the normal approximation will be observed.[25]

FIXED VERSUS VARIABLE PARTITIONING OF MEMORY

The strategies for allocating main memory, under multiprogramming, usually lie near one of two extremes. According to the *fixed partition* strategy, each of the programs sharing the memory is allocated a fixed amount of memory for its private use. According to the *variable partition* strategy, the amount of space allocated to each program depends on its working set size. The principal argument advanced for fixed partitioning is simplicity of implementation; a recent study by Coffman and Ryan, however, suggests that the cost of implementing variable partitioning is more than adequately compensated by the saving in memory requirement.[8]

Under the assumption that n programs whose memory demands are normally distributed working set processes with the same mean s and variance σ^2, Coffman and Ryan compared a fixed partition and a variable partition strategy with respect to a "safeness factor" as a performance measure (the safeness factor is related to the volume of paging and overall efficiency). In both cases the total mount of main memory was M pages. In the fixed partition case, each program is allocated a block of b pages and the total memory capacity is $M = nb$; a block is considered safe (unsaturated) whenever the instantaneous value of working set size for the program using that block does not exceed b, and the multiprogrammed memory is considered safe whenever all the blocks are safe at the same time. In the variable partition case, each program is allocated an amount of space equal to its working set size; this system is considered safe if the sum of the working set sizes does not exceed the memory capacity M. The values of S_f, the safeness factor for fixed partitioning, and S_v, the safeness factor for variable partitioning, are readily determined under the assumption of normally distributed working set size. The results are suggested in Figure 6. Whenever the variance σ^2 of working set size is small enough so that the block size b is at least $s+2\sigma$ [equivalently, σ is less than $(b-s)/2$], the two strategies are competitive; but when σ becomes large, the variable partition strategy is considerably safer than the fixed partition strategy.

Under the assumption of variable partitioning and a working set memory management policy, it is possible to specify the equipment configuration (relative amounts of processor and memory resources) of a computer system.[14]

CONCLUSIONS

Originally introduced as a means of understanding the vagaries of multiprogrammed memory management, the working set model has proved useful both as an aid for guiding one's intuition in understanding program behavior, and as a basis for analyses and simulations of memory problems. To summarize: it has led to a sharpening of the notions "working information" and "locality of reference"; it has aided in the development of multiprogrammed memory management strategies which guarantee each program a given level of efficiency and which prevent thrashing; it has quantified the relationship between memory demand and paging rate; it has led to better understanding of paging algorithm behavior in both demand and prepaging modes; and it has enabled preliminary analysis of fixed *v.* variable partitioning of multiprogrammed memories and of system equipment configurations. Approximations to the working set principle have been used successfully in many contemporary systems both large and small, demonstrating the viability of the model: systems reporting success include the RCA Spectra 70/46,[9,23,27] certain TSS/360 installations,[18] the TENEX system for the PDP-10,[4] and MULTICS. Many systems have used, successfully, approximations to the working set model, and at least one is including hardware capable of implementing the moving-window definition of working set.[22]

Many extensions to the model are possible and worthy of future investigation. For example, the working set can be defined as the set of most recently used blocks of program code, the assumption of fixed sizes pages being eliminated. Or, it may be useful in some situations to partition the working set into two disjoint subsets, the instruction working set and the data working set, and to implement the working set principle separately for both these working subsets. Or, none of the foregoing considerations has allowed for the possibility of sharing—overlapping working sets—in certain cases the effects of sharing have been quantified, a notable reduction in total memory demand being observed.[12]

REFERENCES

1 Proceedings of a Symposium on Storage Allocation
ACM Comm ACM 4 10 October 1961

2 L A BELADY
A study of replacement algorithms for virtual storage computers
IBM Sys J 5 2 1966 pp 78-101

3 L A BELADY C J KUEHNER
Dynamic space sharing in computer systems
Comm ACM 12 5 May 1969 pp 282-288

4 D G BOBROW J D BURCHFIEL D L MURPHY R S TOMLINSON
TENEX—A paged time sharing system for the PDP-10
Comm ACM 15 3 March 1972

5 B BRAWN F GUSTAVSON
Program behavior in a paging environment
AFIPS Conference Proceedings Vol 33 Fall Joint Computer Conference 1968 pp 1019-1032

6 B BRAWN F GUSTAVSON
Sorting in a paging environment
Comm ACM 13 8 August 1970 pp 483-494

7 E G COFFMAN JR A C McKELLAR
Organizing matrices and matrix operations for paged memory systems
Comm ACM 12 3 March 1969 p 153ff

8 E G COFFMAN JR T A RYAN
A study of storage partitioning using a mathematical model of locality
Comm ACM 15 3 March 1972

9 W M DEMEIS N WEIZER
Measurement and analysis of a demand paging time sharing system
Proc 24th National Conference ACM August 1969 pp 201-216

10 P J DENNING
Memory allocation in multiprogrammed computer systems
MIT Project MAC Computation Structures Group Memo No 24 March 1966

11 P J DENNING
The working set model for program behavior
Comm ACM 11 5 May 1968 pp 323-333

12 P J DENNING
Resource allocation in multiprocess computer systems
MIT Project MAC Technical Report MAC-TR-50 PhD Thesis May 1968

13 P J DENNING
Thrashing—Its causes and prevention
AFIPS Conference Proceedings Vol 33 Fall Joint Computer Conference 1968 pp 915-922

14 P J DENNING
Equipment configuration in balanced computer systems
IEEE Trans Comp C-18 11 November 1969 pp 1008-1012

15 P J DENNING
Virtual memory
Computing Surveys 2 3 September 1970 pp 153-189

16 P J DENNING
Third generation computer systems
Computing Surveys 3 4 December 1971

17 P J DENNING S C SCHWARTZ
Properties of the working set model
Comm ACM 15 3 March 1972

18 W DOHERTY
Scheduling TSS/360 for responsiveness
AFIPS Conference Proceedings Vol 37 Fall Joint Computer Conference 1970 pp 97-112

19 E L GLASER J F COULEUR G A OLIVER
System design for a computer for time sharing application
AFIPS Conference Proceedings Vol 27 Fall Joint Computer Conference 1965 pp 197-202

20 M L GREENBERG
An algorithm for drum storage management in time sharing systems
Proceedings 3rd ACM Symposium on Operating Systems Principles October 1971

21 J S LIPTAY
The cache
IBM Sys J 7 1 1968 pp 15-21

22 J B MORRIS
Demand paging through utilization of working sets on the Maniac II
Los Almos Scientific Labs Los Alamos New Mexico 1971

23 G OPPENHEIMER N WEIZER
Resource management for a medium scale time sharing operating system
Comm ACM 11 5 May 1968 pp 313-322

24 B RANDELL C J KUEHNER
Demand paging in perspective
AFIPS Conference Proceedings Vol 33 Fall Joint Computer Conference 1968 pp 1011-1018

25 J RODRIGUEZ-ROSELL
Experimental data on how program behavior affects the choice of schedular parameters
Proceedings 3rd ACM Symposium on Operating Systems Principles October 1971

26 D SAYRE
Is automatic folding of programs efficient enough to displace manual?
Comm ACM 13 12 December 1969 pp 656-660

27 N WEIZER G OPPENHEIMER
Virtual memory management in a paging environment
AFIPS Conference Proceedings Vol 34 Spring Joint Computer Conference 1969 pp 234ff