Technical Report CS74025-R

Working-Set-Like Memory Management Algorithms[1]

Felix L. Lam[2]

Domenico Ferrari[3]

1. Work reported herein was supported in part by the U. S. Navy under Contract (NESC) N00039-71-C-0255.

2. Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061.

3. Computer Science Division, Department of Electrical Engineering and Computer Sicence, University of California, Berkeley, California 94720.

Abstract

This paper considers the design and evaluation of memory management algorithms to be applied to multiprogramming computer systems with virtual memory. The operation of the Denning working set algorithm is studied and is recognized to be a replacement process of time indices based on a rule closely related to the replacement rule of the First-In-First-Out replacement algorithm. Basing on these analyses, a framework in the time domain is then proposed. A duality rule capable of transforming a replacement algorithm in the space domain into a working-set-like algorithm (retention algorithm) in the time domain is designed. Some properties of these newly-designed retention algorithms are derived. The performances of some retention algorithms with respect to their space duals are experimentally studied by simulation. Results show generally better performance for retention algorithms than for their space dual replacement algorithms.

Key Words and Phrases: Memory management, space sharing, dynamic storage allocation, multiprogramming, virtual memory, working sets, replacement algorithms, program behavior, retention algorithms, space-time duality, demand-paging.

CR Categories: 4.30, 4.32, 4.35, 4.6

I. INTRODUCTION

This paper is concerned with the problem of dynamic memory management in a multiprogrammed computer system capable of supporting multiple virtual address spaces [1]. The basic hardware configuration, which includes a hierarchical memory system consisting of two levels, is presented in Fig. 1. There are in the system a number of processors having direct access to main memory, but not to auxiliary memory. Therefore, information to be processed must be resident in main memory at the time it is used by the processors. In general, main memory is of very limited size because it is made up of fast and therefore expensive memory devices, whereas auxiliary memory has a much larger capacity, but is relatively slow.

The rationale behind a memory hierarchy is the basic size-cost-performance tradeoff. If the memory hierarchy is intelligently managed so that information needed by the processors is found for most of the time in main memory, then a memory system with access time close to that of the main memory, but of a larger capacity and at a lower cost, is achieved. Finding an intelligent way to use a memory hierarchy is the objective of the memory management problem.

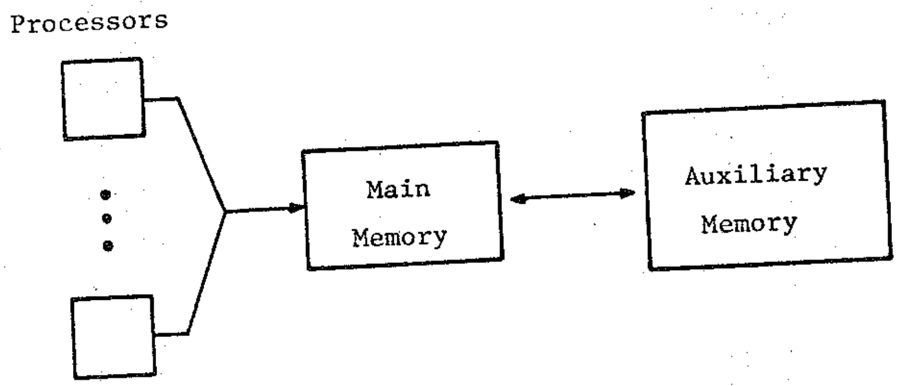

Figure 1. Basic System Hardware

The system in Fig. 1 is assumed to be multiprogrammed even if it may contain only one central processor (i.e., several processes are being concurrently executed and therefore parts of their programs and data must be in main memory at any instant). The partitioning of main memory may be static (i.e., each program is given a fixed portion of memory for as long as it executes) or dynamic (when the partitions separating the precesses in main memory are allowed to vary in time). Since the memory requirement of a program is likely to vary during execution, dynamic partitioning can provide, if the moving of partitions is properly handled, better memory utilization and higher system efficiency [2].

We also assume that the portion of main memory allotted to a process (either statically or dynamically) is in general not sufficient to contain all the information of that process. In other words, only portions of a process' virtual address space can be mapped into that process' allotted physical memory space at any given time.

A well-designed memory management scheme has to determine for each process which part of its virtual address space should be placed into its current main memory allotment so that the goal of minimizing information flow between levels of the memory hierarchy can be achieved. In other words, a process' working information (i.e., that part of its virtual address space having the best chance to be used in the immediate future) should be estimated. If main memory is assumed to be dynamically partitioned in the system in Fig. 1, the method used to dynamically estimate a process' working information should fit nicely into such a memory management scheme.

The problem of program working information estimation has attracted a lot of attention in the past, and a fair number of algorithms have been proposed to tackle it. Some of these algorithms have actually been implemented either in hardware or software in existing systems. To facilitate their discussion, we shall assume that in our system virtual memory is implemented by the paging technique, i.e., that memory management deals with uniform-size blocks called <u>pages</u> [1]. Furthermore, we assume that paging is done on a demand basis. This means that no pages are loaded into main memory until they are referenced. Then whenever a page is needed and not found in main memory, an interrupt, called a <u>page fault</u>, is generated, and page loading is performed through this page fault mechanism.

Under these assumptions, the task of estimating a program's working information can be simplified. In the case of static memory partitioning in which a program is allocated a fixed number of page frames for execution, these page frames are first filled up one at a time. No memory management actions are taken until an additional page is needed and no vacant page frame is found. Then an algorithm, called a <u>replacement algorithm</u>, is activated that selects for removal from main memory one page among those currently present in memory to make room for the incoming page. Different ways to select a page for replacement result in different replacement algorithms. If main memory is dynamically partitioned, it is desirable that an algorithm used to estimate a program's working information has the additional capability of estimating the program's current memory demand so as to facilitate the dynamic partitioning of main memory among several programs. As page loading is done on demand, this amounts to determining how many and which of the previously loaded pages retain in main memory as the estimate of a program's current working information.

In a paging system, the representation of a program's dynamic behavior can also be simplified, since an address trace, i.e., the sequence of virtual addresses generated by the program during one of its executions can be replaced by a string of page numbers called a page trace or a reference string. Let $a_1a_2 \ldots a_t \ldots a_\ell$ be an address trace representing a program, where a_t is a virtual address referenced at virtual time t (virtual time is taken to be discrete, $t = 1,2,3,\ldots,\ell$). Then, if r_t is the page containing address a_t, $r_1r_2 \ldots r_t \ldots r_\ell$ is the page trace. Since virtual time is measured discretely, t is actually an index to the reference r_t in the page trace. Consequently, we will refer to t as the time index of the reference r_t. Note that, in general, the lengths of (virtual) time intervals between successive references in the reference string differ from each other. But these time intervals are actually very short with respect to the time required to move a page between levels of the memory hierarchy so their variations can indeed by neglected in our study. In the next section, we shall review most of the methods proposed so far to estimate a process' working information.

II. REPLACEMENT ALGORITHMS AND THE DENNING WORKING SET ALGORITHM

As mentioned in Section I, replacement algorithms are used to estimate a program's working information in a fixed-space environment, and different replacement algorithms are characterized by different replacement rules, which, in turn, can be traced to some (generally heuristic) models of program behavior. More specifically, suppose that a program has n pages and is given m page frames in main memory for execution, where $n > m \geq 1$. Then, once the m page frames are occupied and an additional page is referenced that is not one of those pages currently contained in the m page frames, a page fault is generated. A replacement algorithm is then activated which selects one out of the set of m pages for removal from main memory according to its replacement rule.

It is clear that the above situation corresponds to a uniprogramming system in which only one program is given control of the system, or to a statically-partitioned multiprogramming system in which each program is given a fixed portion of main memory for execution. Replacement algorithms applied in these situations are said to be <u>local</u> algorithms as they deal with a single program. Only when replacement algorithms are locally applied can we talk about the models of program behavior underlying them. Indeed, local application of replacement algorithms was the framework assumed when they were first proposed and studied [3 - 5].

However, the same replacement algorithms can be applied to multiprogramming systems without any modifications, provided that the virtual address space in such applications is taken to be the sum of virtual address spaces belonging to all processes partially loaded in main memory. Thus, no distinction is made between pages belonging to different concurrent programs even though these programs can be logically independent. This is called the <u>global</u>, or <u>system-wide</u>, application of replacement algorithms. Since global replacement algorithms may cause an excessive paging activity (the thrashing phenonenon [6]), we shall restrict ourselves to the local application of replacement algorithms.

Given any program with n pages and main memory space of m page frames with $m < n$, the requirement placed on a replacement algorithm as a tool in memory management is to make "wise" replacement decisions so that a minimum number of page faults is generated when this program is executed in the allocated main memory of m page frames. The questions to ask are: (i) whether a minimum can actually be attained, and (ii) what information is required to do so. These questions were first answered by Belady [4] who proposed the MIN replacement algorithm. The principle of optimality underlying MIN states that at each replacement the page that will be referenced next in the most distant future should be replaced. Accordingly, knowledge of the future reference pattern is needed in order to make a current replacement decision. Thus, even though MIN yields a minimum number of page faults when a program completes execution, it is nonetheless unrealizable. It should be noted that MIN assumes no model of program behavior and can be applied to any program.

Realizable replacement algorithms, however, cannot wait for future information and must rely on past information about a program's referencing pattern to make replacement decisions. Different replacement rules use this past information in different ways and each particular rule reflects a particular assumption about how a program makes references to its pages, or how a program behaves, which we call an underlying model of program behavior. Inspired by the true principle of optimality discussed above, all these algorithms use an "informal principle of optimality" that states that the page to be replaced is that which has the longest expected time until next reference [7]. Of course, the program model underlying a replacement algorithm plays an important role in the estimation of a page's expected time until next reference. Thus, after initial loading, a replacement algorithm always attempts to keep in main memory the m pages that have a better chance of being references again as dictated by its underlying program model. In this way, replacement algorithms fulfill their role as estimators of a program's working information. But they do not estimate a program's memory demand, as they are designed for a fixed-space environment.

Let us consider, for example, the Least Recently Used (LRU) replacement algorithm. When a page faults occurs, LRU selects the page that has not been referenced for the longest time. As with any other replacement algorithms, the idea behind LRU is that a page that has not been needed during the recent past may have little chance of being referenced in the near future.

LRU is the most popular stack replacement algorithm. A replacement algorithm is classified as a stack algorithm [5] if for every page trace and at every point in time along the page trace, the "inclusion property" of memory contents holds, that is,the set of pages contained in an m-page-frame memory is a subset of the pages contained in an (m+1) - page - frame memory, for all $m + 1 \leq n$. A sufficient condition for a replacement algorithm to qualify as a stack algorithm is that it induces a total ordering of all previously references pages at every point in time and bases its replacement decisions on this priority list. In stack processing, at every point in time, every page in the program has a position, finite or infinite, in the stack. All previously referenced pages have a finite stack position; a page at the kth position in the stack at that instant is contained in a memory of at least k page frames, but not less. Pages that have not been referenced before are assumed to have a stack position at infinity. The stack is updated at every reference in the page trace and updating is based on the priority list induced by the replacement algorithm. The stack position of the page just referenced before the updating of the stack is called the stack distance; clearly, a distance string corresponds to any given reference string for a given stack algorithm.

The model of program behavior underlying the LRU replacement algorithm (known as the LRU stack model [11]) can be obtained by assuming the LRU distance string to be a sequence of independent and identically distributed random variables and by superimposing a probabilistic structure on the LRU stack positions: for any stack position i, $1 \le i \le n$, p_i is defined to be the probability that the page in this stack position will be referenced next (of course, these probabilities sum to 1). If, in addition, these probabilities are monotonically decreasing ($p_1 \ge p_2 \ge .....\ge p_n$), then since pages are arranged in the LRU stack according to their recency of reference, the page that is least recently referenced then has the smallest probability of being referenced next and hence is the candidate for replacement if it is in memory.

It can be shown [8] that the monotonically decreasing probability distribution defined on the LRU stack positions in this model does guarantee the implementation of the "informal principle of optimality"in page replacement for the LRU replacement algorithm. That is to say, the least recently used page, and hence the page with the smallest probability of being referenced next according to the LRU stack model of program behavior, is indeed the page with the longest expected time to its next reference.

The question of whether there is a "best" replacement algorithm for a given program or set of programs must unfortunately be given a negative answer: our as yet limited experimental findings (see for example [9]) seem to indicate that a program does not generally behave according to any single program model; rather, its behavior may be represented as a "chronological" mixture of several models.

Unlike replacement algorithms, the <u>Denning Working Set</u> algorithm (DWS) tries to dynamically estimate not only which subset of pages a program requires, but also the cardinality of such a subset, i.e., the memory demand of a program. DWS produces a subset of a program's pages at every point t in time, called the working set at time t, according to the program's past referencing pattern, and uses it as an estimator of the program's current working information. Not only are the contents of such working sets a function of time, so is their cardinality.

If $r_1 r_2 \ldots r_t \ldots r_\ell$ is a program reference string, then the working set of pages at time t ($1 \leq t \leq \ell$) is the set of distinct pages referenced in the interval $[t-T+1, t]$ inclusively, where T is a parameter of DWS called the window size. The working set at time t is denoted by $W(t,T)$, and its cardinality by $\omega(t,T)$. Conceptually, the procedure for computing the working set according to the DWS algorithm can be considered as the one of sliding a (backward) window of constant size T along the reference string and extracting the set of distinct pages referenced within this window.

The most important difference between DWS and (local) replacement algorithms is that DWS, which has the capability to estimate an individual program's memory demand, is suitable to use in systems with dynamic memory partitioning. Also, Denning [6] recommends a working set principle for memory management which states that a program may be active (i.e., eligible to use a processor) only if its working set is loaded in main memory. Denning shows that, if the working set principle is followed, the thrashing phenomenon may not occur. Thus, DWS is a memory management algorithm which can interact with the scheduling component of the operating system to avoid thrashing.

How is the procedure for computing the working sets by DWS related to program referencing patterns? An answer to this question will be given in the next section.

III. SPACE-TIME DUALITY BETWEEN MEMORY MANAGEMENT ALGORITHMS

In Section II, we have seen that, conceptually, the application of the DWS algorithm involves sliding a window of constant size T along the reference string and at every time instant using the references covered by this window to identify the working set of pages. Now it should be clear that for any given reference string of length ℓ, any subset of time indices of the set $\{1,2,\ldots,\ell\}$ uniquely defines a set of distinct pages. Then the process in which the DWS algorithm computes a program's working set can be viewed as one in which a subset of time indices is collected at every time instant (i.e., the set of time indices within the constant-size window) for the purpose of identifying the working set at that instant. We shall now show that this collection process can indeed be interpreted as a replacement process.

When $t \leq T$, the window covers the references $r_1 r_2 \ldots r_t$, and when $t > T$, it covers the references $r_{t-T+1} \ldots r_t$. If the window is thought of as a container capable of holding T time indices, then it is first filled up with the initial T time indices. At $t = T$, the window is full and contains the set of time indices $\{1,2,\ldots,T\}$. At $t = T+1$, the window will contain the time indices $\{2,3,\ldots,T,T+1\}$. This change in the contents of the window from $t = T$ to $t = T+1$ is clearly a replacement process in which the time index 1 is removed from the window to make room for the time index $T + 1$. The same replacement procedure repeats itself at $t = T+2$ and beyond, up to $t = \ell$.

This alternative description of the process involved in applying the DWS algorithm would be uninteresting if it were not for the strong analogy it bears to the process involved in applying any replacement algorithm. While replacement algorithms are designed for a constant-space framework, the DWS is seen to operate with a constant-size window. While a memory space of constant size can contain a fixed number of pages, a window of constant size is seen to be able to contain a fixed number of time indices. When a fixed-size memory space is full and a new page is needed, a page is chosen for replacement by the incoming page: this is the replacement process associated with replacement algorithms. When the fixed-size window is full (for $t \geq T$), a time index is chosen for replacement by the next time index: this is the replacement process associated with the DWS algorithm.

There are two apparent differences between these two types of replacement process. First, for replacement algorithms, a replacement of pages is not necessary for every reference even when the memory space is full. This is so because references at different times can be made to the same page. But a replacement of time indices is necessary for every reference in applying the DWS algorithm once the window is full, as no two time indices are the same. Second, once the memory space is full, a replacement algorithm uses the fixed number of pages currently in main memory as an estimate of a program's working information. But under the DWS algorithm, the working set size can change dynamically, as references corresponding to different time indices can be made to the same page. These two differences are indeed complementary in nature. On one hand, no replacement (of pages) is needed for every reference but no (or just a constant) memory demand of a program is estimated. On the other hand, a replacement (of time indices) is necessary for every reference but the memory demand of a program is estimated dynamically. In fact, this is a classical case in which the complexity (and hence the cost of implementation) of an algorithm is traded for its performance.

As replacement algorithms are characterized by replacement rules, it is of immediate interest to find out what rule guides the DWS replacement process. It is easy to see that, by proper interpretation, the replacement rule associated with DWS is closely related to that for the First-In First-Out (FIFO) replacement algorithms, which assigns the lowest priority to the page that has been referenced first.

At time $t \geq T$, the window contains time indices $\{t-T+1, t-T+2, \ldots, t\}$; at $t+1$, the contents of the window change to $\{t-T+2, t-T+3, \ldots, t, t+1\}$. Clearly, in order to admit the time index $t+1$ into the window, the time index $t-T+1$ is selected for removal, and this time index always corresponds to the page that has been first referenced among all pages in the current working set. Note that there may be more than one time index in the window corresponding to this page in the working set; in such cases, $t-T+1$ proves to be the smallest such time index. Thus, the replacement rule (of time indices) for the DWS algorithm can be stated as follows: Replace the smallest time index that corresponds to the page which has been referenced first among all pages in the current working set. In other words, in the replacement process associated with the DWS algorithm, the time index replaced corresponds to the lowest-priority page in the working set, and the page that is assigned the lowest priority among all pages in the working set is the one that has been first referenced.

Thus, both the replacement rules of FIFO and DWS assign the lowest priority to the page which was referenced first. For this reason, FIFO and DWS are said to have dual replacement rules (in fact, the replacement rule of DWS will be called a retention rule). We shall now define a framework in the time domain built around the constant-size window in which DWS operates, and define a principle of duality between replacement algorithms in the space and in the time domains.

We have seen that replacement algorithms operate in a memory space of constant size, whereas the DWS algorithm operates with a window of constant size. Since a constant-size memory space consists of a fixed number of page frames each of which can hold a page, it is possible to conceptualize the constant-size window as consisting of a fixed number of <u>time frames</u> each of which can hold a time index. To complete the conceptual framework in the time domain, the term "window" will be replaced by "memory span". The reason for this is that, unlike the DWS algorithm which always keeps consecutive time indices so that it makes sense to talk about a window, other DWS-type algorithms may not always do that and therefore cause the window to be "broken". While a framework in the space domain consists of a memory space with a fixed number of page frames, a dual framework in the time domain consists of a memory span with a fixed number of time frames. Algorithms operating in the space framework to estimate a program's working information are known as replacement algorithms. We call <u>retention algorithms</u> those operating in the time framework to estimate a program's working information. The dual components of these frameworks are listed in Table I.

Table I. Dual Components of the Space and Time Frameworks

Space	Time
Memory Space	Memory Span
Page Frame	Time Frame
Page	Time Index
Replacement Rule	Retention Rule
Replacement Algorithm	Retention Algorithm

It has been seen that the DWS algorithm is a retention algorithm which bases its replacement of time indices on a rule dual to the replacement rule of FIFO. The resulting set of time indices contained in the memory span (window) is then used to identify the working set. Thus, one may wonder whether and how other retention algorithms may be derived from known replacement algorithms. Indeed, the following duality rule can be used to create a retention algorithm from a given replacement algorithm.

Duality Rule: Given a replacement algorithm X, the dual retention algorithm DX operating in the dual framework in the time domain replaces the (smallest) time index corresponding to the lowest priority page in the working set designated by the replacement rule of X.

As an example, the retention rule of LRUT, the dual retention algorithm of LRU defined by this duality rule, calls for the replacement of the (smallest) time index corresponding to the least recently used page in the working set. LFUT, the dual retention algorithm of the LFU (Least Frequently Used) replacement algorithm, is similarly defined. In Tables II and III, we illustrate by an example how the LRUT and LFUT retention algorithms work. Note that time indices contained in the memory span are not always consecutive and that the resulting working set size varies dynamically in much the same way as with the DWS algorithm. Note also that, in Table III, laxicographic ordering is used to break ties for the LFU priority list.

IV. SOME PROPERTIES OF RETENTION ALGORITHMS

In this section, we present a few properties of retention algorithms in the form of propositions. To this end, the formal definitions of some of the concepts introduced informally in the previous section need be given.

Table II: The LRUT Retention Algorithm

t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
r_t	A	B	A	B	A	A	B	B	D	B	E	E	E	C	F	A	F	F
LRU	A	B	A	B	A	A	B	B	D	B	E	E	E	C	F	A	F	F
Priority		A	B	A	B	B	A	A	B	D	B	B	B	E	C	F	A	A
List									A	A	D	D	D	B	E	C	C	C
											A	A	A	D	B	E	E	E
														A	D	B	B	B
															A	D	D	D
Contents of	1	1	1	1	5	5	5	8	8	8	8	12	12	12	15	15	15	15
Memory Span		2	2	4	4	6	6	6	9	9	11	11	11	14	14	14	17	17
of 3 Time			3	3	3	3	7	7	7	10	10	10	13	13	13	16	16	16
Frames																		
Working	A	A	A	A	A	A	A	A	D	D	E	E	E	E	E	A	A	F
Set		B	B	B	B		B	B	B	B	B	B		C	C	C	F	
															F	F		
Contents of	1	1	1	1	5	5	5	5	9	9	11	11	13	13	13	13	17	17
Memory Span		2	2	2	2	6	6	6	6	10	10	10	10	14	14	14	14	18
of 4 Time			3	3	3	3	3	8	8	8	8	8	12	12	12	16	16	16
Frames				4	4	4	7	7	7	7	7	12	11	11	15	15	15	15
Working	A	A	A	A	A	A	A	A	A	D	E	E	E	E	E	E	A	A
Set		B	B	B	B	B	B	B	B	B	B	B	B	C	C	C	C	F
									D						F	F	F	
																A		

Table III: The LFUT Retention Algorithm

t		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
r_t		A	B	A	B	A	A	B	B	D	B	E	E	E	C	F	A	F	F
Frequency Counts	A	1	1	2	2	3	4	4	4	4	4	4	4	4	4	4	5	5	5
	B	0	1	1	2	2	2	3	4	4	5	5	5	5	5	5	5	5	5
	C	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
	D	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
	E	0	0	0	0	0	0	0	0	0	0	1	2	3	3	3	3	3	3
	F	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	2	3
LFU Priority List		A	A	A	A	A	A	A	A	A	B	B	B	B	B	B	A	A	A
			B	B	B	B	B	B	B	B	A	A	A	A	A	A	B	B	B
										D	D	D	E	E	E	E	E	E	E
												E	D	D	C	C	C	F	F
															D	D	D	C	C
																F	F	D	D
Contents of Memory Span of 3 Time Frames		1	1	1	1	1	6	6	6	6	6	6	6	6	6	6	6	6	6
			2	2	4	5	5	5	5	5	5	11	12	13	14	15	16	16	16
				3	3	3	3	7	8	9	10	10	10	10	10	10	10	17	18
Working Set		A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
			B	B	B			B	B	D	B	B	B	B	B	B	B	F	F
												E	E	E	C	F			
Contents of Memory Span of 4 Time Frames		1	1	1	1	1	1	7	8	9	10	10	10	10	10	10	10	17	18
			2	2	2	5	5	5	5	5	5	5	5	5	5	5	5	5	5
				3	3	3	3	3	3	3	3	11	12	13	14	15	16	16	16
					4	4	6	6	6	6	6	6	6	6	6	6	6	6	6
Working Set		A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
			B	B	B	B		B	B	D	B	B	B	B	B	B	B	F	F
												E	E	E	C	F			

Definition 1: The alphabet N = {A,B,C,...} of a program is a finite set of pages the program makes reference to at least once during its execution.

Definition 2: A program reference string $R = r_1r_2..r_t..r_\ell$ is a finite sequence of references, each of which is an element of the alphabet N. For any reference r_t, $1 \le t \le \ell$, t is called the time index of this reference and the finite set of positive integers $\{1,2,...,\ell\}$ is called the index set I of R.

Definition 3: A time frame is an information container which either is empty or contains any time index from the index set I.

Definition 4: A memory span is a finite collection of time frames. A memory span of T time frames, $T > 0$, is said to have capacity or size T. The subset of time indices from the index set I contained in a memory span of capacity T at time t is denoted by M(t,T).

Definition 5: Given a program reference string R and a memory span of capacity T, the working set at time t, $1 \le t \le \ell$, denoted by W(t,T), is the set of distinct pages corresponding to time indices contained in the memory span, i.e., M(t.T). The cardinality of W(t,T), denoted by $\omega(t,T)$, is called the working set size.

Note that Definition 5 generalizes the definition of the term "working set" given by Denning [1, 10]. In fact, the working set resulting from Definition 5 (by any retention algorithm) is considered to be an estimate of a program's working information and the working set size its memory demand. With these preliminary definitions, we can proceed to define retention algorithms.

Definition 6: A memory management scheme used to dynamically estimate a program's working information and its memory demand is called a retention algorithm if an only if

(1) it has a retention rule which, when called for, selects a time index for removal from the memory span of T time frames; a retention rule may maintain a priority list of pages used by the program; in case that a tie occurs among several time indices, there must be a tie-breaking rule; and (2) it operates according to the flowchart shown in Fig. 2.

In the last section, we have seen that useful retention rules can be designed from the knowledge of replacement rules through the space-time duality transformation. This idea is formalized in the following definition.

Definition 7: Given a replacement algorithm X with a replacement rule RRX, then the retention algorithm DX is said to be <u>the dual of X in the time domain</u> if and only if the retention rule RRDX of DX reads: Select for removal the smallest time index from the memory span that corresponds to the page in the working set considered the lowest priority page by RRX.

Note that, in the above definition, selecting the smallest time index among a few time indices all corresponding to the same page is the tie-breaking rule adopted. It can be immediately recognized that the following property holds for the retention algorithms defined above.

<u>Proposition 1</u>: Given any program reference string $R = r_1 r_2 \ldots r_t \ldots r_\ell$, for all $1 \le t \le \ell$, the page kept in a memory space <u>of capacity 1</u> being managed by any replacement algorithm is identical to the (necessarily single) page in the working set corresponding to the time index kept in a memory span <u>of capacity 1</u> being managed by any retention algorithm.

We now consider the duals of an important class of replacement algorithms, the one of stack algorithms. We recall that a sufficient condition for a replacement algorithm to qualify as a stack algorithm is that it induces a priority ordering of all previously referenced pages at every point in time and

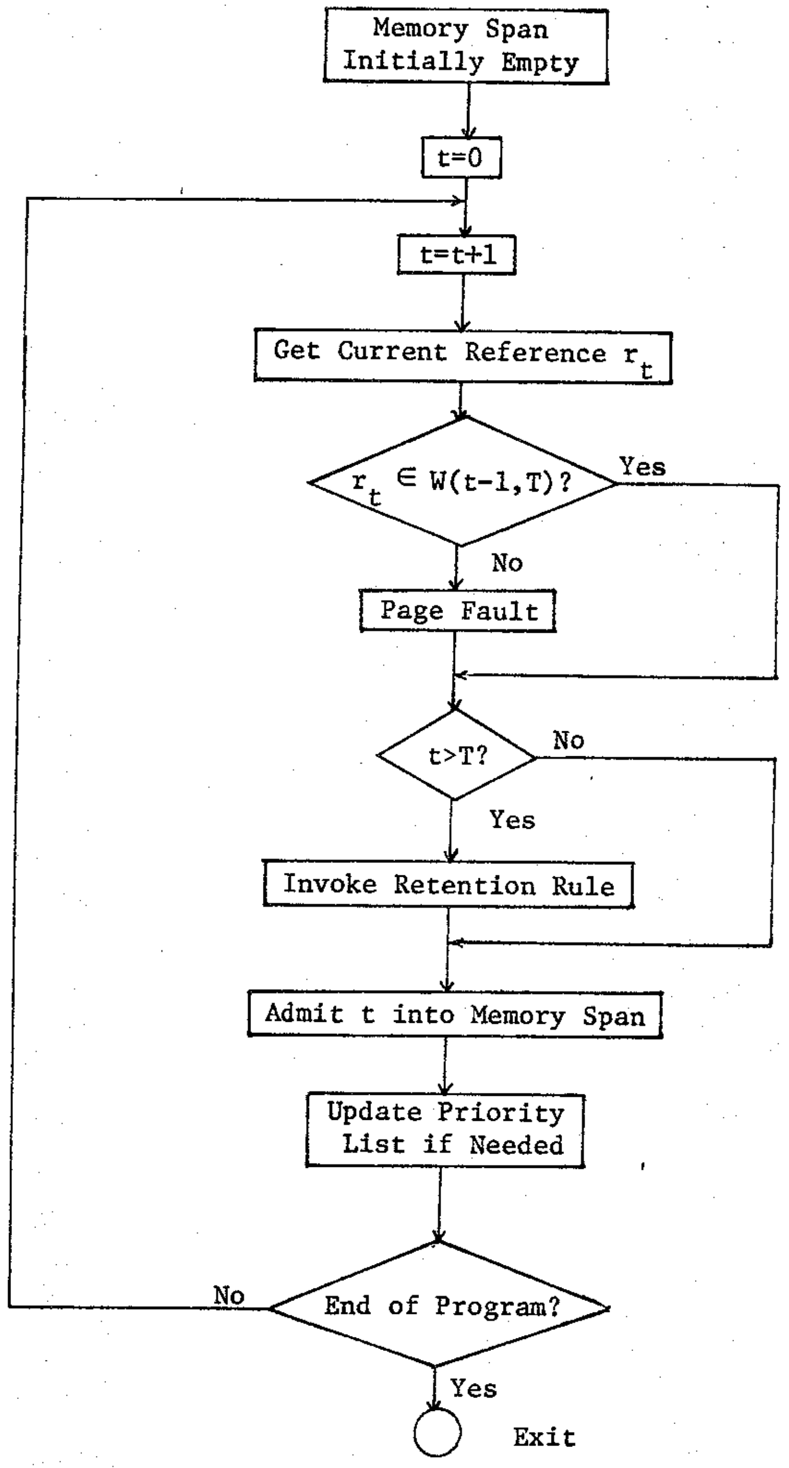

Figure 2. Flow Chart for Retention Algorithm Operations

bases its replacement decisions on this ordering. It has also been shown that priority lists can be constructed for every stack algorithm [15]. Because of the priority list maintained by the stack algorithms, the resulting dual retention algorithms possess some properties that are not generally true. These properties will only be stated here. Their proofs, which are often easy to construct, can be found in [8].

Proposition 2: Let X be a stack algorithm and DX its dual retention algorithm. Let $R = r_1 r_2 \ldots r_t \ldots r_\ell$ be any program reference string and let T be any constant such that $1 \leq T \leq \ell$. Then when DX is applied to manage a memory span of capacity T, at every t, $1 \leq t \leq \ell$, $M(t,T) \subseteq M(t,T+1)$ and $W(t,T) \subseteq W(t,T+1)$.

It should be noted the Proposition 2 holds for dual retention algorithms of the class of stack algorithms because of the availibility of a priority list that orders all previously referenced pages. Furthermore, since $W(t,T) \subseteq W(t,T+1)$ implies that $\omega(t,T) \leq \omega(t,T+1)$, we conclude that if all conditions in Proposition 2 are satisfied, then at every point in time, the working set size is a non-decreasing function of the memory span capacity T. It should also be noted that, unlike stack algorithms, maintaining a priority list of all previously referenced pages is only a sufficient condition for the conclusions of Proposition 2. A case in point is the DWS algorithm, which is the dual retention algorithms of a non-stack algorithm, FIFO, and yet possess the inclusion properties of memory span and working set described in the statement of Proposition 2.

In a stack algorithm, all previously referenced pages are arranged in a stack, which is updated at every reference according to the priority list induced by the stack algorithm. The pages occupying the first k stack positions will be kept in a memory space of capacity k managed by the stack algorithm in question. In the case of LRU, the stack and the priority list are identical.

Because the LRU priority ordering of pages is based on the recency of reference, it follows that a memory space of capacity k managed by LRU always contains the k highest priority pages, namely, the k most recently referenced pages. This remark is the basis for proving the following lemma.

Lemma 1: Let LRUT be the dual retention algorithm of the LRU replacement algorithm. Let $R = r_1 r_2 \ldots r_t \ldots r_\ell$ be any program reference string and let T be any constant such that $1 \leq T \leq \ell$. Also, let t be any time during the program's execution, i.e., $1 \leq t \leq \ell$. Suppose that LRUT is applied to manage a memory span of capacity T and that the resulting working set at time t, W(t,T), has a size m, i.e., $\omega(t,T) = m$. Then, if LRU is applied to manage a memory space of capacity m and if the set of pages kept in this memory space at time t is denoted by B(t,m), it is $W(t,T) = B(t,m)$.

Lemma 1 states that, under LRUT and at any time t, the working set of size m consists of exactly the same pages that will be contained at the same time t in a memory space of capacity m managed by LRU. The proof is based on the fact that with LRU the stack and the priority list are identical. This lemma can be used to prove the following proposition.

Proposition 3: Let $R = r_1 r_2 \ldots r_t \ldots r_\ell$ be any program reference string over an alphabet of cardinality n. At any time t, $1 \leq t \leq \ell$,

(i) if T is any constant such that $1 \leq T \leq \ell$, then there exists a constant $k_1 \leq T$ such that the pages kept in a memory space of size k_1 under LRU are identical to the working set W(t,T) resulting from using LRUT; and

(ii) if m is any constant such that $1 \leq m \leq n$, then there exists a constant $k_2 \geq 1$ such that the working set $W(t,k_2)$ resulting from using LRUT is identical to the pages kept in a memory space of size m under LRU.

It should be noted that, in the above proposition, the constants k_1 and k_2 vary not only from program to program, but also from time to time during the execution of any program.

V. PERFORMANCE OF RETENTION ALGORITHMS

Retention algorithms can estimate the memory demand of a program; this capability provides valuable information to the scheduler and allows it to avoid thrashing. From the viewpoint of system operation, this capability possessed by retention algorithms has a significant impact because thrashing is a state in which the computer system practically ceases to do any more useful work. However, it is not at all clear whether or not this advantage is accompanied by an improvement of some relevant system performance measures such as system throughput rate or response time. To gain some insight into the performance of some replacement algorithms and their dual retention algorithms, an experimental comparison between them has been performed.

As with different replacement algorithms, the retention algorithms we designed in the previous sections through the space-time duality transformation are different from DWS in that they monitor a program's working information according to different retention rules. Therefore, another matter of interest in testing out these newly designed algorithms is their relative performance with respect to the DWS algorithm.

These comparisons would be facilitated if there existed a typical program to serve as a yardstick. Without such a typical program, extensive experiments, performed on real or simulated systems, must be carried out before any firm conclusions can be drawn.

Only a limited experimental study was carried out in our investigation. In this study, we employed simulated page traces generated by a program behavior model known as the simple LRU stack model [11]. Experimenting with real page traces has the advantage that results are more realistic. But local variations in program referencing patterns in a page trace and the very nature of the program sometimes not only make selection of a trace or portions of it quite difficult, but also subject the results to skepticism. This difficulty can be resolved if a fairly wide spectrum of representative program traces are used and enough experimental results collected. However, this solution is definitely an expensive one. On the other hand, if only some average statistics for programs managed by different algorithms are of interest, simulated program traces may adequately serve the purpose, as long as the program model for their generation reflects the essential characteristics of the dynamic behaviors in question. We feel that this is the case in our study and therefore have taken the latter, less expensive approach.

Two page traces were generated for our study. Since it was the page referencing pattern, not the page numbers, that was of interest, pages were arranged in ascending numerical order in the initial LRU stack. Stack distance distributions were obtained from those of real-life programs reported in the literature. The first reference string, which we call trace 1, had an assumed stack distance distribution identical to that of reference string #6 in [11] with 20 pages. Trace 2 was generated from the stack distance distribution of the DCDL (Digital Control and Design Language) compiler reported in [12] with 43 pages. In the second case, however, the probabilities of referencing the first six stack positions, which accounted for 98.34% of all probabilities were not shown in the curve. These probabilities were found from a rather straightforward, if arbitrary, extrapolation of the reported curve with the constraint that all probabilities should add up to 1.0.

Both traces had one million references, but it was soon discovered that only a portion of each should be used in our experiments for cost considerations. To select a portion of a trace to use, two sets of tests on trace 1 with the DWS algorithm were performed. In both sets of tests, the size of the memory span (or the window size in the case of DWS) ranged from 100 to 1000 in steps of 100, and values for the average working set size and the average inter-page-fault time were measured in each case.

In the first set of tests, three non-overlapping sections of the same length (100,000 references) were compared. The results were seen to be close enough for our purpose and the section which appeared to have characteristics between those of the other two was chosen to be the section to use in our experiments. In no case was the difference between performance measures over 6.5% and the majority of these differences were well below 2%. In the second set of tests, the effects of the section length were examined and 100,000 references was found to be a sufficient length for the selected section of the trace.

Thus, the portion of trace 1 starting at the 500001st reference with a length of 100,000 references was selected for our experiments. An identical choice was made for trace 2 on the grounds that both traces were generated with the same program model and hence could be expected to display similar dynamic behaviors. In the sequel, we shall refer to these portions of traces 1 and 2 as Trace I and Trace II respectively.

Finally, the page trace generation process was checked by processing Traces I and II with the LRU replacement algorithm and determining the relative frequencies of reference of different LRU stack positions. Since the page trace generation process is equivalent to a sequence of independent and identically distributed random variables which represent the LRU stack distances, the strong law of large numbers predicts that the relative frequency of referencing stack distance k should approach, for a sufficiently long trace, the fixed probability of referencing this stack distance which was assumed when the trace was generated. Also, the results of this last set of tests were completely satisfactory (details of all tests are reported in [8]).

Having selected Traces I and II, four dual algorithm pairs, i.e., eight algorithms, were simulated. These were the FIFO, LRU, LFU and MFU replacement algorithms and their respective dual retention algorithms, DWS, LRUT, LFUT and MFUT. These algorithms were chosen basically for their relative popularity. In comparing them, three performance measures were of interest: average inter-page-fault time, efficiency and space-time product. In general, given a page trace and an algorithm for memory management (either a replacement algorithm or a retention algorithm), all these performance measures are closely related to the number of page faults induced by the algorithm in processing the page trace. Of course, the number of page faults is in turn a function of the size of the memory space in the case of a replacement algorithm and of the size of the memory span in the case of a retention algorithm.

The performance measures selected will now be defined. Let $r_1 r_2 \ldots r_\ell$ be a page trace having a length ℓ and using n pages. Let X be a memory management algorithm; if X is a replacement algorithm, let the memory space capacity be m page frames with $1 \le m \le n$, and if X is a retention algorithm, let the memory span capacity be T time frames with $1 \le T \le \ell$. Let f be the number of page faults resulting from processing the page trace using algorithm X. In the case where X is a retention algorithm, let $\omega(t,T)$ be the working set size at any time t, $1 \le t \le \ell$. Finally, let T_m and T_s be the access times to main and auxiliary memories respectively, and R be the ratio T_s/T_m.

(a) The average inter-page-fault time, a, is the average number of references between the occurence of two consecutive page faults. Thus

$$a = \frac{\ell}{f} \quad \text{references.}$$

Since a well-designed memory management algorithm is expected to produce few page faults, the average inter-page-fault time is a good indicator of this aspect of memory management as the length ℓ of the reference string is fixed. Note that the reciprocal of a is sometimes referred to as the average paging rate, or the missing page probability.

(b) The <u>efficiency</u>, e, for a program's execution is defined as the fraction of real processing time spent in executing the program. Of course, the rest of the real processing time is assumed to be spent in page waits (here, as is usual in this kind of study, user-initiated I/O activities are neglected).

$$e = \frac{\ell \cdot T_m}{\ell \cdot T_m + f \cdot T_s} = \frac{1}{1 + \frac{f}{\ell} \cdot R} \quad .$$

Note that in the above expression, only main memory access time, not instruction execution time, contributes to the program execution time. This is justified because instruction execution time is generally negligible compared to memory access time in most modern computers. Note also that if

$$\frac{f}{\ell} \cdot R \gg 1, \quad \text{then } e \simeq \frac{\ell}{f} \cdot \frac{1}{R} = \frac{a}{R} \quad .$$

Since R is a constant for an experiment, the efficiency will be a scaled average inter-page-fault time in these cases. In fact, this turned out to be the case in most of our experiments and consequently we only plotted efficiency in reporting our results. In general, since the speed ratio R of the memory hierarchy is in the order of 10^4 to 10^5 or higher, efficiency is very low unless f is small compared to ℓ. An obvious way to achieve high efficiency is to allocate sufficient memory space to the program. But this solution is generally neither an economical not an optimal way to utilize main memory in a multiprogramming environment, as a constantly large memory demand by one program tends to interfere with the execution of the concurrent programs and hence may downgrade the overall system performance.

(c) The <u>space-time product</u> is considered to be a measure proportional to the cost of storage. Belady and Kuehner [13] define the space-time product during the real time interval (t_o,t_1) as

$$C = \int_{t_o}^{t_1} S(t)dt$$

where S(t) is the amount of storage occupied by the program at any time t in the interval (t_o,t_1). If the execution of a program is considered a discrete process, we can rewrite the above integral, as Chu and Opderbeck [12] do, in the following way:

$$C = \sum_{i=1}^{\ell} S_i T_m + \sum_{i=1}^{f} S_{t_i+1} \cdot T_s$$

where S_i is the number of allocated page frames prior to the ith reference and t_i is the time when the ith page fault occurs. For replacement algorithms, since the memory allocation to a program is a fixed number m, we have

$$C = m \cdot \ell \cdot T_m + f \cdot m \cdot T_s$$

Dividing both sides by $\ell|T_m|$ gives

$$C_s = \frac{C}{\ell|T_m|} = \frac{T_m}{|T_m|} \cdot m \cdot (1 + \frac{f}{\ell} \cdot R) \text{ page·seconds}$$

As the value of ℓ and T_m are fixed for an experiment, C_s is actually a scaled space-time product. The reason we divide C by the absolute value of T_m rather than by T_m is just to preserve the unit for the scaled space-time product. For retention algorithms, memory demand may vary with time as represented by the working set size $\omega(t,T)$; thus the space-time product becomes

$$C = \sum_{t=1}^{\ell} \omega(t,T) \cdot T_m + \sum_{i=1}^{f} \omega(t_i+1,T) \cdot T_s,$$

and a similar scaling gives

$$C_s = \frac{C}{\ell |T_m|} = \frac{T_m}{|T_m|} \cdot \left[\bar{\omega}(T) + \frac{R}{\ell} \sum_{i=1}^{f} \omega(t_i+1,T) \right]$$

where $\bar{\omega}(T) = \frac{1}{\ell} \sum_{t=1}^{\ell} \omega(t,T)$ is the average working set size.

In designing our experiments, it was discovered that counting the number of page faults, f, was not as straightforward for retention algorithms as for replacement algorithms operating in a fixed-space environment. For a fixed-space environment, if the allocated space is filled and a page fault occurs, the page chosen for replacement is removed and will no longer exist in main memory. Thus, a later reference to this removed page requires moving it back into main memory and hence always generates a page fault. On the other hand, retention algorithms monitor a program's working set whose size varies dynamically. When a page drops out of the working set, that is, no more time indices corresponding to this page are contained in the constant-size memory span, this page is no longer considered by the retention algorithm to be part of the program's working information, and hence the page frame it occupies is available for use by the same or other, concurrently executing programs. However, there is no need to remove this page from main memory unless the page frame it occupies is indeed needed for placing another page. Therefore, when this page is referenced again at a later time, there is a chance that it is still in main memory and so this reference will not be a page fault. In this case, we say that the page is reclaimed.

The probability that a page which has dropped out of the working set is reclaimed is a function of the system workload, more precisely, of the instantaneous memory demands of all the concurrently executing programs. In our experiments, we consider a fixed probability p to reclaim a page. Thus, when a page dropped out of the working set, a random number (uniformly distributed in [0,1]) was generated for it, which would be compared with the fixed probability p to determine whether this page could be reclaimed if it would be referenced at a later time. It is clear that for a series of experiments with p as a parameter the number of page faults corresponding to the case p=0 is an upper bound. Similarly, this case gives an upper bound for space-time product and a lower bound for efficiency.

In experiments with replacement algorithms, for each page trace and for each value of memory space capacity, values for the average inter-page-fault time, efficiency, and scaled space-time product were computed using the expressions presented above and a speed ratio R = 10,000. Likewise, the same three performance measures were computed in the cases with retention algorithms for each page trace and for each value of memory span capacity. With retention algorithms, the performance measures were also functions of the page reclamation probability p. In addition, a value for average working set size was also collected in each experiment with retention algorithms. With these data, a plot of a performance measure versus the memory space capacity could be obtained for each page trace and for each replacement algorithm tested, and a plot of a performance measure versus the memory span capacity for each page trace and for each retention algorithm tested.

To compare the performances of different algorithms, in particular, the performances of a replacement-retention dual algorithm pair, the problem arises in comparing plots of the performance measures as functions of different independent variables. To get a meaningful comparison, we decided to plot performance measures on the basis of average memory demand. In the case of replacement algorithms, we regard the (fixed) memory space capacity to be the average memory demand of a program as estimated by the replacement algorithms. Retention algorithms, on the other hand, do have the ability to dynamically estimate a program's memory demand and in fact, a program's average memory demand as estimated by a retention algorithm is just the average working set size. Therefore, the average working set size curve can be used to obtain plots of performance measures versus average memory demand. Let PM denote a performance measure, let T denote the memory span capacity and let $\bar{\omega}$ denote the average working set size. Now for each retention algorithm tested and for each page trace, two plots, PM vs. T and $\bar{\omega}$ vs. T, can be obtained. Suppose that k is an integer falling in the range of values of $\bar{\omega}$, then we can obtain the corresponding value T_k from the average working set curve, that is, the $\bar{\omega}$ vs T plot. Then, from the PM vs T plot, a value PM_k corresponding to T_k can also be obtained. Finally. a plot of PM vs k is available which consists of the points (k, PM_k). This final plot then gives the performance of the retention algorithm on the basis of average memory demand. These procedures are schematically illustrated in Figure 3.

This transformation requires justification. The question is, whether the average working set curve is a one-to-one function. The answer to this question is negative as a simple example will show: for the reference string ABCABC, the average working set size using DWS is $\bar{\omega}(3) = \bar{\omega}(4) = 2.5$. However, since LRUT, LFUT and MFUT are dual retention algorithms of LRU, LFU and MFU which are stack algorithms, Proposition 2 states that their resulting working set size at any time is a non-decreasing function of the memory span capacity, that is, $\bar{\omega}(t,T) \leq \bar{\omega}(t,T+1)$. This in turn implies that $\bar{\omega}(T) \leq \bar{\omega}(T+1)$ as $\bar{\omega}(T) = \frac{1}{\ell} \sum_{t=1}^{\ell} \omega(t,T)$. The same is also true for the DWS algorithm. Therefore, for all four retention algorithms we tested, the average working set size is a non-decreasing function

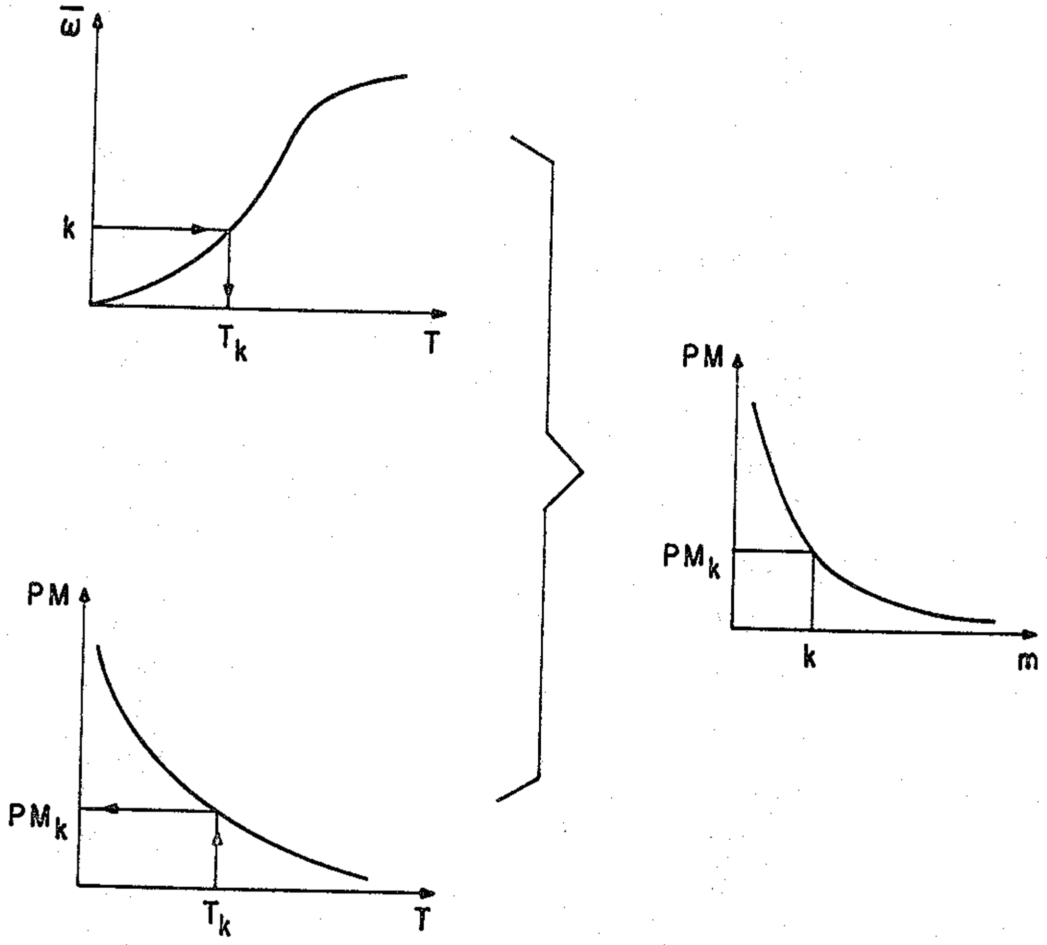

Figure 3. Produres in Obtaining Performance Curve Through Space-Time Capacity Transformation

of T. Hence, if for an integral value k, there correspond a group of values T_k, these values are necessarily consecutive. In these cases, we choose the smallest such T_k to break the tie. However, in our experiments, no such cases occurred. This is also easy to understand, for if $T_1 \neq T_2$, then $\bar{\omega}(T_1) = \bar{\omega}(T_2)$ requires $\bar{\omega}(t,T_1) = \bar{\omega}(t,T_2)$ for all t. Thus, with t varying between 1 and 10^5 and with T_1 and T_2 differing for much more than 1, it is highly unlikely that $\bar{\omega}(T_1)$ is equal to $\bar{\omega}(T_2)$, as was evidenced from the results of our experiments.

Figures 4 through 7 summarize some of the results for Trace I, and Figures 8 through 11 for Trace II. For each pair of algorithms, there are two figures, one for efficiency and one for scaled space-time product so that the relative performances of these dual algorithms can be readily compared. Note also that the scaled space-time product so that the relative performances of these dual algorithms can be readily compared. Note also that the scaled space-time product is computed by $C_s = \frac{1}{\ell|T_m|} C$, where ℓ is the page trace length and T_m is main memory access time. For our experiments, $\ell = 10^5$ and a typical value for T_m will be 10^{-6} sec., thus C_s is roughly 10 times bigger than C. Similar curves were obtained for the LFU-LFUT and MFU-MFUT pairs and are reported in [8].

For the retention algorithms DWS and LRUT, quite different values of average working set size, efficiency and space-time product were obtained for the same memory span capacity. However, if we compare their performances on the basis of equal average memory demand, they seem almost indistinguishable from each other (see Figures 4 - 7 and 8 - 11). Thus, with respect to the traces we tested, these two retention algorithms are almost equivalent in performance. But for the same memory span capacity, DWS seems to always make a higher estimate of memory demand than LRUT. This characteristic of DWS may become disadvantageous in a heavily loaded system in which memory is always in short supply.

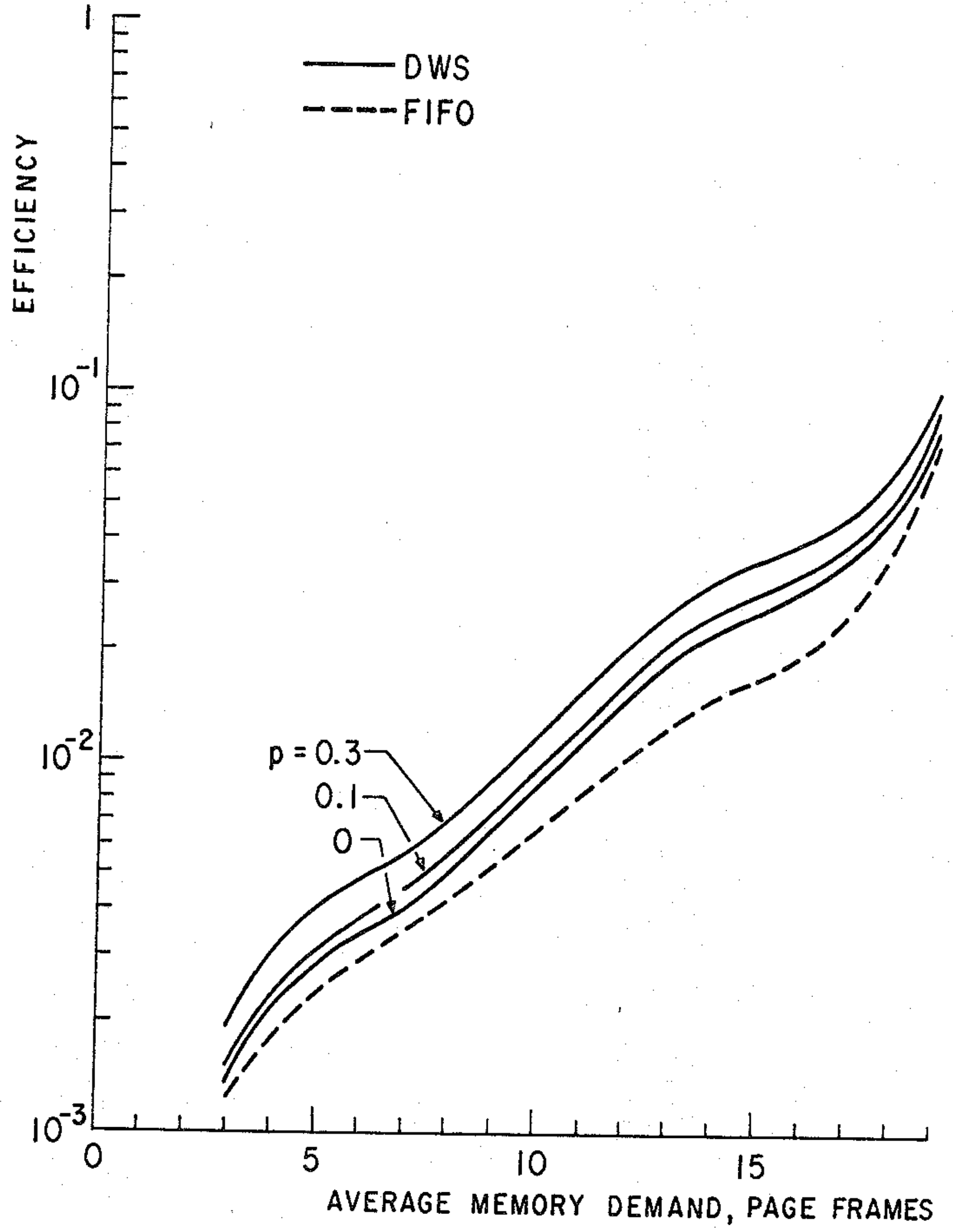

Figure 4. Efficiency vs. Average Memory Demand for Trace I With FIFO and DWS

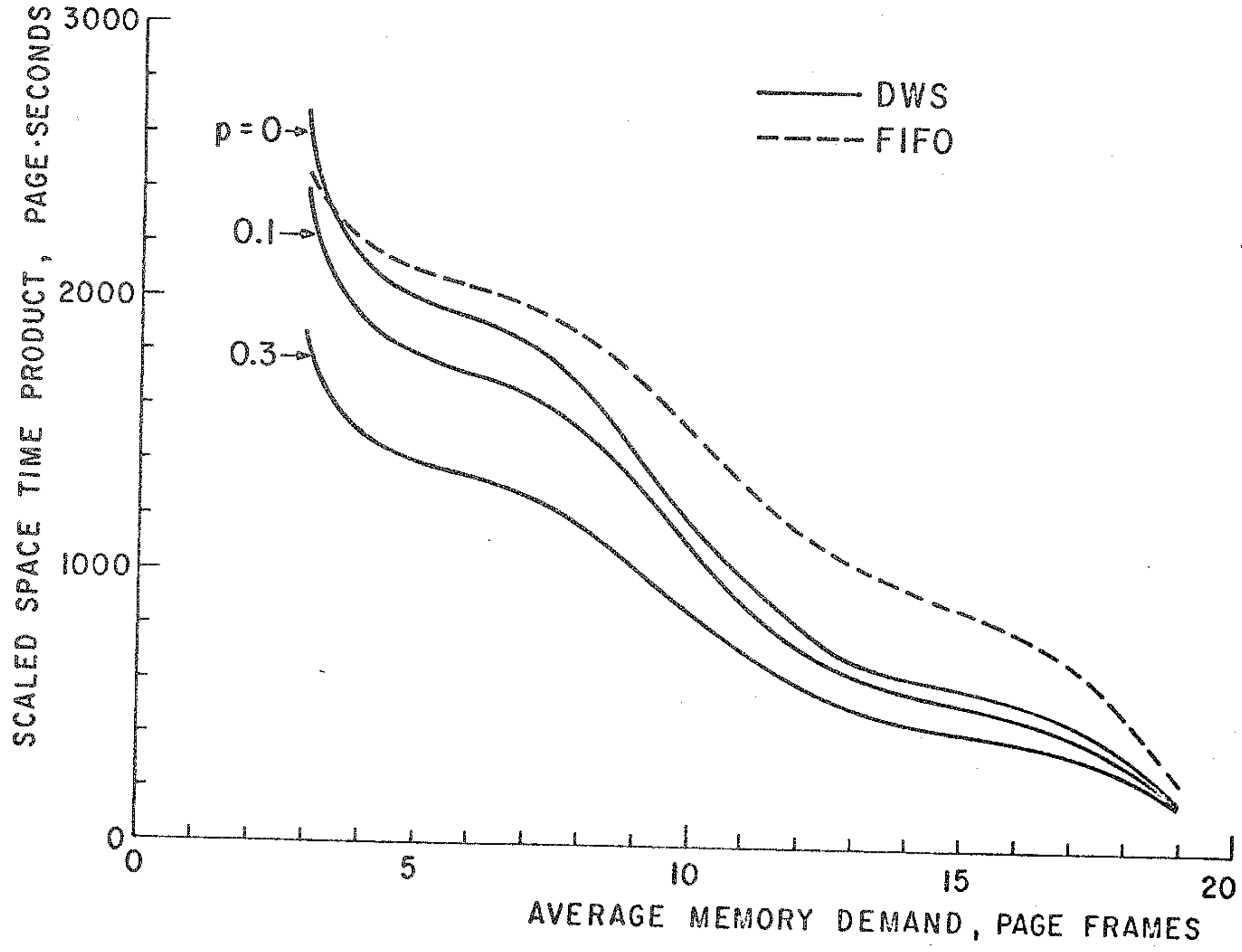

Figure 5. Scaled Space Time Product vs. Average Memory Demand for Trace I With FIFO and DWS

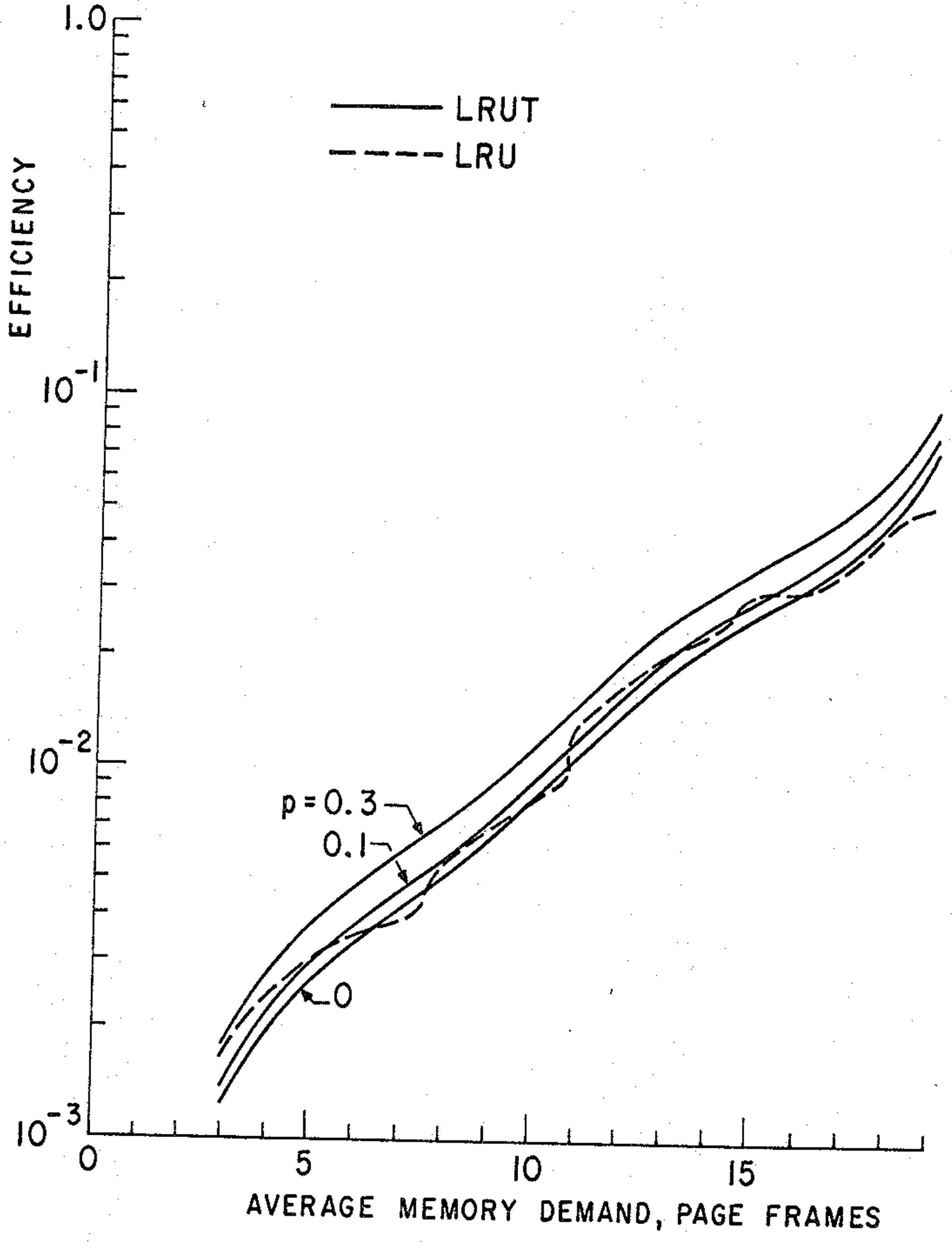

Figure 6. Efficiency vs. Average Memory Demand for Trace I With LRU and LRUT

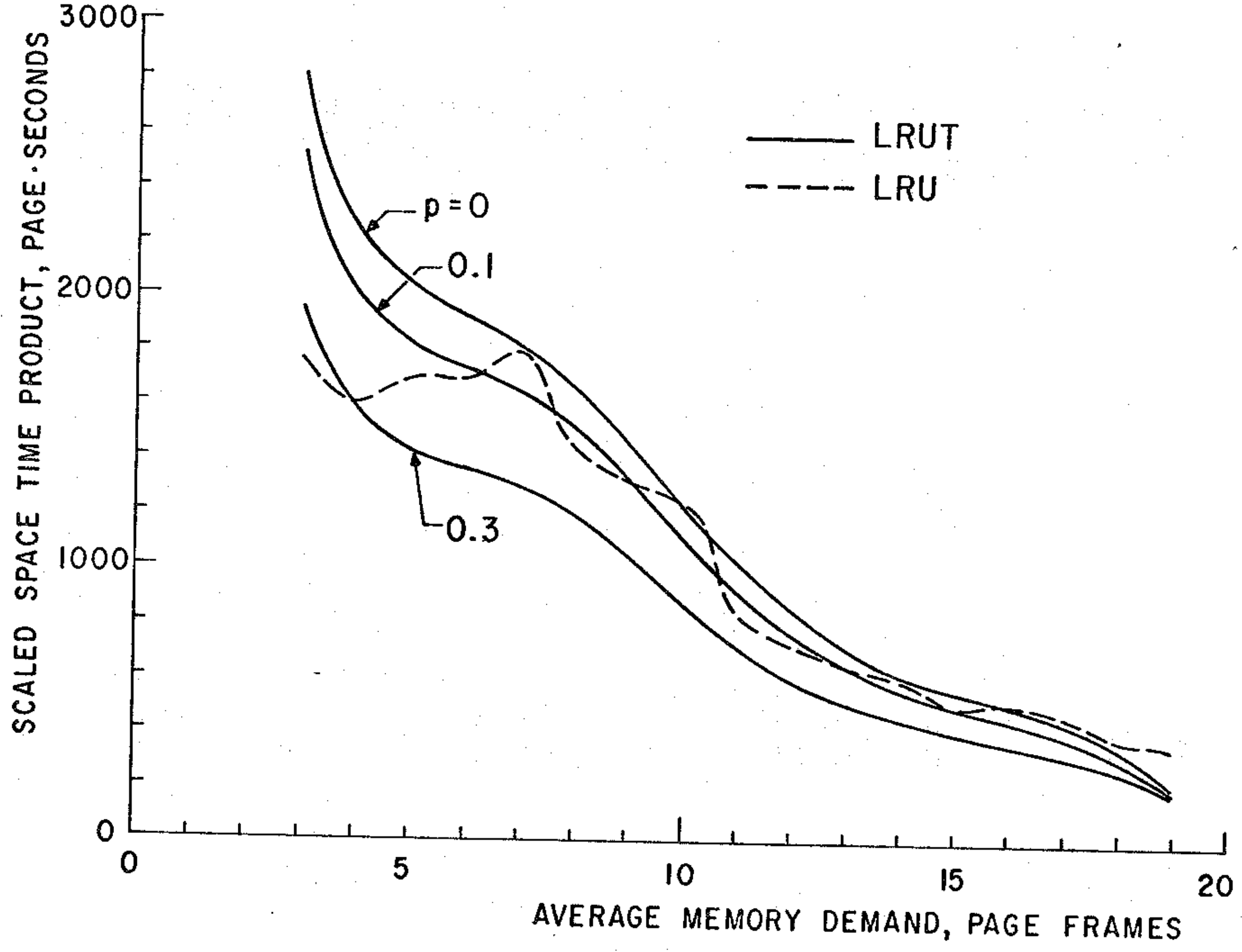

Figure 7. Scaled Space Time Product vs. Average Memory Demand for Trace I With LRU and LRUT

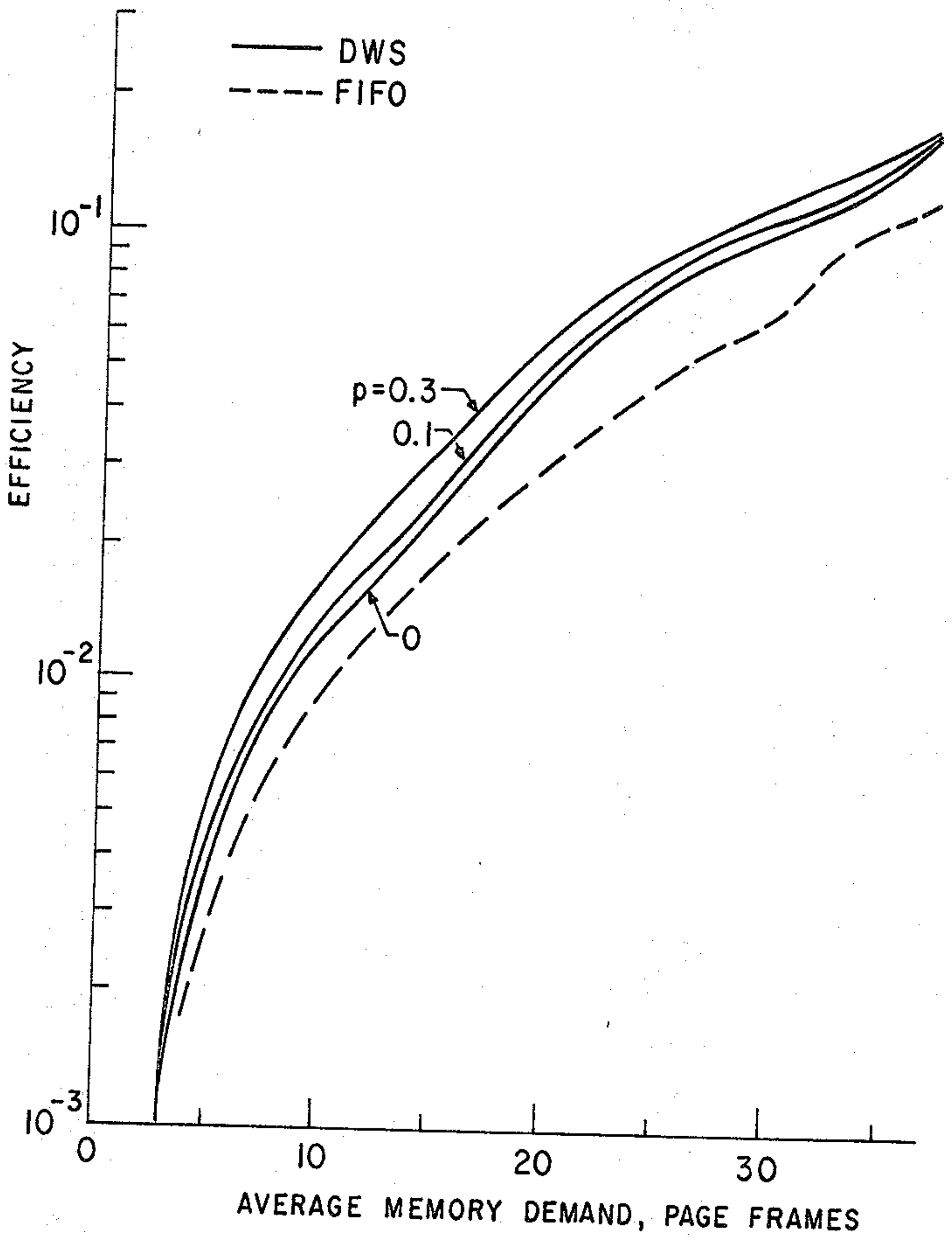

Figure 8. Efficiency vs. Average Memory Demand for Trace II With FIFO and DWS

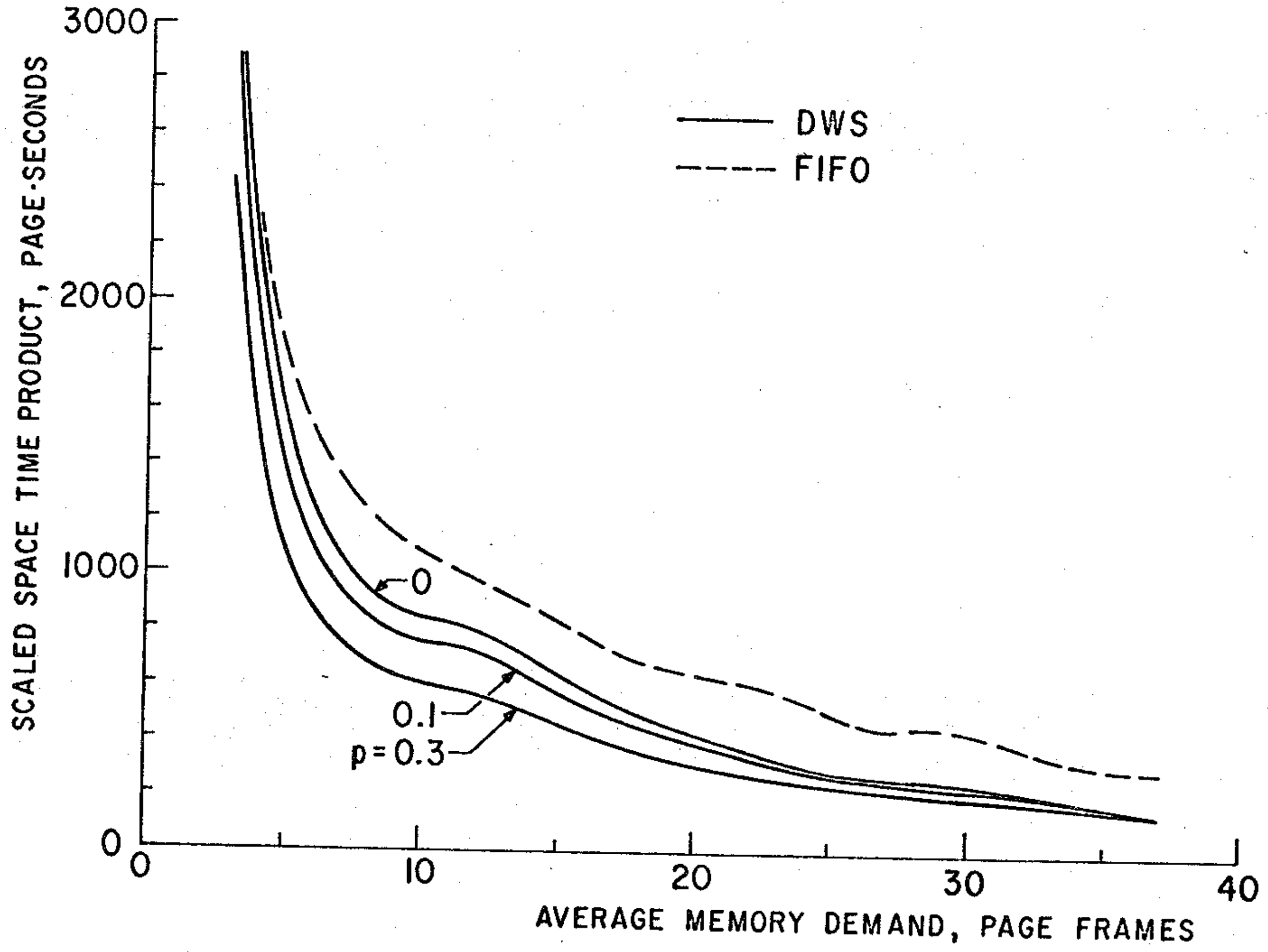

Figure 9. Scaled Space Time Product vs. Average Memory Demand for Trace II With FIFO and DWS

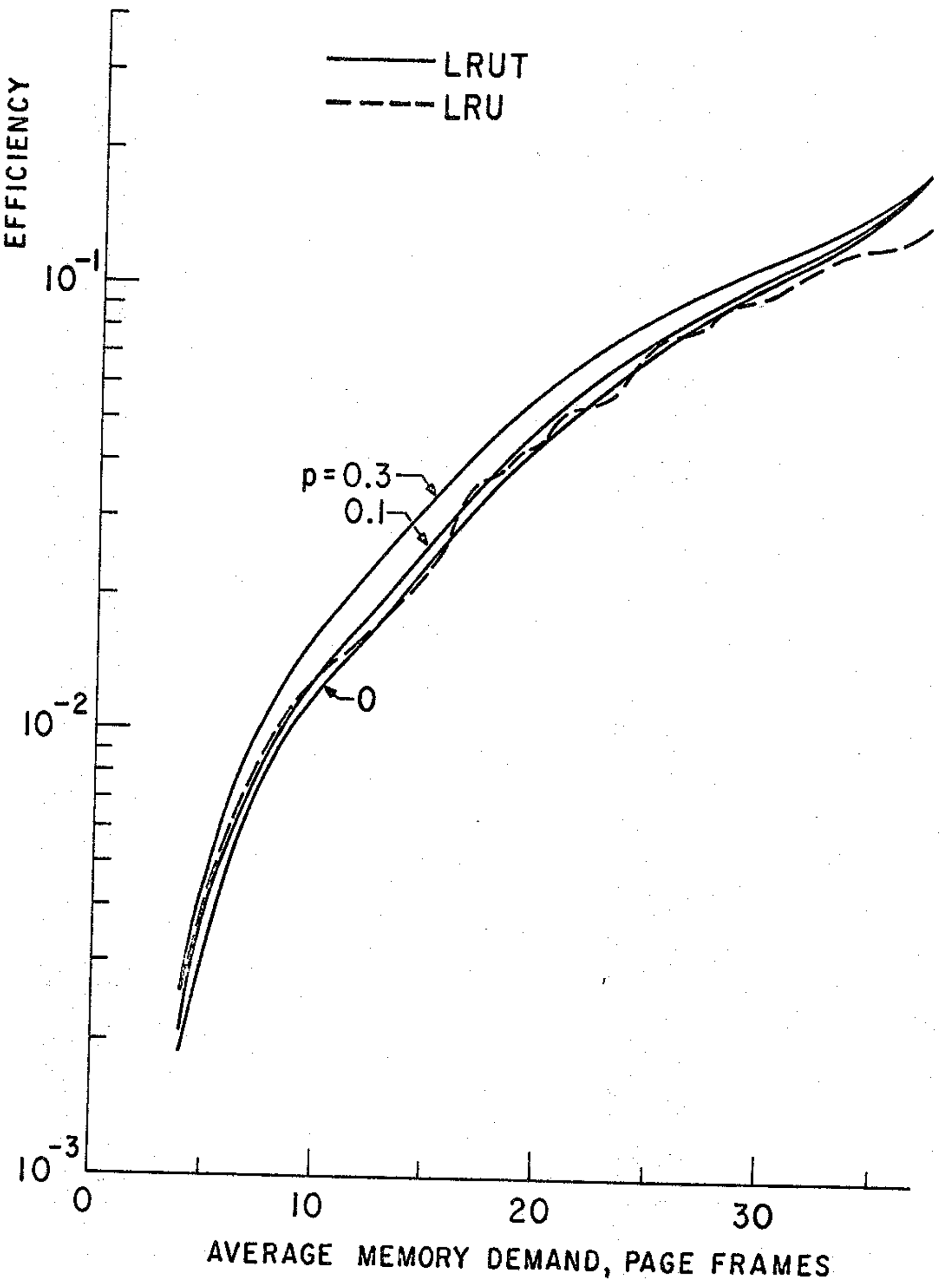

Figure **10.** Efficiency vs. Average Memory Demand for Trace II With LRU and LRUT

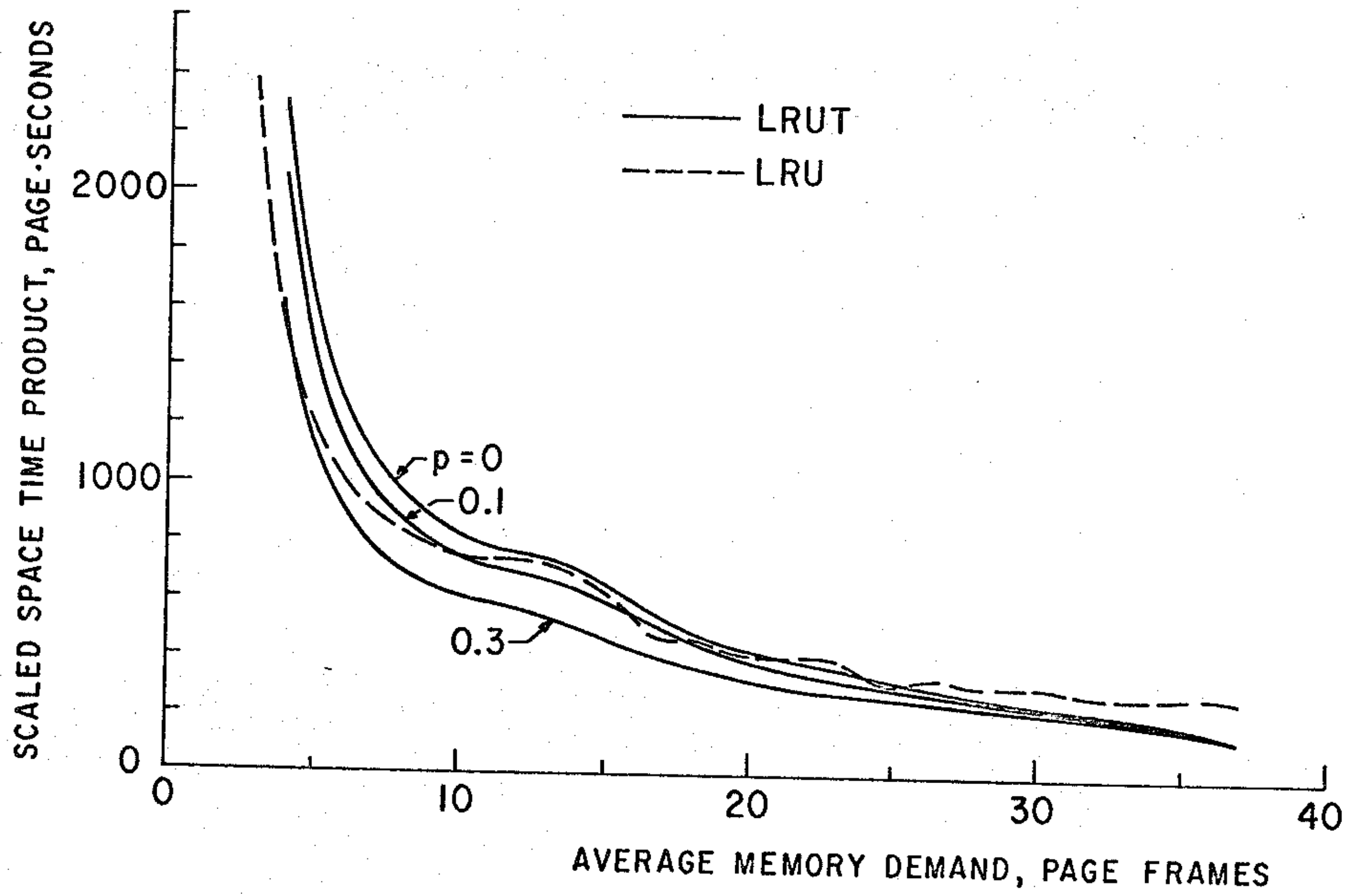

Figure 11. Scaled Space Time Product vs. Average Memory Demand for Trace II With LRU and LRUT

As expected, due to the nature of the page traces used, LRU has the best performances among all four replacement algorithms. In addition, LRU performs better than both DWS and LRUT when the page reclamation probability is close to zero, i.e., for p=0 and p=0.1. When there is a higher probability to reclaim a page (p=0.3), both DWS and LRUT outperform LRU. In contrast, FIFO is seen to be outperformed by DWS and LRUT for all values of p. It should be stressed that, even though a program trace generated according to the simple LRU stack model is inevitably biased towards LRU, these results seem to indicate that for a reasonable probability to reclaim a page both DWS and LRUT have better performances than LRU. Since both DWS and LRUT are retention algorithms, we may conclude that, just from considerations of the performance measures we have chosen the idea that a memory management algorithm should have the capability to dynamically estimate a program's memory demand is a valid one. Finally, it must be emphasized that the performance data presented here represent the performances of memory management algorithms with respect to a single program running in a multiprogramming environment. However, it is not straightforward to relate these performances to overall system performance such as system throughput rate.

VI. CONCLUDING REMARKS

A general method for the design of algorithms, which we call retention algorithms, for dynamic memory management has been developed. The retention algorithms we designed have different retention rules, and hence different underlying program behavior models, for computing which parts of a program constitute its working information at any time during its execution. In this regard, it is of interest to study the relative performances of these retention algorithms with respect to each other and, in particular, with respect to the DWS algorithm.

In our experiments, it was found that, for the page traces tested, DWS and LRUT were almost indistinguishable in performance and that they outperformed LFUT and MFUT by almost an order of magnitude in the performance measures we chose. However, due to the page trace generation process we used and to the limited number of traces we tested, these findings are by no means conclusive. To establish the relative merits of these algorithms, more experiments must be performed so that at lest some statistical conclusions can be drawn. It also appears that real program traces are preferred for such purposes, as any program trace generation model inevitably biases the results in favor of some algorithms. These experiments can be performed on real systems or in simulated environments. A further refinement in the experiments seems to be in the area of page reclamation. More realistic and sophisticated schemes, such as one in which the probability to reclaim a page depends on the length of time since it left the working set, can be employed to reflect the real operation of a multiprogramming system. Moreover, it would be more satisfactory to obtain the probability distribution through measurements on real systems. Choices of other system performance measures that can relate more readily to overall system performance than those employed in our experiments seem desirable. On a higher level, experiments with these algorithms in a multiprogramming setting, rather than with individual programs, can provide valuable information in evaluating various memory management schemes.

From the implementation viewpoint, all the newly-designed retention algorithms appear to require, like DWS, large amounts of information, and do not seem to be efficient unless implemented in hardware. However, it might turn out that some variations of these algorithms, in which the rigid requirement that a replacement of time index from the memory span at every instant of time is relaxed, could be a lot easier to implement. Furthermore, the determination of the memory span capacity for any of these algorithms may be a fruitful research topic. If separate working sets are kept for procedures and for data, it may be interesting to investigate how these new algorithms can be applied.

In conclusion, we feel that further research work in this area may help establish the merit and applicability of retention algorithms as well as shed some more light on the behavior of programs.

REFERENCES

[1] P. J. Denning, "Virtual Memory", Computing Surveys, vol. 2, pp. 153-189, Sept. 1970.

[2] E. G. Coffman and T. A. Ryan, "A study of storage partitioning using a methematical model of locality", Comm. ACM, vol. 15, pp. 185-190, March 1972.

[3] T. Kilburn et al., "One-level storage systems", IRE Trans. Electron. Comput., vol. EC-11, pp. 223-235, Apr. 1962.

[4] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer", IBM Syst. J., vol. 5, pp. 78-101, 1966.

[5] R. L. Mattson et al., "Evaluation techniques for storage hierachies", IBM Syst. J., vol. 9, pp. 78-116, 1970.

[6] P. J. Denning, "Thrashing: Its causes and prevention", in 1968 Fall Joint Comput. Conf. Proc., AFIPS Conf. Proc., vol. 33. Washington, D. C.: Thompson, pp. 915-922, 1968.

[7] A. V. Aho, P. J. Denning and J. D. Ullman, "Principles of optimal page replacement", J. ACM, vol. 18, pp. 80-93, Jan. 1971.

[8] F. L. Lam, "Program working information estimation by a space-time duality approach", Ph.D. Thesis, University of California, Berkeley, Aug. 1974.

[9] J. M. Thorington and J. D. Irwin, "An adaptive replacement algorithm for paged-memory computer systems", IEEE Trans. Comput., vol. C-21, pp. 1053-1061, Oct. 1972.

[10] P. J. Denning, "The working set model for program behavior", Comm. ACM, vol. 11, pp. 323-333, May 1968.

[11] J. R. Spirn and P. J. Denning, "Experiments with program locality", in 1972 Fall Joint Comput. Conf. Proc., AFIPS Conf. Proc., vol. 41. Montvale, N. J.: AFIPS Press, pp. 611-621, 1972.

[12] W. W. Chu and H. Opderbeck, "The page fault frequency replacement algorithm", in 1972 Fall Joint Comput. Conf. Proc., AFIPS Conf. Proc., vol. 41. Montvale, N. J.: AFIPS Press, pp. 597-609, 1972.

[13] L. A. Belady and C. J. Kuehner, "Dynamic space-sharing in computer systems", Comm. ACM, vol. 12, pp. 282-288, May 1969.

[14] E. G. Coffman and P. J. Denning, Operating Systems Theory. Englewood Cliffs, N. J.: Prentice-Hall, p. 276, 1973.

[15] E. G. Coffman and N. D. Jones, "Priority paging algorithms and the extension problem", in Proc. 11th Switching and Automata Theory Symp., Oct. 1971.