Technical Report CS75003-R

STORAGE REDUCTION THROUGH MINIMAL SPANNING TREES

A. N. C. Kang
R. C. T. Lee
C. L. Chang
S. K. Chang

March 1975

A. N. C. Kang is with the Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061

R. C. T. Lee is with the Naval Research Lab, Washington, D.C. 20375

C. L. Chang is with IBM, San Jose, California

S. K. Chang is with IBM, Yorktown Heights, New York 10958

Abstract

In this paper, we shall show that a minimal spanning tree for a set of data can be used to reduce the amount of memory space required to store the data. Intuitively, the more points we have, the more likely our method will be better than the straightforward method where the data is stored in the form of a matrix. In Section 3, we shall show that once the number of samples exceeds a certain threshold, it is guaranteed that our method is better. Experiments were conducted on a set of randomly generated artificial data and a set of patient data. In the artificial data experiment, we saved 23% for the worst case and 45% for the best case. In the patient data experiment, we saved 73% of the memory space.

Section 1. Introduction

Recently, because of the progress made in computer technology, it is customary to store more data than to discard them. In many instances, the amount of data is so large that it is desirable to reduce the memory requirement of the records. In this paper, we shall introduce a method to reduce the memory space required.

Our method is based on our observation that similar records do not have to be stored in their entireties. Suppose we have to store two identical records A and B. We may simply store record A in its entirety and for record B, we merely provide a pointer to record A.

If two records are similar, but not identical, we may still reduce the necessary memory space by providing a pointer from one record to another and indicating precisely the differences between these records. For example, consider the following two records:

	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}	X_{11}	X_{12}
R_1	a	a	b	b	c	a	e	a	b	c	b	a
R_2	a	a	c	b	c	a	e	a	d	c	b	a

We note that these two records are different only at X_3 and X_9. The two records can therefore be stored as follows:

R_1	a	a	b	b	c	a	e	a	b	c	b	a
R_2	1	3	c	9	d							

For record 1, since it is used as a reference point, the entire record has to be stored. For R_2, the first location stores the pointer (pointing to record 1). After the pointer, the values of those variables which are different between R_1 and R_2 are stored. The meaning should be obvious.

In the case of R_2, the memory content tells us that the value of X_3 is c and the value of X_9 is d; all other variables have the same values as the variables in R_1. R_2 can therefore be reconstructed quite easily by searching back from R_2 to R_1.

It is possible that we have a third record R_3 which is different from R_2 only in X_7. Suppose in R_3, X_7=a. We may now provide a pointer pointing from R_3 to R_2 as follows:

R_3 2 7 a

Altogether, the three records will be stored in the memory as follows:

R_1 a a b b c a e a b c b a

R_2 1 3 c 9 d

R_3 2 7 a

Since the lengths of the records are different now, it is necessary for us to have another array to store the pointers pointing to these records. We need 3 locations for this array. Totally, we need 3+12+5+3=23 memory locations. Without this mechanism, we need 12x3=36 locations. We have saved (36-23)/36=36% of the memory space.

For a rather large set of data, we can expect the existence of many identical or similar records. It is therefore usually desirable to use our method to reduce the memory storage requirement. However, once we have a large set of records, it is no longer easy to know how to arrange the pointers. In [Lee and Chang 1973], it is pointed out that minimal spanning tree concept can be applied to arrange the pointers. We shall discuss their ideas in the next section.

Section 2. The Minimal Spanning Tree Concept and its Applications to Storage Reduction.

Let us first introduce briefly the concept of minimal spanning trees [Prim 1957].

Definition.

Given a set S of points, a spanning tree of S is a connected graph G of S that satisfies the following conditions:

(1) Every point of S is on G.

(2) G contains no loops.

Definition.

A minimal spanning tree of a set S of points is a spanning tree whose total length is a minimum among all possible spanning trees.

We shall assume that a record is in the form of an m-dimensional vector. For two records

$$R_i = (x_1^i, x_2^i, \ldots, x_m^i)$$

and $R_j = (x_1^j, x_2^j, \ldots, x_m^j)$,

we shall define a function $f(x_k^i, x_k^j)$ as follows:

$$f(x_k^i, x_k^j) = 0 \text{ if } x_k^i = x_k^j$$

$$= 1 \text{ if otherwise.}$$

We shall define the distance between R_i and R_j as

$$d_{ij} = \sum_{k=1}^{m} f(x_k^i, x_k^j).$$

The distance we have defined essentially counts the number of differences between two records. Two identical records will have the distance equal to 0 and two totally different records will have the distance equal to m (the number of variables).

Consider the set of records in Table 1. Based upon the above definition of distances among records, we can now construct a minimal spanning tree as shown in Fig. 1.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
R_1	a	b	a	b	a	a	a	c	a	a
R_2	a	b	a	b	a	a	a	b	a	a
R_3	b	c	a	b	c	a	a	c	a	b
R_4	a	a	a	b	a	a	a	b	a	c
R_5	a	b	a	b	c	a	a	a	a	b
R_6	a	b	a	b	a	a	a	b	a	a

Table 1

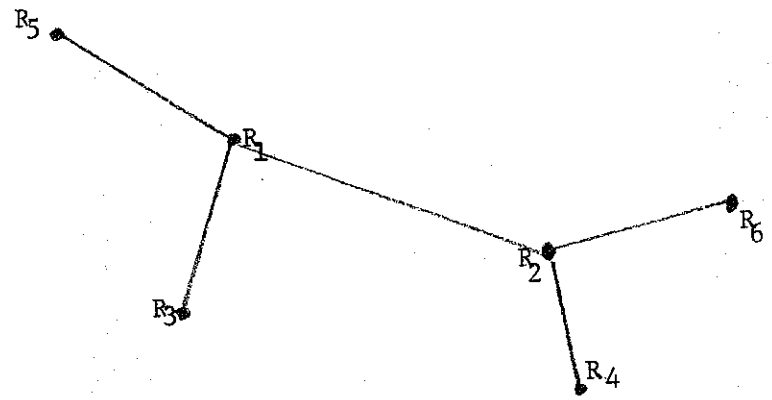

Fig. 1

We can now denote any point as the root of the tree and redraw the minimal spanning tree as a directed tree. Let us assume that we choose R_1 as the root. The tree in Fig. 1 is now redrawn in Fig. 2.

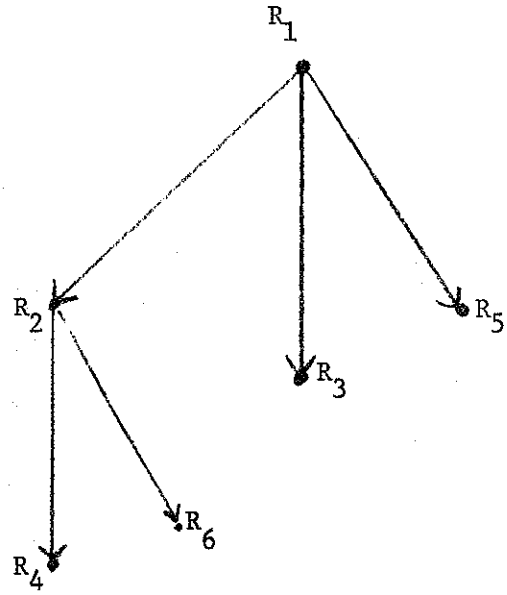

Fig. 2

Based upon the tree in Fig. 2, we can store the data according to the following rule:

(1) The root of the tree will be used as a reference point. The record corresponding to it will be stored in its entirety.

(2) For every node i, the following information is to be stored:

(a) Node j: the immediate predecessor of node i.

(b) If the value of x_k of record i is different from the value of x_k of record j, this fact should be stored.

For the set of data in Table 1, we can store the data as follows:

R_1	a	b	a	b	a	a	a	c	a	a
R_2	1	8	b							
R_3	1	1	b	2	c	10	b			
R_4	2	2	a	10	c					
R_5	1	8	a	10	b					
R_6	2									

Table 2

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 $1 \leq m \leq n$, and if X is a retention algorithm, let the memory span capacity be T time frames with $1 \leq T \leq \ell$. 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, $1 \leq t \leq \ell$. 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 <u>average inter-page-fault time</u>, 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 efficiency, 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 space-time product 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

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

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

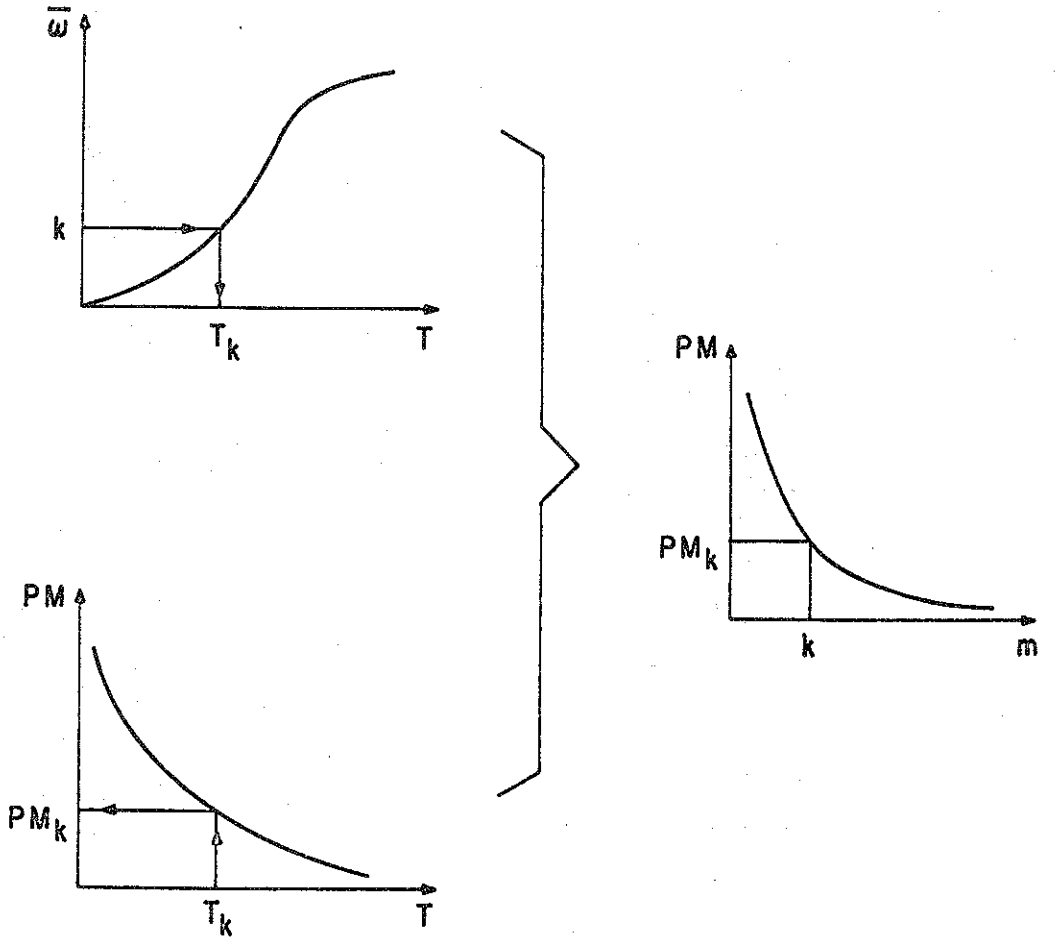

Figure 3. Produres in Obtaining Performance Curve Through Space-Time Capacity Transformation

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.

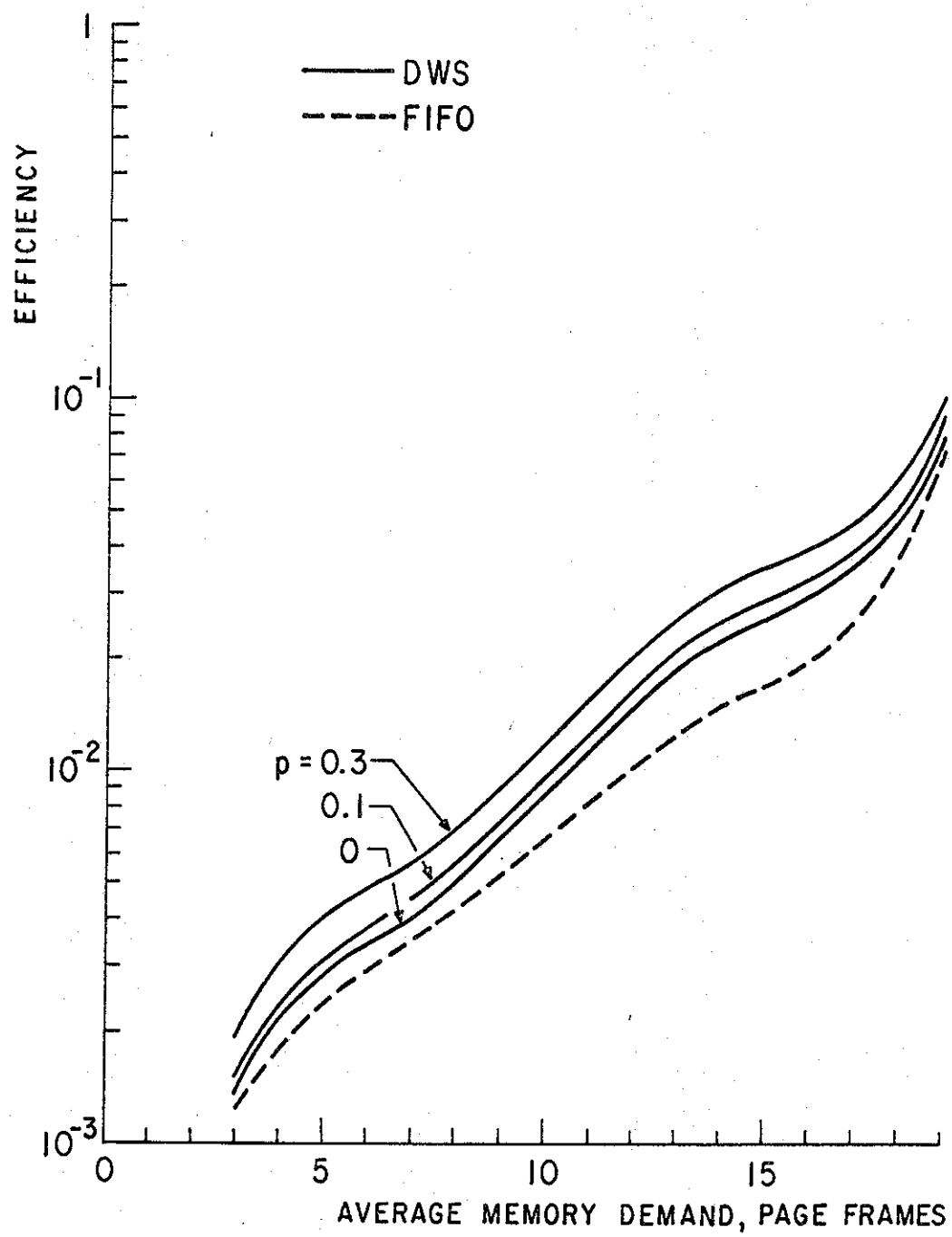

Figure 4. Efficiency vs. Average Memory Demand for Trace I With FIFO and DWS

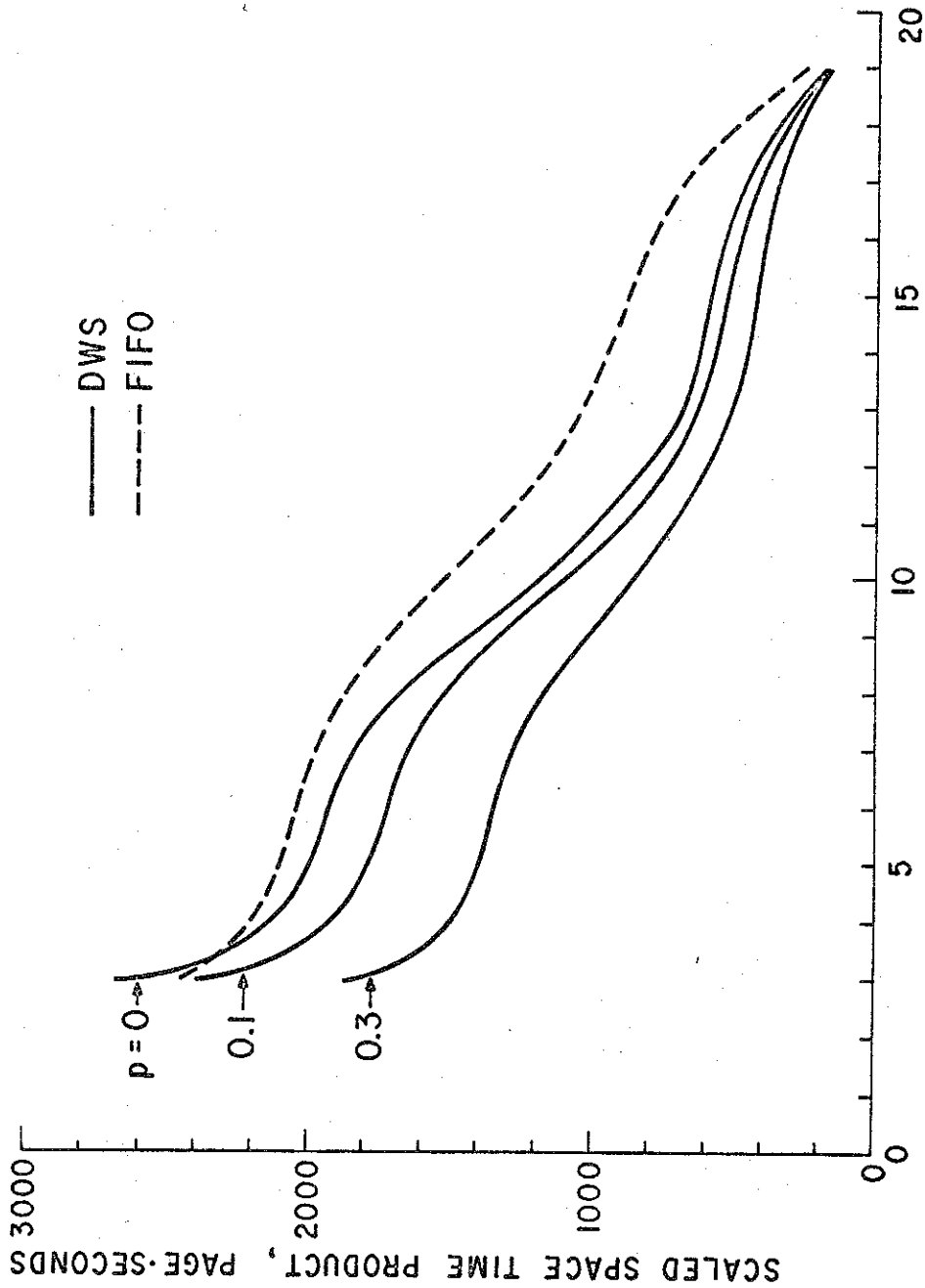

Figure 5. Scaled Space Time Product vs. Average Memory Demand for Trace I With FIFO and DWS

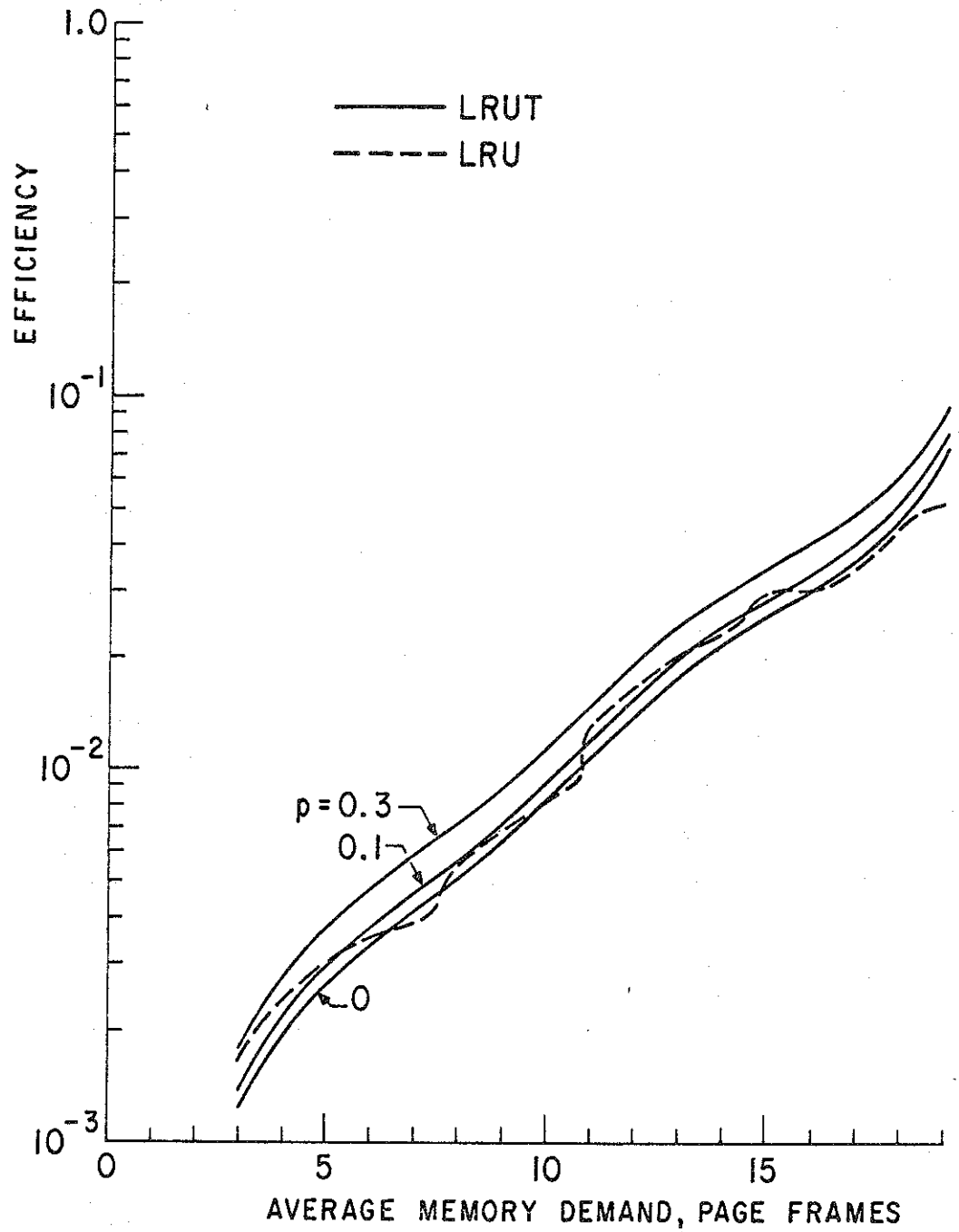

Figure 6. Efficiency vs. Average Memory Demand for Trace I With LRU and LRUT

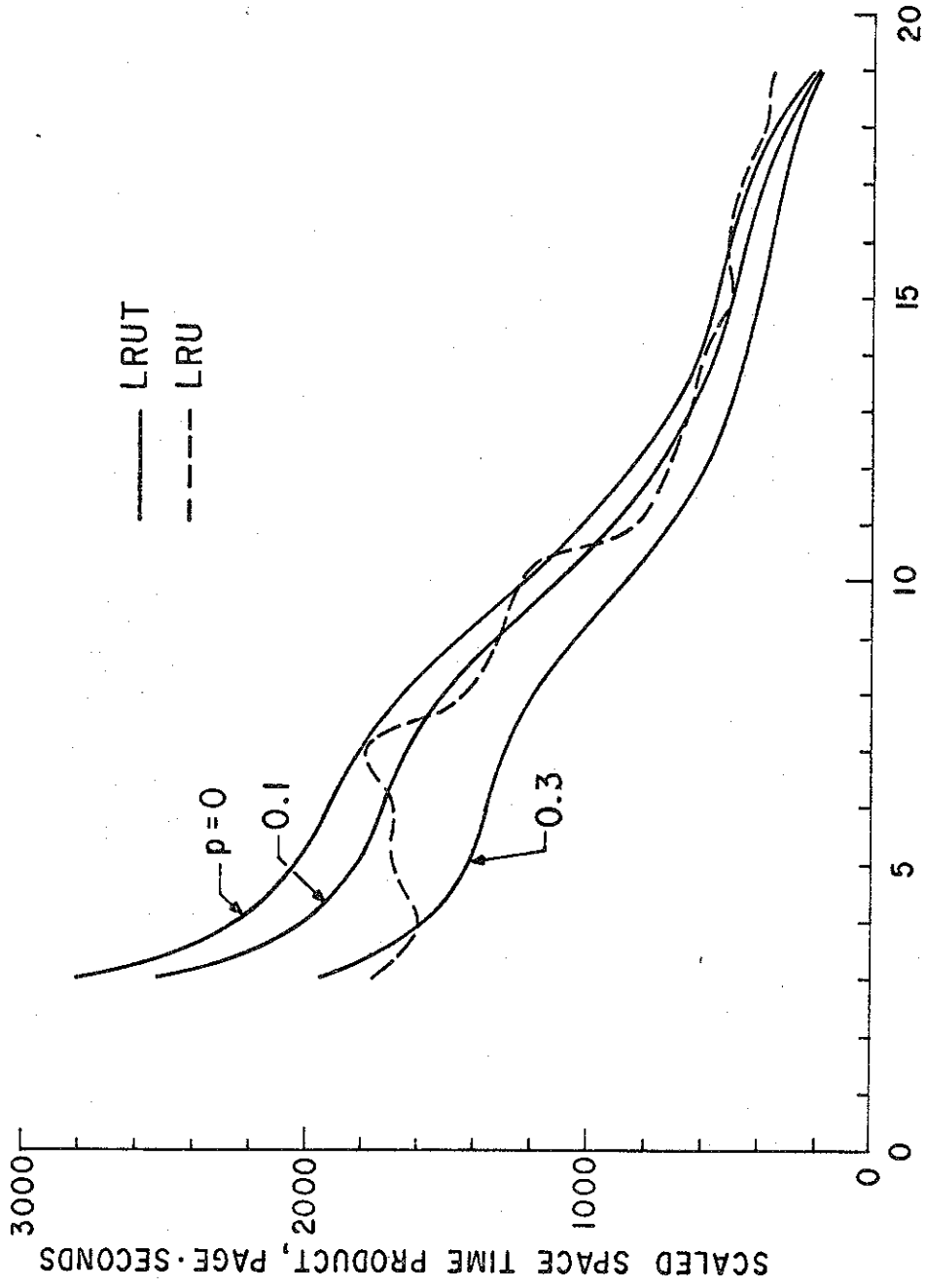

Figure 7. Scaled Space Time Product vs. Average Memory Demand for Trace I With LRU and LRUT

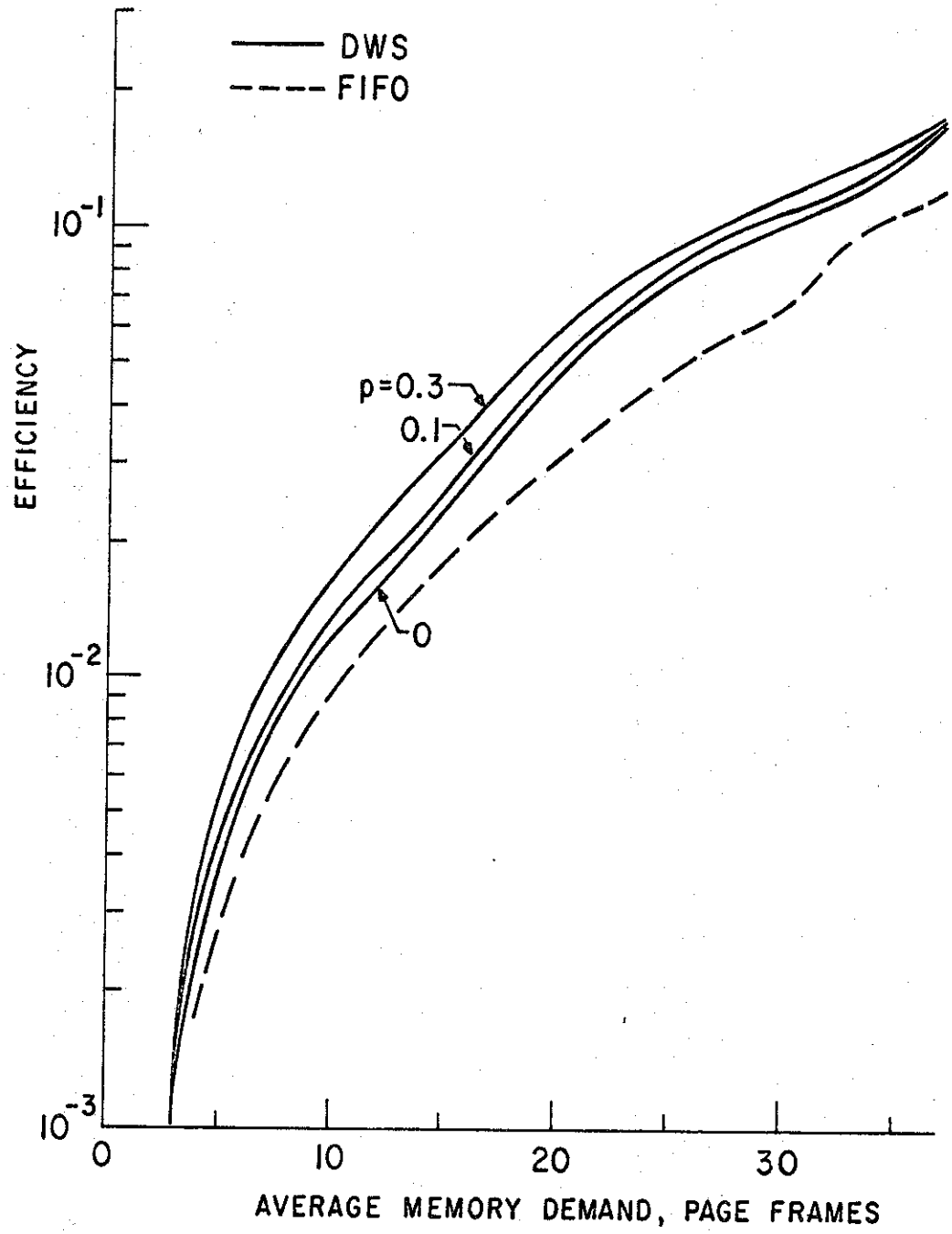

Figure 8. Efficiency vs. Average Memory Demand for Trace II With FIFO and DWS

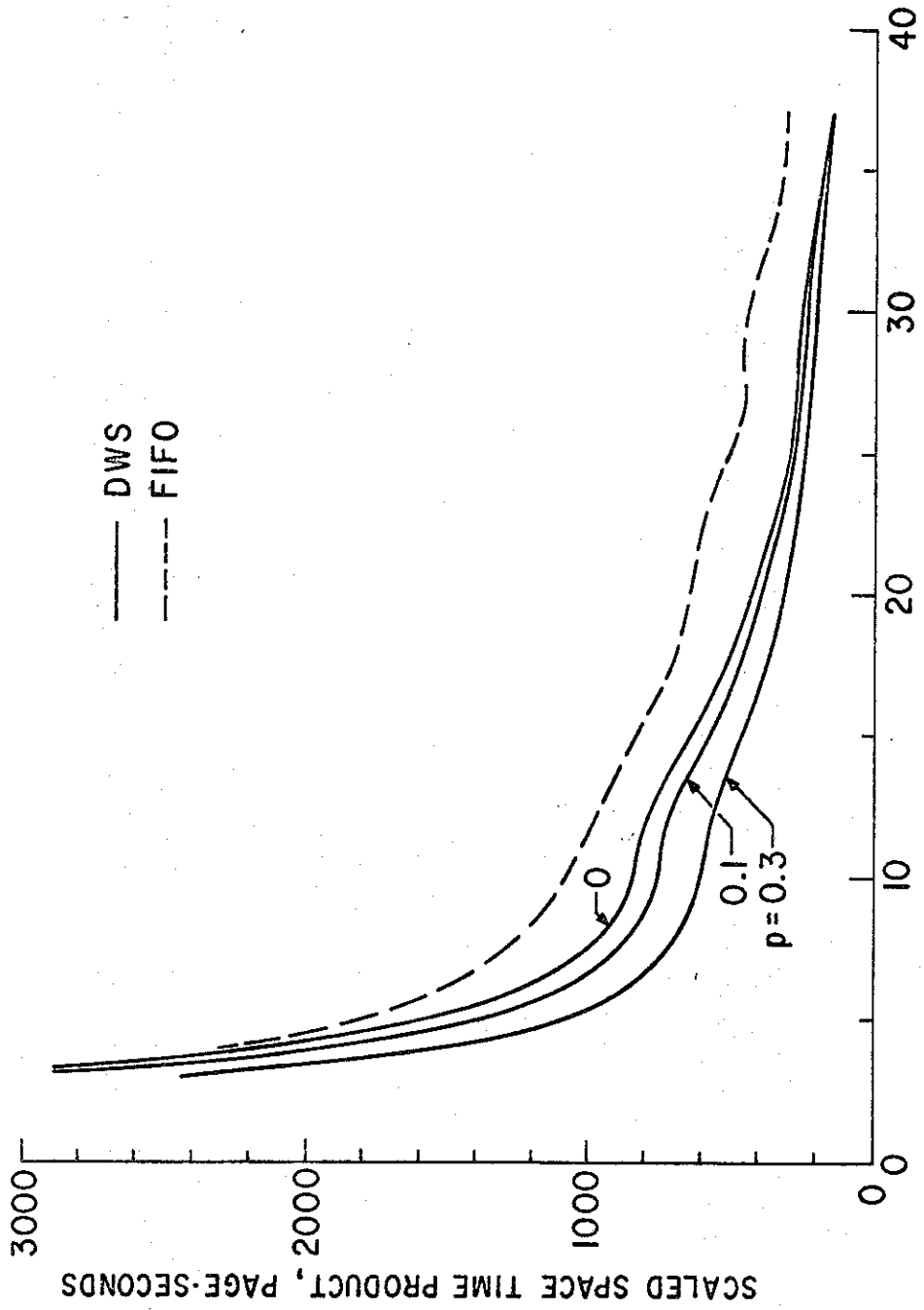

Figure 9. Scaled Space Time Product vs. Average Memory Demand for Trace II With FIFO and DWS

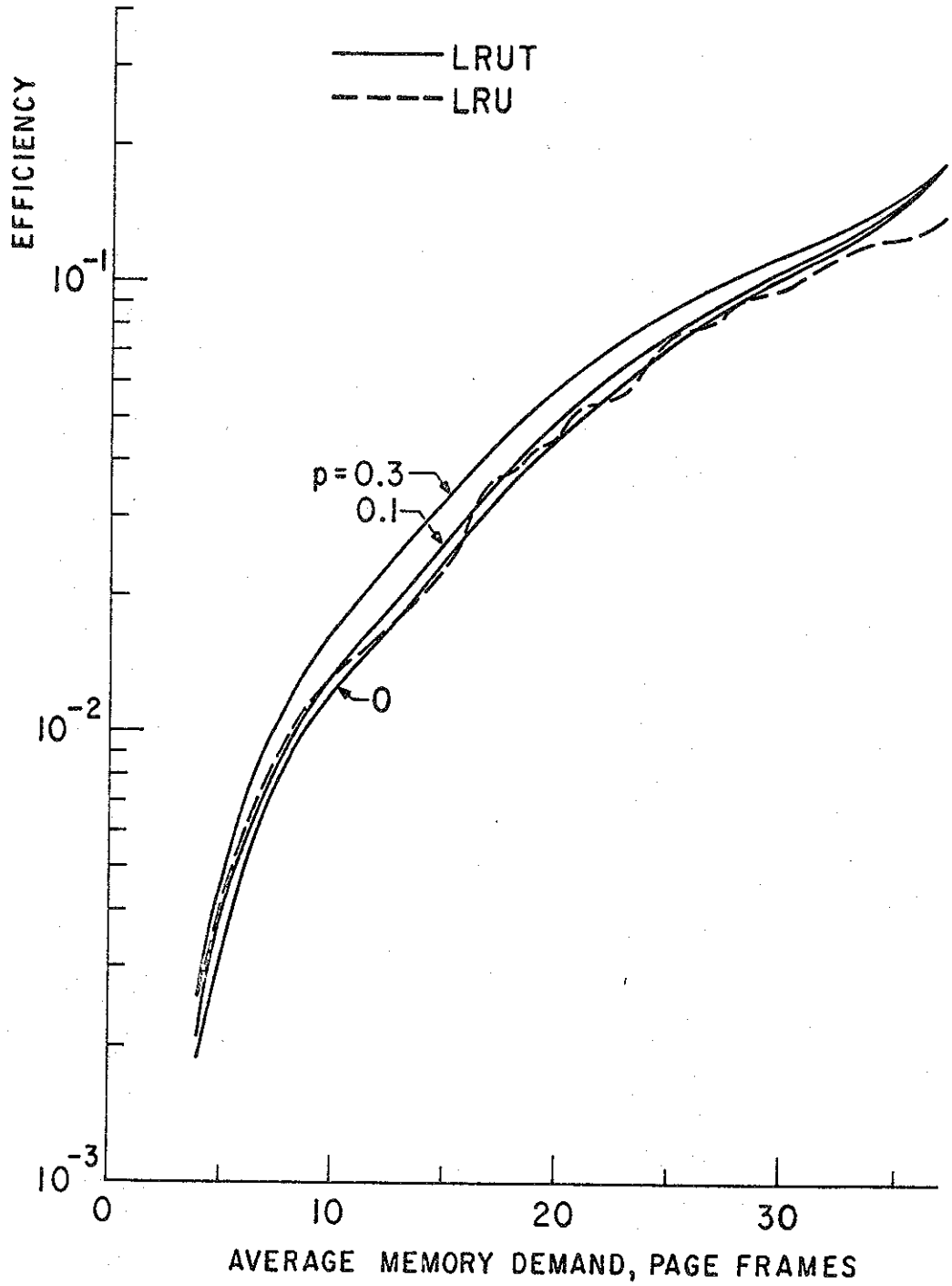

Figure 10. Efficiency vs. Average Memory Demand for Trace II With LRU and LRUT

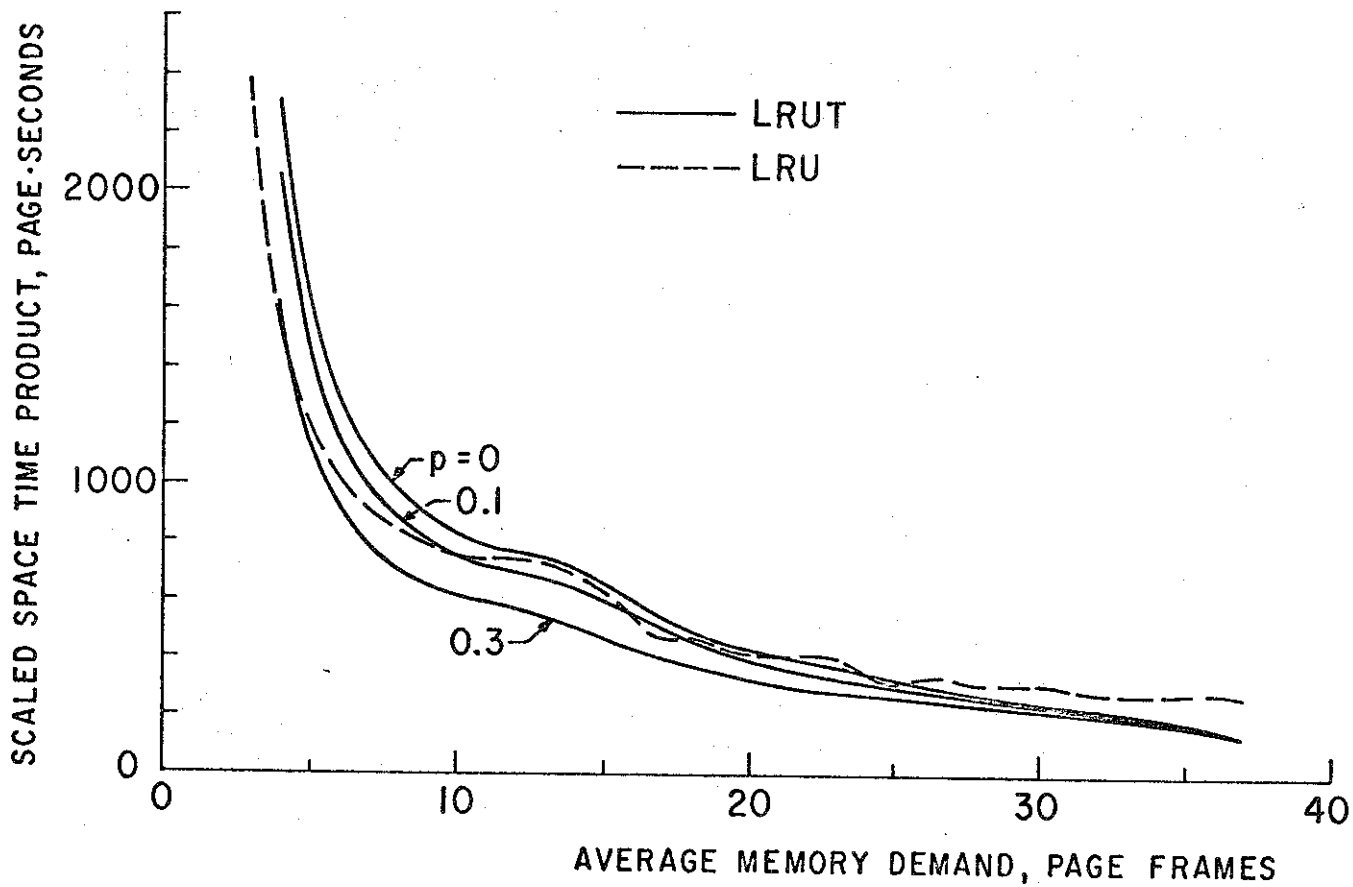

Figure 11. Scaled Space Time Product vs. Average Memory Demand for Trace II With LRU and LRUT

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.