SU-SEL 77-008 SU-326-P.39-18

SEQUENTIAL PREFETCH STRATEGIES FOR INSTRUCTIONS AND DATA

by

B. RAMAKRISHNARAU

January 1977

Technical Report No. 131

The work described herein was supported by the U.S.Energy Research and Development Administration under contract No. EY-76-S-03-0326-PA 39. Computer time was made available by the Stanford Linear Accelerator Center.

DIGITAL SYSTEMS LABORATORY
STANFORD ELECTRONICS LABORATORIES
STANFORD UNIVERSITY . STANFORD, CALIFORNIA

SU-SEL 77-008 SU-326-P.39-18

SEQUENTIAL PREFETCH STRATEGIES FOR INSTRUCTIONS AND DATA

by

B. RAMAKRISHNA RAU

January 1977

Technical Report No. 131

Digital Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

The work described herein was supported by the U.S. Energy Research and Development Administration under contract No. EY-76-S-03-0326-PA 39. Computer time was made available by the Stanford Linear Accelerator Center.

SU-SEL 77-008 SU-326-P.39-18

DIGITAL SYSTEMS LABORATORY
STANFORD ELECTRONICS LABORATORIES
STANFORD, CALIFORNIA

TECHNICAL REPORT NO. 131

SEQUENTIAL PREFETCH STRATEGIES FOR INSTRUCTIONS AND DATA

BY

B. RAMAKRISHNA RAU

ABSTRACT

An investigation of sequential prefetch as a means of reducing the average access time is conducted. The use of a target instruction buffer is shown to enhance the performance of instruction prefetch. The concept of generalized sequentiality is developed to enable the study of sequentiality in data streams. Generalized sequentiality is shown to be present to a significant degree in data streams from measurements on representative programs. This result is utilized to develop a data prefetch mechanism which is found to be capable of anticipating, on the average, about 75% of all data requests.

The work described herein was supported by the U.S. Energy Research and Development Administration under Contract No. EY-76-S-03-0326-PA 39. Computer time was made available by the Stanford Linear Accelerator Center.

Sequential Prefetch of Instructions and Data

1. Introduction.

A number of operations need to be performed in the course of executing an instruction; the instruction must be fetched and then decoded, the addresses of the operands must be generated, the operands must then be fetched and, lastly, the operation specified by the opcode must be carried out. The instruction decode, address generation and the final execution are generally accomplished in one processor cycle. As the speed of the processor is increased, the time spent in fetching the instructions and operands rapidly dominates the total instruction processing time and limits the performance of the processor. Overlapping and pipelining can mask the memory access time only partially and on the occurrence of conditional branches the access time makes itself felt once again.

Consequently, a good deal of effort has been spent in developing techniques for minimizing the memory access time. The use of a faster technology is the most direct approach, but is limited by the economics of the situation. A more cost-effective solution lies in the use of a memory hierarchy wherein the information which is most likely to be referenced is held in a small and fast buffer. Ideally, the fastest level would present an access time similar to that of a register. This would permit the operands to be accessed, operated upon and the results stored, all in one cycle. Such a buffer would present an access time of zero cycles.

A buffer that is required to operate this fast would necessarily have to be quite small. The performance of a buffer managed by a demand policy, (i.e., information is moved up to this buffer only if it is referenced and it is not already present), depends on the tendency for programs to re-reference information while still present in the buffer. With a buffer of very small capacity, we would find that by and large, the information would be displaced by the time it was re-referenced, resulting in rather poor performance. In such a situation, the use of anticipatory policies can be quite effective.

Perhaps the simplest anticipatory policy is that of sequential prefetch. Based on the assumption that instructions follow one another sequentially, one might prefetch the word that sequentially follows the one being decoded currently. This will result in part of the access time being masked by the decoding of the previous word and reduces the effective access time. In general, one could prefetch the d words following the one being decoded. This is the conventional prefetch of degree d.

All measurements reported in this paper were made on one or more of five trace tapes which represent a sample of the workload that might be found at a general purpose computer center. The tapes were created by an instruction-by-instruction trace of programs executing in the user mode on a 360 compatible architecture. The results should be applicable at the qualitative level to similar register oriented architectures.

Sequential Prefetch of Instructions and Data

The trace tapes are:

043 - Fortran execution

049 - Cobol execution

050 - Cobol compilation

051 - Fortran compilation

052 - Cobol Sort execution

Sequential Prefetch of Instructions and Data

2. Instruction Prefetch

The obvious presence of sequentiality in the instruction stream has resulted in sequential prefetch being widely employed in medium and high speed processors. Given a sufficiently high degree of prefetch, all sequential instruction fetches appear to have zero access time and the full memory latency is seen only on requests which are non-sequential to their preceeding requests. On the other hand, an increase in the degree of prefetch is accompanied by an increase in the total number of requests made to the memory. This increased traffic will, eventually, cause memory interference and effectively a longer access time.

Estimates of both the average access time (neglecting any memory interference) and the traffic can be derived from the run length statistics. A run is defined to be the series of requests starting with a non-sequential request and terminated by (but not including) the next non-sequential request. The number of runs will be exactly equal to the number of non-sequentialities in the request stream. Let the frequency of non-sequential requests be n (per request). Then the average number of instruction words decoded in a run is given by 1/n. If the degree of prefetch is d then the number of requests during each run will be increased by the d prefetched words which are not used. The average number of requests made per run is (1/n + d). The average number of requests made per instruction word that is actually decoded is given by

$$n(1/n + d) = (1 + nd). \qquad (1)$$

The memory traffic is a linear function of the degree of prefetch.

An approximate figure for the average access time is arrived at from the run length distribution (Fig.1). The first request of a run always experiences the full latency of the memory. If the degree of prefetch is d, then the next d requests see zero latency since their access time is masked by the access time of the first request. Subsequent requests see the access time of the memory less the time that it takes to decode the d previous requests. Let the average time to decode an instruction word be r cycles (this allows for the presence of multiple instructions in a word) and let the memory access time be T cycles. All runs of length 1 through d+1 have a total access time of T per run. Each increment to the run length beyond d+1 increases the total access time for the run by $(T-rd)^+ = \max(0,T-rd)$. Thus if p(i) is the probability that a run is of length i, the average access time is given by

$$n\left(\sum_{i=1}^{d+1} Tp(i) + \sum_{i=d+2}^{\infty}\left(T + (i-d-1)(T-rd)^+\right)p(i)\right)$$

The assumptions which have been made implicitly are that:

1. The memory is interleaved and can service requests at the sustained rate of one every (decode) cycle.

2. The width of a memory module is one word, or conversely, a word is defined to be the number of bits available from one access of

a memory module. This is also the width of each register in the prefetch buffer.

3. The degree of interleaving is at least as large as the degree of prefetch plus one (d+1).

4. Interleaved memory interference is ignored. (The interference can be approximately accounted for by a suitable adjustment in the access time, T).

5. On the occurrence of a non-sequentiality, the entire prefetch buffer is invalidated.

Computationally, a more convenient formula for the average access time is given by

$$nT + n(T-rd)^{+}\left((1/n - \sum_{i=1}^{d+1} ip(i)) - (d+1)(1 - \sum_{i=1}^{d+1} p(i))\right) \qquad (2)$$

where use is made of the fact that

$$\sum_{i=1}^{\infty} p(i) = 1$$

$$\sum_{i=1}^{\infty} ip(i) = 1/n$$

Fig.2 is a plot of the average access time and traffic as functions of the degree of prefetch where for the sake of illustration we have

assumed that T=10 cycles. This assumption is maintained throughout. We see that the average access time is limited below by the fact that non-sequentialities must necessarily see the full memory access time.

This being the case, any attempt to further reduce the average access time should focus on the non-sequential requests. One strategy is to retain the targets of previous non-sequentialities in a fast buffer referred to as a target instruction buffer [RAU77]. In view of the looping that exists in programs, there is a good chance that the target of a non-sequentiality will be found in the target instruction buffer. In fact, to completely eliminate the penalty of a non-sequentiality it is necessary to buffer, in addition to the target word, the d succeeding words, where d is the degree of prefetch, so that the entire prefetch buffer is filled with zero latency. It was found that buffering a couple of the most recent targets produced a significant improvement but that additional buffering was of negligible value. Fig.3 compares the conventional prefetch strategy with the strategy which couples prefetch with a target instruction buffer which retains the two most recent targets. To allow a basis for comparison, the traffic to main memory is plotted against the average access time with the degree of prefetch as a parameter. The use of a target instruction buffer can be seen to provide a lower access time with relatively little additional traffic as the degree of prefetch is increased. In comparison, the conventional prefetch causes a substantial increase in traffic and is unable to reduce the access time as much as the use of a target instruction buffer

can. If we let h be the fraction of non-sequentialities for which the target is found in the target instruction buffer, then we have

$$\text{Traffic} = 1 + nd(1-h), \tag{3}$$

$$\text{Avg. Acc. Time} = n(1-h)T + n(T-rd)^{+}\left(1/n - \sum_{i=1}^{d+1} ip(i) - (d+1)\left(1 - \sum_{i=1}^{d+1} p(i)\right)\right) \tag{4}$$

since h of the non-sequentialities which would have seen an access time of T now see zero access time.

The target instruction buffer proves superior to a conventional instruction cache of the same capacity if both are constrained to be small in size. This constraint is necessary if we wish to have an effective access time of zero cycles since the addressing for a large cache tends to be more time-consuming. The reason that the target instruction buffer proves superior is that it buffers only the target which is the most crucial section of a sequential run. The conventional cache, on the other hand, replaces the target with subsequent requests if the run is long enough. In the event of a large enough loop, the target is not present when the program branches back to the beginning of the loop. In fact, it can happen that each word is displaced from the buffer just before it is requested. Every request would then go to memory and the buffer would effectively be non-existent.

3.Data Prefetch

The prefetch of operands is easily effected in machines with vector instructions. With the knowledge that a vector operation is underway, it is simple and advantageous to prefetch the vector operands. In architectures such as the 360, the PDP-11, etc., which have no vector instructions (although multiple character instructions may be thought of as such) the identification of vector operations, which have been coded in the form of a loop, is difficult. Accordingly, data prefetch is not employed in such architectures. Our interest is to ascertain the extent to which sequentiality exists in the data streams of such architectures. Secondly, assuming that it does, in fact, exist, we need to outline a procedure for observing it.

This second point is clarified by an example. Let us assume that the two operands are arrays which are being operated upon by an instruction loop. Let these two arrays be stored in memory in two areas starting at locations 100 and 200 respectively. Then, although the individual arrays are accessed in a purely sequential fashion, the resulting stream of requests might be of the form 100, 200, 101, 201, . . . as a result of the interleaving of requests for the two operands. Generally, the sequentiality of a request stream is determined by measuring the fraction of requests that are sequential to the preceeding request. Such a measurement would indicate no sequentiality whatsoever in our example. And yet, we know that the request stream has been obtained by

merging two perfectly sequential request streams. We can observe this hidden sequentiality by generalizing our measurements. If a request is not sequential to the preceeding one, we check whether it is sequential to the one preceeding that. In this manner, we accumulate statistics which indicate how far we need to go back in the request stream to find the sequential predecessor of the current request. Put another way, we obtain the distribution of the interval (measured in requests) between sequential requests. If the probability of short intervals is high, then we may conclude that the request stream is the result of merging two or more fairly sequential streams. In our example, we would measure unit probability of sequential requests being two requests apart. If, on the other hand, the request stream were truely random, we would expect to observe only a very small probability of sequential requests being separated by small intervals.

Using this generalized measure of sequentiality, measurements were conducted upon three of the trace tapes. The ones selected were the traces of Fortran execution, Cobol execution and Cobol compilation. The interval distributions for these three are shown in Figs.4,5,6. In all three we notice a significant probability of sequential requests being separated by short intervals, leading us to the conclusion that generalized sequentiality is prevalent to a great extent in the data stream of such architectures. In addition, it was found that a large number of requests were to the same word as the previous request. This would not show up in the measurements we have outlined but can be used to decrease the average access time.

4. Implementation of Generalized Prefetch

The generalized sequentiality of the data stream can be exploited to permit data prefetch. One method of implementing data prefetch is to take m conventional prefetch buffers and order them in a Least Recently Used (LRU) stack. Thus in Fig.7, the prefetch buffer with degree of prefetch 2, which is in the third position in the stack will be "used" if the next request is to either C or to C+1, and that buffer would then be moved to the top of the stack. In addition, if the request had been to C+1, then C would be discarded and a prefetch request for C+3 would be initiated. This algorithm is formalized below.

Let the Generalized Prefetch Buffer consist of m conventional prefetch buffers, each of degree d (Fig.8). Let these prefetch buffer arrays be ordered by recency of use in an LRU stack, and let the buffer that is in the i-th position in the stack, ($1 \leq i \leq m$), contain the words with addresses A_i, $A_i+1,\ldots,A_i+d$. Then for each request R:

1. Look for the first match between R and A_1, A_1+1, A_2, $A_2+1,\ldots,A_m$, A_m+1 in that order.

2. If a match is found with A_j for $1 \leq j \leq m$, then move the buffer in position j to the top of the stack.

3. If a match is found with A_j+1 for $1 \leq j \leq m$, then set $A_j=A_j+1$, initiate a request to memory for A_j+d and move the buffer in position j to the top of the stack.

4. If no match is found, then invalidate the buffer in position m (the least recently used buffer), set $A_m=R$, initiate requests for A_m+1, A_m+2,...., A_m+d and move the buffer in position m to the top of the stack.

An unanticipated request occurs each time a match is not found. Let us term such a request a "miss". It is of interest to measure the probability of a miss as a function of m, the number of prefetch buffers. Since when looking for a match, we only look at A_i and A_i+1 for each prefetch buffer array, the degree of prefetch is irrelevant in dtermining the miss ratio. It will, of course, affect the performance of the generalized prefetch buffer array in practice since the degree of prefetch determines how much in advance a request is made. However, the miss ratio can be thought of as an upper bound on the performance obtainable if we assume that all anticipated requests have zero access time. Measurements of the miss ratio were made on all five of the trace tapes and are displayed in Figs.9 through 13. It is seen, in general, that m=2 reduces the miss ratio substantially but that there is little to be gained by increasing m beyond 3. With m=3, we find that on the average about 75% of all data requests are anticipated.

As with instructions, the run length statistics can provide us with an estimate of the average access time and the traffic to memory as a function of the degree of prefetch d. The previous formulae (Equations 1 and 2) remain valid but the notion of a run needs to be redefined slightly. A run begins with a miss. Associated with each prefetch buffer array is a counter. The counter of the buffer array that is lowest in the stack is initialized to 1 on the occurrence of a miss. Each request which is not a miss causes the counter to be incremented in the buffer array in which the match was found. Associated with each buffer array at any instant is a run which is currently in progress. Finally, a run is terminated if it corresponds to the lowest buffer array at the time of a miss. The counter value prior to resetting it to 1 is the length of that run.

The run length statistics clearly depend upon the number of prefetch buffer arrays being used. Fig.14 displays sample run length statistics of the data stream for the trace of the Fortran execution. The number of prefetch buffer arrays was set at 3. On comparing the run length statistics for instructions and data, we notice that the latter are more highly skewed,i.e., for the same average run length in both, data tends to have a higher probability of very short runs counterbalanced by a small percentage of runs of very great length. This is characteristic of the data stream with the runs of length 1 probably corresponding to random scalar requests and the long runs corresponding to array operations. The run length statistics may be inserted into the

expressions for traffic and average access time (Equations 1 and 2) to obtain approximate values for these quantities as functions of the degree of prefetch. The result of doing so is shown in Fig. 15 and demonstrates that data prefetch can be quite effective indeed -- almost as effective as instruction prefetch.

The skewness of the data run length distribution means that the average run length of runs greater than 1 in length will be much greater than the average run length of all the runs. This suggests a strategy whereby the traffic can be reduced at the expense of increasing the average access time somewhat. The generalized prefetch is modified so as to activate the prefetch only if the run length is found to be greater than 1. The prefetch mechanism will not be active on the scalar requests thereby reducing the traffic. However, the second request on a run of length 2 or more will not be prefetched resulting in an increased access time. Equations 1 and 2 must be altered slightly, but the line of reasoning is much the same. We now have:

$$\text{Traffic} = n\left(1/n + d\sum_{i=2}^{\infty} p(i)\right) = 1 + nd(1-p(1)) \tag{5}$$

$$\text{Avg. Acc. Time} = 2nT(1-p(1)) + n(T-rd)^{+}\left(1/n - \sum_{i=1}^{d+2} ip(i) - (d+2)\left(1 - \sum_{i=1}^{d+2} p(i)\right)\right) \tag{6}$$

The trade-off is between the traffic generated and the average access

time. Accordingly, the two prefetch strategies are compared in Fig.16 by plotting the traffic versus the average access time with the degree of prefetch as a parameter. The number of prefetch arrays in both cases was 3. For the same average access time, the modified prefetch strategy generates less traffic over the range of access times that it is able to achieve. However, it is unable to achieve as low an access time as can the other strategy.

An attempt to use generalized prefetch for the instruction stream showed that increasing the number of prefetch buffer arrays beyond 1 was of no advantage. The conventional form of prefetch is adequate for the instruction stream. Also, the use of the equivalent of the target instruction buffer for data was not found to be useful. Presumably, data is not referenced in loops the way instructions are.

5. Conclusion

The notion of generalized sequentiality has been introduced to deal with composite, merged request streams such as those which occur in data accessing. A set of measurements have been defined with which we can test the degree to which generalized sequentiality exists in actual programs. The results of conducting these measurements on representative programs demonstrated the existence of a substantial degree of generalized sequentiality in the data stream. An implementation of generalized data prefetch was outlined and its performance was estimated. It was found that on the average about 75% of all data requests could be anticipated and prefetched resulting in a substantial decrease in the average access time. It might be possible to generalize the loop buffer of the 360/91 in a like manner to further improve performance.

In addition, the use of a target instruction buffer was found to enhance instruction prefetch. Also, a modification of the data prefetch strategy was investigated which allowed a trade-off in the average access time and traffic.

Acknowledgements

The plots were constructed using Topdraw, a package written by Roger Chafee at the Stanford Linear Accelerator Center.

Reference

RAU77 Rau,B.R., and Rossmann,G.E., "The effect of instruction prefetch strategies upon the performance of pipelined instruction units", Proceedings of the 4th Annual Symposium on Computer Architecture, Silver Springs, Maryland, 1977.

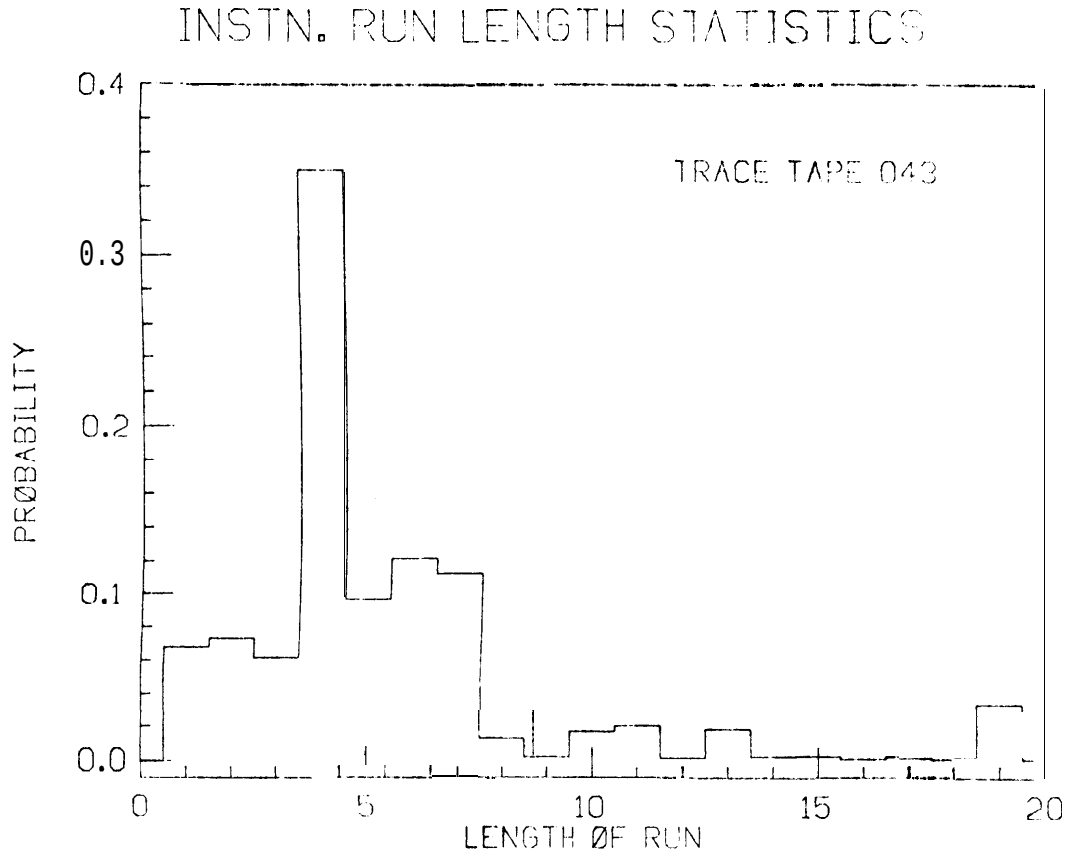
INSTN. RUN LENGTH STATISTICS
TRACE TAPE 043
PROBABILITY
0.4
0.3
0.2
0.1
0.0
0
5
10
15
20
LENGTH OF RUN

FIG. 1

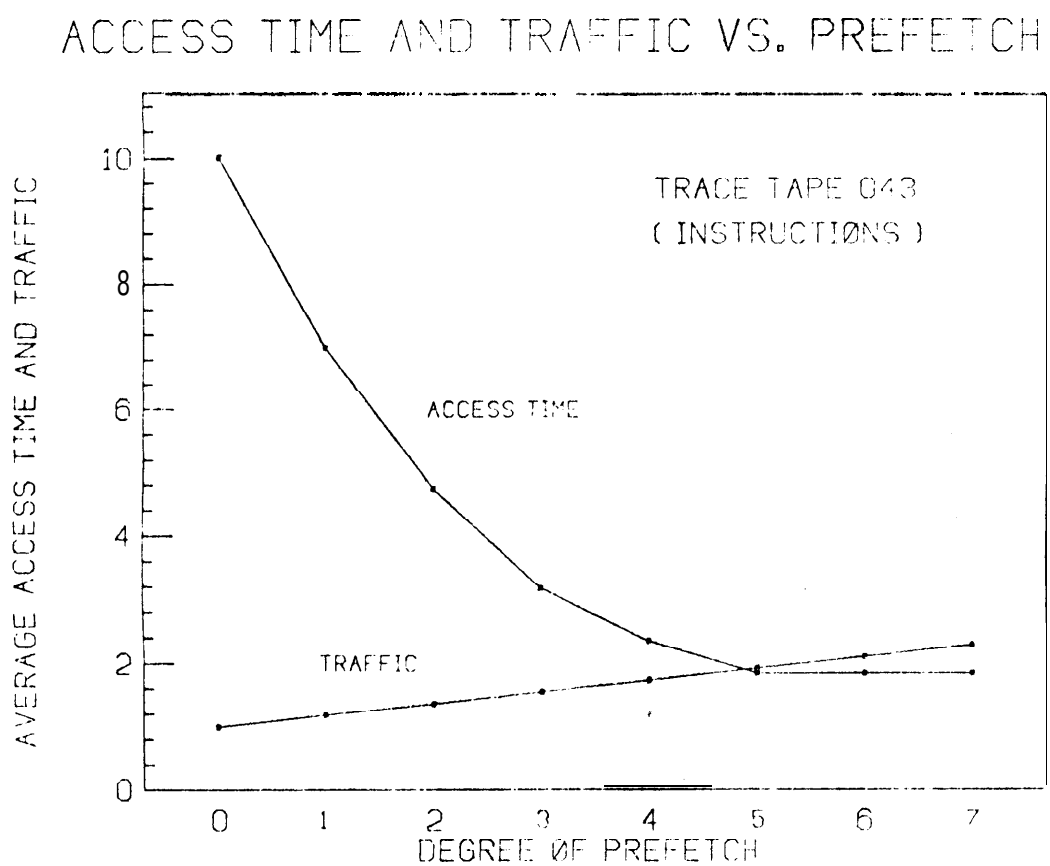
ACCESS TIME AND TRAFFIC VS. PREFETCH
TRACE TAPE 043
(INSTRUCTIONS)
ACCESS TIME
TRAFFIC
AVERAGE ACCESS TIME AND TRAFFIC
10
8
6
4
2
0
0
1
2
3
4
5
6
7
DEGREE OF PREFETCH

FIG. 2

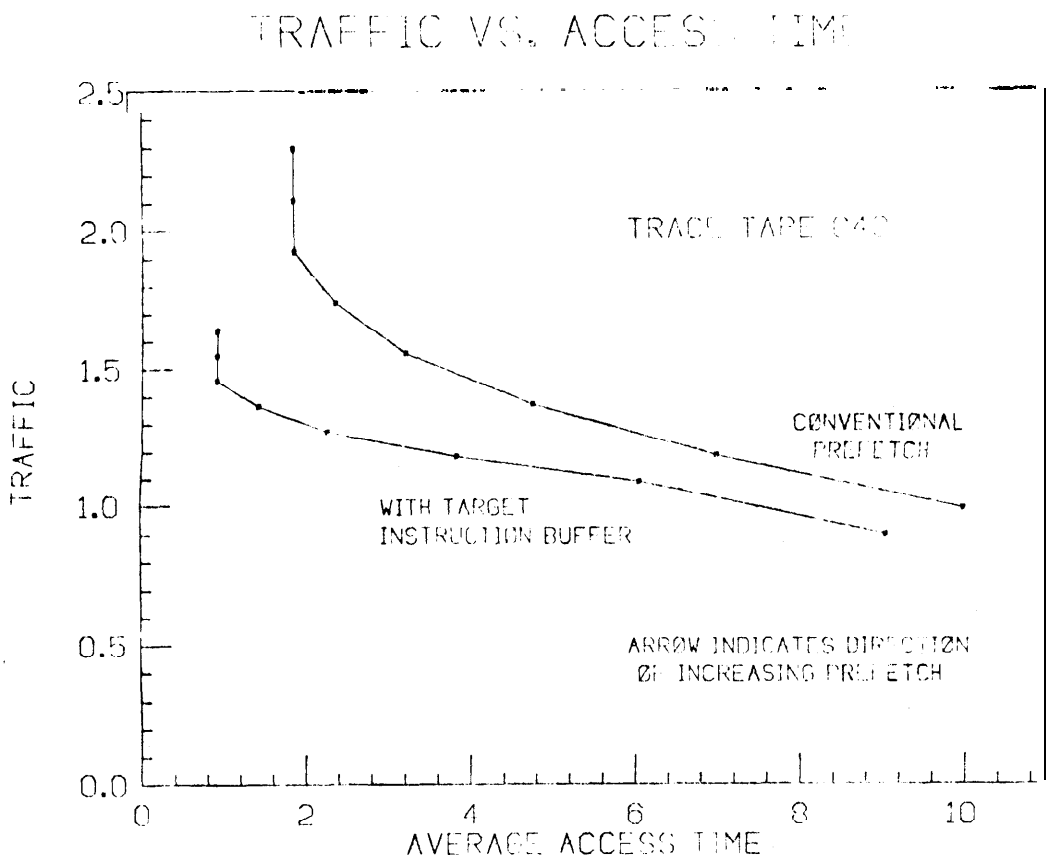
TRAFFIC VS. ACCESS TIME
TRACE TAPE 040
CONVENTIONAL PREFETCH
WITH TARGET INSTRUCTION BUFFER
ARROW INDICATES DIRECTION OF INCREASING PREFETCH
TRAFFIC
AVERAGE ACCESS TIME
2.5
2.0
1.5
1.0
0.5
0.0
0
2
4
6
8
10

FIG. 3

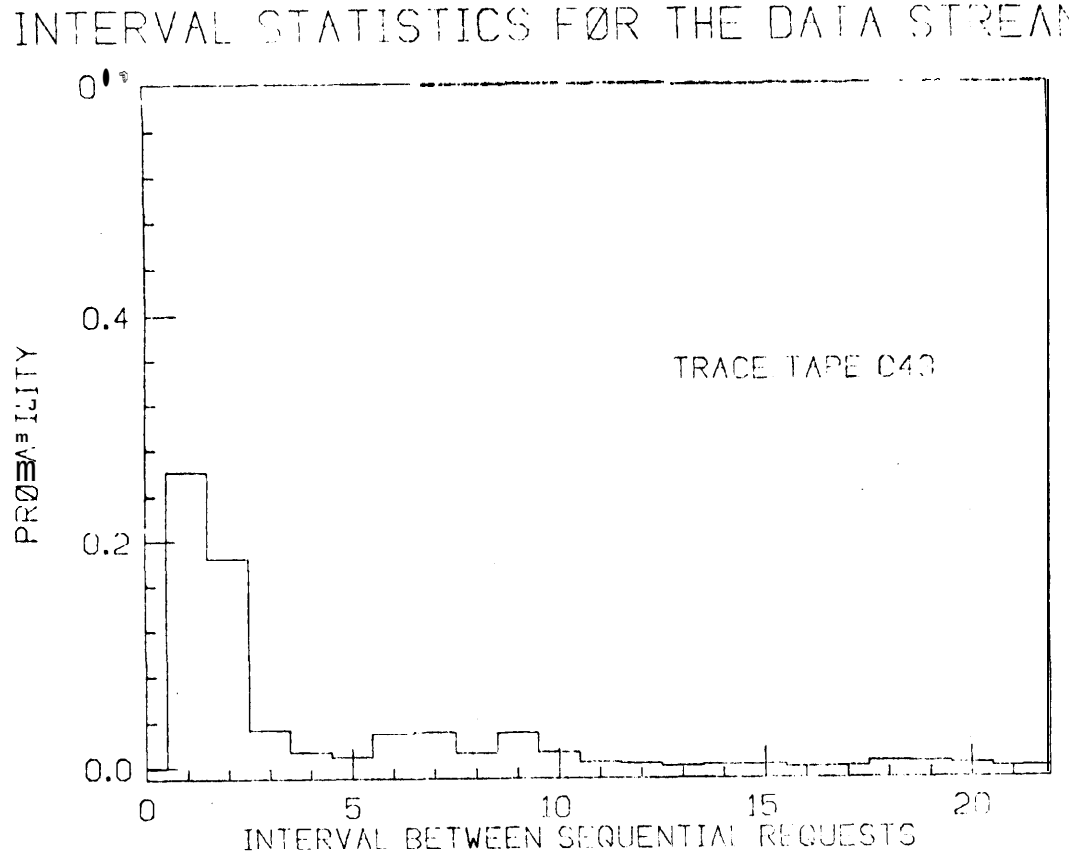
INTERVAL STATISTICS FOR THE DATA STREAM
TRACE TAPE 040
PROBABILITY
INTERVAL BETWEEN SEQUENTIAL REQUESTS
0.4
0.2
0.0
0
5
10
15
20

FIG. 4

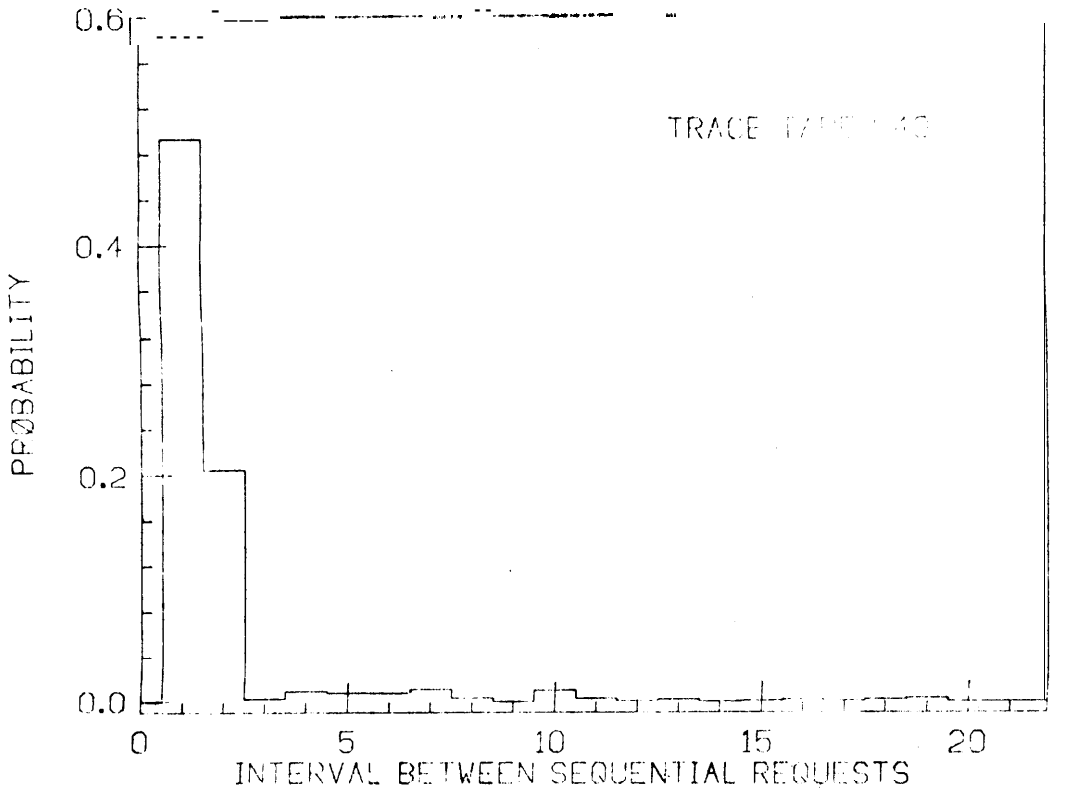
INTERVAL STATISTICS FØR THE DATA STREAM
TRACE TAPE C50
PRØBABILITY
0.6
0.4
0.2
0.0
0
5
10
15
20
INTERVAL BETWEEN SEQUENTIAL REQUESTS

FIG. 5

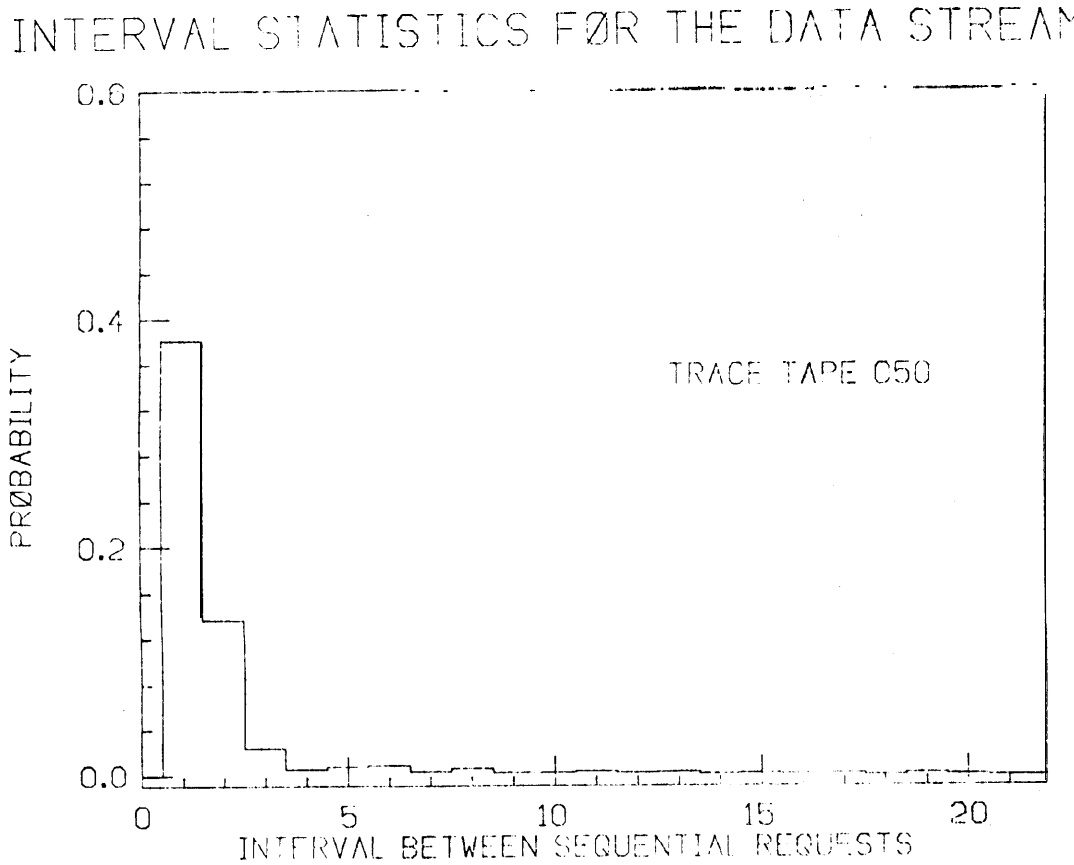
INTERVAL STATISTICS FØR THE DATA STREAM
TRACE TAPE C50
PRØBABILITY
0.6
0.4
0.2
0.0
0
5
10
15
20
INTERVAL BETWEEN SEQUENTIAL REQUESTS

FIG. 6

The Generalized Prefetch Buffer

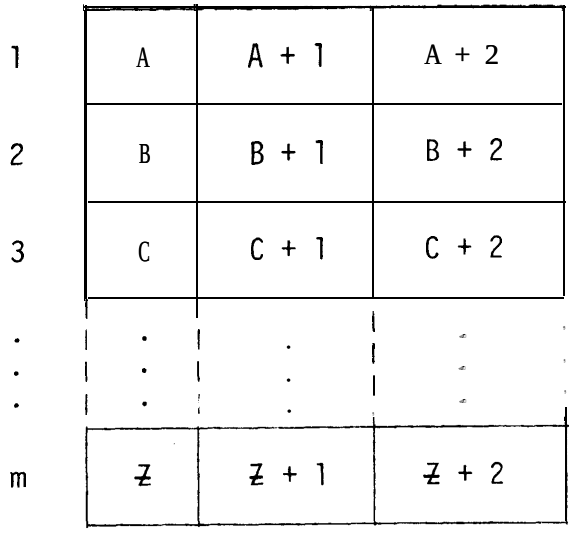

Figure 7

The Generalized Prefetch Buffer

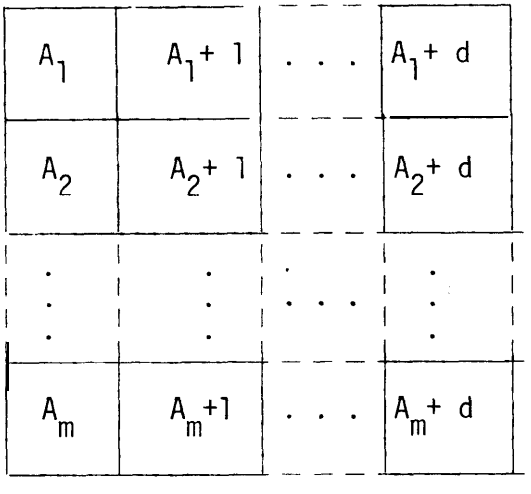

Figure 8

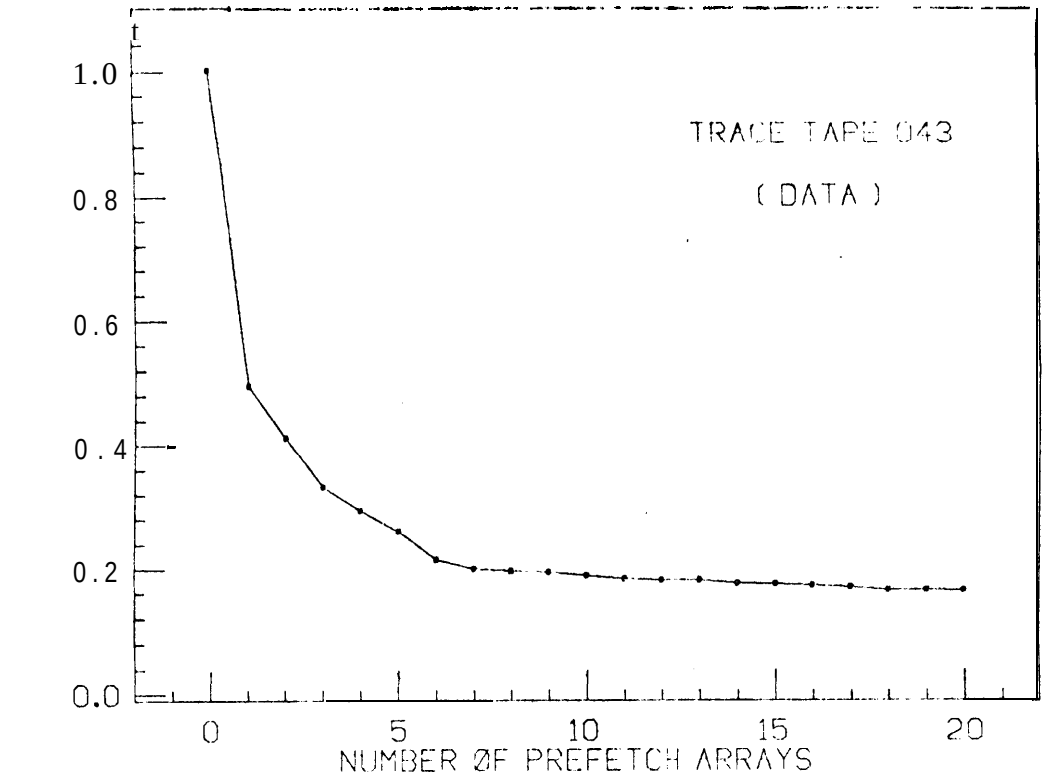
MISS RATE VS. NUMBER ØF PREFETCH ARRAYS
1.0
0.8
0.6
0.4
0.2
0.0
TRACE TAPE 043
(DATA)
0
5
10
15
20
NUMBER ØF PREFETCH ARRAYS

FIG. 9

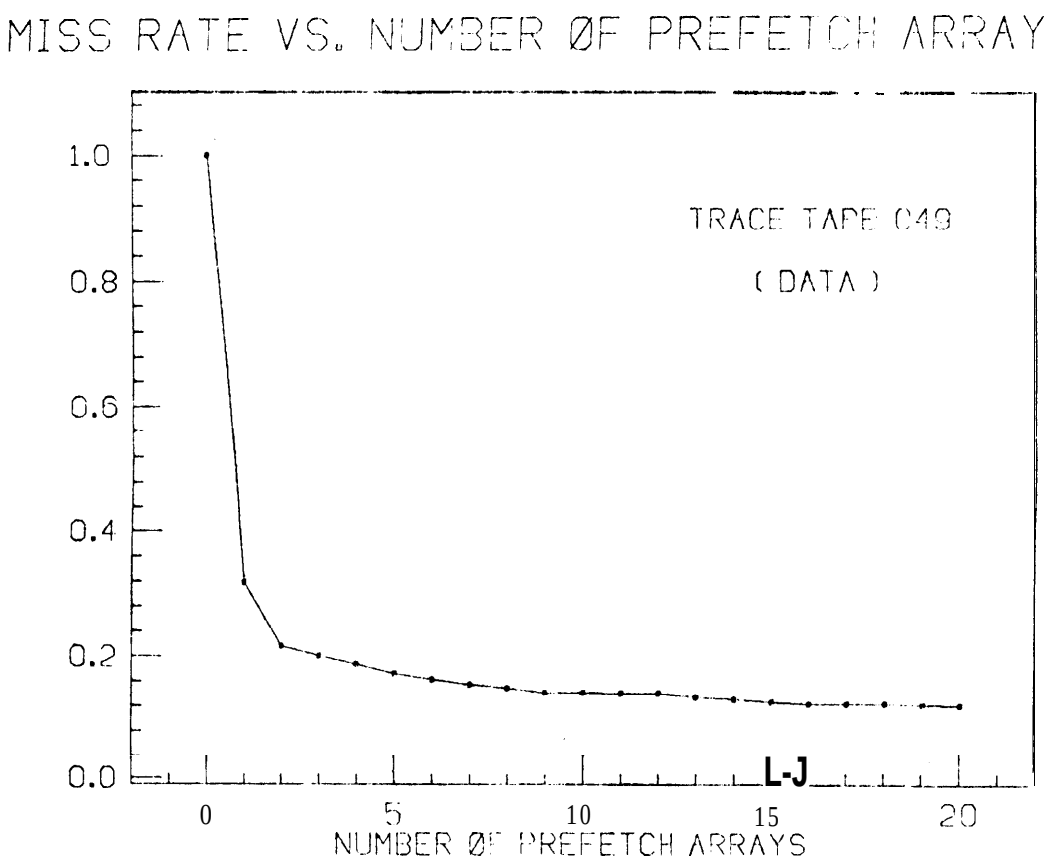
MISS RATE VS. NUMBER ØF PREFETCH ARRAYS
1.0
0.8
0.6
0.4
0.2
0.0
TRACE TAPE 043
(DATA)
L-J
0
5
10
15
20
NUMBER ØF PREFETCH ARRAYS

FIG. 10

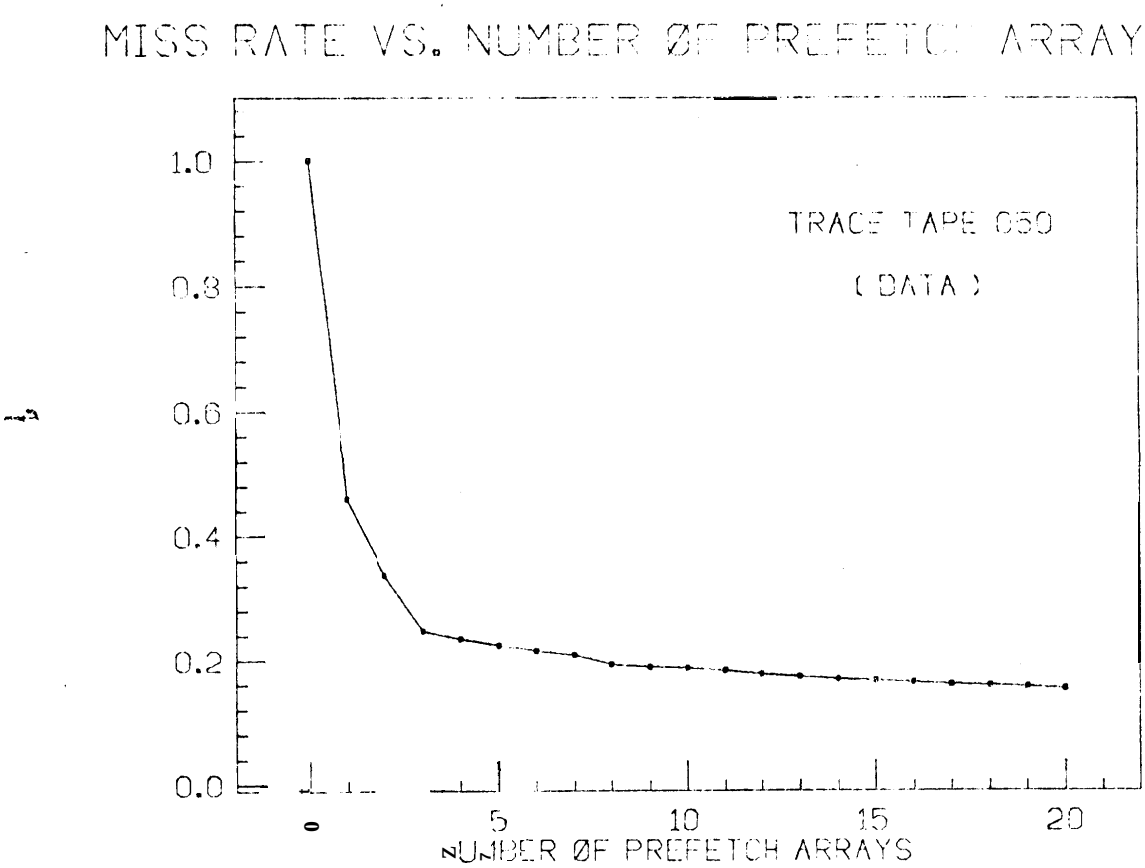
MISS RATE VS. NUMBER ØF PREFETCH ARRAYS
TRACE TAPE 050
(DATA)
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
NUMBER ØF PREFETCH ARRAYS

FIG. 11

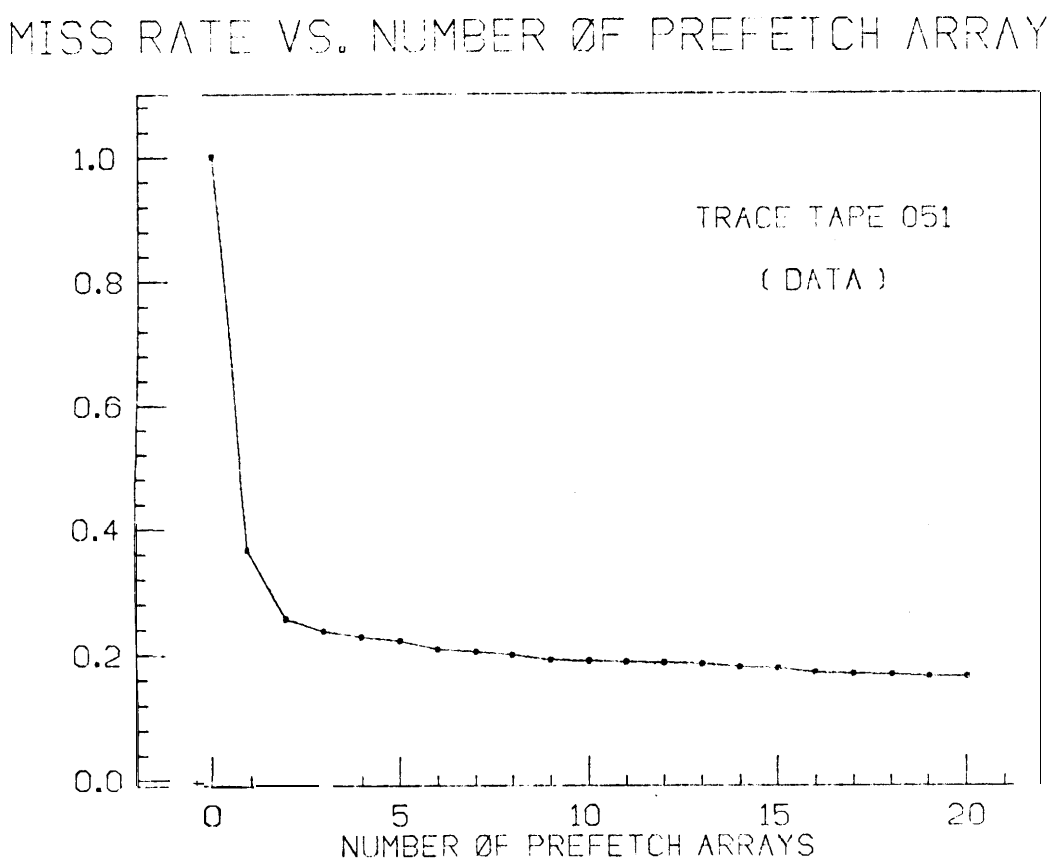
MISS RATE VS. NUMBER ØF PREFETCH ARRAYS
TRACE TAPE 051
(DATA)
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
NUMBER ØF PREFETCH ARRAYS

FIG 12

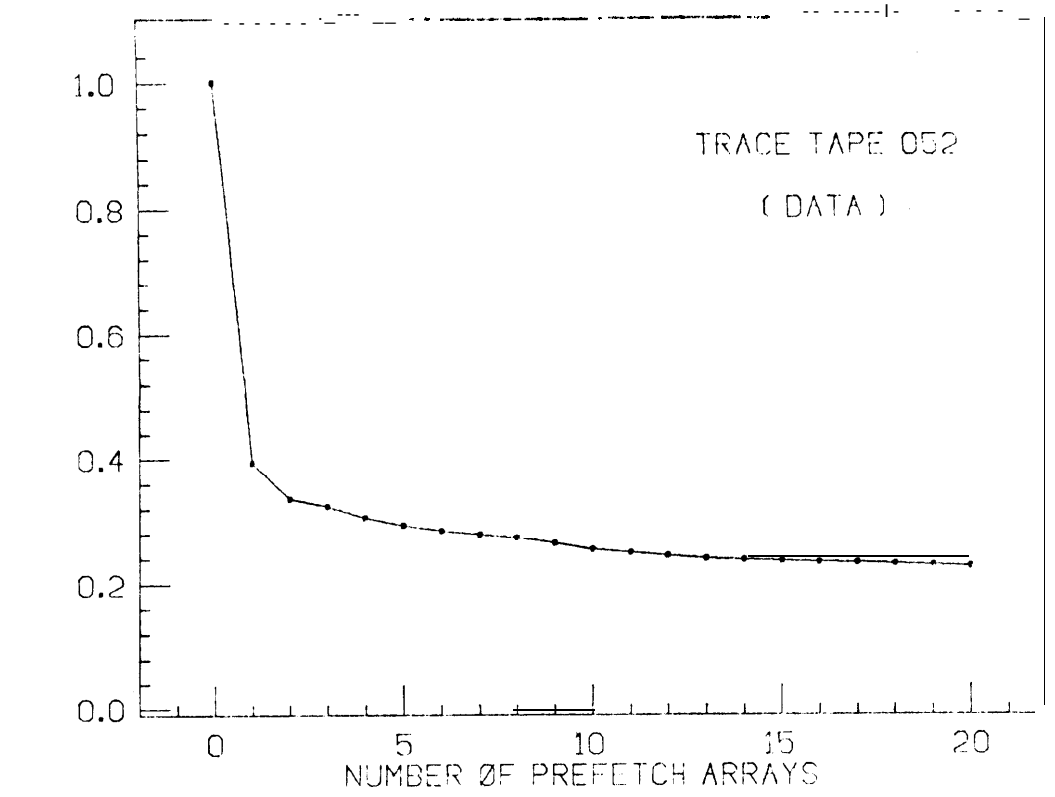
MISS RATE VS. NUMBER ØF PREFETCH ARRAYS
TRACE TAPE 052
(DATA)
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
NUMBER ØF PREFETCH ARRAYS

FIG. **13**

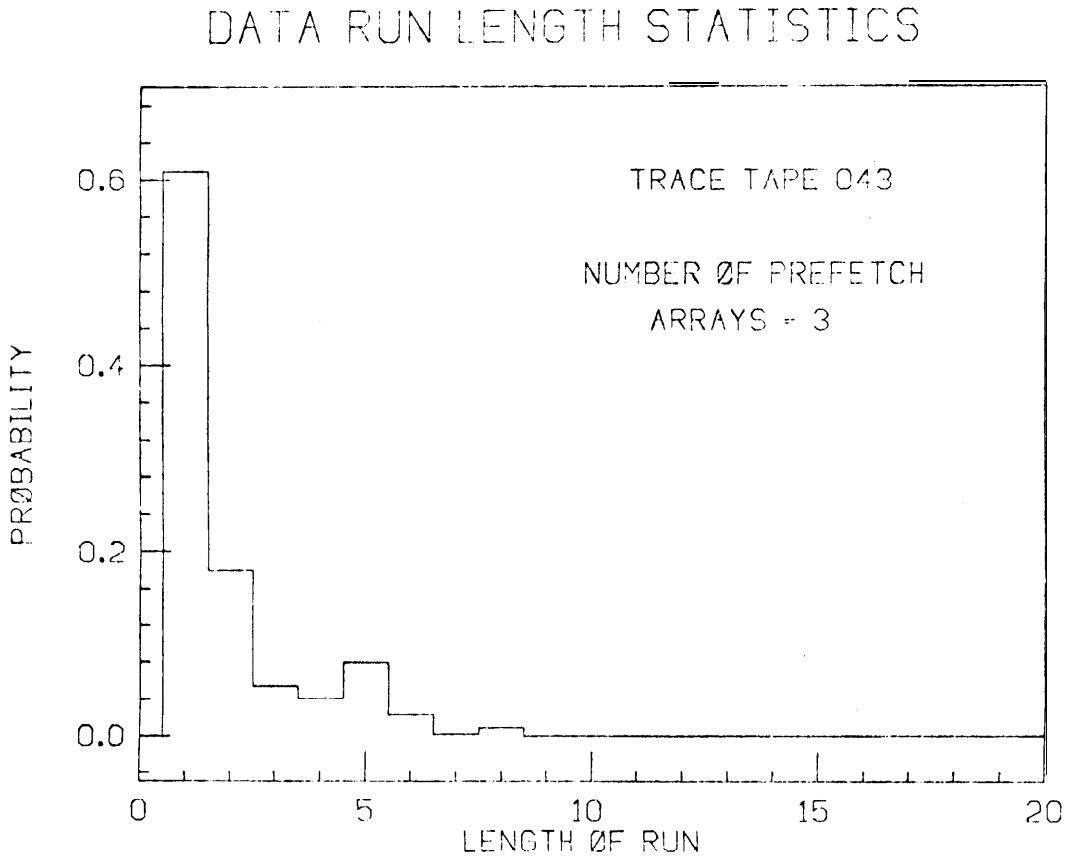
DATA RUN LENGTH STATISTICS
TRACE TAPE 043
NUMBER ØF PREFETCH
ARRAYS = 3
PRØBABILITY
0.6
0.4
0.2
0.0
0
5
10
15
20
LENGTH ØF RUN

FIG. 14

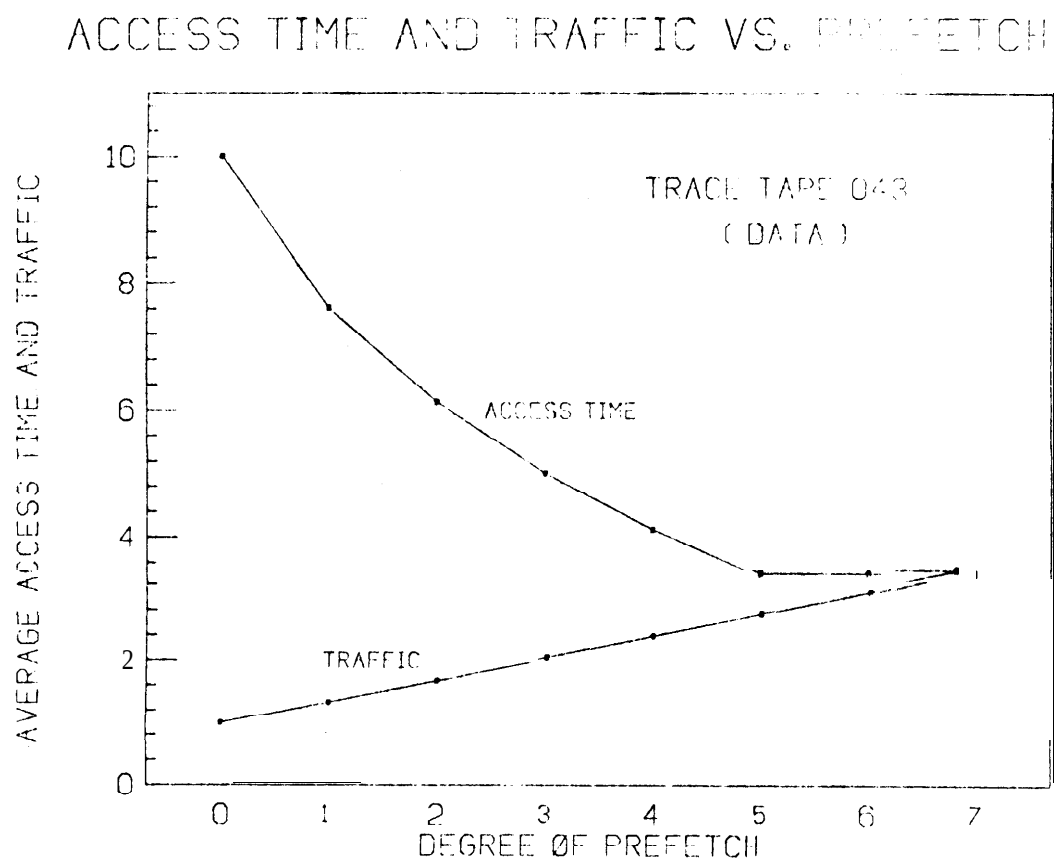
ACCESS TIME AND TRAFFIC VS. PREFETCH
TRACE TAPE 043
(DATA)
ACCESS TIME
TRAFFIC
AVERAGE ACCESS TIME AND TRAFFIC
DEGREE ØF PREFETCH
0
1
2
3
4
5
6
7
0
2
4
6
8
10

FIG. 15

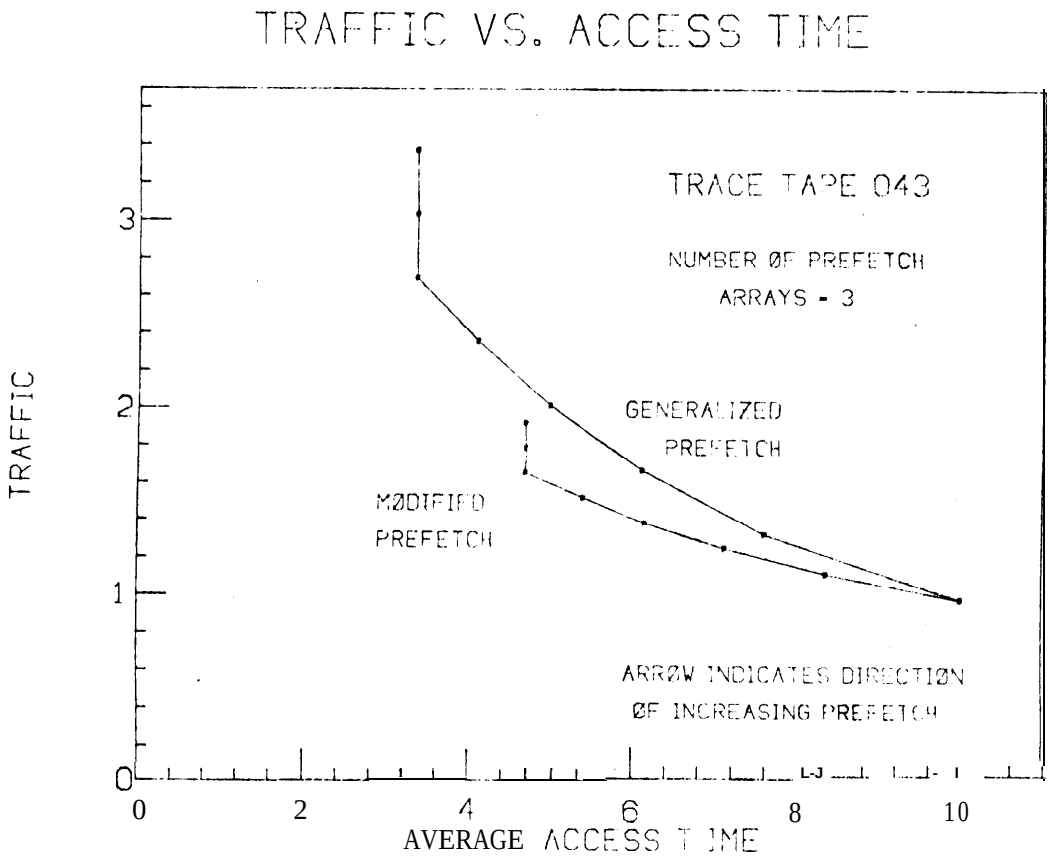
TRAFFIC VS. ACCESS TIME
TRACE TAPE 043
NUMBER ØF PREFETCH
ARRAYS - 3
GENERALIZED
PREFETCH
MØDIFIED
PREFETCH
ARRØW INDICATES DIRECTIØN
ØF INCREASING PREFETCH
TRAFFIC
AVERAGE ACCESS TIME
0
1
2
3
0
2
4
6
8
10

FIG 16