An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine

L. A. BELADY, R. A. NELSON, AND G. S. SHEDLER
IBM Thomas J. Watson Research Center, Yorktown Heights, New York

The running time of programs in a paging machine generally increases as the store in which programs are constrained to run decreases. Experiments, however, have revealed cases in which the reverse is true: a decrease in the size of the store is accompanied by a decrease in running time.

An informal discussion of the anomalous behavior is given, and for the case of the FIFO replacement algorithm a formal treatment is presented.

KEY WORDS AND PHRASES: paging machines, demand paging, replacement algorithm
CR CATEGORIES: 4.30

Introduction

In paging machines [1] all information that is explicitly addressable by the processor is subdivided into uniform units called *pages*. The execution store is similarly subdivided into page-size sections, named *page frames*.

In such machines it is possible to execute a program by supplying it but a few page frames of store. The page containing the first executable instruction is loaded into some available page frame to start the process. Execution continues until some information is not found in the store. The page containing the missing information is then fetched, overwriting some previous page in the store, and so forth. This scheme is called *demand paging*. An instance of a demand for a page which is not in the store is called a *page exception*.

In real life, page size and store size are chosen so that the store accommodates a relatively large number of pages, 32–256, or more. It is often the case, particularly in uniprogramming mode, that the program fits into the store and, beyond the initial load of needed pages, there is no overwriting induced.

In the case of large programs, or in a multiprogramming mode with space-sharing of the store, it is the usual case that the store is filled and yet another page is to be fetched from backup storage. A page frame must be chosen, and its previous content, a page belonging to some program, is usually saved before overwriting. The decision mechanism governing this assignment is normally called the *replacement algorithm*.

When programs overflow the store, the replacements thus induced increase the running time. In a multiprogramming environment without preassigned partitions, a program may run in a variable store size since, depending on the particular replacement algorithm the operating system uses, programs may exchange page frames among themselves.

It is therefore interesting to know how the running time of programs responds to changes in the space supplied. One intuitively expects that smaller space is associated with longer running time, i.e. that the running time versus space curve is monotonically decreasing. This would mean that the classical space-time trade-off in programming also applies to the behavior of paged programs.

This expectation has been confirmed by simulations and experiments on actual computers [2, 3]. In particular, extensive measurements have been performed on the M44/44X machine [4, 5]. This system has variable page size and variable store size up to 184K words. A large number of sample programs have been run several times, varying mainly the size of the store. Results have then been plotted, running-time versus space or, alternatively, number of page replacements induced versus space. The curves have exhibited the monotonically decreasing characteristic. The surprise has come about when at some local values a reversal appeared. With certain real-life programs *the running time is reduced by decreasing the space* in which the program runs. For example, it has been observed that a test program[1] running on the M44/44X using the *FIFO*[2] replacement algorithm with 48 pages of storage capacity induced 170 page fetches. With capacity increased to 52 and everything else unchanged, the number of page fetches

[1] The test program was the so-called FORTRAN test load consisting of compilations of 11 distinct programs.

[2] The rule of FIFO (First-In-First-Out) is the following: choose for replacement the page frame whose content was replaced the longest time ago.

also increased to 202. Similar behavior was observed when several copies of this program were multiprogrammed.

The Anomaly

In order to investigate this anomaly in space-time behavior, it is convenient to represent the system under investigation by a simplified structure. Let the program be represented by a *page-reference string*, in which a *symbol* in the string corresponds to the name (or address) of a page referenced. This string is interpreted as the time sequence of page addresses as generated by a particular program. The type of instruction executed, the value of data fetched or stored or the word-within-page address is relevant only as to how it influences the generation of the page reference string. Viewing the sequence of names of pages contained in the store as the *state* of a given store, a sequence of states is generated as a page-reference string is processed, symbol by symbol, under the direction of a particular replacement algorithm. For example, taking the numerals 1, 2, 3, 4, 5 as the set of symbols corresponding to page names, a possible program can then be represented by the following string:

1 2 3 4 1 2 5 1 2 3 4 5

Using the FIFO replacement algorithm and a 3 page-frame store, the following sequence of states of the store will be generated. The letter x indicates an empty page-frame and underscoring indicates the most recent reference in the string.

$\underline{1}$ x x	
1 $\underline{2}$ x	
1 2 $\underline{3}$	
$\underline{4}$ 2 3	
4 $\underline{1}$ 3	
4 1 $\underline{2}$	
$\underline{5}$ 1 2	
5 $\underline{1}$ 2	(no page exception)
5 1 $\underline{2}$	(no page exception)
5 $\underline{3}$ 2	
5 3 $\underline{4}$	
$\underline{5}$ 3 4	(no page exception)

The result is nine page exceptions. The page replacement activity induced by the above sequence of states can be emphasized by the following notation in which numbers in parenthesis indicate overwritten pages:

(1)	(2)	(3)
(4)	(1)	(2)
5	3	4

It is easily verified that a larger, 4 page-frame store will end up in the following state:

4 $\underline{5}$ 2 3

with ten page exceptions having occurred, and the *anomaly* exists.

Some General Considerations

Some insight into the anomaly can be gained by the following discussion. Consider two stores L and S, a single page-reference string and a replacement algorithm, and the concurrent processing of the string into L and S. Suppose that the string has been processed by the replacement algorithm beyond the point where both stores are full. Suppose further that the following two conditions hold:

(1) L contains at least one page which is also in S.

(2) S contains at least one page which is not in L.

If the next symbol of the reference string is in S but not in L, then a page must be brought into L but not into S. Before that input can be accomplished, a page in L must be selected by the replacement algorithm for overwriting. If the page selected is also in S, after replacement has occurred the two conditons given above still hold, and the number of pages which are in both S and L, as well as the number of pages in S but not in L, are unchanged.

If the occurrence of symbols in the reference string and the activities of the replacement algorithm cooperate so as to preserve this state of affairs for every reference, then paging occurs in L for every reference and paging never occurs in S. Notice that nothing has been said about the relative sizes of L and S. However, if L and S are the same size, then the two conditions given above cannot hold: we are in effect looking twice at one store. L and S are therefore of different sizes and we take it to be the anomalous case if L is larger than S. We note in passing that the conditions require that S contain at least two pages and L contain at least three.

We need to investigate the relationship between the conditions (1) and (2) and the anomaly. The contents of the store S can be divided into two sets, the pages which are also in L and those which are not. Since we are given that some processing of the reference string has already been accomplished, it is the case for the last symbol processed that whatever the paging invoked by that symbol, the corresponding page is now in both L and S. Suppose that the set of pages in S but not in L is empty. Since we postulate a program large enough to cause paging in L, there is some symbol in the reference string not in either L or S. Should that symbol occur, a page will leave both L and S. If L contains l pages, and S contains s pages, the probability that the same page will leave both stores is $1/l$. (This remark supposes that the replacement algorithm is not guided in some way by knowledge of the content of both stores.) If $l = 2s$ then the probability of the set of pages in S but not in L becoming nonempty is $1/2 - 1/l$. As l varies, becoming smaller or larger, the probability is less or greater.

Many measurement runs have been taken on the M44, and many opportunities existed for the anomaly to assert itself, yet it was achieved by only one program. Nevertheless, we feel that the conditions are obtained frequently and that when a program is run in a set of store sizes which differ by 10 percent or so, it frequently happens that some instruction induces a paging activity in one store size and

in a smaller store may not induce a paging activity. We assert that obtaining the conditions and inducing paging in L but not in S does not constitute the anomaly.

We say that we obtain the anomaly in two circumstances. First, given two stores L and S, with L the larger, a replacement algorithm, and the complete processing of a reference string, if there is more paging in L than in S, then the anomaly exists. Second, given a store L and a smaller store S, a replacement algorithm, and a reference string, suppose that the reference string has been processed to some point. The two stores and the replacement algorithm now have some describable states. If we now encounter in the reference string a finite substring which produces more paging in L than in S and, in addition, recreates the states initial to the substring, then the anomaly exists. For whatever the amounts of paging in L and S before encountering the substring, we can create the anomaly by repeating the substring a sufficient number of times. In this case, we can in fact cause the amount of paging in L to exceed that in S by an arbitrary amount. The ratio of paging in L to paging in S may, however, for some store sizes and replacement algorithms, approach some constant.

The present study is restricted to the FIFO algorithm. Future work might well be concerned with the anomaly with respect to other replacement algorithms having only qualitative constraints upon them, such as the following:

1. The algorithm must be the repeated application of a rule for selection which is independent of the number of times it has been invoked.

2. There can be no special symbols or special positions in a reference string or special relationships of symbols and positions which influence its behavior.

3. Its selection must be independent of the relative store sizes: it may not have information that it is being applied to the larger or smaller of two stores or, indeed, that there exists a store of any size other than the size of the store to which it is being applied.

Without such constraints, algorithms can be defined in such a way that the larger store is effectively the smaller of the two, in which case all presently nonanomalous cases will have the inverse space-time characteristics.

It is known that there exist source reference strings which produce the anomaly under AR [4] (a variant of a least recently used replacement algorithm) in those cases where AR can be made to mimic FIFO.

The Anomaly in a Special Case (FIFO Replacement Algorithm)

A production rule is a rule for generating a finite or infinite page-reference string R for processing by a replacement algorithm into L and S. The resulting reference string then displays the anomaly with L and S. We give two production rules for use with FIFO: the cluster rule and the cyclic rule. The number of pages in the large store L is denoted by l, and the number of pages in the small store S, by s. The number of distinct pages to which reference can be made is indicated by a. For convenience, these pages are named, respectively, $1, \cdots, a$.

Cluster Rule

Constraints:[3] $s < l < a < 2s$.

General form: R is the concatenation of the following six strings:

(1) $1, 2, \cdots, l$
(2) $1, 2, \cdots, (l-s+1)$
(3) $(l+1), (l+2), \cdots, a$
(4) $1, 2, \cdots, (l-s+1)$
(5) $(l-s+2), (l-s+3), \cdots, (2l-a+1)$
(6) $(l+1), (l+2), \cdots, a$

The number of page exceptions occurring in L and S for the strings (1)–(6) is indicated below:

	L	S
(1)	l	l
(2)	0	$l - s + 1$
(3)	$a - l$	$a - l$
(4)	$l - s + 1$	0
(5)	$2l - a + 1 - (l - s + 2) + 1$	$2l - a + 1 - (l - s + 2) + 1$
(6)	$a - l$	0
Total:	$l + 1 + a$	$2l + 1$

$$(l+1+a) - (2l+1) = a - l$$

Example. $s = 6$, $l = 8$, $a = 11$.

R: 1,2,3,4,5,6,7,8,/1,2,3,/9,10,11,/1,2,3,/4,5,6,/9,10,11

L: (1) (2) (3) (4) (5) (6) (7) (8)
(9) (10) (11) (1) 2 3 4 5
6 9 10 11
(number of page exceptions: 20)

S: (1) (2) (3) (4) (5) (6)
(7) (8) (1) (2) (3) 9
10 11 4 5 6
(number of page exceptions: 17)

It can be seen that after processing the first five strings, the same amount of paging ($2l+1 = 17$) has occurred in each store. String six is a cluster of ($a-l = 3$) references which causes paging only in the large store.

Cyclic Rule

Constraints:[4] $s < l < 2s - 1$, $a \geq [l+s/2]$.

General Rule: R consists of a prefix string followed by indefinitely many repetitions of a finite substring.

Prefix: $s, (s+1), \cdots, l, 1, 2, \cdots, (s-1)$.

Substring: the two strings

(1) $s, (s+1), \cdots, a, 1, 2, \cdots, (s-1)$,

and

(2) $1, 2, \cdots, a$

are used to produce the substring by selecting alternately an element from string (1) and from string (2) until the strings are exhausted, resulting in: $s, 1, (s+1), 2, \cdots, (s-1), a$

[3] The expression states that the store L must be larger than the store S but less than $2s$-1. A proof of this necessary condition follows in a later section. The cardinality of the alphabet of references must of course exceed the capacity of L, else no paging will occur in L after the initial load. The remaining constraint on a is borne out by examination of the production rule.

[4] The first constraint is identical to that relationship embedded in the constraint for the cluster rule. The remaining constraint on a is a lower bound which can be validated by inspecting the rule.

Example. $s = 6$, $l = 8$, $a = 11$.

R: Prefix: 6, 7, 8, 1, 2, 3, 4, 5

Substring:

									↓			
	(1)	6,	7,	8,	9,	10,	11,	1,	2,	3,	4,	5
	(2)	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11
L:	(6)	(7)	(8)	(1)	(2)	(3)	(4)	(5)				
	9	10	11	6	1	7	2	8				
S:	(6)	(7)	(8)	(1)	(2)	(3)						
	(4)	(5)	(6)	(7)	8	9						
	10	11	1	2								

After processing the prefix and one instance of the subsequence up to but not including the pair (3, 9) indicated by the arrow, 16 page exceptions have been produced in both L and S. From this point on, every reference will produce an exception in L, but only alternate references will cause paging in S: a page-exception ratio of 2 to 1.

Mathematical Model

In a store with a given capacity and under the FIFO replacement algorithm, an input sequence of page references gives rise to a sequence of page exceptions. A particular reference in the input stream contributes to the exception sequence in the event that the referenced page is at that time not present in the store. Given two different store sizes, a reference in the input sequence can be classified as to whether it causes paging in both stores, in neither store, in the small store only, or in the large store only. An input sequence of page references produces the anomaly if the sequence of page exceptions relative to the large store is longer than the sequence of page exceptions relative to the small store. This physical situation can be modeled by the following formal development.

Let $A = \{a_1, a_2, \cdots, a_p\}$ be a finite set and let A^* denote the set of all finite strings over A. If S_1 and S_2 are elements of A^*, then we denote by S_1S_2 that element of A^* which is the concatenation of S_1 and S_2 in that order. The string S_1 concatenated n times with the string S_2 is denoted by $S_1(S_2)^n$. We define string operators, head and tail, as follows. Let

$$I = \{1, 2, 3, \cdots\}, \qquad S = s_1s_2 \cdots s_q \in A^*.$$

The head operator is defined by

$$h: \ A^* \times I \to A^*$$

such that

$$h(s_1s_2 \cdots s_q, k) = \begin{cases} s_1s_2 \cdots s_q & \text{if } q < k, \\ s_1s_2 \cdots s_k & \text{if } q \geq k. \end{cases}$$

Similarly, the tail operator is defined by

$$t: \ A^* \times I \to A^*$$

such that

$$t(s_1s_2 \cdots s_q, k) = \begin{cases} s_1s_2 \cdots s_q & \text{if } q < k, \\ s_{q-k+1} \cdots s_q & \text{if } q \geq k. \end{cases}$$

A *reference string* is an element $R = r_1r_2 \cdots r_q$ of A^* and a *reference* of R is any element r_j $(j = 1, 2, \cdots, q)$ of R. We denote the relation "r_j is a reference of R" by $r_j \prec R$ and denote the number of references of a string R by $\#R$. The null string is denoted by Λ.

Given a positive integer k, we associate with a reference string R an element of A^* called the *exception string*. This association is defined by the function f as follows:

$$f: \ A^* \times I \to A^*$$

such that for all k in I, $f(\Lambda, k) = \Lambda$.

$$f(Ra_j, k) = \begin{cases} f(R, k) & \text{if } a_j \prec t(f(R, k), k), \\ f(R, k)a_j & \text{if } a_j \not\prec t(f(R, k), k). \end{cases}$$

For any $R = r_1r_2 \cdots r_q \in A^*$, we define

$$R_0 = \Lambda, \qquad R_k = R_{k-1}r_k, \quad 1 \leq k \leq q.$$

Now let two positive integers s and l be given with $s < l$. For any reference string $R \in A^*$, we denote $f(R, s)$ by E, $f(R_k, s)$ by E_k, $f(R, l)$ by F, and $f(R_k, l)$ by F_k.

The references r_j of R are classified into four types as follows. The reference r_{k+1} $(0 \leq k \leq q-1)$ is a

(i) *type I reference* if $r_{k+1} \prec t(E_k, s)$ and $r_{k+1} \prec t(F_k, l)$,

(ii) *type II reference* if $r_{k+1} \not\prec t(E_k, s)$ and $r_{k+1} \not\prec t(F_k, l)$,

(iii) *type III reference* if $r_{k+1} \not\prec t(E_k, s)$ and $r_{k+1} \prec t(F_k, l)$,

(iv) *type IV reference* if $r_{k+1} \prec t(E_k, s)$ and $r_{k+1} \not\prec t(F_k, l)$.

The reference r_{k+1} $(0 \leq k \leq q-1)$ is said to be a *contributory reference* of R if r_{k+1} is a type IV reference and $\#E_k \leq \#F_k$. For $1 \leq k \leq q$, we define $d_k = \#F_k - \#E_k$. Now let $s < l$ and A be given. A reference string $R = r_1r_2 \cdots r_q \in A^*$ is said to *produce the anomaly* if $d_q > 0$.

A condition on the storage sizes in which the anomaly can be produced is given by the following theorem.

THEOREM 1. *Let $s < l$ be positive integers and let A be a finite set of cardinality at least $l + 1$. There exists a string $R \in A^*$ which produces the anomaly if and only if $l < 2s - 1$.*

The following lemma is useful in the proof.

LEMMA. *Let $s < l$ be positive integers, A be a finite set and let $R \in A^*$ be given. Let $r_k = a_j \in A$ be a type IV reference and let r_i be the previous occurrence of a_j in R. If r_i is a type II reference or a type IV reference, then $l < 2s - 1$.*

PROOF. Suppose that r_i is a type II reference, and consider the strings E_i and F_i. Since r_i is a type II reference, r_i is the last element in E_i and F_i. Since r_k is a type IV reference, there must have been at least $d = l - s + 1$ type IV references to distinct symbols among $r_{i+1}r_{i+2} \cdots r_{k-1}$. Assume for purposes of contradiction that $2s - 1 \leq l$. Then it follows that $d \geq s$. Each of the d symbols is an element of $t(E_k, s)$, as is r_k. Hence, $t(E_k, s)$ contains at least $s + 1$ elements, a contradiction. The proof for the case in which r_i is a type IV reference is similar.

PROOF OF THEOREM 1. Suppose that R produces the anomaly. Without loss of generality, assume that R contains no type I references. Let $r_k = a_j \in A$ be the first contributory reference of R and let r_i be the previous

occurrence of a_j in R. Since r_k is a type IV reference, we have $l < 2s - 1$ by the lemma, if r_i is either a type IV or type II reference. Assume therefore that r_i is a type III reference. Since r_k is the first contributory reference in R, $\#E_{k-1} = \#F_{k-1}$. This equality implies that among $r_1r_2, \cdots, r_{k-1}$ the number of type III references is equal to the number of type IV references. Since r_i is a type III reference, there must exist a type IV reference $r_j = a_m$ such that the previous occurrence r_n of a_m is a type II reference. Applying the lemma to r_j and r_n we conclude that $l < 2s - 1$.

To prove the converse, suppose that $s < l < 2s - 1$, and without loss of generality, suppose that

$$A = \{1, 2, \cdots, l + 1\}.$$

It is easily verified that the string R produces the anomaly, where

$$R = 1\ 2 \cdots l\ 1\ 2 \cdots (l-s+1)(l+1)\ 1\ 2 \cdots (l-s+1)\ (l-s-2) \cdots l(l-1)$$

This establishes the theorem.

The following theorem provides information about the relative rates of paging in the memories while processing a string which produces the anomaly.

THEOREM 2. *Let s and l be positive integers such that $s < l < 2s - 1$ and let A be a finite set of cardinality p at least $[l + s/2]$. For q sufficiently large there exists a reference string $R = r_1r_2 \cdots r_q \in A^*$ such that $\#E_q/\#F_q$ is arbitrarily close to $\frac{1}{2}$.*

PROOF. Without loss of generality, we can let $A = \{1, 2, \cdots, p\}$ and consider the string $R = R_1(R_2)^n$ where

$$R_1 = s(s+1) \cdots l\ 1\ 2 \cdots s - 1,$$
$$R_2 = s\ 1\ (s+1)\ 2\ (s+2)\ 3 \cdots p\ (p-s+1)\ 1\ (p-s+2) \cdots (s-1)\ p,$$

and n is a positive integer. It is easily verified that beyond a certain reference in R, each of the references of R occurs as a type IV reference followed by a type II reference. It follows that by taking n sufficiently large, $\#E_q/\#F_q$ can be made arbitrarily close to $\frac{1}{2}$.

The example given below serves to illustrate the construction given in the proof of Theorem 2. Let $s = 4$, $l = 6$, and $p = 8$. Then

$$R_1 = 4\ 5\ 6\ 1\ 2\ 3,$$
$$R_2 = 4\ 1\ 5\ 2\ 6\ 3\ 7\ 4\ 8\ 5\ 1\ 6\ 2\ 7\ 3\ 8.$$

The memory states after the first eighteen references of R are:

(4)	(5)	(6)	(1)	(4)	(5)	(6)	(1)	(2)	(3)
(2)	(3)	(4)	(5)	7	4	8	5	1	6
6	**7**	**8**	**1**						

Each succeeding reference of $R = R_1(R_2)^n$ causes paging in the large memory, but only every other reference causes paging in the small memory.

It is conjectured that if $s < l$ are positive integers and A is a finite set having cardinality such that the anomaly can exist, then for all strings $R = r_1r_2 \cdots r_q \in A^*$, we have $\#E_q/\#F_q \geq \frac{1}{2}$.

Summary

Initiated by the observation of an unexpected phenomenon, a program behavior study has been presented. The study results in showing the existence of programs having the following property: running on a paging machine and using the FIFO replacement algorithm, there are instances when the program runs faster if one reduces the storage space allotted to it.

The study describes methods of constructing programs (i.e. storage-reference strings) which, for two storage sizes and running under the FIFO replacement algorithm, display a (reversed) page-exception ratio of up to 2 to 1.

Though the programs thus constructed may seem too regular and artificial, the fact that a real-life program has been observed behaving similarly, and that the anomaly has also been observed with a multiprogrammed mix of that program, suggests the possibility of FIFO producing the anomaly in subsequences of reference strings.

It should be borne in mind that the strings R, produced by the production rules, can be enlarged in a great variety of ways with references which do not cause paging in either the large or small stores. Essentially dissimilar programs, after deletion of such type I references, may be similar generators of the anomaly.

The anomaly is an indication of inefficiency of the FIFO algorithm, which cyclically assigns page-frames upon demand and ignores any other pattern in the reference string.

It is felt that the anomaly can be eliminated or its occurrence made very infrequent by using other replacement rules which keep track of recent references to pages already in store (see AR-1 in [2]). Some random selection superimposed on FIFO might avoid the anomaly.

The present study is restricted to cases with FIFO and to uniprogramming. A possible and indeed desirable extension of it could be the investigation of the anomaly with some other replacement algorithms.

RECEIVED SEPTEMBER, 1968; REVISED DECEMBER, 1968

REFERENCES

1. RANDELL, B., AND KUEHNER, C. J. Dynamic storage allocation systems. *Comm. ACM 11*, 5 (May, 1968), 297–305.
2. BELADY, L. A. A study of replacement algorithms for a virtual storage computer. *IBM Syst. J. 5*, 2 (1966), 78–101.
3. FINE, G. H., JACKSON, C. W., AND MCISAAC, P. V. Dynamic program behavior under paging. Proc. 1966 ACM Nat. Conf., Thompson Book Co., Washington, D. C., pp. 223–228.
4. O'NEILL, R. W. Experience using a time-sharing multiprogramming system with dynamic address relocation hardware. Proc. AFIPS 1967 Spring Joint Comput. Conf., Vol. 30, Thompson Book Co., Washington, D. C., pp. 611–621.
5. BRAWN, B., AND GUSTAVSON, F. An evaluation of program performance on the M44/44X System, Pt. I. IBM Res. Rep., RC-2083, IBM Corp., 1968. Also as Program behavior in a paging environment, Proc. AFIPS 1968 Fall Joint Comput. Conf., Vol. 33, Pt. 2, Thompson Book Co., Washington, D. C., pp. 1019–1032.