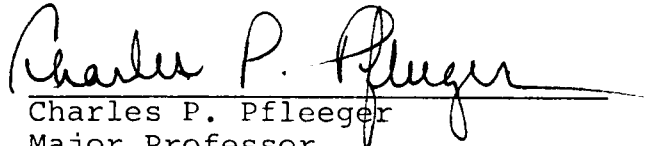
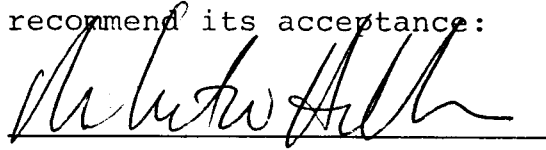
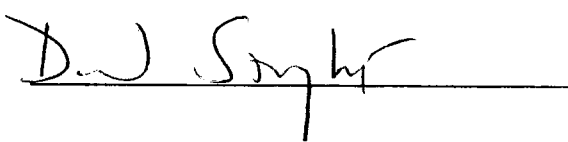


To the Graduate Council:

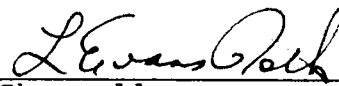
I am submitting herewith a thesis written by James W. Teague II entitled "An Evaluation of Three Globally and Locally Implemented Page Replacement Algorithms Using a Demand Paged Virtual Memory Simulator." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.


Charles P. Pfleeger
Major Professor

We have read this thesis and
recommend its acceptance:

Accepted for the Council:


Vice Chancellor
Graduate Studies and Research

AN EVALUATION OF THREE GLOBALLY AND LOCALLY IMPLEMENTED
PAGE REPLACEMENT ALGORITHMS USING A DEMAND
PAGED VIRTUAL MEMORY SIMULATOR

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

James W. Teague II
August 1981

3062946

To Jim and Aline;
the Meades and Websters;
Don;
SDM;
and Nola

ABSTRACT

This paper presents a survey of some of the more practical methods of page replacement in a demand-paged virtual memory system, and an evaluation of three of these methods is given that was performed using a reference-driven simulator program. A discussion of several page replacement algorithms is given with particular emphasis on a least recently used (LRU) approximation algorithm, a random (RAND) page replacement scheme and a round robin method, also referred to as first in, first out (FIFO). Each of these page replacement algorithms was evaluated in a global application ("paged against the system") and a local implementation ("paged against the process"). Nine sets of test data were used to gather page reference success statistics on each implementation of each page replacement algorithm. These test data made considerations for such factors as performance characteristics of processor-intensive tasks versus those of peripheral-intensive tasks, the interactions between such types of tasks, and physical memory capacity. Results are presented in tabular and graphical form, and related to findings in earlier literature.

TABLE OF CONTENTS

CHAPTER	PAGE
1. INTRODUCTION	1
2. BACKGROUND	6
Terminology and Fundamentals	6
Program Behavior Characteristics	10
Page Replacement Algorithms	17
3. METHODOLOGY	24
Modeling Considerations	24
Test Data	27
4. RESULTS	31
General Discussion	31
RAND Algorithm	32
FIFO Algorithm	35
LRU Approximation Algorithm	44
5. CONCLUSIONS	51
Interalgorithm Comparisons	51
Suggestions for Future Research	59
LIST OF REFERENCES	60
APPENDICES	65
A. TABULAR RESULTS FOR SIMULATIONS	66
B. FORMAT OF SIMULATOR INPUT	69
VITA	72

LIST OF FIGURES

FIGURE	PAGE
2.1. Virtual Address to Physical Address Translation	7
2.2. Parachor Curve	9
2.3. A Reference String and Its LRU Stack Hierarchies	14
2.4. Activity Set Transitions	16
4.1. RAND with Local Implementation	33
4.2. RAND with Global Implementation	34
4.3. RAND with 30 Page Frames	36
4.4. RAND with 50 Page Frames	37
4.5. RAND with 70 Page Frames	38
4.6. FIFO with Local Implementation	39
4.7. FIFO with Global Implementation	40
4.8. FIFO with 30 Page Frames	41
4.9. FIFO with 50 Page Frames	42
4.10. FIFO with 70 Page Frames	43
4.11. LRU with Local Implementation	45
4.12. LRU with Global Implementation	46
4.13. LRU with 30 Page Frames	48
4.14. LRU with 50 Page Frames	49
4.15. LRU with 70 Page Frames	50
5.1. Local Implementations with 30 Page Frames . . .	52
5.2. Local Implementations with 50 Page Frames . . .	53

FIGURE	PAGE
5.3. Local Implementations with 70 Page Frames . . .	54
5.4. Global Implementations with 30 Page Frames . .	56
5.5. Global Implementations with 50 Page Frames . .	57
5.6. Global Implementations with 70 Page Frames . .	58

CHAPTER 1

INTRODUCTION

The concept of organizing computer memories into hierarchial levels in order to utilize the central processor (CPU) and memory more efficiently has received a great deal of attention in the last two decades. A strong motivation to structure memory in this manner arose because of the relationship of storage access times to the cost per unit of memory for various devices: as memory access time decreases, the cost of the device increases. Thus, it seemed intuitively reasonable to keep those more heavily used portions of a program in some amount of faster storage, and the less active parts in a less expensive but larger amount of slower memory.

This process was originally the responsibility of the programmer, who had to divide programs that were too large to be fitted into main memory into functionally related portions called "overlay segments." When the segment occupying main memory finished executing, a new segment was "overlayed" into the space used by the terminating segment. Devising an overlay structure for large, complex programs was often very time-consuming, however, and designating the process to the computer was a logical next step.

Parmelee et al. [1] defined virtual memory as a storage system in which there may be "a distinction between the logical [or virtual] address generated by a program and the physical [or real] address for some real storage device from which the information is actually fetched." There are two major approaches to virtual storage management, paging and segmentation. Paging, which is the type treated in this study, is the method in which main memory is divided into equal-sized blocks which are called "page frames." Programs are also divided into pages that are of the same size as the page frames in memory. Hence any program page will fit into any page frame.

The other type of virtual storage management, segmentation, is similar to paging except that the blocks into which the program is divided are not necessarily the same size. Portions of a program are grouped logically by function, rather than according to some fixed-size unit. Denning [2] has found paging to be superior to segmentation in most cases.

The first computer to use the concept of automatic transfer of data between main memory and a backing store was the ATLAS computer discussed by Fotheringham [3], which was a joint design effort between the Computer Departments of Manchester University and Ferranti Limited. ATLAS used core storage as main memory and drum storage as secondary memory, or backing store.

The viability of virtual memory helped it gain increased popularity in the mid-1960's, and the principles of virtual memory were applied to the design of several commercially available computers at that time. Virtual memory achieved widespread acceptance in spite of people such as Kuehner and Randall [4], who warned that the good performance in virtual storage systems which had been implemented at that time was obtained by the purchase of "rather more" main memory and/or secondary storage than was originally thought necessary.

Virtual memory today is a heavily used means of memory management which is incorporated into the operating systems of some of the most popular computers available. Such computer manufacturers as International Business Machines and Digital Equipment Corporation have found that their virtual storage computers have served the majority of their users quite well.

Each user executing on a virtual storage machine is given the illusion of having all addressable memory on that computer available for use by his/her program. This amount of "virtual" space is limited only by the size of the address field, and the amount of virtual space is almost always significantly larger than the available physical memory.

In a "demand-paged" virtual memory system, programs are divided into pages which are written to backing store before

execution. As a program becomes eligible to begin execution, the page containing the entry point of the program is transferred from secondary storage to main memory. Thereafter, as portions of a program which are resident in main memory reference parts which do not reside in main memory, the pages containing those nonresident portions are brought into primary storage. This process continues in a straightforward manner as long as main memory contains empty page frames. When main memory becomes full and a program references a nonresident page, however, some page frame must be written over to accommodate the new page which is to be read from secondary storage. This process is called "page replacement," and the means by which the operating system decides which page to replace is called a "page replacement algorithm."

This study describes three of the more practical page replacement algorithms that have been formulated, and two methods of implementation for each. These six techniques were evaluated using a program that simulated the paging behavior of a multiprogrammed virtual memory computer.

Chapter II presents background information and terminology that is essential to understand the mechanics of virtual memory. The chapter also discusses the subject of program behavior characteristics and presents a survey of page replacement algorithms.

Chapter III relates the means by which the page replacement algorithms selected for evaluation were tested.

The simulation program that was used is explained in detail, as well as the test data that was employed to gauge the performance of the page replacement algorithms.

Chapter IV discusses the specific implementation details of the three page replacement algorithms. The results of each evaluation are presented, and observations are made.

Chapter V presents conclusions of the study, and some overall comparisons of the page replacement algorithms are made. Finally, recommendations on future research are made.

CHAPTER 2

BACKGROUND

1. TERMINOLOGY AND FUNDAMENTALS

Demand paging, the method of paging with which this study is concerned, is characterized by the fact that a page is "demanded" by a program at the moment it is needed, without advance warning. The demand is implicit, arising out of an attempt to access a page which is not resident in main memory. This implies that not all pages of a program need to be in main memory simultaneously.

The address translation mechanism which converts a virtual address into a real address is a hardware device which works along with tables maintained by the operating system. If a machine has an address field of n bits, and pages which are of size 2^m bytes (where $m < n$), then the leftmost $(n-m)$ bits of the virtual address are treated as the virtual page number, and the rightmost m bits are treated as a byte offset, or displacement, within the page. Address translation is then performed by using the virtual page number as an offset into a "page table." The page table entry for the virtual page may supply a real page number (which, when combined with the byte offset, yields the desired real address), or some type of flag which indicates

that the page is not resident in main memory. This process is illustrated by Figure 2.1.

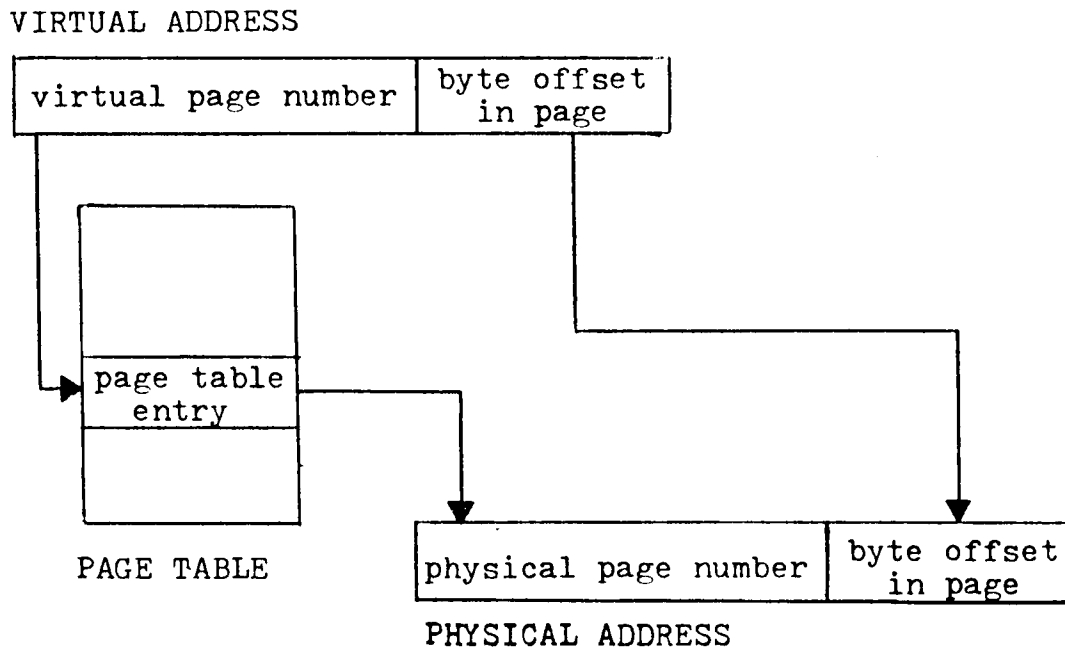


Figure 2.1. Virtual address to physical address translation.

If the page which contains the desired address is not resident in main memory, a page fault or page exception is said to occur. The memory management function of the operating system must then remedy this situation. This involves:

1. Determining where the missing page will reside when read from secondary store (this is the function of the page replacement algorithm);

2. Writing the page which is to be replaced to secondary storage if it has been modified since last read in;
3. Transferring the faulted page to the selected page frame;
4. Updating the page table entries of the newly resident page and the replaced page.

Virtual memory system performance is sensitive to several parameters, one of which is the amount of physical memory available. The effect of increasing or decreasing real memory size on page fault rates can be illustrated by the parachor curve in Figure 2.2. Note that a program's performance falls off rapidly when the set of real pages is reduced below a certain point, and that performance improves more slowly as the physical memory available approaches the amount of virtual space used. Optimal system performance for multiprogramming systems is obtained when the performance of each concurrently executing task lies somewhere in this middle range, since high paging rates require excessive attention by the operating system for input/output (I/O) operations, and low paging rates indicate that the program has more real memory than it needs and could afford to yield some of this memory to programs with a higher paging rate. Denning [5] used the term "thrashing" to describe the situation where a program's high

paging rate adversely affects its performance. The problem created by thrashing is that of excess idle processor time, since programs end up spending too much time waiting for completion of paging I/O.

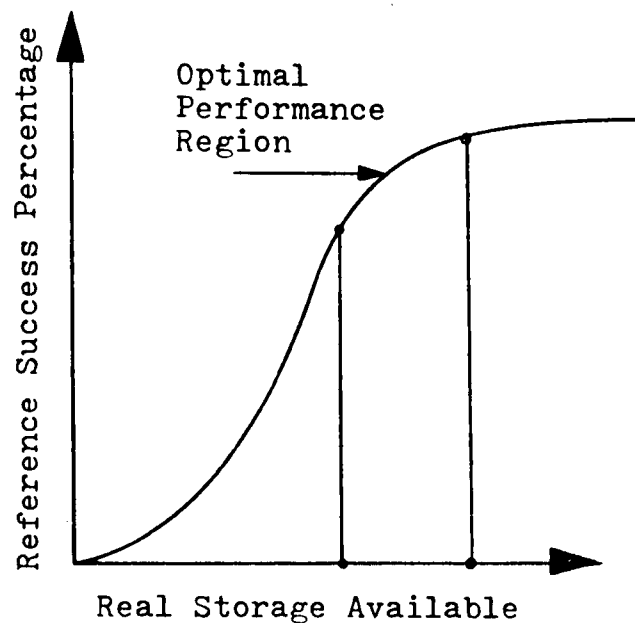


Figure 2.2. Parachor curve.

Another important parameter affecting the performance of virtual memory systems is the size of the pages used. In his early experiments Belady [6] found that small page sizes gave better results in terms of page replacement efficiency. Coffman and Varian [7] and Joseph [8] also

discovered that smaller page sizes tended to enhance system performance. Joseph obtained one set of results which showed that five 128-byte pages brought the page fault rate to a lower level than two 4096-byte pages, and that only after the number of 4096-byte pages exceeded five did the page fault rate improve markedly over that of the smaller page sizes. This behavior can be intuitively explained by the fact that with smaller page sizes a program can have a more "finely adjusted" description of the addresses it references.

Hatfield [9] and Hatfield and Gerald [10] found that achieving good system performance was not as simple as using small page sizes, however. Both articles revealed that the more thoroughly a program references its pages in memory, the poorer the performance obtained by using smaller page sizes. If a program referenced 4096-byte pages heavily, the corresponding number of 2048-byte pages would be referenced identically, but twice as many page faults would be necessary to read the smaller pages into main memory. This would result in twice as much wait time for the program.

2. PROGRAM BEHAVIOR CHARACTERISTICS

The most important factor which influences the performance of a virtual memory system according to Hatfield and Gerald [10] is that of a program's address reference

pattern. To model a program's address pattern, Denning [11], Ferrari [12], and Slutz and Traiger [13] all used the notion of reference strings. A reference string $R = r_1, r_2, \dots, r_k, \dots$ is a sequence of page numbers where r_k is the number of the page containing the k -th address referenced during the execution of a program. Note that these page numbers are not necessarily distinct.

The working set model of program behavior was presented by Denning [5, 14] in 1968 in an attempt to formalize some of the terms and notations used when discussing virtual memory principles. The working set $W(t, T)$ associated with a program at time t is the set of distinct pages referenced by the program during the time interval $(t-T, t)$. T is known as the interval, or window size. In terms of reference strings,

$$W(t, 1) = \{r_t\} ;$$

$$W(t, T) = W(t-1, T-1) \cup W(t, 1) .$$

For example, for $t = 5$ and $T = 3$

$$\begin{aligned} W(5, 3) &= W(4, 2) \cup W(5, 1) \\ &= (W(3, 1) \cup W(4, 1)) \cup W(5, 1) \\ &= (\{r_3\} \cup \{r_4\}) \cup \{r_5\} \\ &= \{r_3, r_4, r_5\} \end{aligned}$$

where r_3 , r_4 , and r_5 are the (possibly) distinct pages referenced at times 3, 4, and 5, respectively.

The working set size $w(t,T)$ is the number of pages in $W(t,T)$. Intuitively, working sets were intended to represent the smallest set of pages that ought to remain in main storage so that a process might operate efficiently, or without thrashing.

A property of program behavior which can be discussed more easily in terms of working sets is the property of "locality of reference." Locality of reference is the tendency that programs have to use the same set of pages for relatively long periods of execution. It is upon this property that virtual memory is founded and continues to be feasible. A program which has "good" locality of reference is one whose references are confined to a small set of pages (which need not be contiguous) for some reasonably long amount of virtual time, or an interval T which is substantially large. An excellent discussion of locality of reference is given in Brawn and Gustavson [15].

Another property of program behavior related to locality of reference is "density of reference" as cited by Morrison [16]. Morrison stated that a page of a program is densely referenced if most of its address space is used when the page is in main memory. Recall the discussion earlier on the effect of page size on virtual memory performance: programs with a high density of reference perform better with larger page sizes, and those programs with less density of reference perform better with smaller page sizes.

Strong empirical evidence has been presented by Knuth [17] and Williams [18] to show a surprising degree of locality of reference in the programs these studies examined. Knuth found that "less than four percent of a program generally accounts for more than half of its running time." Williams reported that "the 50 percent of the pages which are most active account for about 95 percent of all the activity." An examination of the graphical data presented by Williams showed that roughly 20 percent of a program's pages accounted for approximately 80 percent of the activity.

Madison and Batson [19] found the concept of working sets with fixed window sizes inadequate for describing program locality in many cases because there was no means for partitioning a given reference string into a unique sequence of localities. If the window size is one, then each reference constitutes a locality, or working set. On the other hand, if the window size is equal to the entire period of execution, then the set of all pages referenced would constitute a working set. Madison and Batson refined the idea of a least recently used (LRU) stack model used by Spirn and Denning [20]. The LRU stack can be represented by the ordered vector $L(t) = (L_1(t), L_2(t), \dots, L_n(t))$ where n is the number of pages in the program, and $L_i(t)$ is the page identifier (or page number) of the i -th most recently referenced page at time t . If the LRU stack is cut at any

position i , then the topmost i page numbers in the stack $S_i(t) = \{L_1(t), L_2(t), \dots, L_i(t)\}$ are the i most recently referenced pages at time t . These sets $S_1(t), S_2(t), \dots, S_n(t)$ are known as the LRU stack hierarchy. Figure 2.3 shows a reference string and its LRU stack hierarchies.

$$R = 5, 4, 6, 1, 2, 3, 2, 4, 2, 4, 4, 4$$

$$(t = 12)$$

1	4	$S_1(12) = \{4\}$
2	2	$S_2(12) = \{2, 4\}$
3	3	$S_3(12) = \{2, 3, 4\}$
4	1	$S_4(12) = \{1, 2, 3, 4\}$
5	6	$S_5(12) = \{1, 2, 3, 4, 6\}$
6	5	$S_6(12) = \{1, 2, 3, 4, 5, 6\}$

Figure 2.3. A reference string and its LRU stack hierarchies.

Madison and Batson [19] noted that one shortcoming in the LRU stack model was that the stack did not contain enough information to select certain locality sets $S_i(t)$ as being more distinctive than others. For example, the page

at $S_1(t)$ may have been on top of the stack for only a single reference, or for the last 100 references. Madison and Batson finally developed a better method of identifying localities called an "activity set." An activity set at time t was defined as any set of pages in the LRU hierarchy in which every member has been rereferenced since the set has been formed. Further, a "bounded locality interval" is the 2-tuple consisting of an activity set membership and its lifetime. The notation for bounded locality intervals is (A_i, T_i) where A_i is the i -th activity set during a program's execution, and T_i is the lifetime of A_i .

Equipped with the idea of bounded locality intervals, Madison and Batson [19], Freiburger et al. [21] and Courtois and Vantilborgh [22] found strong additional support for the principle of locality, but the most significant new discovery about program behavior was that programs did not gradually drift between localities. Significant discontinuities in the referencing behavior of programs were found between stable phases of execution. This behavior can be illustrated by Figure 2.4.

Probably the most significant factor that contributes to the locality of reference principle is the design and structuring process of a program. Like many complex systems, a program is usually conceived as a set of interrelated procedures, with each procedure responsible for some specific

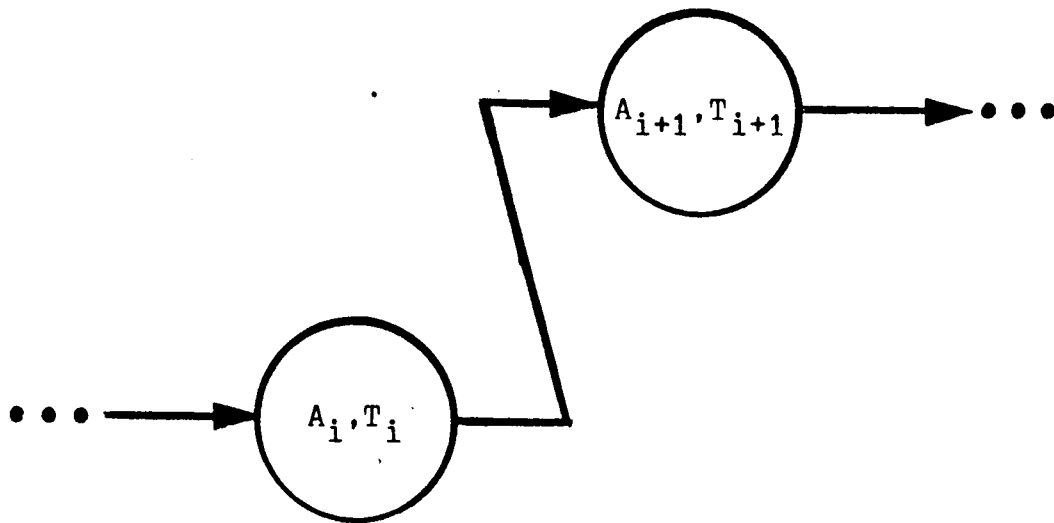


Figure 2.4. Activity set transitions.

function. If the function is complex enough, then the procedure is subdivided into smaller subprocedures until some lowest level of procedures exists. This common programming structure is called top-down design with stepwise refinement. The effect of this type of organization is that the proportion of references within a procedure will be higher than between procedures. The set of data used by a procedure corresponds to a locality or sublocality of the entire program.

As a result, one might intuitively expect programming "style" to have an effect on program behavior in a virtual

memory environment. This has been shown to be true by Brawn and Gustavson [15] and by Innes [23]. Significant improvements in program performance can be obtained by using techniques such as those outlined by Morrison [16], and these usually do not involve radical changes to program structure.

3. PAGE REPLACEMENT ALGORITHMS

The page replacement algorithm for a given virtual memory system is the method by which a page is selected to be replaced when main memory is full and a program has faulted a page. If the selected page has been changed somehow since it was last read from secondary storage, it must be written back to backing storage before the main memory page frame can be overwritten. If an up to date copy exists on the secondary storage medium, then the selected page frame can be overwritten immediately. A formal definition of page replacement algorithms is given later in this paper.

The goal of a page replacement algorithm is to reduce program page faults by selecting the page to be replaced as the one which will remain unreferenced for the longest period. This, in its strictest sense, is impossible since the algorithm would need to possess some capability to reliably predict a program's future behavior. This type of future behavior data is rarely known to a strong enough degree to permit construction of a predictive page replacement algorithm.

Still, Belady [6] and Prieve and Fabry [24] have discussed such unrealizable algorithms as bases for comparison for the more conventional page replacement algorithms.

Belady [6] categorized page replacement algorithms into three classes:

Class 1—Class 1 algorithms assume that all pages are equally likely to be referenced at any time.

No data is kept regarding past uses of memory.

Class 2—Class 2 algorithms use data collected about previous page usage patterns, assuming that future program behavior will be identical.

Class 3—Class 3 algorithms use data collected about previous page references and also about those pages that have not been referenced. Data is therefore available on every page of a program.

There are two class 1 algorithms evaluated in this study, a random (RAND) page replacement scheme and a first in, first out (FIFO) replacement method. RAND is a fairly simple, straightforward algorithm that selects a page to be replaced from all pages available to the executing program at random.

The FIFO page replacement algorithm cycles a page pointer through physical page frames which indicates the page to be selected for replacement. As a page is replaced, the pointer is incremented to the next longest resident page. In this

manner, the page selected to be replaced is the page which has occupied a frame in main memory for the longest period. Both RAND and FIFO give fair performance in most cases, and are very simple to implement.

The class 2 page replacement algorithms are more difficult to implement, but usually will yield better performance than class 1 algorithms. By far the most popular class 2 algorithm is the least recently used (LRU) approximation page replacement algorithm. LRU approximation assumes that pages can be divided roughly into two categories: those with high frequency usage and those with a lower frequency of use. A bit is associated with each physical page frame, and when that page is referenced this "reference" bit is set to 1. When pages are selected for replacement, those pages whose reference bit is 0 are selected. Periodically, reference bits must be reset to 0's according to some scheme.

Another class 2 algorithm is the Denning working set (DWS) algorithm as discussed by Denning [14]. The DWS scheme is actually very close to the LRU method—the main difference is that the DWS uses some type of timer for each page frame where the timer is set at each reference. Pages selected for replacement are those having expired timers. Such timer schemes are impractical for the reasons cited by Madison and Batson [19] regarding the inadequacy of a fixed window size, or in this context, a fixed timer delay period. If the delay

period was too short, all of a program's page timers would be expired if the processor became devoted to other programs for long periods of time, and DWS performance might degrade to that of RAND. A software implementation was also proposed, but due to the inherent slowness of programmed solutions as compared to those in hardware, this solution was considered even more impractical.

Other class 2 algorithms are those such as the page fault frequency (PFF) scheme as discussed by Gupta and Franklin [25] and Chu and Opderbeck [26]. The idea behind the PFF algorithm is to bring in an entire set of faulted pages when the page fault frequency rises above some critical level P . This results in an increase in the number of allotted page frames and usually reduces the page fault frequency. If, on the other hand, the page fault frequency falls below P , those page frames which have not been referenced since the last page fault occurred are freed. The PFF algorithm implies a kind of implementation called "local" implementation which will be discussed later.

The page survival index (PSI) as discussed by Bard [27, 28] is a predictive parameter intended to be used in conjunction with some page replacement algorithm. The page survival index relates the number of page faults that an unreferenced page survives before being replaced. This value can be used to indicate when the number of page frames allocated to a program should be increased or decreased.

The page replacement algorithm used by the ATLAS computer mentioned by Fotheringham [3] was the only class 3 algorithm encountered in the research of this study. The ATLAS algorithm was termed the drum transfer learning program, and it kept activity data on every virtual page of a program.

The idea behind the selection of a page replacement algorithm is to choose one as simple as possible to keep operating system overhead down, and at the same time to use an algorithm which is sophisticated enough to provide good performance in most cases. Most of the time, however, even the simplest algorithms will yield acceptable results, so many computer designers use either a FIFO or LRU approximation scheme.

For each page replacement algorithm there are two implementations available, as discussed in Levy and Eckhouse [29]. The first implementation is termed "global" page replacement, or "paging against the system." With global page replacement, all concurrently executing programs compete for the same set of physical page frames. This permits one program to "steal" pages from another, and to acquire page frames until its execution is interrupted. Thus the number of page frames assigned to a particular user will not necessarily be the same as for any other user, and the number of page frames for a user changes over time.

The second type of implementation, which was mentioned in the discussion of the PFF algorithm, is that of local

page replacement, or "paging against the process." Under local page replacement a faulting program must have one of its own pages replaced in order to accommodate the faulted page. More generally, a program is given a limit to the number of page frames it may occupy in main memory, and it may not steal page frames from another process.

This study considers local and global implementations of the RAND, FIFO, and LRU approximation, avoiding the PFF algorithm, DWS algorithm, and the PSI parameter. PFF and DWS were intended for use as local algorithms only, and involve dynamic activity set sizes, a feature not available on the simulator used for this study. Bryant [30] has found it difficult to predict the working set sizes for the DWS scheme, making it an even less practical scheme. Belady et al. [31] and Franklin and Gupta [32] have reported anomalies among RAND, FIFO and LRU, but these algorithms give satisfactory results in most instances, and the latter two are commonly implemented on commercially available machines.

Aho et al. [33] gave a precise formal description of the FIFO and LRU page replacement algorithms. Suppose that the set $N = \{1, 2, \dots, n\}$ is the set of pages of a given n -page program, $M = \{1, 2, \dots, m\}$ is the set of page frames of memory, and that $1 \leq m \leq n$. The reference string model of program addressing is again used where $r_t = x$ means that the program references page x at time t , or the t -th reference. (This

description is actually for a local scheme, but can be readily generalized to a global scheme.)

S is called a memory state where $S \subseteq N$, representing the subset of pages actually in memory. Z represents the set $\{S | S \subseteq N \text{ and } \#S \leq m\}$ where $\#S$ denotes the number of elements in set S . Let M and N be given. A page replacement algorithm for M and N is a system $A = (Q, q_0, g)$ where

1. Q is a set of control states for the algorithm,
2. q_0 in Q is the initial control state,
3. $g: Z \times Q \times N \rightarrow Z \times Q$ is the allocation mapping.

Let x denote the page to be placed in memory, and the sequence $Y = (y_1, y_2, \dots, y_m)$ as the pages to be removed from memory when $\#S = m$. The allocation functions for FIFO and LRU are

$$g_{\text{FIFO}}(S, q, x) = \begin{cases} (S, q) & \text{if } x \in S \\ (S + x - y_m, q') & \text{if } x \notin S \end{cases}$$

$$g_{\text{LRU}}(S, q, x) = \begin{cases} (S, q'') & \text{if } x \in S \\ (S + x - y_m, q') & \text{if } x \notin S \end{cases}$$

where $q' = (x, y_1, \dots, y_{m-1})$,

$q'' = (x, y_1, \dots, y_{k-1}, y_{k+1}, \dots, y_m)$ where $x = y_k$,

$S + x$ denotes the set $S \cup \{x\}$,

$S - x$ denotes the set $S - \{x\}$.

CHAPTER 3

METHODOLOGY

1. MODELING CONSIDERATIONS

The simulator that was programmed for this study was designed to model a multiprogramming virtual memory system. Important items considered in this design were speed of execution and ease of use, but not at the expense of being detailed enough to be realistic. The simulator concentrated on modeling variables such as job mixture, physical memory size, page replacement algorithms and their implementation, instead of highly technical factors such as page transfer delays discussed by Brandwajn [34]. Chamberlin et al. [35], Bard [36, 37] and Anderson et al. [38] all made considerations for the types of jobs executing, although these studies were not specifically concerned with page replacement algorithms and fault rates.

The available physical memory for the simulated system was alterable by the user by means of input data. Simulated processor quanta were dispatched in a round robin fashion, with the length of the simulation being specified in the number of complete processor allocation cycles. Other system parameters were the page replacement algorithm to be used, whether the algorithm was to be implemented globally or locally, and how many jobs were in the simulation.

Each simulated job was allowed to have a different number of virtual pages. A job class parameter was also supplied for each job which designated how relatively I/O-bound or processor-bound the simulated program was to be. This job class parameter was specified as an integer between one and ten, inclusive, with one representing the most I/O-bound and ten being the most processor-intensive. This class number was simply multiplied by ten, and the result designated the number of simulated page references the job made each time it was dispatched processor time.

Each simulated job could describe up to twelve activity sets, with each having a different page reference frequency pattern. These activity sets could be manufactured to correspond to the phases of program execution, such as an input phase, a processing phase, and an output phase. A duration in terms of processor allocation cycles was specified for each activity set. Hence, the user of the simulator was able to define disjoint bounded locality intervals as supported by Madison and Batson [19], Freiburger et al. [21] and Courtois and Vantilborgh [22].

Page reference frequency patterns for each activity set were specified as percentages that were converted to range values to be associated with each page. For example, if five pages were specified with reference frequencies of 10, 20, 40, 25, and 5 percent, the range values would be

.10, .30, .70, .95 and 1.00. A pseudo random number generator URAND, as described by Forsythe et al. [39], was used to produce a random value between zero and one. By searching the range values, a page reference was simulated to the first page whose range value was greater than or equal to the random number. In this manner the user could describe any kind of page reference frequency pattern desired. The reference frequency percentage total was required to be within 2 percent of 100 when the input was read or else an error message was generated. The simulator also had a facility for specifying some set of pages to be resident in memory at the time the simulation begins. This was designed to reduce the effects of startup faulting, the tendency for programs to fault heavily when they begin execution.

The pseudo random number generator used was motivated by Knuth [40] and verified by testing before being used in the simulator. A pseudo random number generator produces an identical sequence of "random" values if the same starting value (or "seed") is given, which was precisely what was desired for this study. Since this simulator was event-driven by page references, it was a reference-driven type of simulator as opposed to a fault-driven simulator as described in Chu and Opderbeck [26].

Mappings from virtual page to physical page and vice versa were performed very simply. A two dimensional array

was dimensioned to the number of users by the number of user virtual pages. If the virtual page was not in "memory" the contents of the array for that element was zero. If the virtual page was resident, that element of the array contained the element number of a singly dimensioned array which represented physical page frames. The mapping back to the virtual page from the physical page array was performed by placing the value of 100 times the user number plus the virtual page number in that element of the page frame array. For example, if element 17 of the page frame array contained a 914, then physical page frame belonged to user 9, virtual page 14. At most 10 users with 20 virtual pages each were allowed. The page frame maximum was 100.

2. TEST DATA

There were three base sets of test data used in the evaluation. These sets of data represented an I/O-bound job mixture, a heterogeneous job mixture, and a processor-intensive job mixture. Each set of data simulated 10 concurrently executing programs. In all test data the job classes were oriented somewhat toward the I/O side of the spectrum, since many computer centers give I/O-intensive processes priority because of frequent interruptions that allow other tasks to execute, and because of the greater availability of I/O devices than processor time.

The processor-bound jobs in the test data were given a higher turnover of activity sets to simulate a rapid execution of instructions and use of memory. A realistic example of this might be some large matrix processing program. The more I/O-intensive tasks had a lower turnover of activity sets to simulate the smaller amount of progress made for each processor allocation.

All activity sets used in the test data consisted of eight pages, four of which accounted for 80 percent of the page references, and four others which were referenced the remaining 20 percent of the time. Transitions from one activity set to another were triggered by expiration of the lifetime of the old activity set, and the new activity set was indicated on input as eight different pages of the user's total number with a similar 80/20 reference frequency pattern. A range value adjustment was performed as each new activity set became effective.

There were three available page replacement algorithms in the simulator, namely RAND, FIFO and LRU, and each of these algorithms could be implemented globally or locally. For global implementation of the algorithms, each faulting program was allowed to steal pages from any other program (including itself), based on the direction of the replacement algorithm. With local implementation, the number of physical page frames was divided by the number of users, and each user

was limited to that number of page frames. If a page was selected for replacement, it was chosen from the faulting process' allotted number of page frames. Since all activity sets were the same size, this idea of an identical local page allocation for each program seemed reasonable.

Three sizes of physical memory were chosen for the test data. The smaller size, 30 frames, would not allow all of the most heavily referenced portion of a program's activity set to reside in memory for local implementations. The next larger size of memory, 50 frames, allowed the most heavily referenced pages to reside in memory as well as one of the less frequently referenced pages. The largest memory size, 70 frames, allowed all but one page of a program's activity set to reside in memory.

There were three sets of test data that represented the type of job mixture, three page replacement algorithms, two implementations of each algorithm, and three sizes of physical memory; hence 54 simulation executions were performed.

Statistics were collected for the number of successful page references (those that did not generate page faults) and the number of unsuccessful page references (those that did generate page faults). These counts of successes and failures were kept for each program and converted to a reference success percentage by dividing the number of

successes by the total number of page references. An overall reference success percentage was then computed and used to report the results for the simulation execution.

CHAPTER 4

RESULTS

1. GENERAL DISCUSSION

Bar charts were chosen to illustrate the results obtained in this study. All graphs are shown with the vertical axis representing the reference success percentage, or the percentage of the time a reference was made to a page that was in main memory. Also, all charts are shown with clusters of three bars: the dark bar represents the I/O-intensive job mixture, the cross hatched bar stands for the heterogeneous job mixture, and the light bar symbolizes the processor-bound job mixture.

Six of the graphs illustrate how the amount of physical memory affected performance for each algorithm/implementation pairing, and are discussed in the sections pertaining to their respective page replacement algorithms. The remaining figures show how the local and global implementations compared for each algorithm/memory size pairing. These are also discussed in the corresponding algorithm's section.

Overall comparisons between page replacement algorithms are presented in Chapter 5, and Appendix A contains the tabular results for the simulations.

2. RAND ALGORITHM

A separate procedure for generating pseudo random number sequences for page replacement was used so as not to bias or change the address reference string produced by the original pseudo random number generator. The generator used for random page replacement was constructed identically to the generator for the page reference strings except for a different seed, or initial value. In this manner, reference strings for each simulation were identical.

Figure 4.1 illustrates the reference success percentage for the RAND page replacement algorithm with a local implementation, and Figure 4.2 displays the same information for a global version of RAND.

Note the difference in sensitivity to the type of job mix for the local and global implementations of the algorithm. This can be explained by the fact that global implementations "punish" the I/O-bound tasks, which address fewer pages before being interrupted, by having their pages stolen by the more processor-intensive tasks, which reference more addresses and accumulate more pages in memory each time they are dispatched. With local implementations, no task may steal pages from another and thereby affect its performance. This makes local schemes attractive for job mixes that may contain high priority realtime tasks that are dormant for long periods of time. These realtime tasks are then somewhat protected from

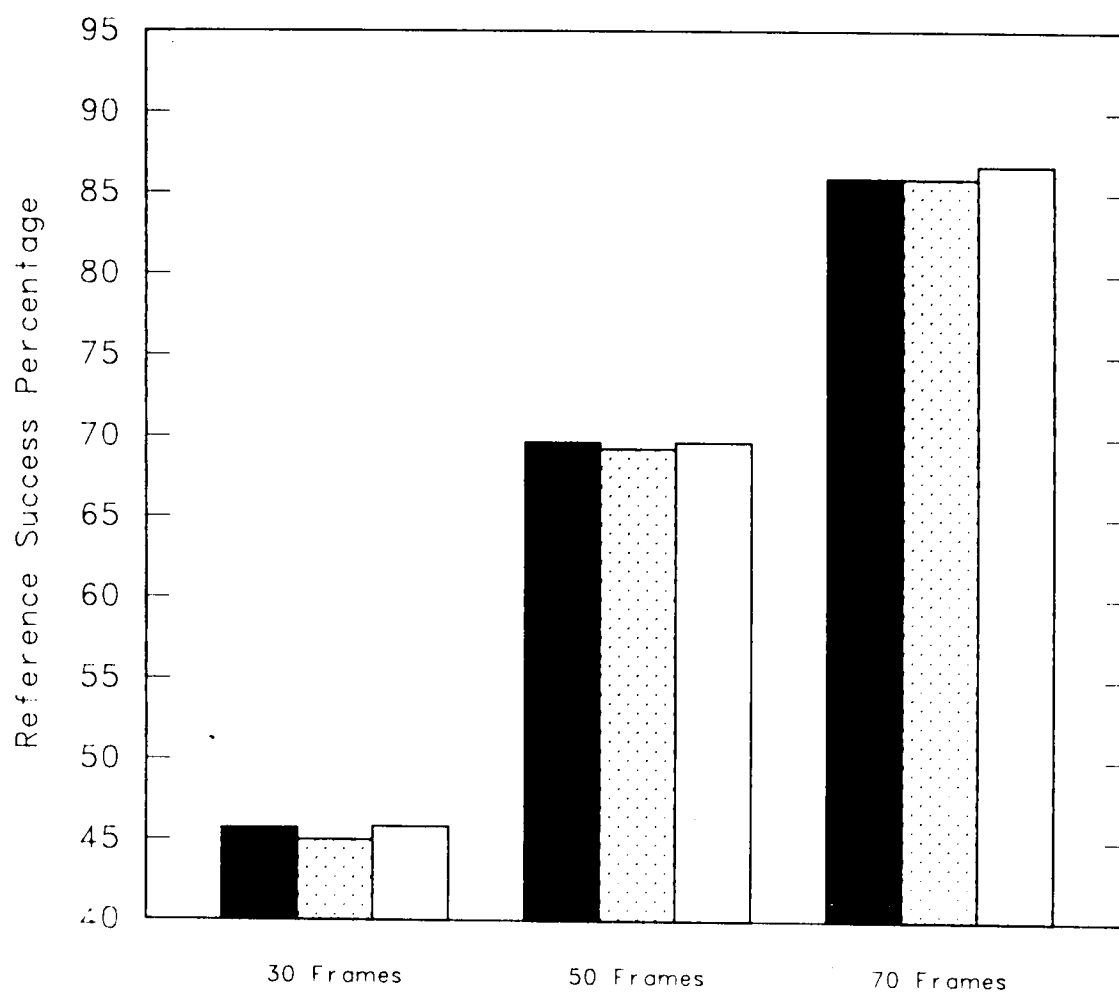


Figure 4.1. RAND with local implementation.

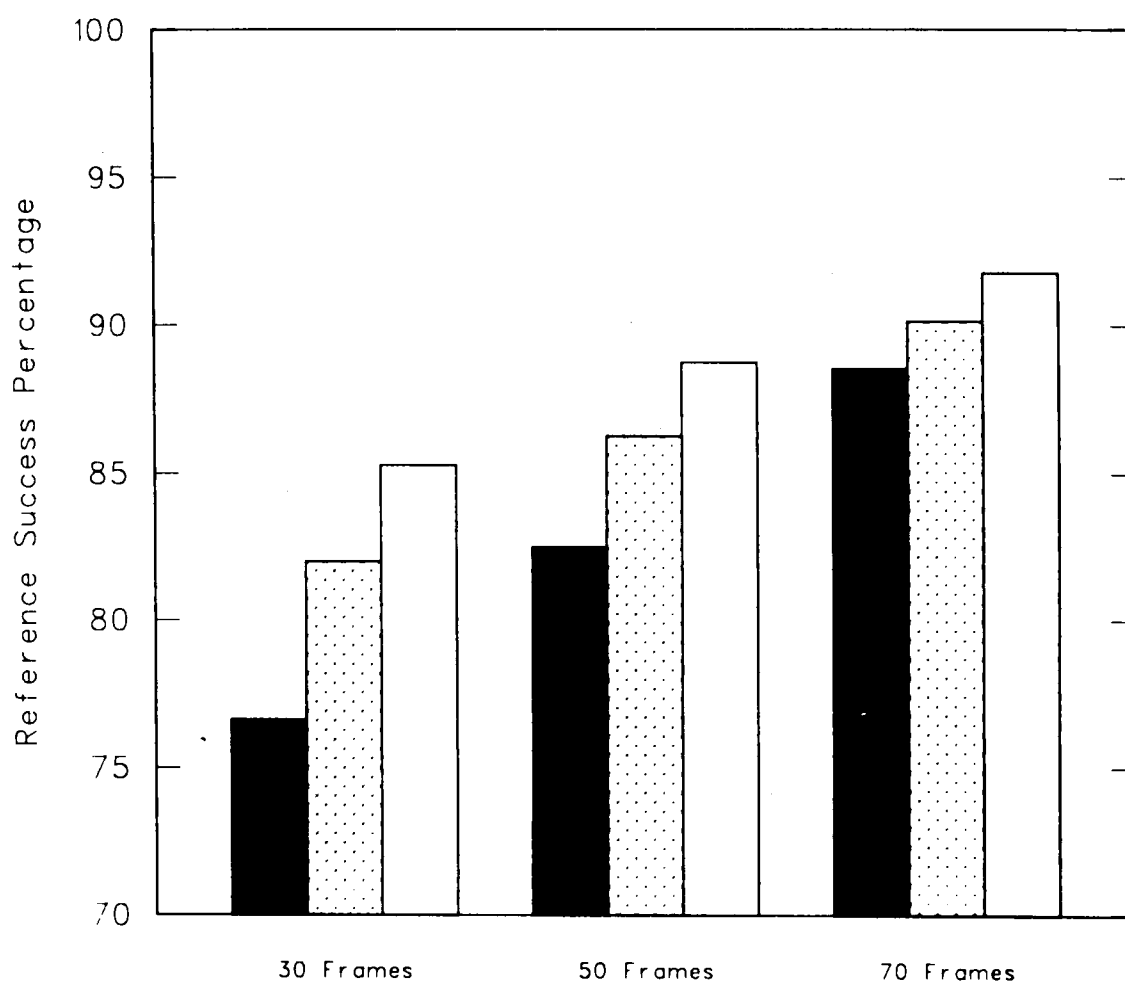


Figure 4.2. RAND with global implementation.

having their response time degraded by having to fault pages that were stolen by other programs.

The local implementation shows a type of behavior called a parachor, a curve whose improvement decreases with each increment of memory. The global implementation seems to minimize the effects of limited memory, and shows a much slower, more nearly linear increase in reference successes.

Figures 4.3-4.5 show how the local implementation's performance approaches that of the global version's as memory size is increased. Additional, unreported data have shown that with a memory size of 80 page frames, which is large enough to contain all pages of a working set, the local implementation's paging efficiency surpasses that of the global implementation.

3. FIFO ALGORITHM

Figures 4.6 and 4.7 again show the difference in sensitivity to the type of job mixture for the local and global implementations, and by tracing a line through the tops of the bars in Figure 4.6 a parachor curve is created. Surprisingly, the results of the 30 frame global implementation and the 50 frame global implementation are identical, even though Belady [6] has shown RAND and FIFO performances to be fairly constant over a wide range of memory sizes.

Figures 4.8-4.10 again show how the local algorithm's performance approaches that of the global version as memory

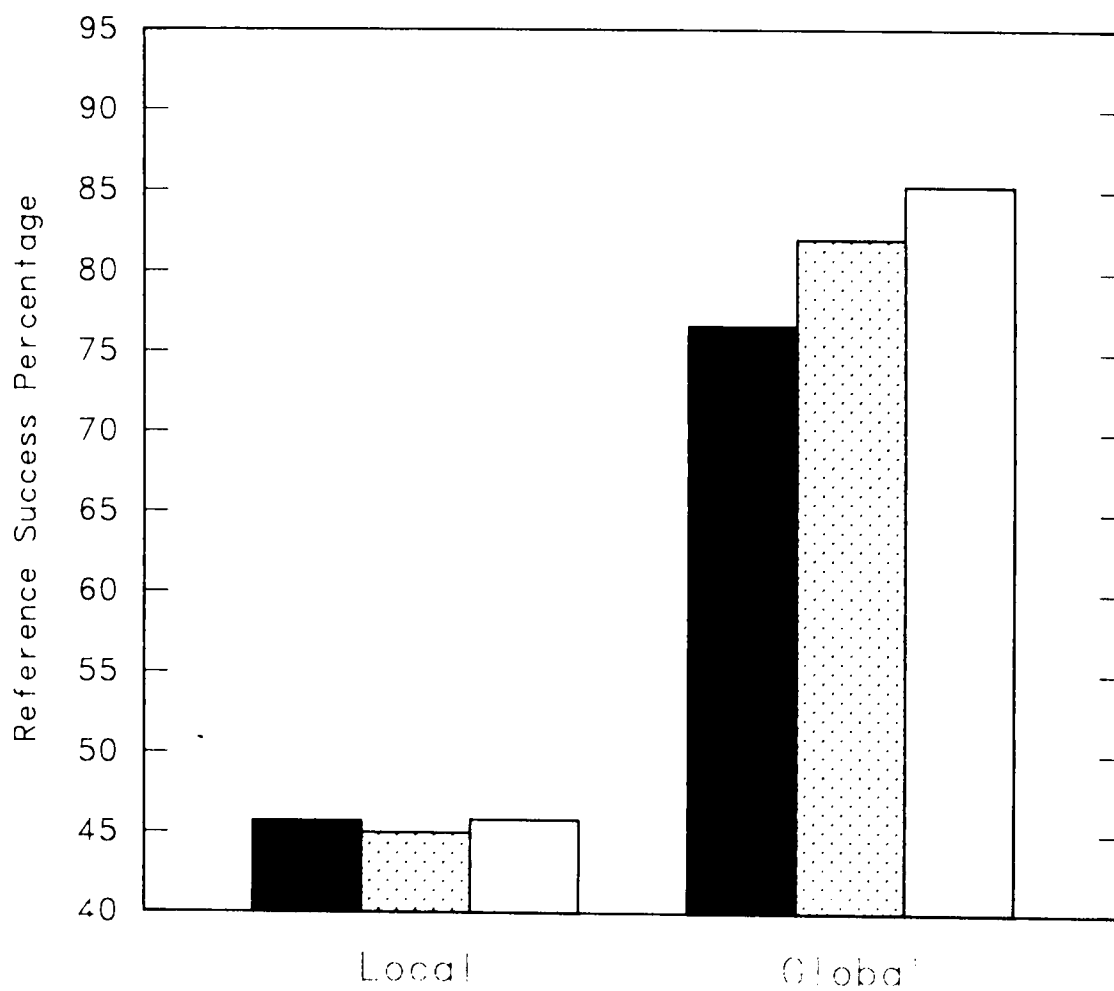


Figure 4.3. RAND with 30 page frames.

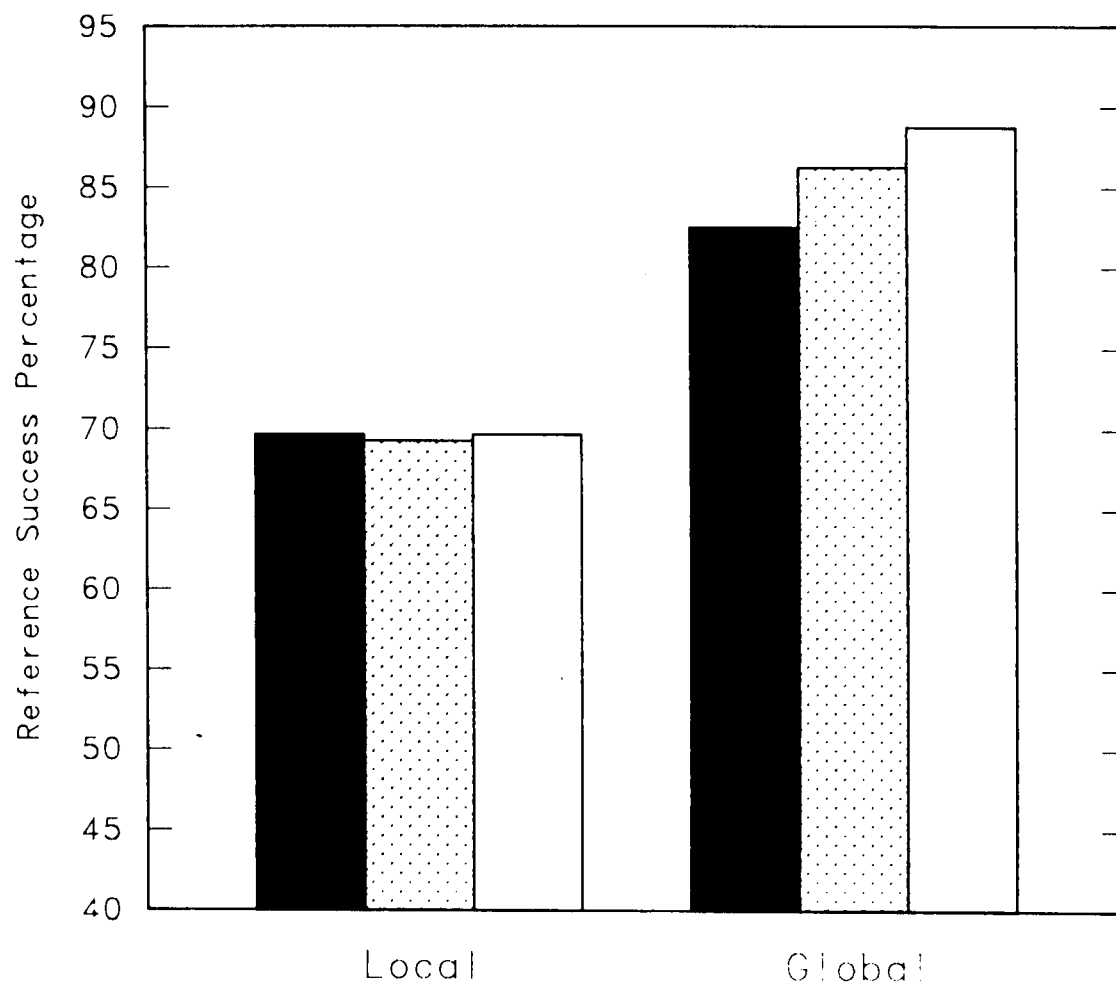


Figure 4.4. RAND with 50 page frames.

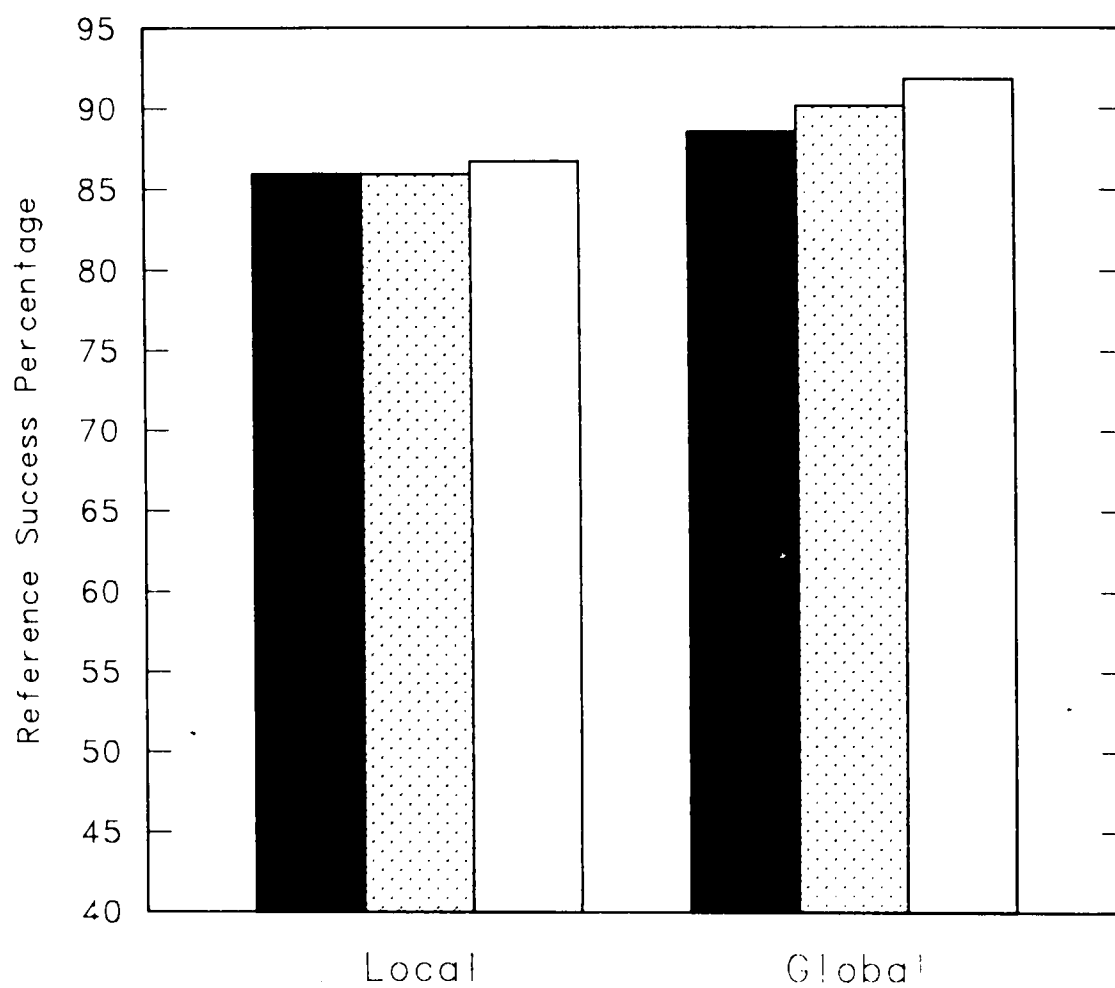


Figure 4.5. RAND with 70 page frames.

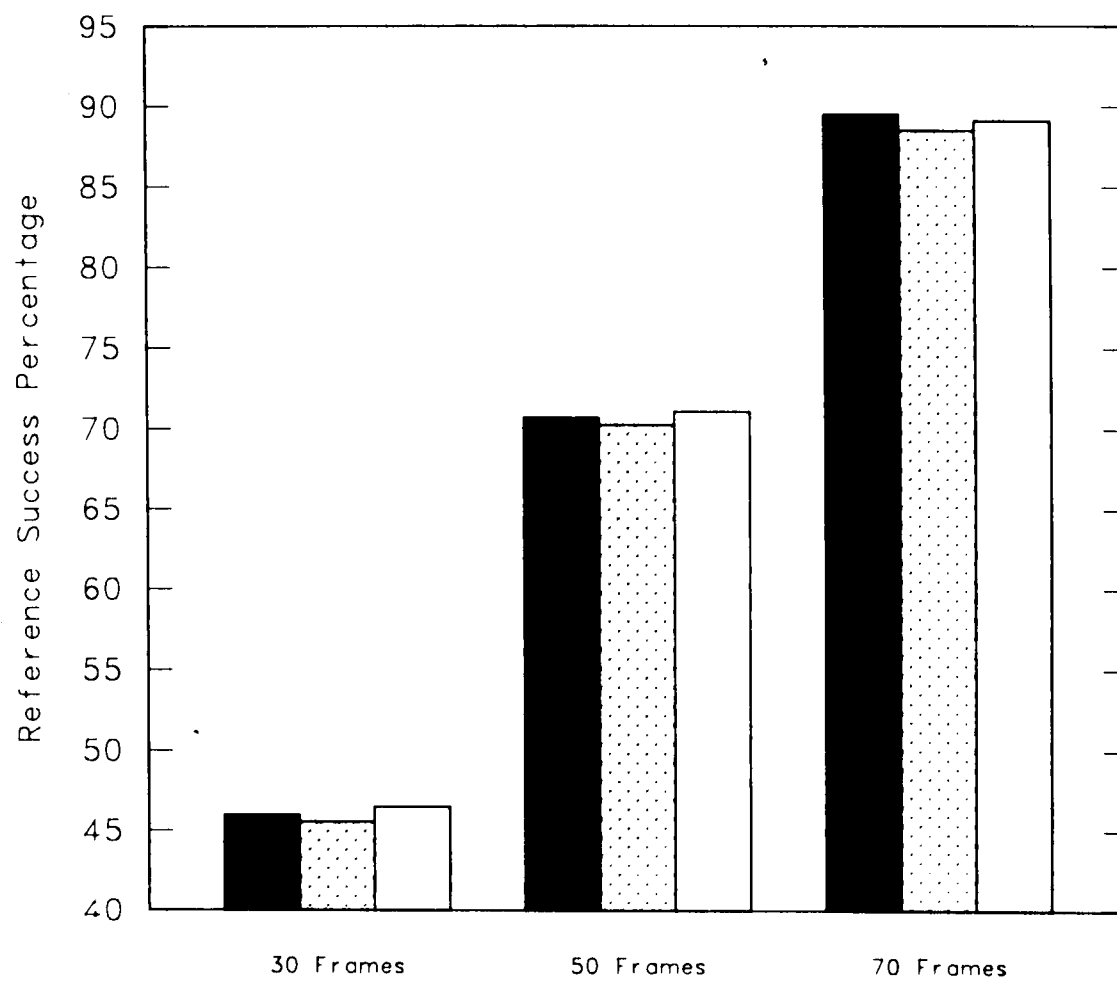


Figure 4.6. FIFO with local implementation.

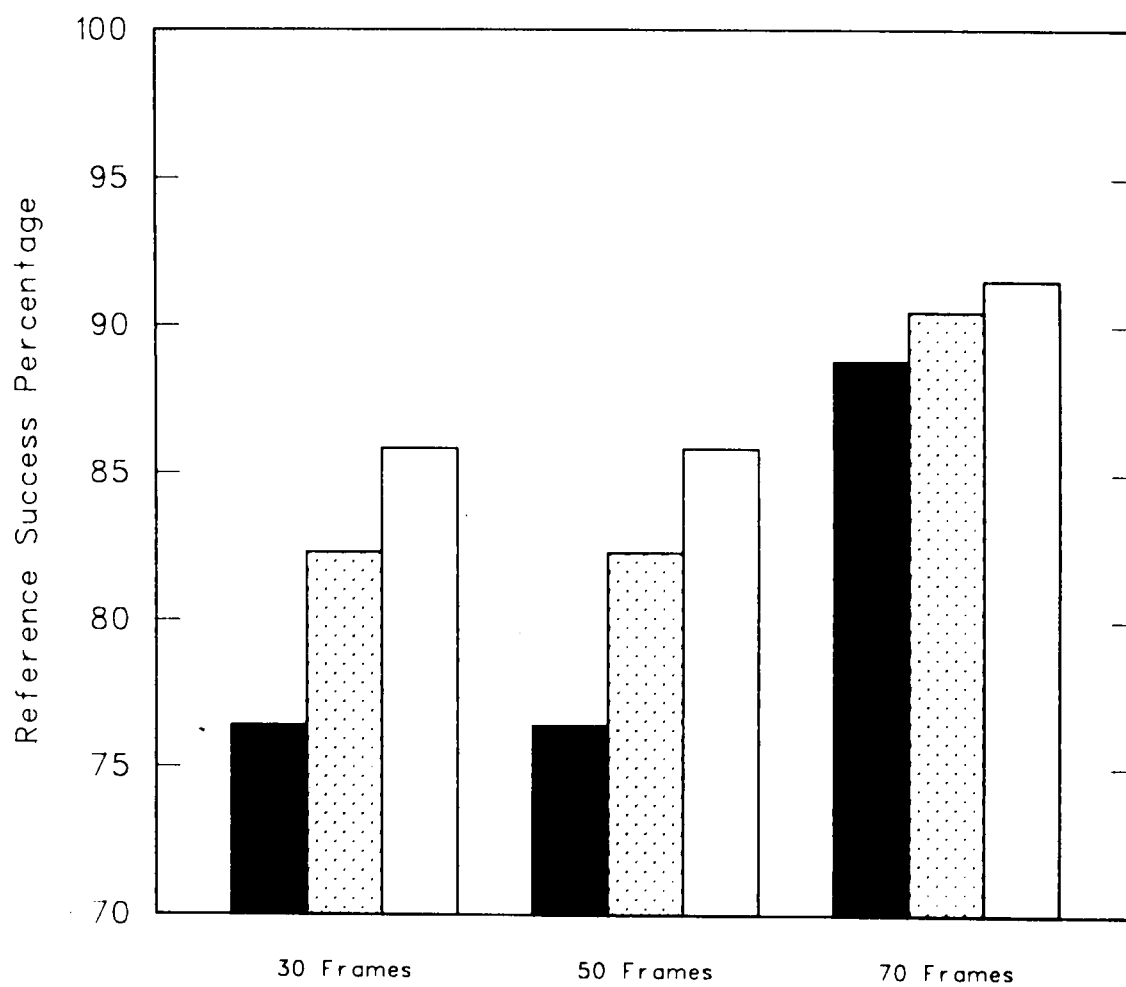


Figure 4.7. FIFO with global implementation.

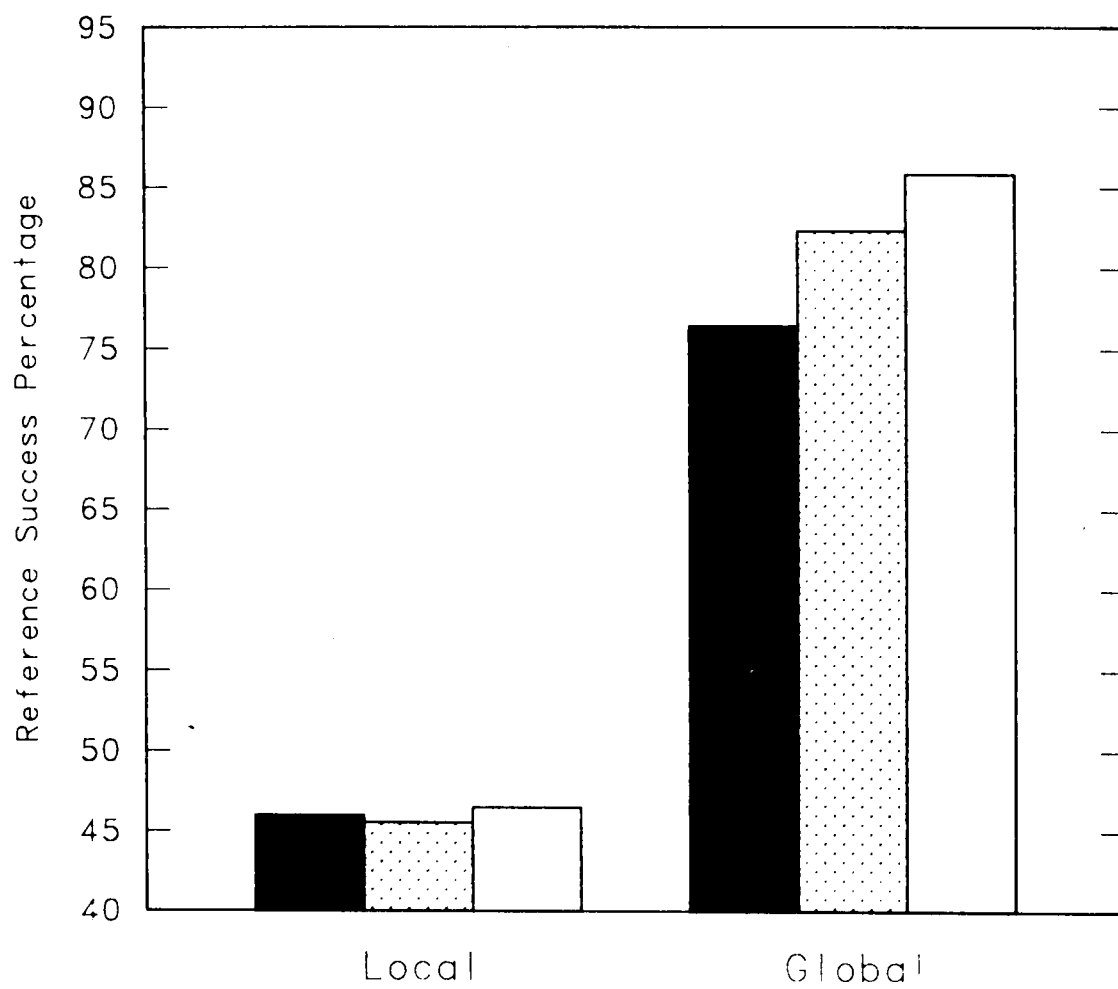


Figure 4.8. FIFO with 30 page frames.

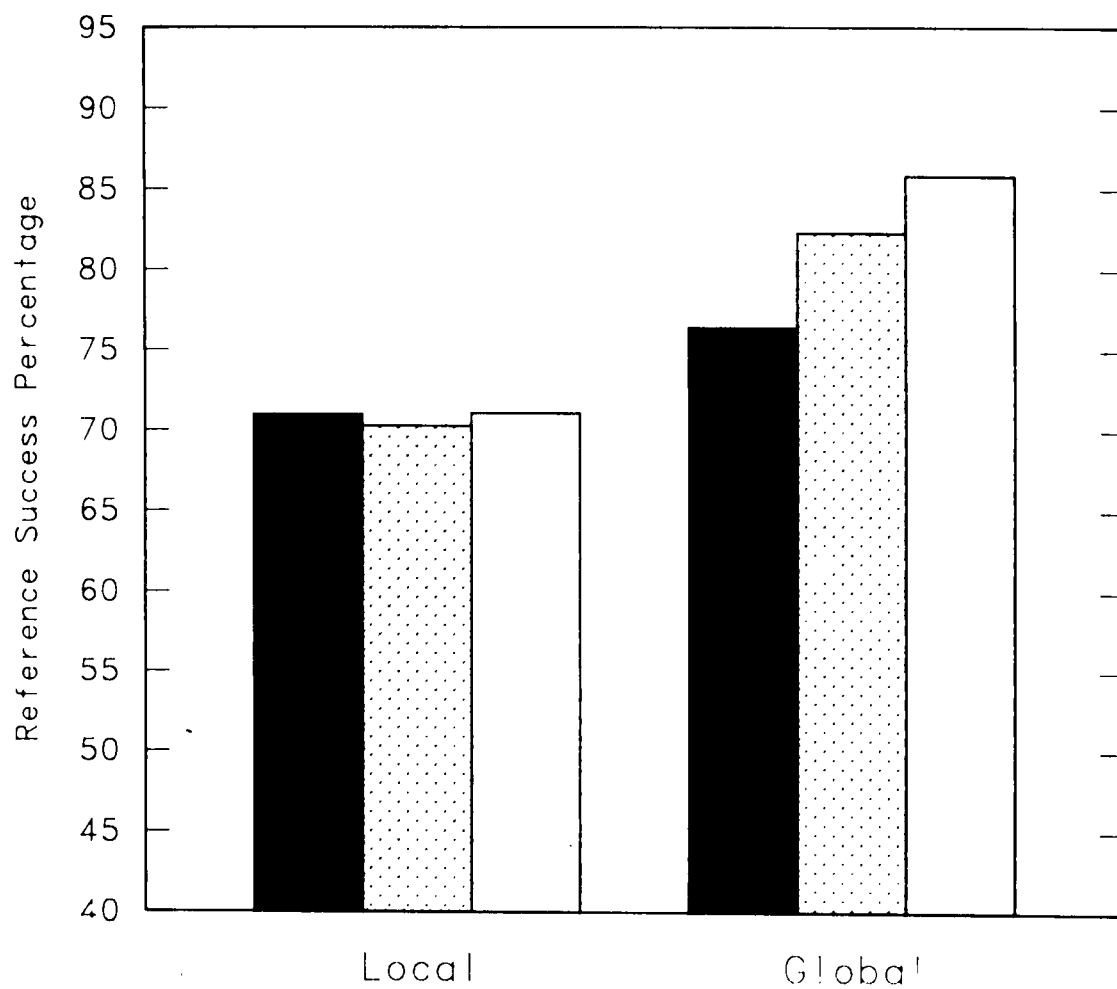


Figure 4.9. FIFO with 50 page frames.

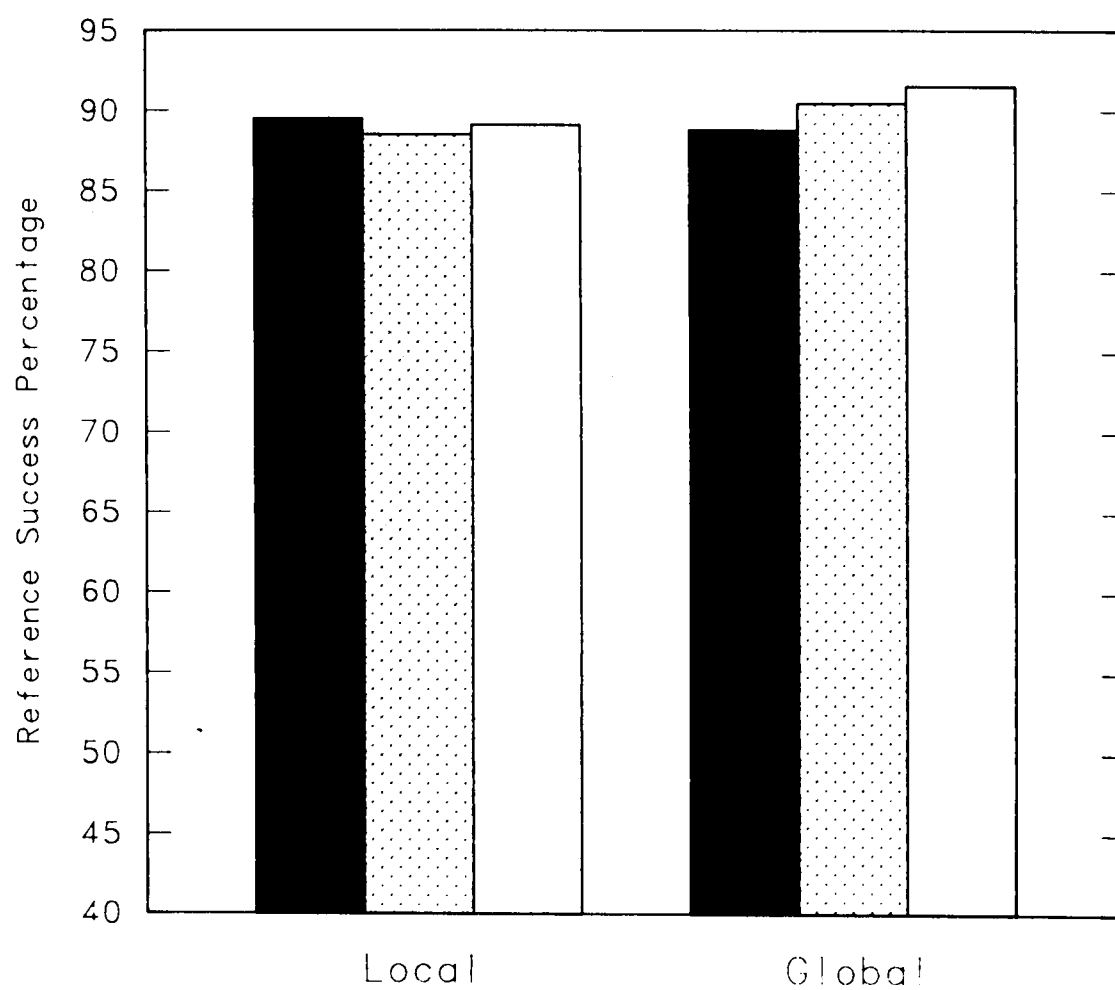


Figure 4.10. FIFO with 70 page frames.

size increases. The FIFO performance differences at the largest memory size were significantly less than those for RAND. In the I/O-bound simulation the local implementation's performance even surpassed that of the global version.

4. LRU APPROXIMATION ALGORITHM

The particular type of LRU approximation used for this simulator was that as described by Madnick and Donovan [41]. A single reference bit was associated with each physical page frame, and this bit was set to 1 every time the virtual page which occupied the frame was referenced. A cyclic page frame pointer was kept, similar to FIFO. When a page was needed for removal, the reference bits were examined starting from where the pointer was left after the last page replacement. Reference bits that are 1 are reset to 0 during the sequential search for the first page with a 0 reference bit. When this page was found, it was chosen for replacement, and the pointer was incremented to point immediately after the selected one.

Local LRU is insensitive to the type of job mixture as shown in Figure 4.11; this was also the case with RAND and FIFO. Figure 4.12 demonstrates a similarity between the 30 frame simulation and the 50 frame simulation of the global version.

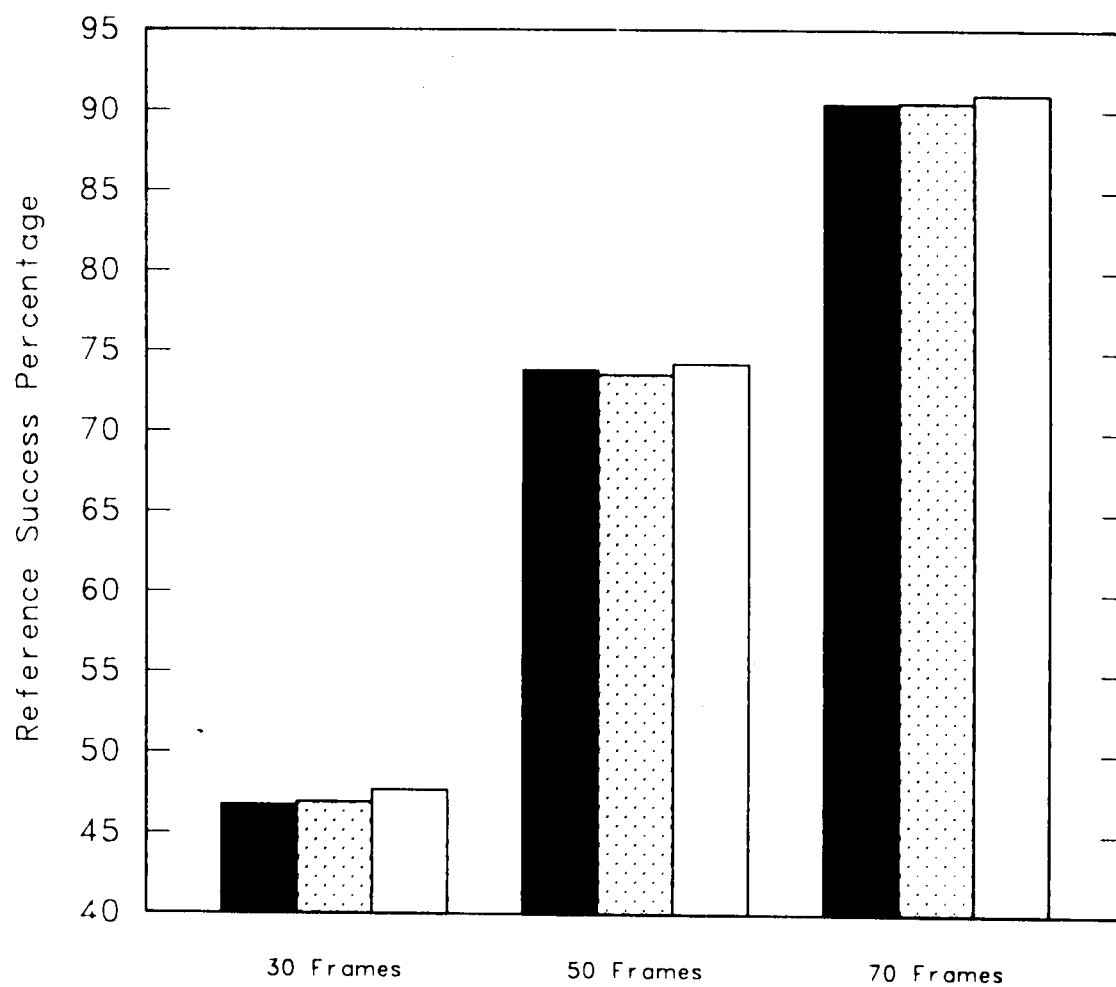


Figure 4.11. LRU with local implementation.

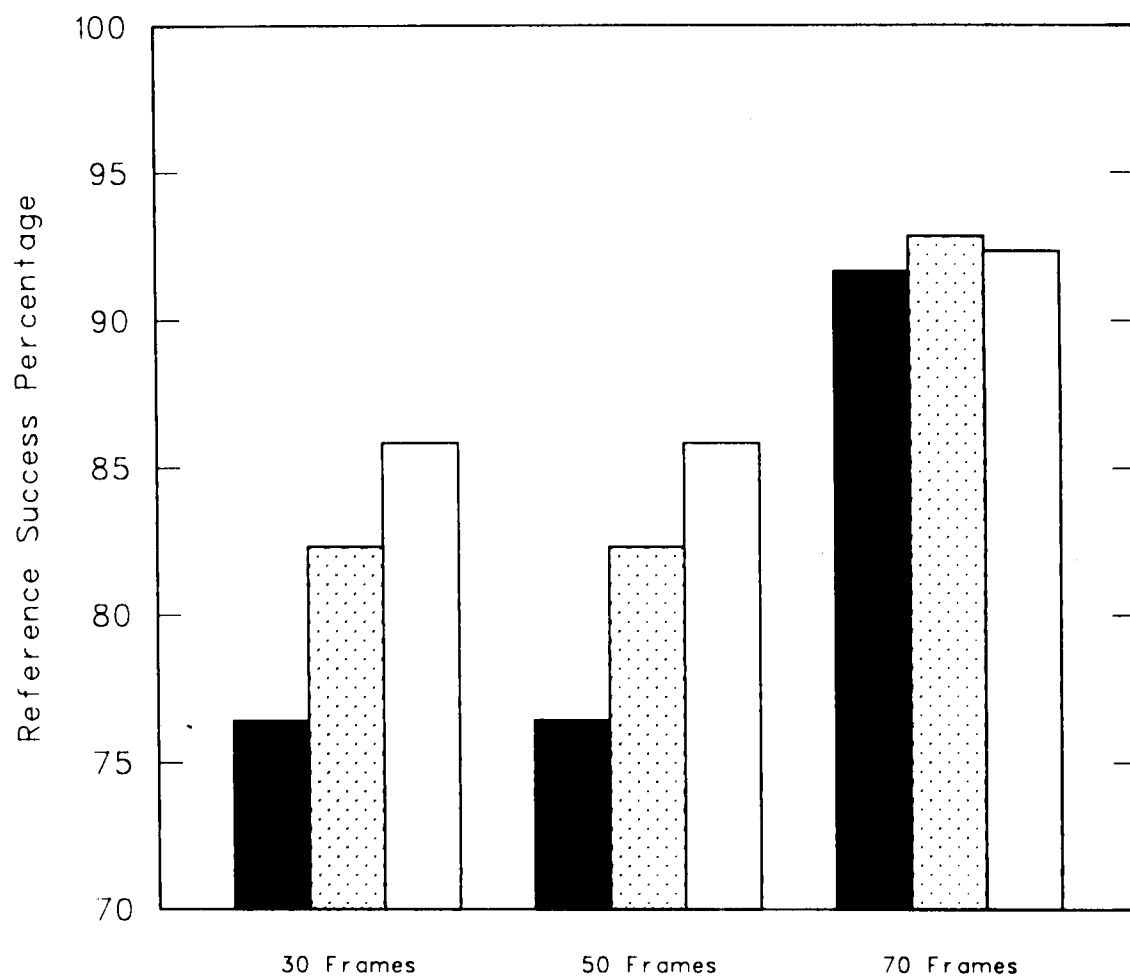


Figure 4.12. LRU with global implementation.

Figures 4.13-4.15 again illustrate how locally implemented performance approaches that of global versions as the local allotment of memory becomes large enough to accommodate all of a program's activity set.

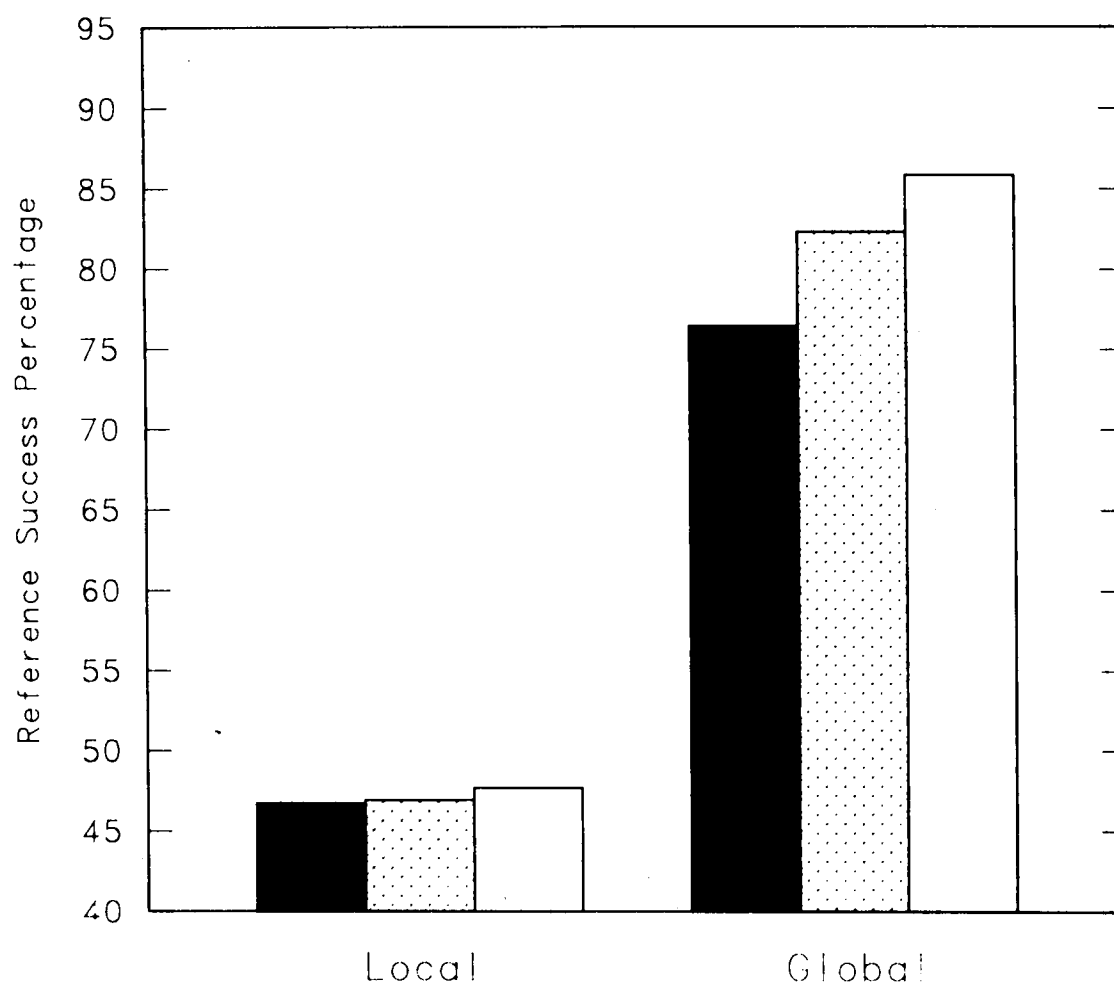


Figure 4.13. LRU with 30 page frames.

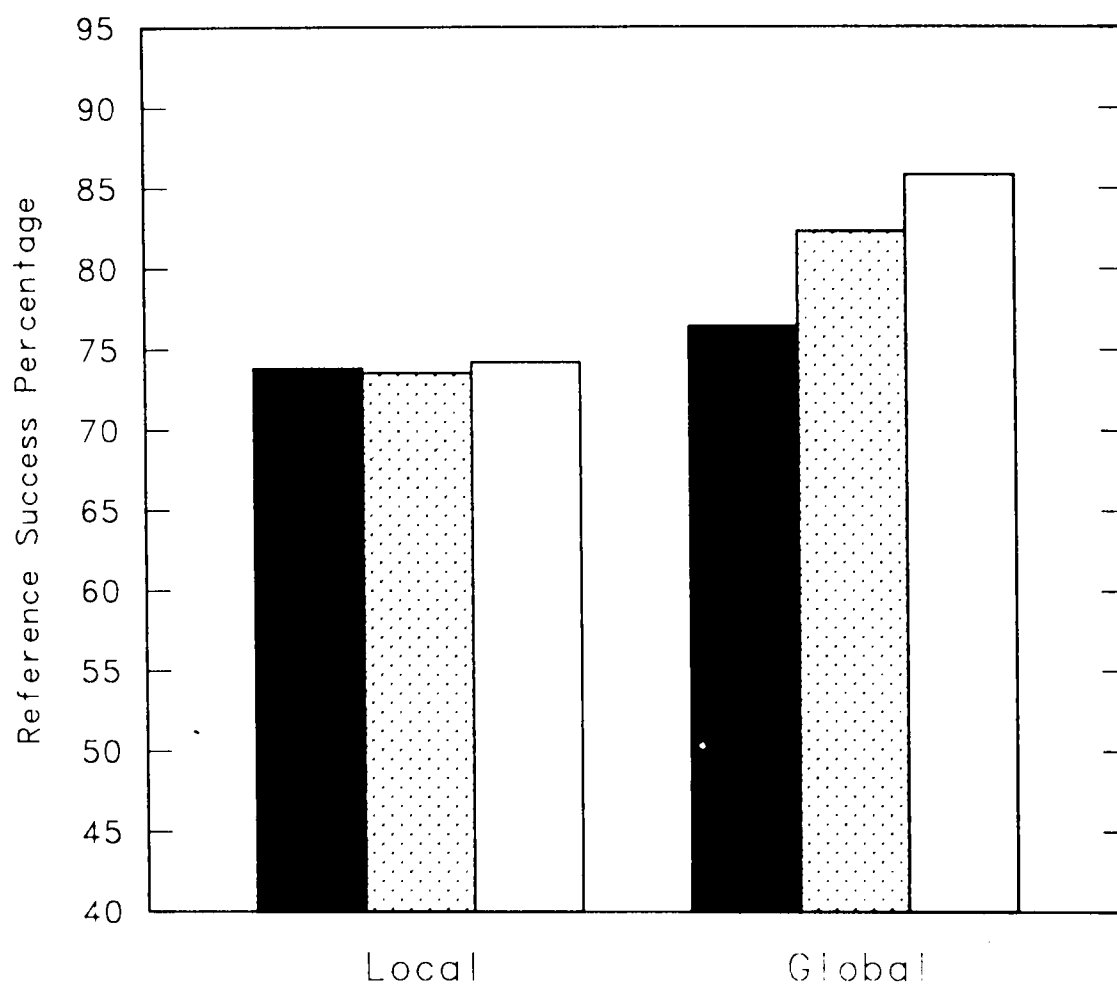


Figure 4.14. LRU with 50 page frames.

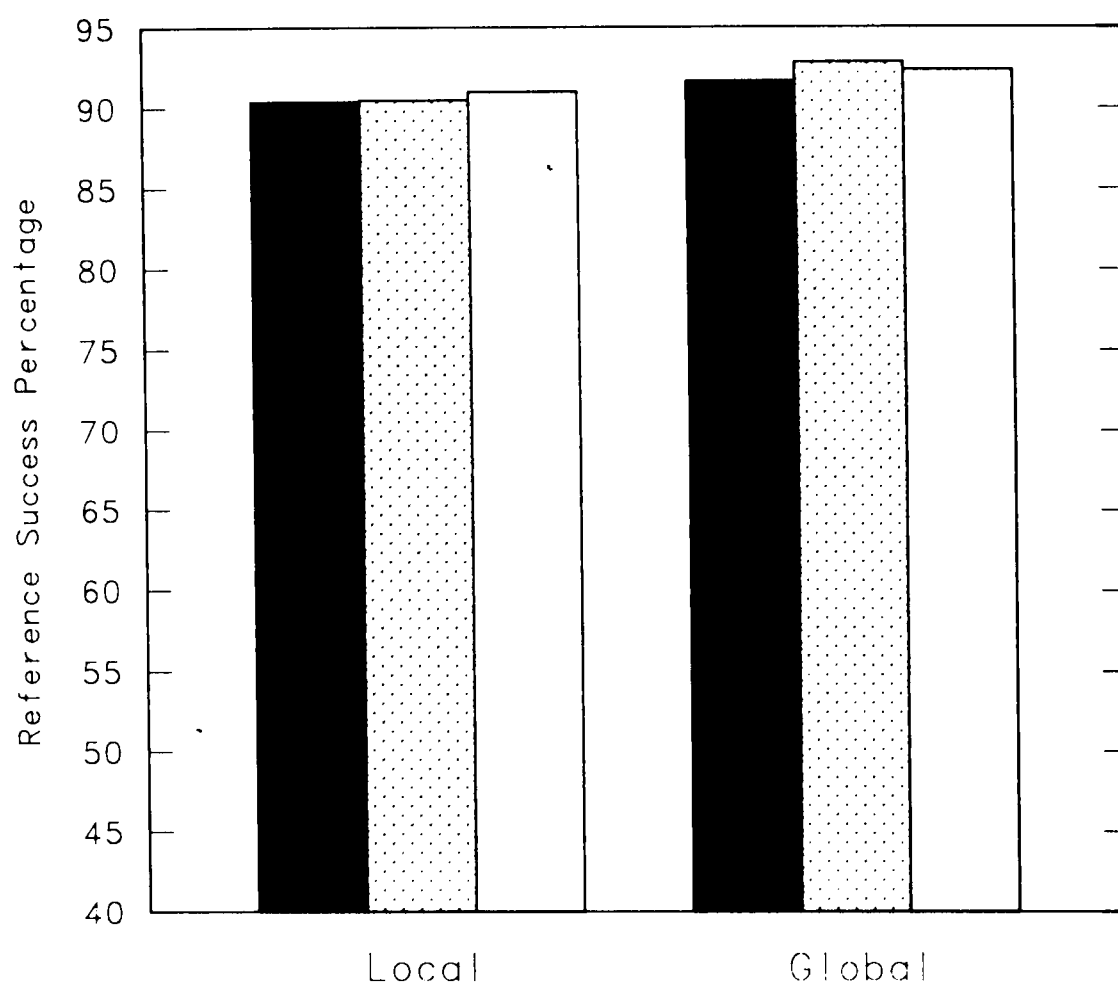


Figure 4.15. LRU with 70 page frames.

CHAPTER 5

CONCLUSIONS

1. INTERALGORITHM COMPARISONS

Some significant findings have already been discussed in the sections dealing with the results of the individual algorithms. The goal of this section is to present some comparisons and observations between the page replacement algorithms rather than within them.

The original portrayal of the parachor curve was that of a single program competing with itself for memory, called a local scheme in the context of this study. Global methods may be thought of similarly, as if a single program were looping through executions of several working sets, stealing pages from itself. This relation will help in the comparison of results presented in this study to observations in previous literature.

As displayed in Figures 5.1, 5.2, and 5.3, the results for the local schemes of the page replacement algorithms for all memory sizes were as expected. RAND gave the poorest and most constant performance. FIFO yielded the next best performance and was only slightly more sensitive to variations in memory size. LRU produced the best page reference percentages in all cases.

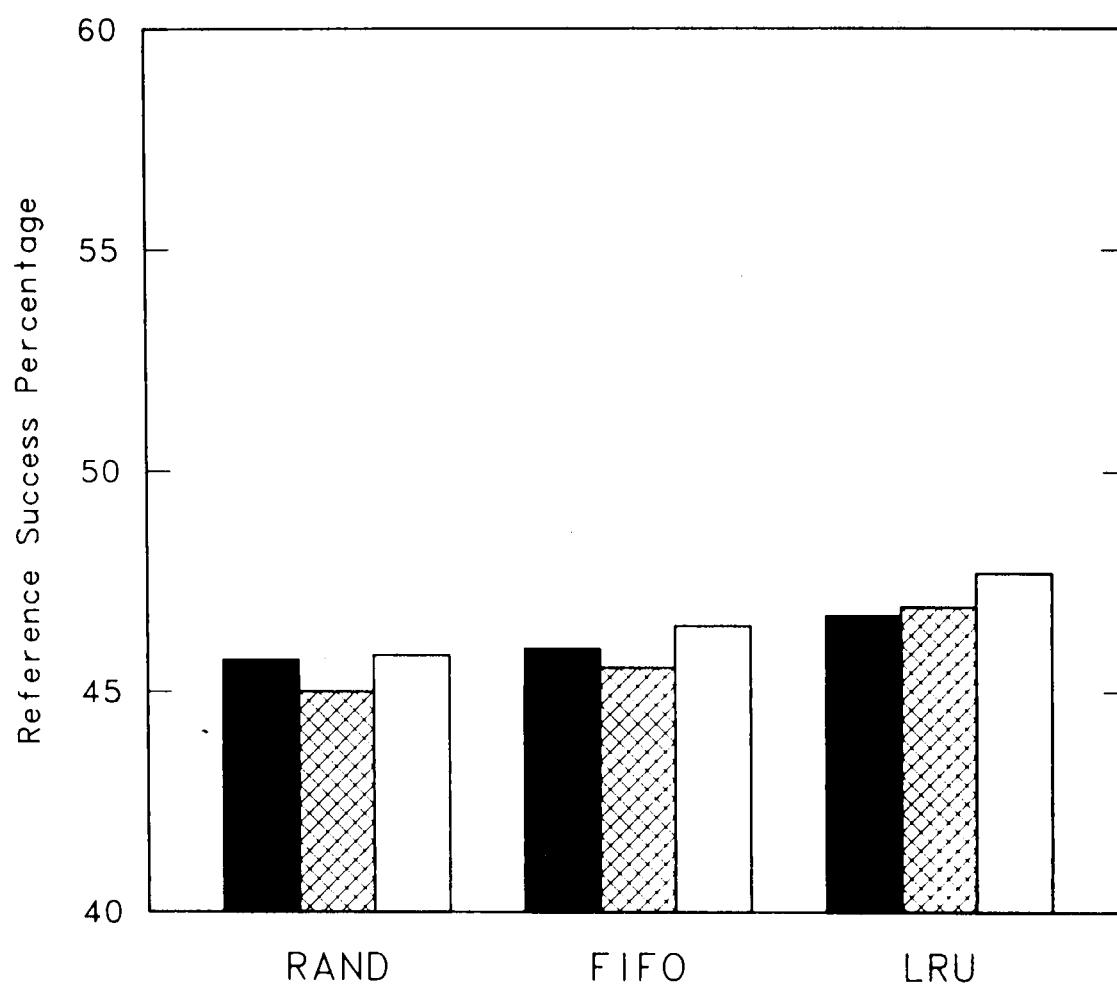


Figure 5.1. Local implementations with 30 page frames.

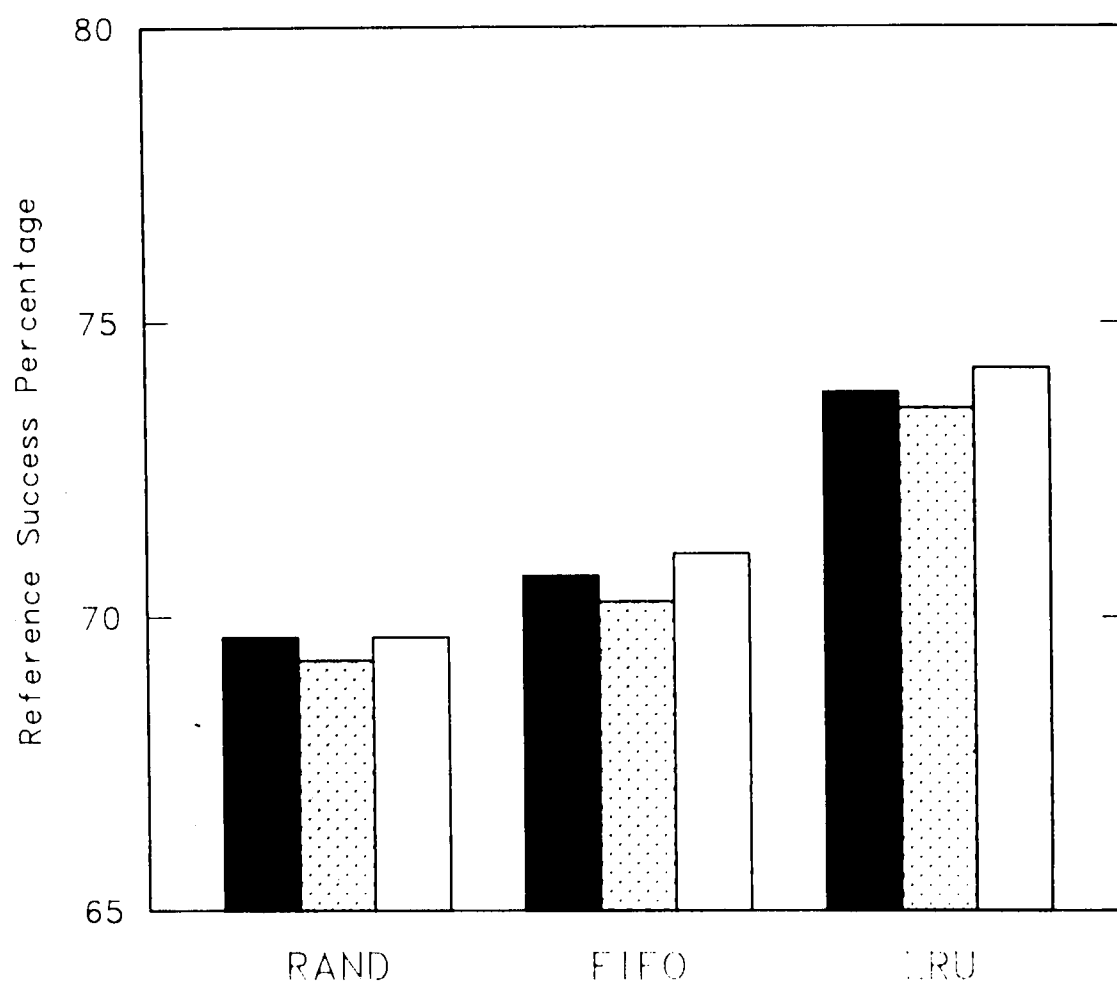


Figure 5.2. Local implementations with 50 page frames.

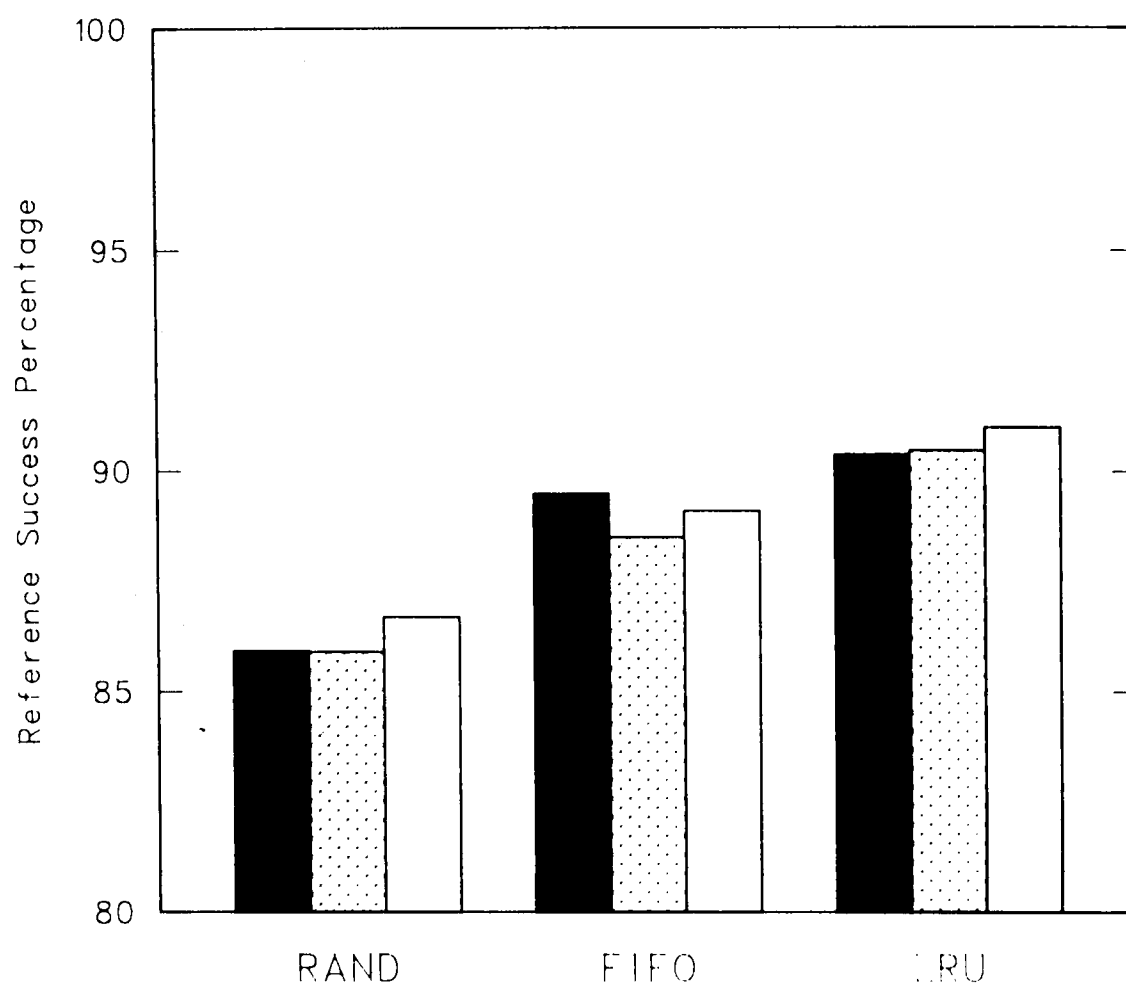


Figure 5.3. Local implementations with 70 page frames.

The most surprising observation is the similarity of the performances of the page replacement algorithms for global implementations, as is illustrated by Figures 5.4, 5.5, and 5.6. For the smallest memory size of 30 frames, the results were almost identical. At the next larger memory size the differences were only slightly more pronounced. Only at the largest memory size of 70 frames did differences show up between all three of the page replacement algorithms.

The intuitive reasoning for this is that with a small effective memory size any page replacement algorithm is bound to make more misguided decisions since there are fewer page frames in which to keep the most frequently used pages. Franklin and Gupta [32] and Belady [6] have stated that in these situations even RAND can outperform LRU, when the opposite is almost always the case. One such situation was encountered in this study.

RAND outperformed FIFO and LRU in their global versions at a physical memory size of 50 frames. Half of this situation can possibly be explained by the fact that with a FIFO scheme, each page is guaranteed to be removed from physical memory within a certain number of page faults, assuring that those pages most heavily referenced pages will be removed as well. With RAND, however, this is not necessarily the case. A heavily referenced activity set may be "missed" by RAND for long enough to finish execution. Of

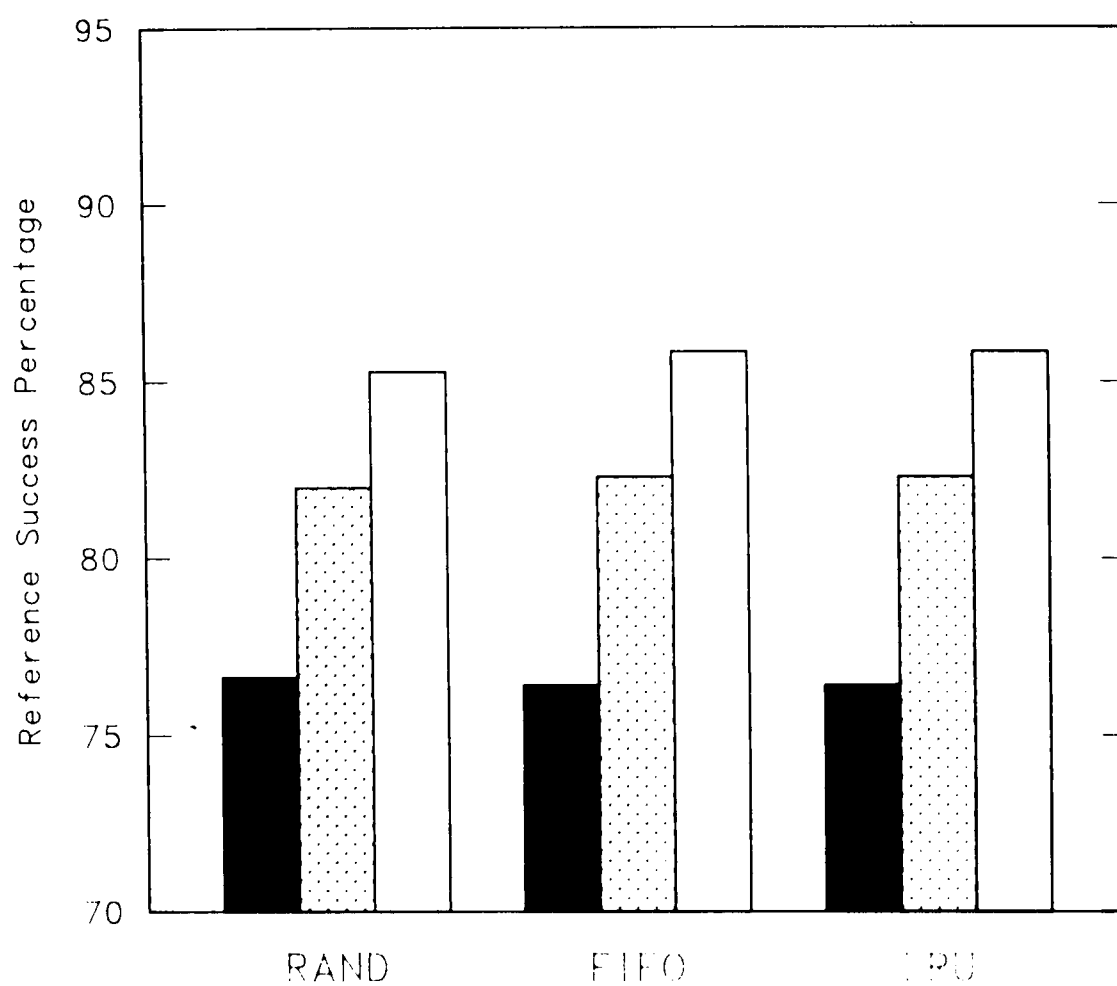


Figure 5.4. Global implementations with 30 page frames.

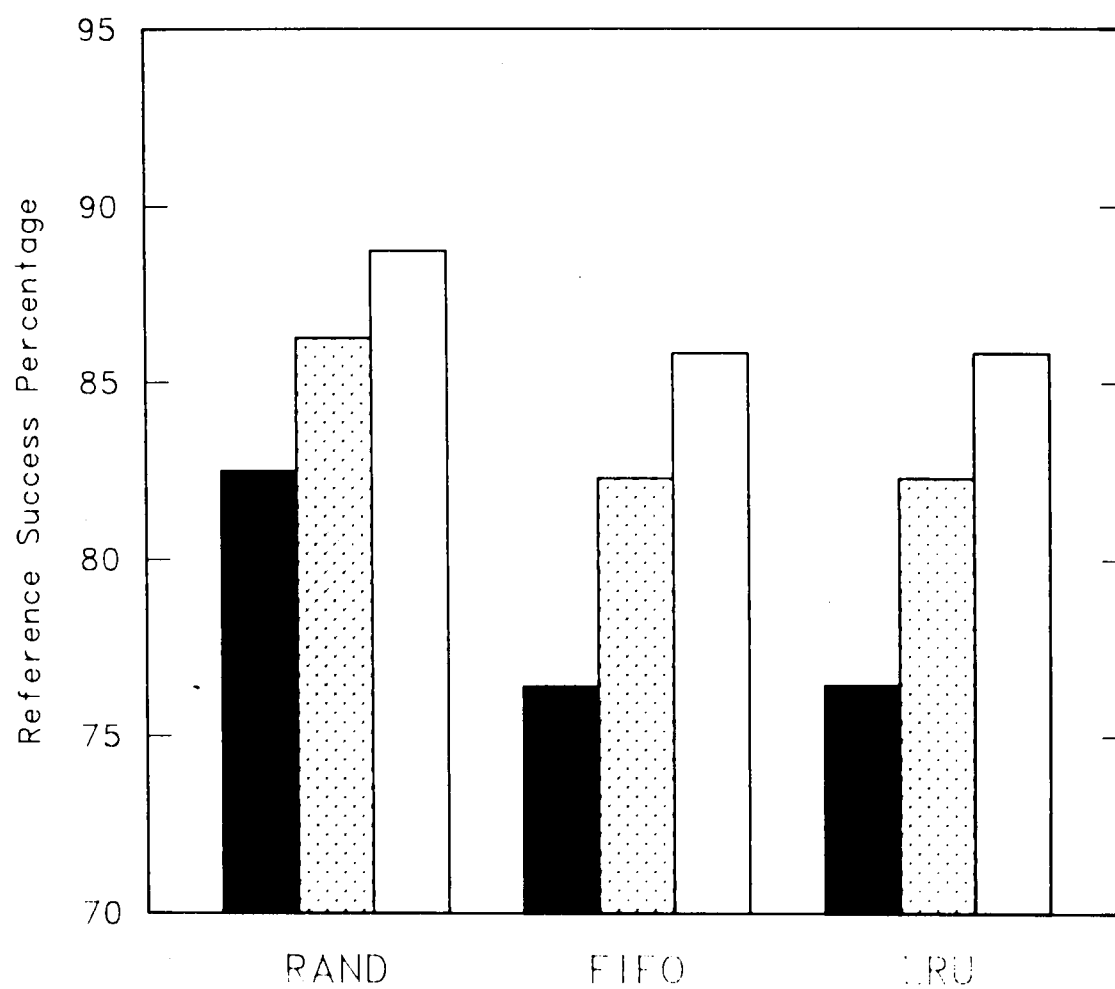


Figure 5.5. Global implementations with 50 page frames.

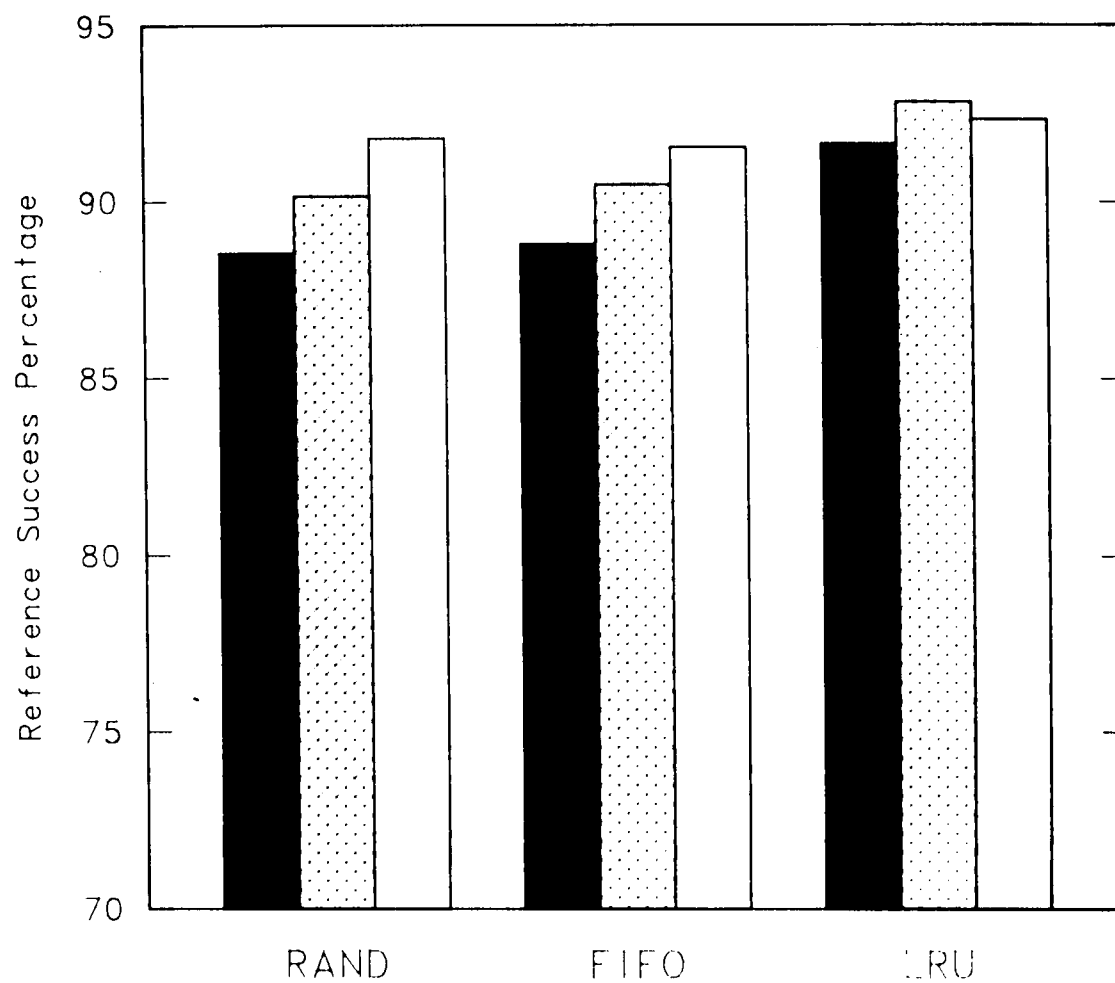


Figure 5.6. Global implementations with 70 page frames.

course, it is just as likely that RAND would perform more poorly than FIFO, which was the situation most often observed.

2. SUGGESTIONS FOR FUTURE RESEARCH

Two items not incorporated by the simulator used in this study could provide some interesting results if implemented. The first item is the ability to cause a looping string of page references. Since the reference strings were produced by a random number generator, recurring loops were almost certainly not produced. Program behavior involving large loops can degrade the performance of several page replacement algorithms, especially in situations where the entire set of pages involved in the loop cannot fit into the allotted number of page frames.

The second feature is the capability to dynamically allocate a program's page limit in local implementations. Two page replacement algorithms that were not evaluated in this study, the page fault frequency (PFF) algorithm and the Denning working set (DWS) algorithm, rely on this idea in order to work properly.

It was interesting to note the closeness in performance of the local and global implementations at the largest amount of physical memory. One further set of test data, with 80 page frames of main memory would probably allow local performance to surpass global performance. One such unreported test case has given support to this assumption.

LIST OF REFERENCES

LIST OF REFERENCES

1. Parmelee, R. P., T. I. Peterson, C. C. Tillman and D. J. Hatfield, "Virtual Storage and Virtual Machine Concepts," IBM Systems Journal, 11, 2, 1972, pp. 99-130.
2. Denning, P. J., "Virtual Memory," Computing Surveys, 2, 3, 1970, pp. 153-189.
3. Fotheringham, J., "Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store," Communications of the ACM, 4, 10, 1961, pp. 435-436.
4. Kuehner, C. J., and B. Randell, "Demand Paging in Perspective," AFIPS Fall Joint Computer Conference, 33, 1968, pp. 1011-1017.
5. Denning, P. J., "Thrashing: Its Causes and Prevention," AFIPS Fall Joint Computer Conference, 33, 1968, pp. 915-922.
6. Belady, L. A., "A Study of Replacement Algorithms for a Virtual Storage Computer," IBM Systems Journal, 5, 2, 1966, pp. 78-101.
7. Coffman, E. G., and L. C. Varian, "Further Experimental Data on the Behavior of Programs in a Paging Environment," Communications of the ACM, 11, 7, 1968, pp. 471-474.
8. Joseph, M., "An Analysis of Paging and Program Behavior," Computer Journal, 13, 1, 1970, pp. 48-54.
9. Hatfield, D. J., "Experiments on Page Size, Program Access Patterns, and Virtual Memory Performance," IBM Journal of Research and Development, 16, 1, 1972, pp. 58-66.
10. Hatfield, D. J., and J. Gerald, "Program Restructuring for Virtual Memory," IBM Systems Journal, 10, 3, 1971, pp. 168-192.
11. Denning, P. J., "On Modeling Program Behavior," AFIPS Spring Joint Computer Conference, 40, 1972, pp. 937-944.
12. Ferrari, D., "Improving Locality by Critical Working Sets," Communications of the ACM, 17, 11, 1974, pp. 614-620.

13. Slutz, D. R., and I. L. Traiger, "A Note on the Calculation of Average Working Set Size," Communications of the ACM, 17, 10, 1974, pp. 563-565.
14. Denning, P. J., "The Working Set Model for Program Behavior," Communications of the ACM, 11, 5, 1968, pp. 323-333.
15. Brawn, B. S., and F. G. Gustavson, "Program Behavior in a Paging Environment," AFIPS Fall Joint Computer Conference, 33, 1968, pp. 1019-1032.
16. Morrison, J. E., "User Program Performance in Virtual Storage Systems," IBM Systems Journal, 12, 3, 1973, pp. 216, 237.
17. Knuth, D. E., "An Empirical Study of FORTRAN Programs," Software-Practice and Experience, 1, 2, 1971, pp. 105-133.
18. Williams, J. G., "Experiments in Page Activity Determination," AFIPS Spring Joint Computer Conference, 40, 1972, pp. 739-747.
19. Madison, A. W., and A. P. Batson, "Characteristics of Program Localities," Communications of the ACM, 19, 5, 1976, pp. 285-294.
20. Spirn, J. R., and P. J. Denning, "Experiments with Program Locality," AFIPS Fall Joint Computer Conference, 41, 1972, pp. 611-621.
21. Freiburger, W. F., U. Grenader, and P. D. Sampson, "Patterns in Program References," IBM Journal of Research and Development, 19, 3, 1975, pp. 230-243.
22. Courtois, P.-J., and H. Vantilborgh, "A Decomposable Model of Program Behavior," Acta Informatica, 6, 3, 1976, pp. 251-275.
23. Innes, D. R., "Exploiting the Least Recently Used Page Replacement Algorithm," Software-Practice and Experience, 7, 2, 1977, pp. 271-273.
24. Prieve, B. G., and R. S. Fabry, "VMIN—An Optimal Variable-Space Page Replacement Algorithm," Communications of the ACM, 19, 5, 1976, pp. 295-297.
25. Gupta, R. K., and M. A. Franklin, "Working Set and Page Fault Frequency Paging Algorithms: A Performance Comparison," IEEE Transactions on Computers, C-27, 8, 1978, pp. 706-712.

26. Chu, W. W., and H. Opderbeck, "Analysis of the PFF Replacement Algorithm via a Semi-Markov Model," Communications of the ACM, 19, 5, 1976, pp. 298-304.
27. Bard, Y., "Characterization of Program Paging in a Time-Sharing Environment," IBM Journal of Research and Development, 17, 5, 1973, pp. 387-393.
28. Bard, Y., "Application of the Page Survival Index (PSI) to Virtual Memory System Performance," IBM Journal of Research and Development, 19, 3, 1975, pp. 212-220.
29. Levy, H. M., and R. H. Eckhouse, Jr., Computer Programming and Architecture: The VAX-11, Digital Press, 1980.
30. Bryant, P., "Predicting Working Set Sizes," IBM Journal of Research and Development, 19, 3, 1975, pp. 221-229.
31. Belady, L. A., R. A. Nelson, and G. S. Shedler, "An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine," Communications of the ACM, 12, 6, 1969, pp. 349-353.
32. Franklin, M. A., and R. K. Gupta, "Computation of Page Fault Probability from Program Transition Diagram," Communications of the ACM, 17, 4, 1974, pp. 186-191.
33. Aho, A. V., P. J. Denning, and J. D. Ullman, "Principles of Optimal Page Replacement," Journal of the ACM, 18, 1, 1971, pp. 80-93. ✓
34. Brandwajn, A., "A Model of a Virtual Memory System," Acta Informatica, 6, 4, 1976, pp. 365-386.
35. Chamberlin, D. D., S. H. Fuller and L. Y. Liu, "An Analysis of Page Allocation Strategies for Multiprogramming Systems with Virtual Memory," IBM Journal of Research and Development, 17, 5, 1973, pp. 404-412.
36. Bard, Y., "Experimental Evaluation of System Performance," IBM Systems Journal, 12, 3, 1973, pp. 302-314.
37. Bard, Y., "Performance Analysis of Virtual Memory Time-Sharing Systems," IBM Systems Journal, 14, 4, 1975, pp. 366-384.
38. Anderson, H. A., Jr., M. Reiser, and G. L. Galati, "Tuning a Virtual Storage System," IBM Systems Journal, 14, 3, 1975, pp. 246-263.

39. Forsythe, G. E., M. A. Malcolm, and C. B. Moler, Computer Methods for Mathematical Computations, Prentice-Hall, Inc., 1977.
40. Knuth, D. E., The Art of Computer Programming: Seminumerical Algorithms, 2, Addison-Wesley, 1969.
41. Madnick, S. E., and J. J. Donovan, Operating Systems, McGraw-Hill, 1974.
42. Smith, A. J., "Bibliography on Paging and Related Topics," ACM Operating Systems Review, 12, 4, 1978, pp. 39-56.

APPENDICES

APPENDIX A

TABULAR RESULTS FOR SIMULATIONS

Table A.1. Overall reference success percentages for simulations involving the RAND page replacement algorithm.

		RAND Local	RAND Global
30 page frames	I/O-bound job mix	45.718	76.624
	heterogeneous job mix	45.000	82.000
	processor-bound job mix	45.833	85.272
50 page frames	I/O-bound job mix	69.655	82.500
	heterogeneous job mix	69.271	86.260
	processor-bound job mix	69.663	88.742
70 page frames	I/O-bound job mix	85.920	88.534
	heterogenous job mix	85.906	90.146
	processor-bound job mix	86.699	91.787

Table A.2. Overall reference success percentages for simulations involving the FIFO page replacement algorithm.

		FIFO Local	FIFO Global
30 page frames	I/O-bound job mix	45.977	76.408
	heterogeneous job mix	45.552	82.312
	processor-bound job mix	46.498	85.841
50 page frames	I/O-bound job mix	70.963	76.408
	heterogeneous job mix	70.260	82.312
	processor-bound job mix	71.074	85.841
70 page frames	I/O-bound job mix	89.483	88.793
	heterogeneous job mix	88.510	90.479
	processor-bound job mix	89.103	91.538

Table A.3. Overall reference success percentages for simulations involving the LRU page replacement algorithm.

		LRU Local	LRU Global
30 page frames	I/O-bound job mix	46.724	76.408
	heterogeneous job mix	46.927	82.312
	processor-bound job mix	47.692	85.841
50 page frames	I/O-bound job mix	73.807	76.437
	heterogeneous job mix	73.542	82.312
	processor-bound job mix	74.223	85.841
70 page frames	I/O-bound job mix	90.374	91.638
	heterogeneous job mix	90.469	92.833
	processor-bound job mix	90.986	92.332

APPENDIX B

FORMAT OF SIMULATOR INPUT

The parameters for each simulation must be read from the SYSIN input file. Almost all of the data items are "free format;" i.e., the items need not reside in particular columns of the card image input. However, column one has a special significance: if an asterisk appears in this column, the rest of the card image is flagged as a comment, and the remaining 79 columns are ignored. Otherwise column one must remain blank. Comment cards improve data readability, and may be inserted anywhere the user desires except where noted below.

1. Number of title lines card—A positive integer (NTITLE) on this card signifies the number of title cards that describe this simulation. The maximum value is 10.
2. Title card(s)—NTITLE card(s) of simulation title.
3. Physical memory size card—A positive integer (NSYSPAG) relates the number of physical page frames for this simulation. The maximum is 100.
4. Page replacement algorithm card—A character string (PRALG) specifies the type of page replacement algorithm to be used in this simulation. Acceptable values are 'RAND', 'FIFO', or 'LRU'. The single quotes must be supplied to delimit the character string.

5. Implementation card—A character string (ALGTYPE)

designates the type of implementation of PRALG.

Acceptable values are 'LOCAL', or 'GLOBAL'. The single quotes must be supplied to delimit the character string.

6. Number of cycles card—A positive integer (NCYCLES)

denotes the duration of this simulation in terms of the number of complete processor dispatch cycles.

Each user in the simulation will be dispatched NCYCLES times.

7. Number of users card—A positive integer (NUSERS)

specifies the number of eligible users in this simulation. The maximum value is 10.

(There are NUSERS sets of the following cards)

7.1 User job class card—A positive integer (JOBCLAS)

relates the relative I/O-boundedness or processor-boundedness of this user. If JOBCLAS is 1, the user is extremely I/O-bound; if JOBCLAS is 10, the user is heavily processor-bound. Any integer between these extremes is also acceptable.

7.2 User virtual pages card—A positive integer (NUSRPAG)

designates the number of virtual pages in this program. The maximum value is 20.

7.3 User activity sets card—A positive integer (REFSETS)

denotes the number of activity sets for this user.

The maximum number of activity sets is 12.

(There are REFSETS sets of the following cards for each user)

7.3.1 Locality interval card—A positive integer (REFCHNG)

specifies the lifetime of this activity set in terms of complete processor dispatch cycles.

7.3.2 Reference percentage cards—NUSRPAG floating point

values denote the reference frequency percentage of each virtual page. There must be NUSRPAG values, even if the activity set described is smaller than NUSRPAG. For example, if NUSRPAG is six, and the activity set consists of only four pages, the percentages could be coded as 25.0, 25.0, 25.0, 25.0, 0.0, and 0.0. The total of these values must be greater than 98.0 and less than 102.0.

The simulator user may designate pages in the initial activity set to be resident in memory when the simulation begins, in order to deemphasize the effects of startup faulting. To do this, simply specify the reference percentages of the pages to be resident as negative.

Comment cards may not be interspersed with this data.

VITA

James William Teague II was born in Oak Ridge, Tennessee on July 13, 1956. He attended elementary schools in that city and was graduated from Oak Ridge High School in June 1974. The same month he entered East Tennessee State University, and in March 1978 he received a Bachelor of Science degree in Computer Science. In April 1978 he accepted employment with the Union Carbide Corporation Nuclear Division as a Computing Analyst.

He entered the Graduate School of The University of Tennessee, Knoxville, in September 1978 as a part time student while still employed with Union Carbide. In September 1980 he took an educational leave of absence from Union Carbide to accept a Graduate Teaching Assistantship with The University of Tennessee and attend school full time. He received the Master of Science degree with a major in Computer Science in August 1981.

The author is a member of Phi Kappa Phi, Kappa Mu Epsilon, Gamma Beta Phi, the Association for Computing Machinery, and the Institute of Electrical and Electronics Engineers. Mr. Teague will be employed with Digital Equipment Corporation in Nashua, New Hampshire after graduation.