THE EFFECT OF PROGRAM STRUCTURE ON PROGRAM BEHAVIOUR IN VIRTUAL MEMORY SYSTEMS

COLIN T. SCOTT

Submitted for the degree of
Doctor of Philosophy
of the
University of Edinburgh
1980

During part of the period of research for this thesis I was supported by a research studentship from the Science Research Council. The work is my own, except where explicit acknowledgement is made in the text, and this thesis has been composed by myself.

I would like to thank my various supervisors, Professor S. Michaelson, Mr J. Ellenby and Dr A. Wight for the assistance they gave me throughout this period of research. Finally I would like to thank my mother for her expert typing and Anne for all her patience and encouragement.

ABSTRACT

The background, both in terms of theory and practice, to current memory management systems is presented. It is suggested that current paged memory management systems have serious operational deficiencies, particularly with respect to the behaviour of page replacement algorithms. Examples of these operational deficiencies are presented.

Consequently, an alternative approach to memory management, based on the notion of a segment, is developed. In this system, the segments are determined at compile time based on a knowledge of the structure of the high-level language program. This segment information is passed to the run-time system which uses this information as the basis of its memory allocation policy.

An experimental implementation of such a system for PASCAL programs has been achieved and results from this system are presented.

CONTENTS

1. INTRODUCTION

The behaviour of programs with respect to their residency in a storage hierarchy, even in the most restricted case of a two-level system, has provoked a great deal of research. General characteristics of program behaviour have been proposed and these have, within the context of demand paging systems, stimulated the development of replacement algorithms which depend upon some subset of these characteristics.

The work reported in this thesis calls into question deductions made from these characteristics, and aims to show that real programs can frequently burst out of the restrictions that are theoretically imposed upon them. This leads to poor program behaviour and a general reduction in efficiency of computer systems using such algorithms.

The major flaw in such approaches is that programs are considered to be relatively unstructured "black-boxes" which generate storage references in some predictable but poorly-understood fashion. It is the major contention of this thesis that programs do currently, and, with developments in programming languages, will in the future, show distinct structure which is known at compile time. This thesis maintains that, if such structural information can be passed to the run-time system, then that system can satisfactorily tailor itself to the needs of the running programs. Such a system is adaptable to each program currently running on it, and does not attempt to fit each program into the strait-jacket of "average" or "normal" behaviour.

The structural information gained could, it is suggested, be incorporated into a more general form of the capability (Den 66) called the "operational capability". This, it is suggested, is a unifying concept which creates an efficient run-time environment for programs.

The remaining chapters of this thesis are as follows:-

Chapter 2	-	Historical Background
Chapter 3	-	Theoretical Development
Chapter 4	-	Behavioural Characteristics of Conventional Memory Management Systems
Chapter 5	-	A proposal for Memory Management Systems based on a Knowledge of Program Structure
Chapter 6	-	An Experimental Implementation
Chapter 7	-	Results
Chapter 8	-	Conclusions
Chapter 9	-	References

2. HISTORICAL DEVELOPMENT

2.1 Introduction

In this chapter the origins of the current state-of-the-art in automatic memory management are traced. This function is carried out in an environment which consists of a paged virtual memory space, filled by page-on-demand strategies and freed by a standard page-replacement algorithm.

2.2 The Problem

Hansen (Han 73) states that:-

"Store management raises three basic questions:

1. What is the appropriate unit of storage to assign to computations?
2. How are these units placed in an internal store prior to their use?
3. How are they referenced by computations during execution?"

Following Hansen a number of features of storage systems can be identified. Firstly, to the user of a high-level language a virtual store exists. This consists of data identified (or addressed) by text strings called identifiers. Consequently a virtual store can be considered to be a mapping of identifiers into values:

$$\text{Virtual store: identifier} \rightarrow \text{value} \qquad (2.1)$$

On the other hand, the physical store is made up of locations identified by consecutive numbers called addresses.

Since these locations hold values of one form or another, physical store can be thought of as a mapping from addresses to values:

physical store: address $\rightarrow$ value (2.2)

In order to complete the link between the user and the "real machine" some process must be carried out, before a program is run, which associates identifiers with addresses. This is the store allocation process which defines an intermediate mapping of identifiers into addresses:

store allocation: identifier $\rightarrow$ address (2.3)

These three mappings are of fundamental importance to the storage management process.

Mapping (2.2) is clearly outwith the control of the software designer, yet what is provided at this hardware level has a significant effect on what can be achieved by systems programmers and user programmers alike. In this field alone, variations exist from the potentially bit-addressable B1700 (Wil 72) to the 512-bit storage accesses performed in CDC Star (Pur 74).

However if all the above factors are considered, a significant amount of useful information about the storage management task can be obtained.

2.3 The Appropriate Unit of Storage to Assign to Computations - Early History

The simplest answer to this problem was to assign the whole of the available store to a computation. This approach, attractive in its simplicity, had a number of disadvantages. Firstly store was almost invariably wasted. If a sufficient amount of store was to be available for most problems, particularly the relatively large ones, then for the smaller problems during their running time (possibly large) amounts of store were unused. Even in modern storage hierarchies such wastefulness of a relatively expensive resource would not be tolerated. Equally, the need to deal with large problems meant that large amounts of store had to be available, thereby compounding the first problem.

Two problems arose from this technique:-

1. How to deal with wasted space in store?
2. How to accommodate large programs whose total memory requirements were larger than the available main storage space?

The simplest solution to the first problem was by means of partitions of main store. This technique, used in OS 360 MFT (IBM 71), operated as follows:-

Any memory not used by the control program was divided into partitions (see Figure 2.1).

Figure 2.1

The size of each partition was set by the operator and its associated priority was determined by its position relative to other partitions. PO was reserved for jobs of the highest priority, while Pn was reserved for jobs of the lowest priority. When a job was initiated it was allocated a partition for the class of the job.

This technique has the advantage of allowing multi-programming, but does not successfully overcome the problem of wasted memory space. With this technique, jobs do not normally fill their allocated partition completely and consequently, as with the whole memory approach storage is wasted.

The second problem, i.e. how to accommodate programs whose storage requirements were greater than the available storage, was first solved using overlays.

This method requires that the programmer divides his program into sections, one of which must be designated the main section. The remaining sections are called dependent sections. By using a linking loader, the main section at run time could call in the dependent sections for execution. By placing the main section in the available memory and sharing the rest of the memory among the dependent sections, the main section can replace dependent sections when they are no longer needed with other dependent sections. This technique was used in operating systems for CDC-6000, UNIVAC 1108, GE 635, and IBM System 360 (Lan 69).

This technique allows a user to utilise small amounts of physical storage for large programs. However, there are some significant drawbacks:-

1. The user is responsible for the division of his program into its main and dependent sections.
2. Careful job preparation is required, so that the relation between the main section and its dependents is clear.
3. References between dependent sections should be minimised.
4. The amount of main store allocated to the main section and its dependents is fixed during the <u>entire</u> execution period. This implies that dynamic space variations cannot be utilised and that the maximum amount of memory required be allocated initially.
5. All of the sections of a job must be available at linkage time.

This overlaying technique was most severely criticised by Sayre (Say 69), who compared results by Brawn et al. (Bra 68) and measurements on a <u>demand paging</u> unit built by Belady et al., against manual "folding" techniques (such as overlaying) and concluded that

> "... a folding mechanism will probably become a normal part of most computing systems"

Sayre gives six reasons for his support of "automatic folding":-

1. Programming Cost I

   Manual folding is difficult to do and get right.

2. Programming Cost II

   Once folded for a particular size of memory, a program will not run efficiently in another size of memory.

3. Multiprogramming and Timesharing

   Once folded, a program <u>must</u> have the size of memory it was folded for. This is not a good starting point for systems which involve the dynamic sharing of memory among programs.

4. System Availability

   Since a pre-folded program must have the memory it was folded for, this will be a significant drawback if that amount of memory is temporarily unavailable due to system failures.

5. Design Predictability

   The performance of a program will depend critically on how well it is folded.

6. Retention of Technical Options

   The large amount of investment in pre-folded programs does not take account of technological advance (for example) making more memory available.

2.4 The Appropriate Unit of Storage to Assign to Computations - Pages and Segments

Largely due to the unsatisfactory nature of overlaying and partitioning - static memory allocation - other techniques, collectively known as dynamic memory allocation, were being developed. These were based on two units:-

1. Pages
2. Segments.

2.4.1 Pages

The aim of this technique was to ease a number of the problems mentioned above by dividing a program's address space into equal fixed-size areas called pages. Main store was also divided into identical fixed-size areas called page frames. A number of page frames would be allocated to a program during its run and these would be filled with program pages as necessary. All addressing was done in terms of these pages.

With the adoption of this technique, the pages belonging to a program could become scattered throughout store in order to take advantage of any unused page frames that might become available. This meant that the addresses used by the program (virtual addresses) had to be translated into the correct physical addresses before they were used to access the main store.

Figure 2.2 shows a possible arrangement for address mapping under a paging scheme. Each user has a page table which contains an entry for each of the user's pages. If a page is in main store then the main store address is included in this entry, whereas if it is currently resident in backing store then the backing store address is given.

An address within a program is of the form of a pair:-

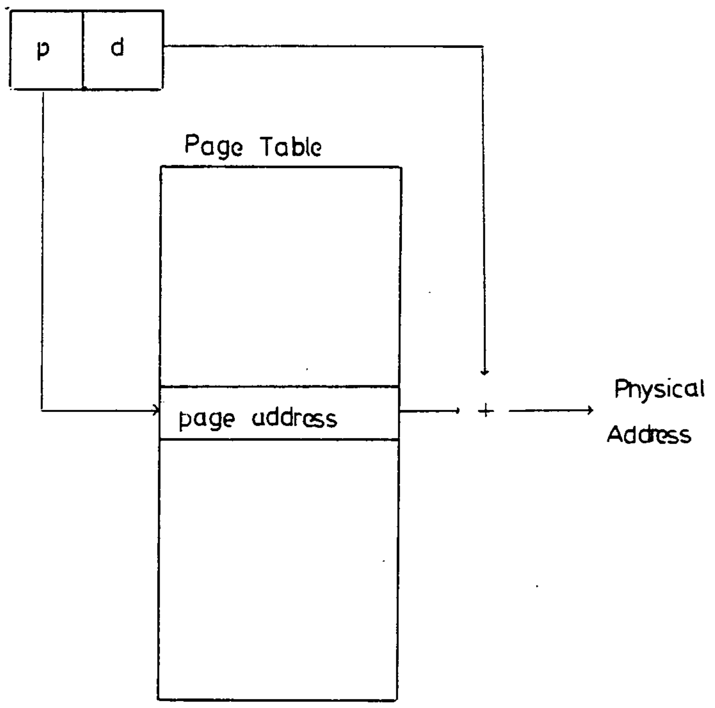


Figure 2.2

<page no. , displacement>

where the displacement gives the position of the addressed item within the specified page. To access any item, the entry for the specified page is examined in the page table. If that page is in the main store then the main store address of the start of the page has the displacement added to it to give the physical address of the item. If the page is not in main store, then it is brought into some free position in main store and the page table updated accordingly. The above procedure is then followed to obtain the desired physical address. In practice, the page table itself may be held in main store and each user will be given a hardware register to indicate the base of his page table. Such an addressing scheme is usually performed by hardware, but even so this results in two store accesses for every word accessed (one for the page table and one for the word itself). Some computer systems overcome this by holding current page table entries in associative memories and attempting to ensure that all current page descriptors are in these memories at all times.

This technique was introduced on the Atlas computer (Kil 62). In this system main store was divided into blocks of 512 words. This system also yielded one of the earliest page replacement algorithms which will be discussed later.

Within a paged system placement of pages in main store is not a significant problem, since any freed page frame will accommodate any page. The real problems are:-

a) when to bring pages into store, and

b) what page should be removed, if necessary, to accommodate the new page.

The simplest way to deal with a) is to page on demand. That is to say, a page is brought into store if and when a program requests an item on that page. Consequently, the most significant study refers to page replacement algorithms. The case in support of the use of demand paging algorithms will be put forward in Chapter 3 and this chapter restricts itself to a study of available page replacement techniques.

2.4.1.1 Atlas Loop Detection

This technique, described by Baylis et al. (Bay 68) assumes a strictly cyclic pattern of use of the blocks (pages) within a given program. For each page of store two parameters are computed:-

t - the time the block has been idle in core store since last being accessed,

T - the total time the block remained idle the last time it was written to backing store.

Measurement of both t and T are made in terms of process time. The implication of the cyclic strategy is that if $t > T$ then the block is no longer in use in the current cycle and can be written out to backing store. If no block satisfies this property then the block (excluding the current blocks) with the largest $(T - t)$ is the best candidate for replacement.

In the studies presented (Bay 68) this algorithm was compared with two others:-

a) selecting a candidate for replacement at random

b) selecting the page with the largest t.

It appeared from the studies that, <u>overall</u>, the system behaved best under (b) and that the loop detection method was about 10% worse than (b). However this was explained by the non-cyclic nature of the Supervisor program which was also included in the study. The feeling of the study was that, although the cyclic strategy was inferior to (b), strategy (b) penalised programs with cyclic behaviour to such an extent that the loop detection method represented the "safest" approach. This was particularly true if, as was thought likely, cyclic programs could dominate the job mix over a period of time.

The authors did conclude that applying an algorithm based on store usage was worthwhile but that the particular algorithm had only marginal effect.

2.4.1.2 Least Recently Used (LRU)

This method is exactly the alternative (b) mentioned above. That is to say, the page replaced is that page that has remained unreferenced for the longest time.

Two types of LRU can be distinguished:-

a) Global LRU - The replaced page is that page which has not been referenced for the longest period of <u>real</u> time, regardless of the task to which it belongs. This technique has been used in CP/67 (Ale 69), (Bay 68b), Multics (Org 69), MTS (Ale 69), VS1 (IBM), VS2 (IBM).

b) Local LRU - This allocates a <u>fixed</u> number of memory pages per task. The least recently used selection is made from pages belonging to the task which generated the page fault. This has been implemented in the original IBM version of TSS (IBM 70).

2.4.1.3 First-in-First-out (FIFO)

This is probably one of the simplest algorithms to implement, the page which has resided in main store for the longest time is chosen to be replaced. This technique has been used on the B5500 (Bat 69). Belady (Bel 66) has shown that this algorithm can behave quite well in most cases. However, it is possible (Bel 69b) that it will <u>increase</u> the number of page transfers made by a program when the main store made available to that program is <u>increased</u>.

2.4.1.4 Working Set Algorithm

This technique developed by Denning (Den 70) involves the examination of the pages that have been referenced in a fixed process time interval before the current reference.

All pages in this set, known as the Working Set of the program, remain in core. All others are marked as candidates for removal. Some implementations of this technique exist where the replaced page is the least recently used page which does not belong to the Working Set of any program (Doh 70).

This algorithm, as will be shown below, has been extensively analysed and with LRU forms the basis for much of the work done on the analysis of program behaviour.

2.4.1.5 Page Faulty Frequency Algorithm (PFF)

This algorithm was first suggested by Chu and Opderbeck (Chu 72). It attempts to dynamically control the rate of page faults by varying the memory space allocated to a program.

The PFF algorithm measures the inter-page fault intervals during execution of the program. At page fault times, it compares these intervals with previously selected threshold T. If the inter-page fault time exceeds T then all the pages in main memory belonging to the program that have not been referenced since the last page fault are candidates for removal. Otherwise no page is removed and the program's allocation is increased by one page.

A modification to this algorithm was suggested by Sadeh (Sad 75) wherein a program is prevented from collecting all its pages in main memory (otherwise no page faults would be generated during its remaining execution). This is achieved by placing a limit, z, on the inter page fault interval. Whenever this limit is reached a memory allocation is made without waiting for a page fault to occur.

The operation of this algorithm is described in (Chu 76) among others.

These, then, represent the major page replacement algorithms that have been proposed and studied.

2.4.1.6 Other Techniques

A mention must be made at this point of pre-paging. The aim of this technique, wherein a page is brought into main store before it is referenced, is to reduce or eliminate page waits so that CPU utilisation can increase. Pre-paging involves a balance between initiating the page fetch early enough to overcome the delays involved in the use of backing store with high latency periods, and initiating the fetch late enough to ensure that the page does not wait around in memory for a significant period before it is referenced (if at all). Studies on EMAS (Whi 73) by Adams (Ada 75), (Ada 76) indicate that pre-paging does tend to outweigh any disadvantages caused by moving in unwanted pages. On the other hand, Hoare and McKeag (Hoa 72) concluded that pre-paging is not only difficult to use but may be actually prejudicial in its effect on system performance.

Another technique worthy of mention at this point is that of page recapture. The idea behind this approach is that when a page replacement algorithm marks a page as free, it may be some time before that page is actually overwritten by an incoming page. This is due to the fact that many algorithms free a number of page frames when, perhaps, only one is needed at that time. Consequently, the system remembers what the contents of a page frame are, whether that page is marked as free or not. It has been shown in the studies by Adams mentioned above, that recapture can play a significant role in the operation of a system. It is only fair to point out however, that any success that recapture might display tends to imply the failure of the replacement algorithm in that pages are being marked as free (and consequently not needed) only to be needed again after a very short time.

2.4.2 Segments

As will be shown in Chapter 3, one of the major problems with paging systems is the choice of an appropriate page size. Again the problem is a matter of balancing conflicting requirements:-

1. If the page size is too small:-
    - i) the size of the page tables increases and this implies a loss of main memory space, if the tables are held in store (Table fragmentation).

ii) the unit of transfer chosen may be inappropriate for the devices involved.

iii) if large amounts of store are required, a comparatively large number of pages must be freed and transferred, thus increasing the system overheads.

2. If the page size is too large:-

    i) the region of the store required by a program may be considerably less than a page, but a whole page must be allocated to it. This results in a waste of space within pages (Internal fragmentation).

These matters are dealt with by Randell (Ran 69). In this paper he suggests that few designers have reduced single page sizes below 1024 words because of the overheads involved in storing and processing page tables. However it transpires that the logical unit of transfer (the segment) can frequently be small (eg 60 words (McK 67)). Although this mean is small, the variation appears to be quite large and some designers have provided two page sizes (Cor 65) to attempt to attack this problem. Randell further comments, however, that compilers and programming conventions are likely to have a considerable effect on the mean segment size (but less likely to remove the problem of the variation in sizes). This remark will be considered at a later stage.

To return to the notion of segmentation as such, this technique was introduced by Dennis (Den 65). He introduced the concept of a <u>name space</u>, that is, the set of addresses a program can generate, and contrasted with this the <u>memory space</u> of physical memory locations that are accessible to a program. Dennis proposed that:-

1. A computation should have the use of a name space sufficiently large that all information it references may be assigned unique names, and such that the re-allocation of information within its name space is never necessary.
2. Data objects of a computation should be expandable without re-allocation of name space.
3. Information referenced in common by several computations should have the same name for all computations that reference it.
4. A protection mechanism should operate in name space to permit access by a computation only in an authorised manner.

Dennis claimed that this could be achieved by a sytem in which information was addressed by a two component address:-

<segment name, word address>

A segment is an ordered collection of words with an associated segment name.

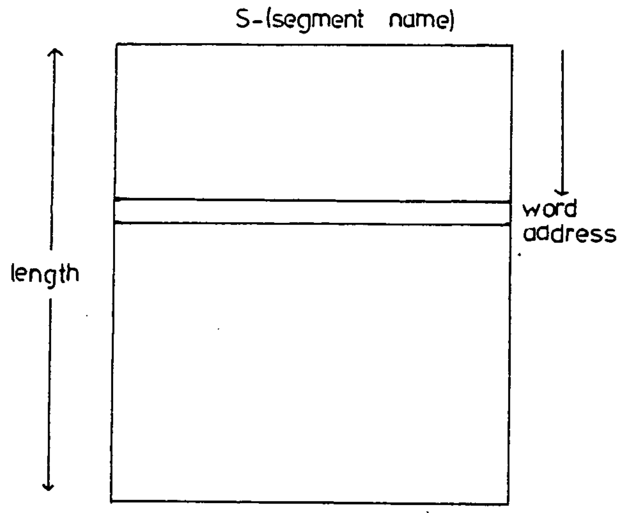


Diagram 2.3

A particular word in a given segment S is accessed as shown in diagram 2.3.

To use such a system, programs and data are split into segments which consist of related information. In much the same fashion as a paging system, the base addresses of all segments belonging to a computation are kept in a single table called a segment table. More information must be retained than for a paged system because no limitation has been put on the length of such a segment. Consequently, the storage management system must have the segment length available to it at all times (see Diagram 2.4).

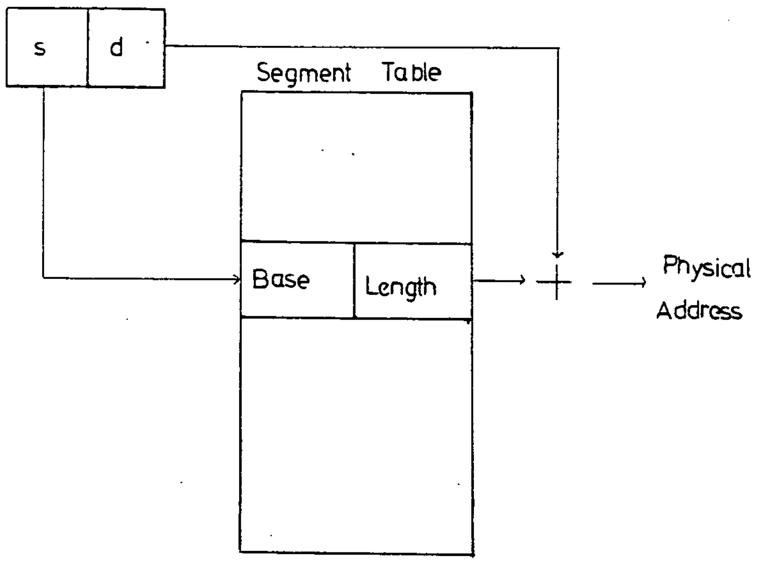


Diagram 2.4

A consequence of the choice of segments means that, in theory at least, such a system would not be susceptible to internal fragmentation. A proliferation of small segments would lead to the same table fragmentation as in a paged system. However such a system is prone to another form of fragmentation as will be shown below.

Unlike a paging system, the placement of segments poses a problem. With a paged system any free page frame can, by definition, accommodate any page. However, since segments are of variable size, the same is not true. Consequently, suitable space must be found in some other way for a desired segment in main store. A typical memory layout is shown in Diagram 2.5.

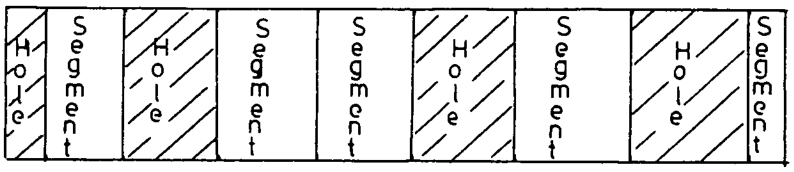


Diagram 2.5

This random pattern of holes and segments has been caused by the allocation and de-allocation of segments.

If a new segment is required in memory then, given that the length of a segment will be fixed during its lifetime, a suitable hole can be chosen by one of the following algorithms:-

First Fit	- a segment is placed in the first hole large enough to hold it,
Best Fit	- a segment is placed in the smallest hole capable of holding it.

Knuth (Knu 69) has shown that, contrary to expectation, the First Fit algorithm tends to be superior. As it also tends to be easier to implement it has been used in the B5500 MCP (McK 71). Knuth also suggested a third approach, known as the Buddy System, which involves maintaining holes of fixed sizes on lists. The sizes chosen are 2,4,8,....2^k words so that a 2^h hole can be split into two adjoining 2^{h-1} holes, and similarly two adjacent holes of the same size can if necessary be coalesced into one hole of the next larger size. This technique attempts to tailor the hole sizes to the requests that might be made on them. But it seems a rather complex task to maintain these lists in the appropriate fashion. However Knuth states that it does marginally outperform the other two techniques mentioned above.

A problem with segmentation is that small holes tend to proliferate and there comes a point at which it is impossible to find a suitable hole for a required segment, although the total free space is sufficient to meet its needs. This loss of space has been called External Fragmentation (Ran 69) and can be overcome by moving all used segments to one end of store (see Diagram 2.6). This technique is known as compaction.

Compaction is a time-consuming business since large amounts of information must be moved from one place in store to another (see Chapter 3). It is suggested that in a well-designed system compaction occurs so rarely that processor time spent on this relocation is negligible.

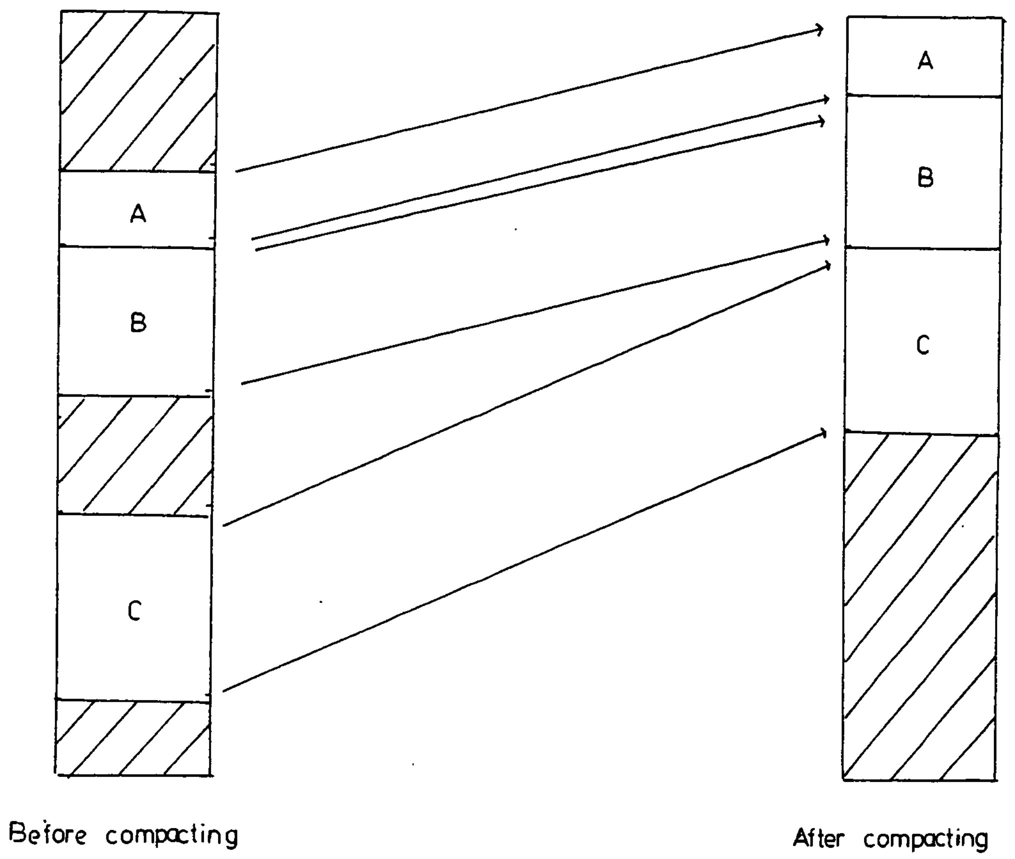


Diagram 2.6

Both paging and segmentation have their drawbacks in the utilisation of main storage. However attempts have been made to combine the best of both systems.

2.4.3 Paging and Segmentation

Such a combined technique has been proposed by Arden et al. (Ard 66). This system involves a three-component address for informations:-

<Segment Number,Page Number,Address>

The requirements for such a system are shown in Diagram 2.7. The segment table is defined by a Segment Table Register which contains the Segment Table Length (STL) and the address of the

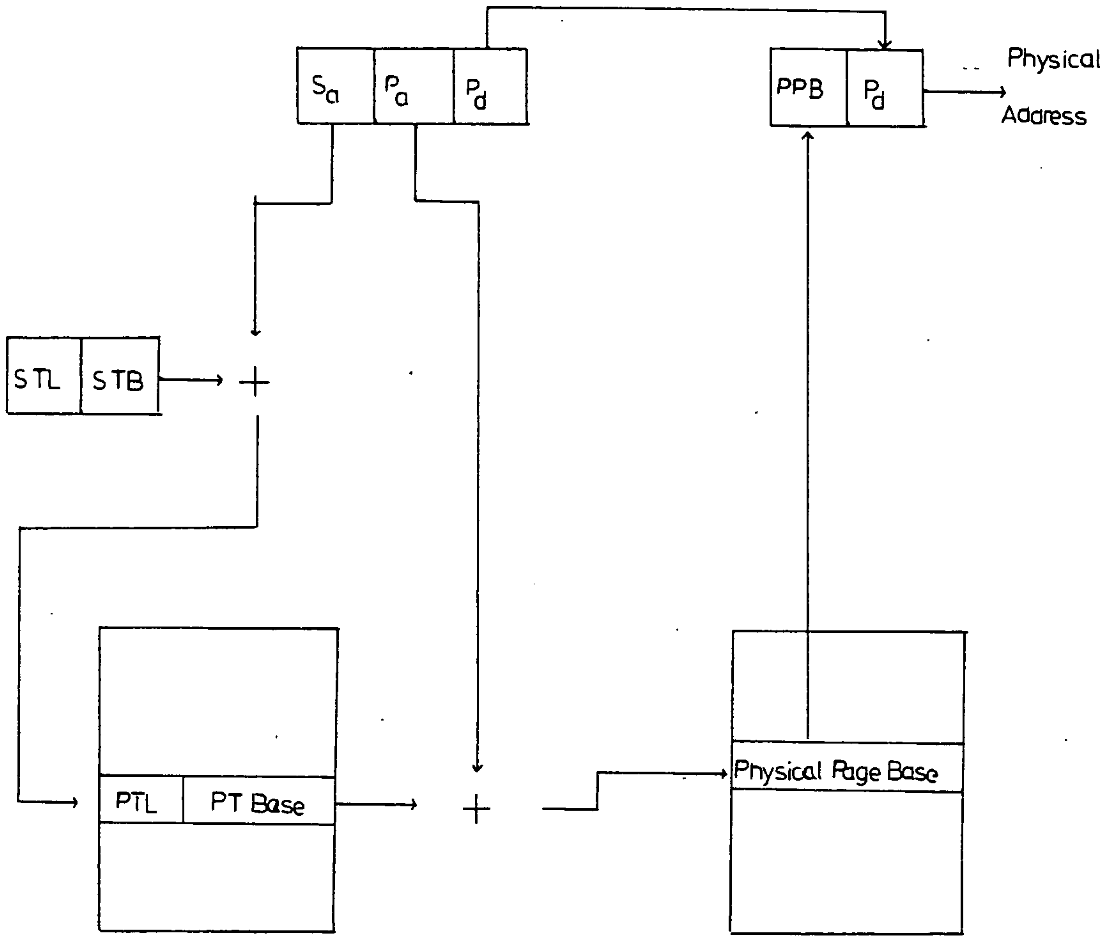


Diagram 2.7

Segment Table Base (STB). The required segment is used as an index to the Segment table whose entries consist of a pair:-

<Page Table Length, Page Table Base>.

The Page Table Base is the base address of the required page table. The required page is then used as an index to the page table which contains the base address of the required page. This base address then has the displacement added to it in the usual way to provide the final physical address of the required information.

2.5 Conclusion

This chapter has attempted to trace the development of memory management systems to the present day. Some indications of the reasons for the development of the current demand paged systems have been given. Much more of the motivation for the choices that have been made lies in the theoretical analysis of program and paging behaviour that has also developed. This is considered in the next chapter.

It is also useful to note, at this stage, that good theoretical analysis is of vital importance in this field. It is often the case that, despite the apparent simplicity of the techniques described above, implementation may be difficult and costly in real systems. Also, it is true that it is difficult to evaluate the benefits that may accrue from these features <u>alone</u> in real situations.

3. THEORETICAL DEVELOPMENT

3.1 Introduction

In this chapter the theoretical development of current storage management systems is investigated. This theoretical work has tended to be carried out in parallel with the actual implementation of the techniques, and this has, perhaps, overly restricted the areas of theoretical study. However in this chapter the arguments for the conventional approaches to storage management are put forward.

3.2 Storage Utilisation in Segmented Systems

In his paper on virtual memory (Den 70), probably the most influential paper in this area, Denning identifies three policies that must be considered in storage management systems:-

1. Replacement policies
   - which information is to be removed from memory.
2. Fetch policies
   - when information is to be loaded.
3. Placement policies
   - where information is to be put in memory.

Replacement and fetch policies are much the same for paged and non-paged systems, but, as will be shown, placement policies for non-paged systems are considerably more complex than those for paged systems.

If a non-paged system is considered, two important results can be derived:-

PROPOSITION 3.1   The Fifty-Percent Rule

If a segmented memory system is in equilibrium having n segments and h holes (see Figure 3.1), where n and h are large, then h is approximately n/2.

PROOF

Consider an arbitary segment s, then it is necessary to find the probability, p, that this segment has a right neighbour. During the residency of a segment in store, half the transactions to the region on its right are insertions and half are deletions (because the system is an equilibrium). This implies

$$p = 1/2$$

$\Rightarrow$ No. of segments with holes as right neighbours

$$= np = n/2$$

$\Rightarrow$ No. of holes = n/2.

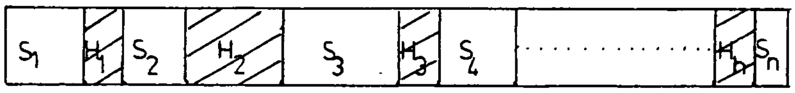


Figure 3.1

PROPOSITION 3.2 The Unused Memory Rule

If a segmented memory system is in equilibrium and

f = the fraction of memory occupied by holes

s_o = the average segment size

ks_o = the lower bound on the average hole size ($k \geq 0$)

then

$f \geq k/(k+2)$

PROOF

Let the memory size = m words. By Proposition 3.1, if there are n segments in memory then there are n/2 holes. The total amount of space occupied by holes is

$$m - ns_o$$

and the average space occupied per hole is therefore

$$2(m - ns_o)/n$$

Now since it has been assumed that

$$2(m - ns_o)/n \geq ks_o$$

$$\Rightarrow (n/m)s_o \leq 2/(k+2)$$

$$\Rightarrow f = (m - ns_o)/m = 1 - (n/m)s_o$$

$$\geq 1 - 2/(k+2) = k/(k+2).$$

Diagram 3.2 shows the relationship between f and k graphically. The curve in the diagram represents a lower bound on the fraction, f, of unused memory. It can be seen that as the average hole size becomes large with respect to the average segment size, i.e. $k \to \infty$, then so the fraction of unused memory becomes large $f \to 1$.

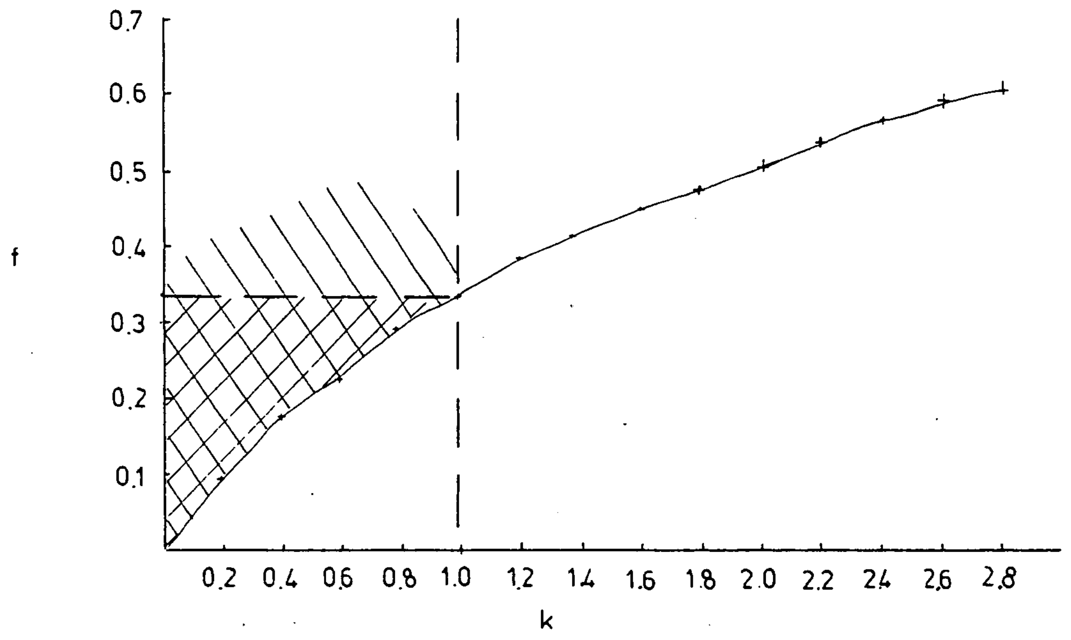


Diagram 3.2

situation is thus, for large k, we have a number of holes, n/2, whose average size is considerably greater than the average segment size.

Two states may be distinguished:-

1. There is insufficient work waiting in the system, consequently memory is under-utilised. Herein a large f is reasonable.
2. There is sufficient work waiting in the system. If this work has the same segment size profile, then it would seem to be reasonable that there are segments waiting to be loaded which will fit into some of the available holes.

In this latter case, the action of loading another segment reduces the average hole size.

Since, if n increases to n', and the amount of memory allocated increases from ns_o to $n's_o$, then the amount of memory unused decreases from $m - ns_o$ to $m - n's_o$. Consequently the average hole size decreases from $2(m - ns_o)/n'$.

It is clear therefore that the average hole size, e_o , must lie in the range

$$0 \leq e_o < s_o \qquad (3.1)$$

otherwise case 2 above applies. Consequently an <u>upper</u> bound can be placed on the average hole size. Hence

$$ks_o \leq e_o < s_o$$

$$0 \leq k < 1.$$

Thus k must be restricted to the range shown in Diagram 3.2.

Using equation 3.1, the following may be derived:-

Given $e_o < s_o$

and $m = ns_o + (n/2)e_o$

then $m < ns_o + (n/2)s_o$

$\Rightarrow$ $m < (3ns_o)/2$

$\Rightarrow$ $s_o > (2m)/(3n)$

Now since

$$f = (m-ns_o)/m$$

$$\Rightarrow \quad f < (m-(n(2m)/3n\ )/m \; = \; 1-2/3 \; = 1/3$$

Consequently, in practice a management system can achieve

$$k/(k+2) \leq f < 1/3 \qquad (3.2)$$

This is the area shown by the cross-hatching in Diagram 3.2

Three placement algorithms were considered by Knuth (Knu 68), which have been discussed in 2.4.2 above.

The following compaction result is reported by Denning (Den 70):-

PROPOSITION 3.3   Compaction Result

Suppose a non-paged memory system is in equilibrium immediately after compaction, a fraction f of memory being unused; suppose that each segment is referenced an average r times before being deleted and that the average segment size is s_o. Then the fraction F of the time the system spends on compaction satisfies

$$F \geqslant (1 - f)/(1 - f + (f/2)(r/s_o))$$

PROOF

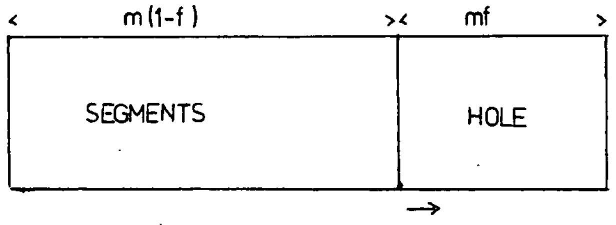


Diagram 3.3

Diagram 3.3 shows the memory state immediately after compaction.

If it is assumed that a segment is referenced each time unit, then a segment is deleted every r time units, and, since the system is in equilibrium, a new segment is inserted every r time units. Consequently, the boundary moves at the rate s_o/r.

The system will operate happily for $t_o = fmr/s_o$, the time for the boundary to cross the hole.

Since the compaction operation requires at least two operations for each of the $(1 - f)m$ words to be moved, then t_c, the time taken for compaction satisfies:-

$$t_c \geqslant 2(1 - f)m$$

Consequently, the time spent compacting as a fraction of the total time is

$$F = 1 - t_o/(t_o + t_c)$$

$$\Rightarrow \quad F \geqslant 1 - t_o/(t_o + 2(1 - f)m\ )$$

$$\Rightarrow \quad F \geqslant (t_o + 2(\ 1 - f)m - t_o)/(t_o + 2(1 - f)m\ )$$

$$\Rightarrow \quad F \geqslant (\ 1 - f)(2m/(fmr/s_o + 2(\ 1 - f)m)\ )$$

$$\Rightarrow \quad F \geqslant (\ 1 - f)/(\ (\ 1 - f) + (f/2)(r/s_o)\ )$$

Diagram 3.4 shows a plot of F against f

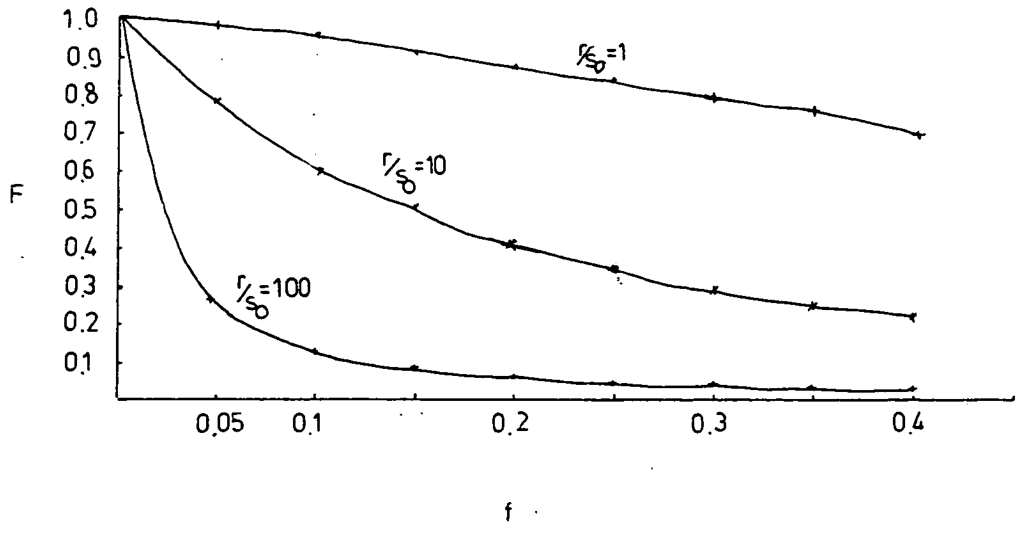


Diagram 3.4

The use of relation 3.2 has enabled the range of f to be considerably reduced over that presented by Denning.

It is perhaps even more clear now, in diagram 3.3, that only in a situation where compaction is carried out relatively infrequently due to <u>high reference density</u> in segments can compaction be tolerated (i.e. r/s_o large).

Denning because of the overhead of compaction and the possibility of a large amount of unused memory (without the benefit of relation 3.2), discounts segmentation and turns to paged systems.

3.3 Paged Systems - Page Size

The simplicity of paged systems in terms of their implementation and the consequently high number of successful implementations, has prompted much theoretical interest. Equally, theoretical investigations have shown that although the underlying idea is simple, what actually goes on in a paged system is not at all clear and model building is necessary in order to achieve some understanding of the real situation.

Placement policies as such have no relevance to paged systems since all that is required in order to place k pages is that k page frames be freed.

Using Denning (Den 70) again as a starting point, the following proposition is relevant:-

PROPOSITION 3.4 Optimal Page Size Result

Let z be the page size and s_o the average segment size; Suppose c_1 is the cost of losing a word to table fragmentation and c_2 is the cost of losing a word to internal fragmentation, and let $c = c_1/c_2$. If $z \ll s_o$ then the optimal page size is approximately $(2cs_o)^{\frac{1}{2}}$.

PROOF

The cost for any given z is

$c(z) = c_1 s_o/z + c_2 z/2$ (Since if $z \ll s$ then $z/2$ words will be wasted in internal fragmentation)

This has an optimal value when $c'(z) = 0$

$$\Rightarrow \quad -(c_1 s_o)/z_o^2 + c_2/2 = 0$$
$$\Rightarrow \quad z_o^2 = 2c_1 s_o/c_2$$
$$\Rightarrow \quad z_o = (2cs_o)^{\frac{1}{2}}$$

It is fairly reasonable to assume, in a system where page tables are held in store, that $c = 1$. Consequently, it can be shown that

$$z_o = (2s_o)^{\frac{1}{2}} \qquad (3.3)$$

Although this is in itself an important relationship, it is useful to note that if available data on segment sizes (Bat 70) implies that $s_o \leq 1000$ words then equation 3.3 implies that $z_o \leq 45$ words. This is somewhat contradictory to the current practice of page sizes of 512 or 1024 words. Although there are other good system reasons for these choices of page sizes it is essential to observe that such sizes will necessarily increase the amount of space wasted within a page. Not because the page does not contain information but because the page contains information that is not relevant to the segment currently being accessed.

Hatfield (Hat 72) examined the effect of varying the page size on system behaviour. The time to process a page fault has three components:-

a - the access time to the device where the page resides

b - the time to transfer the page

c - the software overhead.

It can be argued that a and c remain more or less constant irrespective of the page size. Consequently, if two page sizes b_1 and b_2 are considered then the relative costs can be shown as follows:-

$$(a + b_1 + c)/(a + b_2 + c)$$

Now the time to transfer the page, b, is given by

$b = zt_r$ where t_r is the transfer rate for the device.

If c is assumed to be small compared with a + b then

relative cost $\approx (a + b_1)/(a + b_2)$.

If actual figures are substituted, for example

$a = 50 \times 10^{-3}$ secs

$b = 5 \times 10^{-6}$ secs/word (for disk storage)

$\Rightarrow$ relative cost $\approx (5000 + 5z_1)/5000 + 5z_2)$

$\Rightarrow$ relative cost $\approx (1000 + z_1)/(1000 + z_2)$

This shows that unless page sizes are very large, or z_1 and z_2 are very different then the page size does not have a significant effect on the system overhead (see Table 3.5).

z_1	z_2	Relative Cost
256	512	0.83
256	1024	0.62
256	2048	0.41
512	256	1.20
512	1024	0.75
512	2048	0.49
1024	256	1.61
1024	512	1.33
1024	2048	0.66
2048	256	2.42
2048	512	2.01
2048	1024	1.50

Table 3.5

For example, an eight-fold increase in the page size from 256 to 2048 words causes a little less than a factor of 2.5 increase in the overheads.

It is useful to note that these figures are very much dependent on the storage devices being used.

3.4 Paged Systems - Demand Paging

The term demand paging refers to a process whereby pages are only brought into main storage when a program refers to them.

To fully present the notions of demand paging it is necessary to introduce some formal representation of program behaviour.

Consequently the following definitions are presented:-

DEFINITION 3.1

Let $N = \{0,1,2,\ldots,n-1\}$ be the set of pages of a program.

DEFINITION 3.2

Let $S(t)$ be the set of pages belonging to a program that are in main store after the reference at time t. This is somteimes known as the Store Set of the program at time t.

DEFINITION 3.3

The memory references of a program are denoted by:-

$r(1), r(2), r(3), \ldots\ldots r(k)$ where $r(t) \in N$

and $r(t)$ is the page referenced at reference t. A sequence of such references is known as a reference string.

PROPOSITION 3.5 The Principle of Locality (Den 70).

During any interval of execution a program tends to favour a subset of its pages, and this set of favoured pages tends to change its membership slowly.

<u>PROOF</u>

Denning maintains that this is an experimentally observed phenomenon, but formalises the notion as follows:-

<u>DEFINITION 3.4</u> Reference Density

The reference density for a page i is denoted by a(i,k) where

$$a(i,k) = \Pr[(\text{reference } r(k) = i)]$$

<u>DEFINITION 3.5</u> Ranking

A ranking R(k) of a program's pages is an ordering

$$p_0, p_1, \ldots\ldots\ldots, p_{n-1} \quad \text{where } p_i \in N, \forall i,\ 0 \leq i \leq n-1$$

such that

$$a(p_0,k) \geq a(p_1,k) \geq \ldots \geq a(p_{n-1},k).$$

Such a ranking is <u>strict</u> if

$$a(p_0,k) > a(p_1,k) > \ldots > a(p_{n-1},k).$$

<u>DEFINITION 3.6</u> Ranking Change

There is a ranking change at reference k if

$$R(k) \neq R(k-1)$$

<u>DEFINITION 3.7</u> Ranking Lifetime

A ranking lifetime is the number of references between consecutive ranking changes.

Now can be stated:-

<u>PROPOSITION 3.5a</u>   The Principle of Locality

> The rankings R(k) are strict and the expected ranking lifetimes long.

This will be considered in more detail later. For completeness, it is necessary to include alternative definitions of locality due to Madnick (Mad 73).

<u>DEFINITION 3.8</u>   Temporal Locality

> If the logical addresses a(1), a(2),.... are referenced during the time interval t - T to t, there is a high probability that these same addresses will be referenced during the time interval t to t + T.

<u>DEFINITION 3.9</u>   Spatial Locality

> If the logical address a is referenced at time t, then there is a high probability that a logical address in the range a - A to a + A will be referenced at time t + 1.

These definitions probably have a greater intuitive appeal than those of Denning.

<u>DEFINITION 3.10</u>   Paging Algorithm

> A paging algorithm gives S(t + 1) as follows:-
>
> $$S(t+1) = S(t) + X(t+1) - Z(t+1)$$
>
> where X(t + 1) is the set of pages brought in at

time t + 1 and Z(t + 1), the replaced page set, is a possibly non-empty subset of S(t). It is possible that at a given time t' both X(t') and Z(t') are empty, and this represents no change in the storage allocation for a program.

DEFINITION 3.11 Strict Demand Paging Algorithm (Spi 77)

A strict demand paging algorithm gives S(t + 1) as a function of S(t):-

$$S(t+1) = \begin{cases} S(t) & \text{if } r(t+1) \in S(t) \\ S(t) + r(t+1) - Z(t+1) & \text{if } r(t+1) \notin S(t) \end{cases}$$

A variation of this type of algorithm which allows pages to be removed at any time rather than just at the time of a page fault is given below:-

DEFINITION 3.12 Loose Demand Paging Algorithm

A loose demand paging algorithm gives S(t + 1) as a function of S(t):-

$$S(t+1) = \begin{cases} S(t) - Z(t+1) & \text{if } r(t+1) \in S(t) \\ S(t) + r(t+1) - Z(t+1) & \text{if } r(t+1) \notin S(t) \end{cases}$$

In opposition to pre-paging, demand paging algorithms have been used because of the difficulty ascribed to the prediction involved in a pre-paging algorithm. This in itself would not be sufficient to justify demand paging as an acceptable approach if demand paging were shown to be considerably more expensive than pre-fetching.

The following result, due to Mattson et al. (Mat 70), has been used to support the case for demand paging:-

PROPOSITION 3.6

Given any reference string and replacement algorithm, (not necessarily using demand paging) another replacement algorithm exists that uses demand paging and causes the same or fewer page faults.

This result is intuitively reasonable, since pre-paging can be considered as only causing page faults to occur earlier than they would have done under demand paging. If the page movements are done too soon then it is possible that a removed page will be referred to before the page that has been brought in.

Aho, Denning, and Ullman (Aho 71) have given a generalisation of this result, which requires the following definition:-

DEFINITION 3.13 The Cost of Replacement Algorithms

If $h(k)$ denotes the cost of an operation that places k (≥ 1) pages in memory, where $h(k) \geq h(1) = 1$, then the cost for processing a reference string

$R = r(1), r(2), \ldots r(n)$

with a given algorithm A starting from an initial memory allocation S is given by:-

$$C(A,S,R) = \sum_{t=1}^{n} h(|X(t)|)$$

where $X(t)$ is the set of pages brought into memory at time t.

If A is a demand paging algorithm then $|X(t)| \leqslant 1$, for $1 \leqslant t \leqslant n$ and consequently

$$C(A,S,R) = \sum_{t=1}^{n} X(t) .$$

The following can now be stated:-

<u>PROPOSITION 3.7</u>

If A is a paging algorithm, and further if $h(k) \geqslant k$, for $k \geqslant 1$ and $h(1) = 1$, then there exists a demand paging algorithm A' such that

$$C(A',S,R) \leqslant C(A,S,R)$$

for all S and R.

Despite the fact that the situation $h(k) < k$ occurs frequently in practice, this result is used as a justification for a restriction of theoretical consideration to demand paging alone. This and the other limitations of the theory will be considered later. A formal representation of current page replacement algorithms in a demand paging environment is given below:-

<u>DEFINITION 3.14</u> First-in First-out (FIFO)

The page which has been in memory for the longest time is replaced.

If $R(p)$, $p \in S(t)$ is defined as

$R(p) = t - t'$ where t' is the latest value of t where $X(t) = \{p\}$.

then an ordering of the pages in S(t) can be defined such that

$S(t) = [p_1, p_2, \ldots. p_k]$

and

$R(p_1) < R(p_2) < R(p_3) < \ldots < R(p_k)$

If $r(t+1) \notin S(t)$ then

$X(t+1) = r(t+1)$

$Z(t+1) = p_k.$

and the new ordering of $S(t+1)$ is

$S(t+1) = [r(t+1), p_1, p_2, \ldots p_{k-1}]$

NOTE: This is a strict demand paging algorithm.

DEFINITION 3.15 Least Recently Used (LRU)

The page in memory which has not been referenced for the longest time is replaced.

If $U(p)$, $p \in S(t)$ is defined as

$U(p) = t - t'$ where t' is the latest t such that $r(t) = p$

then an ordering of the pages in S(t) can be defined such that

$S(t) = [p_1, p_2, \ldots, p_k]$

and

$U(p_1) < U(p_2) < \ldots\ldots < U(p_k).$

If $r(t+1) \notin S(t)$ then

$X(t+1) = r(t+1)$

$Z(t+1) = p_k$

and the new ordering of $S(t + 1)$ is

$$S(t + 1) = [\, r(t + 1), p_1, p_2, \ldots, p_{k-1}] \,.$$

DEFINITION 3.16 Working Set Algorithm (Den 68a).(Den 68b)

The working set of a program is that set of distinct pages referenced in the T most recent references, $r(t - T + 1), \ldots, r(t)$, where T is called the window size.

$$S(t) = W(t,T)$$

where $W(t,T)$ denotes the working set at time t with a window size of T.

DEFINITION 3.17 Page Faulty Frequency Algorithm (Chu 72)

Let t' be the time of the most recent page fault, if a subsequent page fault occurs at time t + 1 then:-

$$S(t + 1) = \begin{cases} S(t) + r(t + 1) & \text{if } t' - t + 1 \leq 1/p \\ \\ W(t,t-t') + r(t + 1) & \text{if } t' - t + 1 > 1/p \end{cases}$$

where p is an estimated page fault frequency parameter.

These definitions correspond to the algorithms that have largely been used in practice. However, two better, theoretically obtainable, algorithms exist:-

DEFINITION 3.18 VMIN Algorithm (Pri 76)

$V(t,T)$, the VMIN set at time t is defined as follows:-

$$V(0,T) = \Phi$$

$$V(1,T) = r(1)$$

and for $t \geq 1$

$$V(t+1,T) = \begin{cases} V(t,T) + r(t+1) \text{ if } r(t) \in W(t+T,T) \\ V(t,T) + r(t+1) - r(t) \text{ if } \\ \qquad r(t) \notin W(t+T,T) \end{cases}$$

In this algorithm a page is replaced if it is not referenced in the next T references. Clearly this involves knowledge of the page reference string in advance.

DEFINITION 3.19 OPT Algorithm (Mat 70)

All pages are assigned a forward distance which for page p is defined, at time t, as

$$FD(p) = F - t$$

$$F = t' \text{ where } t' \text{ is the least } t \text{ such that } r(t') = p \text{ and } t' > t.$$

Consequently, a priority list PL can be defined at time to to be:-

$$PL(t) = [p_1, p_2, \ldots, p_n]$$

where $p_1 = r(t+1)$ and $FD(p_i) < FD(p_{i+1})$.

If a page is never referenced after time t it can be arbitarily assigned a forward distance of infinity.

Thus, the algorithm works as follows:-

$$S(t+1) = \begin{cases} S(t) + r(t+1) - p, & \text{if } r(t+1) \notin S(t) \\ S(t) & \text{if } r(t+1) \in S(t) \end{cases}$$

where $p \in S(t) \; : \forall p' \in S(t) \;\; FD(p) \leq FD(p')$

NOTE: After each reference the priority list must be re-created.

As mentioned above, these algorithms although theoretically obtainable cannot be implemented in practice since a complete "dry run" through the program would be necessary to create the reference string upon which they depend.

The four practical algorithms however need only retain information on the past behaviour of the programs to estimate the future behaviour. The main use of VMIN and OPT is as estimators of the success of the practical algorithms in test situations.

To return to the Principle of Locality, it is possible to measure the effectiveness of a management strategy by its success in estimating the locality at any time T.

Initially, it is sufficient to observe that it is those pages in the current locality which will be referenced in the near future that must be estimated.

Calling upon the notation used to describe the principle of the Working Set algorithm, the temporal locality at time t of width 2T can be defined as follows:-

$$TL(t,T) = W(t,T) \cup W(t+T,\, T)$$

NOTE: In a paged system, spatial locality about referenced addresses normally is automatically handled by the loading of the surrounding page.

Thus it can be stated that an estimator of $W(t + T, T)$ is required.

Apart from the problem of finding a suitable estimator, another difficulty arises, namely the size of T. Coffman and Denning (Cof 73) suggest that for $W(t,T)$ two criteria must be satisfied:-

1. T must be large enough to ensure that the probability of a member of the current locality being missing from the working set is small.
2. T must be small enough to ensure that the probability of more than one inter-locality transition being contained in the working set is small. (An inter-locality transition occurs when a program moves from one favoured subset of its pages to another.)

This can be presented formally:-

Let $r(1), r(2), \ldots, r(t), \ldots$ be the reference string generated by a program, then the Principle of Locality suggests that the program passes through a series of <u>localities</u> $L_1, L_2, \ldots.$ where $L_i \in N$. That is to say, if L, the current locality, is given by

$$L = L_k \quad \text{for } t_i \leq t \leq t_j$$

then

$$r(t) \in L \quad \text{for } t_i \leq t \leq t_j.$$

The management policy generates a sequence of store sets $S_1, S_2, \ldots$ and the aim is that if at time t

i) the store set $= S_1$

ii) the locality $= L_m$

then

$S_1 = L_m$.

In order to support the theoretical analysis of replacement algorithms, Coffman and Denning make the following assumptions about reference strings (Cof 73):-

1. The probability that $r(t + x) = j$, given that $r(t) = i$ $(i, j \in N)$ is independent of t.

2. For any t, and any $i \in N$, there exists a $t' > t$ such that $r(t') = i$.

3. $r(t)$ and $r(t + x)$ become uncorrelated as x becomes large.

Coffman and Denning are aware of the significant restriction placed on reference strings by 1. Over a complete reference string there is no good evidence that 1. should hold. Within a locality, however, 1. is more reasonable. That is to say:-

If $L(k) = \{r(t) : t_1 \leqslant t \leqslant t_2\}$

and $t_1 \leqslant t \leqslant t_2$, $t_1 \leqslant t + x \leqslant t_2$

then it would appear that 1. is intuitively more reasonable.

This is an example of what Spirn (Spi 77) and Denning and Kahn (Den 75) observe to be a difference between <u>micro-</u>

behaviour and macro-behaviour in reference strings. To further expand this notion, Denning and Schwartz (Den 72) give some important properties of localities:-

1. During any interval of time a program distributes its references non-uniformly over its pages.
2. Taken as a function of time, the frequency with which a given page is referenced tends to change slowly.
3. Correlation between immediate past and immediate future patterns of behaviour tends to be high. Whereas the correlation between disjoint reference patterns tends to zero as the distance between them tends to infinity.

In the same paper they make a significant admission:-

"W2: The stochastic mechanism underlying the generation of a reference string is stationary, i.e. independent of the time of origin.

....

Assumption W2 does restrict the results somewhat, limiting the analysis to the context of a single program locality in the following sense. As mentioned above a program passes through a sequence of localities as it generates references. One would expect that whatever non-stationarities exist depend only on the locality. In other words, we could approximate a reference string r as a sequence of substrings

$r = r_1 r_2 \ldots\ldots$

where each substring r_i obeys W2. Therefore the results are applicable locally in a given reference string, but not necessarily globally .... assumption W2 will not be severe as long as the measurement intervals are comparable to or less than the average inter-locality transition time."

This in effect restricts analysis of algorithms to the micro-behavioural phase, and avoids consideration of locality transitions.

It is the contention of this thesis that this and other assumptions place significant restrictions on the utility of page replacement algorithms, and consequently cast doubts on the global validity of demand paging. These contentions are laid out in the following section.

3.5 Assumptions Inherent in the Theoretical Support for Current Algorithms

1. Ignoring all aspects considered above, it would seem that from the point-of-view of system throughput demand paging has a detrimental effect. Stated simply at each page fault occurrence in a demand paged system the program must wait the maximum possible time before its request is satisfied, since the page is only sought once it has been referenced.

2. Following on 1. above, since the models on which paging algorithms are based are effective only <u>within</u> localities, the effects of locality transitions are amplified by demand paging. During a locality transition, a high page activity must be expected. The adoption of demand paging implies that each page fault will be treated singly and no optimisation of, say, disk seeks will be possible.

3. The effectiveness of demand paging is based on a rather unimpressive proof (Propositions 3.6 & 3.7). These propositions admit the <u>existence</u> of optimal demand paging algorithms. However, what is not shown is:-

   a) that this demand paging algorithm can indeed be achieved without a complete pre-determination of the reference string.

   b) that the same demand paging algorithm is optimal in all cases. It seems intuitively likely that OPT should fall into this category but it is reported (Aho 71) that counter-examples can be found.

   c) Proposition 3.7 depends for its proof on assumptions about the cost of a page fetch that do not hold for conventional main store - drum/disk hierarchies.

d) The cost of an algorithm is estimated only in terms of the cost of its page faults. This is not sufficient since this implies that an algorithm which generates no page faults by the simple expedient of holding <u>all</u> of a program's address space in main store is optimal and has zero cost.

4. The optimal choice of page size seems to be dependent on the segment size for a given program. This is in conflict with:-
   a) the need to achieve efficient transfers between backing store and main store.
   b) the convenience of establishing a system-wide norm for page size.

5. All the major replacement algorithms depend on the establishment of an arbitrary system-wide behavioural parameter:-
   a) LRU - requires a stack-length to be fixed.
   b) Working Set - requires a window size to be fixed.
   c) PFF - requires a critical page fault frequency to be fixed.

   As will be shown in the next chapter, the choice of the values for these parameters is critical to the efficient operation of the algorithm in question.

6. As with 2 above, the algorithms presented make no significant attempt to deal with locality transitions, thereby restricting their general effectiveness. To be fair this is perhaps least applicable to the Page Fault Frequency algorithm.

3.6 Conclusion

This chapter has attempted to show the theoretical background of the page replacement algorithms of current systems. It has also shown that some of the claims made are questionable and that conclusions drawn from these propositions are untrustworthy. That the theory has its limitations is undoubtedly true, however it will be shown in the next chapter that despite these limitations, the algorithms can be used to some effect if suitably limited contexts are chosen.

CHAPTER 4 BEHAVIOURAL CHARACTERISTICS OF CONVENTIONAL MEMORY MANAGEMENT SYSTEMS

4.1 Introduction

In this chapter are presented results showing both theoretically and in practical situations, the strengths and limitations of current memory management systems.

4.2 The Working Set Algorithm (Den 70)

This algorithm, which attempts to estimate the current locality of a program by examination of the pages referenced during a fixed time interval in the past, T (called its window), has been extensively studied. But before these studies are considered, it is instructive to examine Denning's own claims for this algorithm.

Denning claims the following:-

> "WORKING SET PRINCIPLE: Suppose memory management operates according to the following rule: A program may run if and only if its working set is in memory, and a page may not be removed if it is a member of the working set of a running program. Then according to the principle of locality, this rule is an implementation of the principle of optimality."

The proof he presents firmly depends on the consideration only of the micro-behavioural characteristics of programs mentioned in Chapter 3. Equally, the related principle of locality (Proposition 3.5a) which he presents depends on estimates of the lengths of localities, i.e. the number of references that a program makes while in a locality.

Much support for his work has been taken from the work by Belady (Bel 66) on program lifetime functions and from the manner in which programs acquire pages on demand from the beginning of a time quantum (Cof 68),(Fin 66).

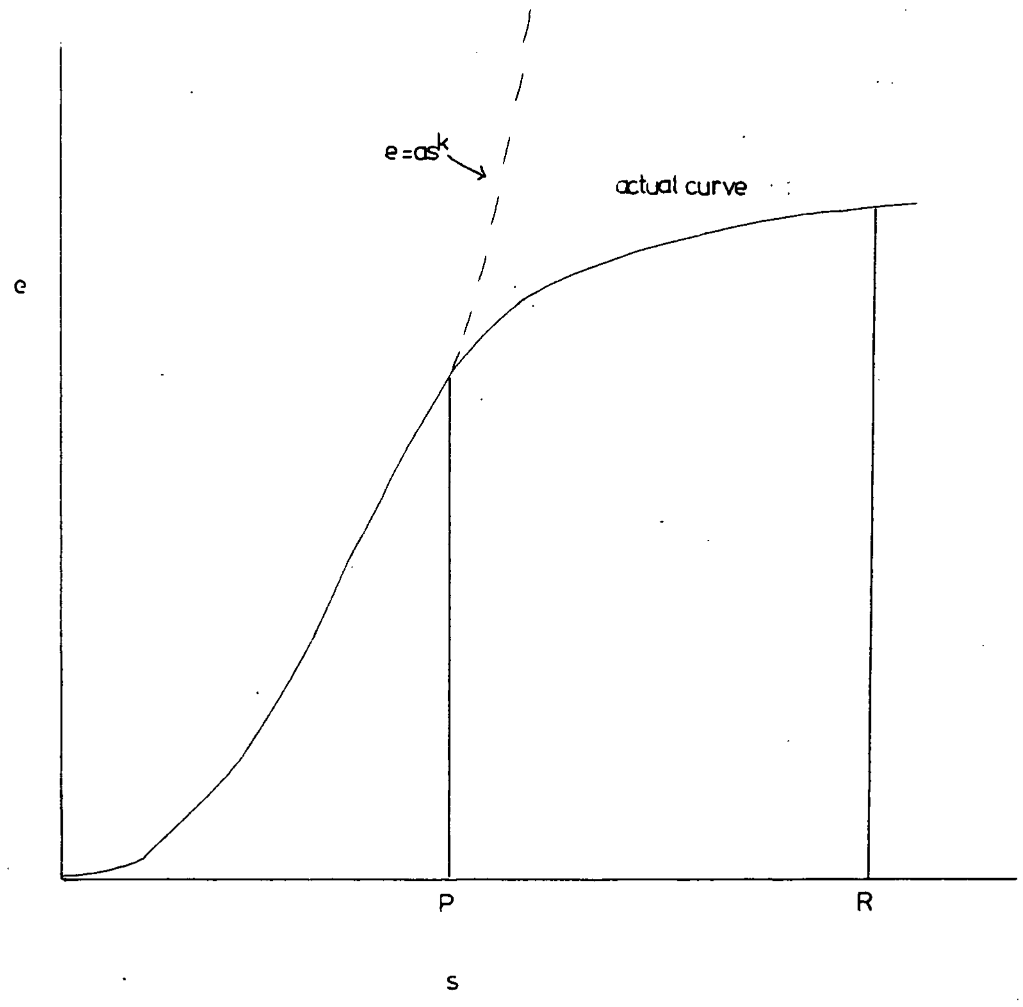
In his paper (Bel 66), Belady describes simulation experiments which together with simulation studies performed on the 360/67 at SDC (Fin 66) and at Princeton (Var 67) lent support to the following major results:-

<u>PROPOSITION 4.1</u> Belady Lifetime Function

> If e is the expected length of time between page faults and s represents the amount of storage assigned to a program, then the relationship between e and s can be approximated by
>
> $$e = as^k$$
>
> where a varies with the individual program and k has been observed to take values in the vicinity of 2.

This relationship is shown graphically in Diagram 4.1

Two points P and R are to be noted on this graph. Firstly R represents the amount of storage required to totally contain the program and P represents the point of divergence between



Relationship of mean execution interval between page and storage allocated.

Diagram 4.1

the approximation and the actual curve. Belady explains this divergence in two ways:-

a) e is the average of all execution intervals, and in the initial loading phase, a program goes through a number of short execution periods which contribute to the reduction of e.

b) if programs are given sufficient space to accumulate their current locality, then little or no paging will occur until a locality change is made.

From this and the work of Coffman and Varian (Cof 68) reported below, Denning extracts the following relationship:-

<u>PROPOSITION 4.2</u> Fault Probability

Let F(A,m,r) denote the number of faults generated as algorithm A processes reference string r under demand paging in an initially empty memory of size m, the <u>fault probability</u> f(A,m) can be defined as follows:-

$$f(A,m) = \sum_{\text{all } r} Pr(r)\ (F(A,m,r)/|r|)$$

where Pr(r) denotes the probability of occurrence of r and |r| represents the length of r.

This, apparently, yields the following graphical relationship (Diagram 4.2):-

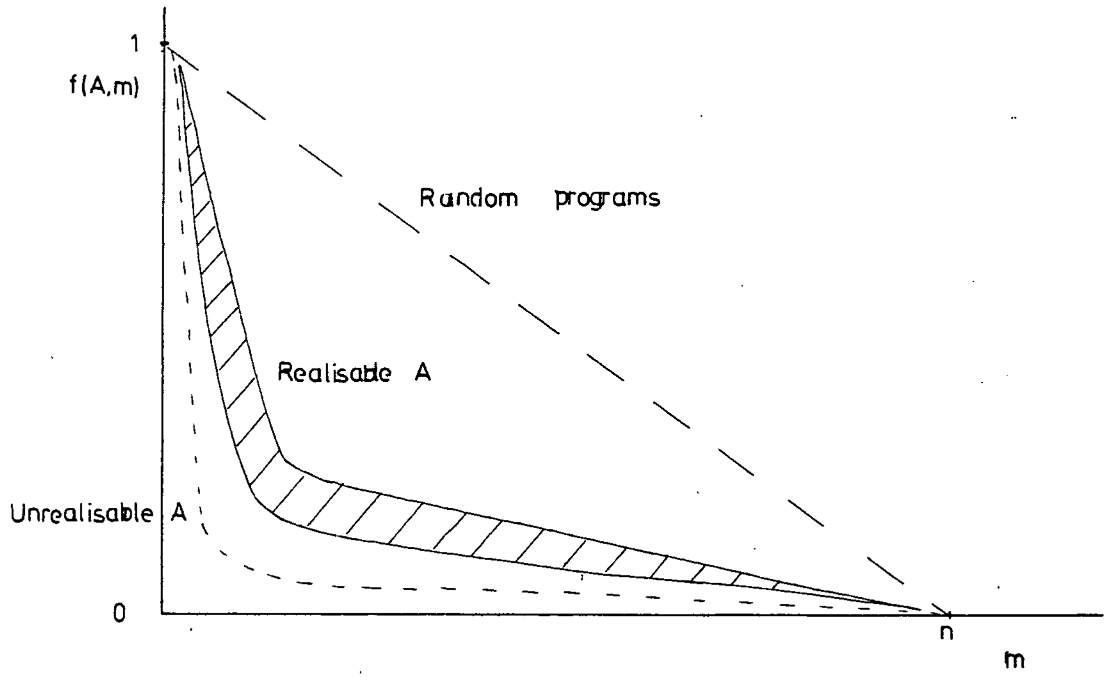


Diagram 4.2

It is stated that "reasonable" algorithms lie in the shaded region on the graph and that the dotted line represents what could be achieved by optimal unrealisable algorithms such as that of Belady. The argument is that for reasonable A, f(A,m) is much more sensitive to m than to A.

The dashed line above is meant to indicate the behaviour that would exist if programs exhibited a random reference pattern.

One of the most unfortunate features of the above diagram is that, although it shows that the number of page faults decreases for "reasonable" algorithms with the increase of allocated memory, it does not give any quantifiable estimates of the behaviour of algorithms either in general or in particular cases.

Further work on the relationship between locality and lifetime functions has been performed by Denning and Kahn (Den 75). These authors again quote considerable experimental evidence supporting the notion of locality (Bry 75), (Hat 71), and (Rod 71). In the same paper they present two important properties of lifetime functions:-

<u>PROPOSITION 4.3</u>

> A lifetime function typically has the convex/concave shape. The convex region is approximated by cx^k, where x is the allocated store size, for some c,k.

<u>PROPOSITION 4.4</u>

> For a given reference string, the Working Set lifetime function will tend to exceed that of LRU for wide ranges of memory allocations.

Evidence for this proposition has been found in the work of Bard (Bar 73), (Bar 75).

As mentioned above, a memory management strategy can best be considered as an estimator of program localities. An ideal estimator is said to have three properties (Den 75):-

a) the store set is always a subset of the current locality set

b) at a locality transition, the resident set contains only the pages in common to the old and new locality sets

c) page faults occur only for first references to entering pages.

The working set algorithm is consequently not an ideal estimator, since at a locality transition old locality pages can remain for up to T references after the transition.

However, if T is short enough to include only one locality transition, then the only penalty is the excess store allocation made to the program. Later some examples will be presented which estimate how significant this over-allocation is.

In another paper (Den 72), Denning and Schwartz establish behavioural characteristics of the Working Set algorithm. Given that:-

$S(T)$ = average working set size

$m(T)$ = missing page rate, i.e. the number of pages per unit time returning to the working set

$f(x)$ = the over-all inter-reference density

$F(x)$ = the over-all inter-reference distribution

n_r = the number of recurrent pages.

PROPOSITION 4.5

$$1 = S(1) \leq S(T) \leq S(T + 1) \leq \{\min n, T + 1\}$$

This states that the average working set size is non-decreasing with T and that the working set size is bounded below by 1 and above by either one more than the current window size or n the number of pages (whichever is the smaller).

PROPOSITION 4.6

$$S(T + 1) - S(T) = m(T)$$

This states that the difference between the average working set size for a window of T+1 and that for a window of T is equal to the missing page rate.

PROPOSITION 4.7

$$0 \leqslant m(T + 1) \leqslant m(T) \leqslant m(0) = 1$$

This states that the missing page rate does not increase with T.

PROPOSITION 4.8

$$m(T) = 1 - F(T) = \sum_{y > T} f(y)$$

This states that m(T) can be regarded as the probability that $x > T$.

PROPOSITION 4.9

$$m(T + 1) - m(T) = -f(T + 1)$$

This states that the difference between m(T + 1) and m(T) is the negative value of the over-all inter-reference density f.

PROPOSITION 4.10

$$(S(T - 1) + S(T + 1))/2 < S(T)$$

This states that the curve S(T) is concave down.

PROPOSITION 4.11

$$\lim_{T \to \infty} S(T) = n_r$$

As $T \to \infty$ the working set size tends to the number of recurrent pages.

PROPOSITION 4.12

$$\lim_{T \to \infty} m(T) = 0$$

As $T \to \infty$ the missing page rate tends to zero.

These properties are far from remarkable, and as such do not provide any insight into the physical operation of the Working Set algorithm. However they do provide two useful indicators to the size of T. As was mentioned above, this arbitrary parameter must be chosen with great care in order to increase the effectiveness of working set policies.

Firstly, if a specified lower bound is placed on the efficiency required of our algorithm, then this implies (in a limited context) an upper bound on the value of m(T). This, in turn,implies a lower bound on T, by Proposition 4.7.

Secondly, the concave down property of S(T) indicates that varying T need not be advantageous.

From a practical point-of-view, it is fairly clear that these considerations do not give a clear indication of how a Working Set algorithm will behave. Much measurement has been done in practical situations, particularly in comparison with LRU strategies. In the remainder of this section, and in the next (dealing with the LRU algorithm itself) these results will be presented.

Spirn and Denning (Spi 72) present the results of their experiments. They compare the behaviour of intrinsic models of locality with the working set algorithm as an estimator of locality. Experiments were carried out on two machines (a PDP-8 and System 360) using both assembly code and FORTRAN programs.

Measurements were made of the average working set size and the missing page probability for each of the techniques studied, compared with possible window sizes.

The work presented in this paper is rather interesting, in that it attempts to compare the behaviour of the intrinsic models with that of real programs by comparing how well the intrinsic compare with the Working Set algorithm. As this thesis has attempted to show, insufficient evidence has been produced to show that the Working Set algorithm is indeed a good estimator of real program behaviour. Even if it were, it has already been admitted that it only presents a model of the micro-behaviour within localities and does not deal with locality transitions. This is contradictory to the claim in this paper (Spi 72):-

> "We are concerned, however, with locality transition behaviour."

However, some useful data can be extracted from these experiments.

For the reference strings used the following statistics can be obtained:-

Reference String	Window Size	$\frac{\text{Working Set Size}}{\text{Program Size}}$ (%)
2	250	25
4	250	50
6	250	40
2	500	26
4	500	63
6	500	63
2	750	28
4	750	70
6	750	65
2	1000	30
4	1000	75
6	1000	69

Figure 4.1

In Figure 4.1, the column headed "Reference String" refers to the identifying number used in the original paper.

It is interesting to note that Spirn later (Spi 77) suggests that window sizes of "practical interest" satisfy:

$$10{,}000 \leq T \leq 100{,}000 \text{ references.}$$

This later statement has also been supported by the work of Rodriguez-Rosell (Rod 73). In this paper, Rodriguez-Rosell commented on the lack of published data on working set behaviour from actual program measurements. The measurements used in his study were made on an assembler program running on the System 360. The minimum value used for the window size was 5000 references.

It is puzzling, therefore, to consider the data in Figure 4.1, particularly considering that in all but one of the reference strings shown, a working set of greater than 70% of the address space available is achieved with a window size of the order of 1000 references. It is equally puzzling that reference strings 4 and 6 are high-level language generated program reference strings (FORTRAN) and that reference string 2 is that of a compiler. Another peculiar feature is that all experiments quoted here were carried out on the System 360, also.

It is useful, at this point, to remember a conjecture of Randell (Ran 69) that programming languages, styles, and conventions might have an effect on the behaviour of programs.

To further support the doubts expressed above, some experimental evidence is presented which shows that, in practical cases, the Working Set algorithm acts as a poor estimator of locality size.

EXAMPLE 4.1.1

On the following page is shown a bubblesort algorithm taken from Wirth (Wir 76). This well-known technique was run under two sets of conditions:-

a) with window size = 10,000 references

b) with window size = 500 references

In both cases the page size was 256 words.

The results are presented on the following pages.

This particular example brings up some interesting points. Firstly, the localities are easily determinable by inspection for the sort part of the program. The references to the address space can be divided into 3 categories:-

i) the current code page

ii) the page containing the index variables i,j,k

iii) the page currently being worked upon in the array.

Consequently, it is to be expected that, since in this example there is only one code page, and since there are few variables other than the array elements, the locality estimate will, at worst, consist of <u>four</u> pages:-

i) the current code page

ii) the page containing the index variables

iii) two (adjacent) pages from the array

```
program bubblesort(input,output);
var
  i,j,k:integer;
  element:array[1..2048]of integer;
begin
   j:=1;k:=-1;
   for i:=1 to 2048 do
   begin
      j:=-(j+1);
      k:=-(k-1);
      element[i]:=i+j-k;
   end;
   for i:=2 to 2048 do
      for j:=2048 downto i do
      begin
         if element[j-1]>element[j] then
         begin
            k:=element[j-1];
            element[j-1]:=element[j];
            element[j]:=k;
         end;
      end;
end.
```

```
******************************
 LRU PFF Wor

WS  Store
  C: 0  1



LRU Store
  C: 0  1



PFF Store
  C: 0  1




******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  1  D:23  2



LRU Store
  C: 0  1  D:23  2



PFF Store
  C: 0  1  D:23  2




******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  1  D:23  2  D: 0  3



LRU Store
  C: 0  1  D:23  2  D: 0  3



PFF Store
  C: 0  1  D:23  2  D: 0  3
```

Algorithms causing page fault.

Contents of memory at page fault for each algorithm

In this case, note the Working set store with a window size of 500 references

```
WS  Store
  C: 0  4  D:23  6  D: 0  7  D:15  8



LRU Store
  C: 0  4  D:23  6  D: 0  7  D:15  8



PFF Store
  C: 0  4  D:23  6  D: 0  7  D:15  8



******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  4  D:23  6  D: 0  7  D:15  13  D:16  14



LRU Store
  C: 0  4  D:23  6  D: 0  7  D:15  13  D:16  14



PFF Store
  C: 0  4  D:23  6  D: 0  7  D:15  13  D:16  14



******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  4  D:23  6  D: 0  7  D:15  13  D:16  269  D:17  270



LRU Store
  C: 0  4  D:23  6  D: 0  7  D:15  13  D:16  269  D:17  270



PFF Store
  D:16  269  D:17  270



******************************
******************************
 LRU PFF Wor

WS  Store
```

Continued in Example 4.1.2

```
WS  Store
  C: 0  1


LRU Store
  C: 0  1


PFF Store
  C: 0  1


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  1  D:23  2


LRU Store
  C: 0  1  D:23  2


PFF Store
  C: 0  1  D:23  2


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  1  D:23  2  D: 0  3


LRU Store
  C: 0  1  D:23  2  D: 0  3


PFF Store
  C: 0  1  D:23  2  D: 0  3


******************************
******************************
 LRU PFF Wor

WS  Store
```

```
PFF Store
  C: 0  4  D:23  6  D: 0  7  D:15  8


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  4  D:23  6  D: 0  7  D:15  13  D:16  14


LRU Store
  C: 0  4  D:23  6  D: 0  7  D:15  13  D:16  14


PFF Store
  C: 0  4  D:23  6  D: 0  7  D:15  13  D:16  14


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  4  D:23  6  D: 0  7  D:15  13  D:16  269  D:17  270


LRU Store
  C: 0  4  D:23  6  D: 0  7  D:15  13  D:16  269  D:17  270


PFF Store
  D:16  269  D:17  270


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  4  D:23  6  D: 0  7  D:15  13  D:16  269  D:17  525  D:18  526


LRU Store
  C: 0  4  D:23  6  D: 0  7  D:15  13  D:16  269  D:17  525  D:18  526
```

```
WS  Store
  C: 0  4  D:23  6  D: 0  7  D:15  13  D:16  269  D:17  525  D:18  781  D:19  782


LRU Store
  C: 0  4  D:23  6  D: 0  7  D:15  13  D:16  269  D:17  525  D:18  781  D:19  782


PFF Store
  D:18  781  D:19  782


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  4  D:23  6  D: 0  7  D:15  13  D:16  269  D:17  525  D:18  781  D:19  1037  D:20  1038


LRU Store
  D:23  6  D: 0  7  D:15  13  D:16  269  D:17  525  D:18  781  D:19  1037  D:20  1038


PFF Store
  D:19  1037  D:20  1038


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  4  D:23  6  D: 0  7  D:15  13  D:16  269  D:17  525  D:18  781  D:19  1037  D:20  1293  D:21  1294


LRU Store
  D: 0  7  D:15  13  D:16  269  D:17  525  D:18  781  D:19  1037  D:20  1293  D:21  1294


PFF Store
  D:20  1293  D:21  1294


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  4  D:23  6  D: 0  7  D:15  13  D:16  269  D:17  525  D:18  781  D:19  1037  D:20  1293  D:21  1549  D:22  1550
```

```
PFF Store
  D:21  1549  D:22  1550


******************************
******************************
 LRU PFF

WS  Store
  C: 0  4  D:23  1806  D: 0  7  D:15  13  D:16  269  D:17  525  D:18  781  D:19  1037  D:20  1293  D:21  1549  D:22  1805


LRU Store
  D:16  269  D:17  525  D:18  781  D:19  1037  D:20  1293  D:21  1549  D:22  1805  D:23  1806


PFF Store
  D:22  1805  D:23  1806


******************************
******************************
 LRU PFF

WS  Store
  C: 0  4  D:23  2058  D: 0  2059  D:15  13  D:16  269  D:17  525  D:18  781  D:19  1037  D:20  1293  D:21  1549  D:22  1805


LRU Store
  D:17  525  D:18  781  D:19  1037  D:20  1293  D:21  1549  D:22  1805  D:23  2058  D: 0  2059


PFF Store
  D:23  2058  D: 0  2059


******************************
******************************
 LRU PFF

WS  Store
  C: 0  2060  D:23  2058  D: 0  2059  D:15  13  D:16  269  D:17  525  D:18  781  D:19  1037  D:20  1293  D:21  1549  D:22  1805


LRU Store
  D:18  781  D:19  1037  D:20  1293  D:21  1549  D:22  1805  D:23  2058  D: 0  2059  C: 0  2060


PFF Store
```

```
******************************
 PFF Wor

WS  Store
  C: 0  31939  D:23  31932  D: 0  31941  D:22  31942


LRU Store
  D:18  781  D:19  1037  D:20  1293  D:21  1549  D:23  31932  C: 0  31939  D: 0  31941  D:22  31942


PFF Store
  D:23  31932  D: 0  31941  C: 0  31939  D:22  31942


******************************
******************************
 PFF Wor

WS  Store
  C: 0  62403  D:23  62396  D: 0  62405  D:22  62287  D:21  62406


LRU Store
  D:18  781  D:19  1037  D:20  1293  D:22  62287  D:23  62396  C: 0  62403  D: 0  62405  D:21  62406


PFF Store
  D:23  62396  D: 0  62405  C: 0  62403  D:22  62287  D:21  62406


******************************
******************************
 PFF Wor

WS  Store
  C: 0  92867  D:23  92860  D: 0  92869  D:21  92751  D:20  92870


LRU Store
  D:18  781  D:19  1037  D:22  62287  D:21  92751  D:23  92860  C: 0  92867  D: 0  92869  D:20  92870


PFF Store
  D:23  92860  D: 0  92869  C: 0  92867  D:21  92751  D:20  92870


******************************
******************************
 PFF Wor

WS  Store
```

```
   D:18   781   D:22   62287   D:21   92751   D:20   123215   D:23   123324   C: 0   123331   D: 0   123333   D:19   123334

PFF Store
   D:23   123324   D: 0   123333   C: 0   123331   D:20   123215   D:19   123334


******************************
******************************
 PFF Wor

WS  Store
   C: 0   153795   D:23   153788   D: 0   153797   D:19   153679   D:18   153798


LRU Store
   D:22   62287   D:21   92751   D:20   123215   D:19   153679   D:23   153788   C: 0   153795   D: 0   153797   D:18   153798


PFF Store
   D:23   153788   D: 0   153797   C: 0   153795   D:19   153679   D:18   153798


******************************
******************************
 LRU PFF Wor

WS  Store
   C: 0   184259   D:23   184252   D: 0   184261   D:18   184143   D:17   184262


LRU Store
   D:21   92751   D:20   123215   D:19   153679   D:18   184143   D:23   184252   C: 0   184259   D: 0   184261   D:17   184262


PFF Store
   D:23   184252   D: 0   184261   C: 0   184259   D:18   184143   D:17   184262


******************************
******************************
 LRU PFF Wor

WS  Store
   C: 0   214723   D:23   214716   D: 0   214725   D:17   214607   D:16   214726


LRU Store
   D:20   123215   D:19   153679   D:18   184143   D:17   214607   D:23   214716   C: 0   214723   D: 0   214725   D:16   214726
```

```
******************************
 LRU PFF Wor

WS  Store
  C: 0  245187  D:23  245180  D: 0  245189  D:16  245071  D:15  245190


LRU Store
  D:19  153679  D:18  184143  D:17  214607  D:16  245071  D:23  245180  C: 0  245187  D: 0  245189  D:15  245190


PFF Store
  D:23  245180  D: 0  245189  C: 0  245187  D:16  245071  D:15  245190


******************************
******************************
 PFF Wor

WS  Store
  C: 0  276377  D:23  276378  D: 0  276379  D:16  276319  D:17  276380


LRU Store
  D:19  153679  D:18  184143  D:15  246534  D:16  276319  C: 0  276377  D:23  276378  D: 0  276379  D:17  276380


PFF Store
  D:23  276378  D: 0  276379  C: 0  276377  D:16  276319  D:15  246534  D:17  276380


******************************
******************************
 PFF Wor

WS  Store
  C: 0  306329  D:23  306330  D: 0  306331  D:17  306271  D:18  306332


LRU Store
  D:19  153679  D:15  246534  D:16  276486  D:17  306271  C: 0  306329  D:23  306330  D: 0  306331  D:18  306332


PFF Store
  D:23  306330  D: 0  306331  C: 0  306329  D:16  276486  D:17  306271  D:18  306332


******************************
******************************
 PFF Wor
```

```
LRU Store
  D:15  246534  D:16  276486  D:17  306438  D:18  336223  C: 0  336281  D:23  336282  D: 0  336283  D:19  336284


PFF Store
  D:23  336282  D: 0  336283  C: 0  336281  D:17  306438  D:18  336223  D:19  336284


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  366233  D:23  366234  D: 0  366235  D:19  366175  D:20  366236


LRU Store
  D:16  276486  D:17  306438  D:18  336390  D:19  366175  C: 0  366233  D:23  366234  D: 0  366235  D:20  366236


PFF Store
  D:23  366234  D: 0  366235  C: 0  366233  D:18  336390  D:19  366175  D:20  366236


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  396185  D:23  396186  D: 0  396187  D:20  396127  D:21  396188
```

```
LRU Store
  D:17  306438  D:18  336390  D:19  366342  D:20  396127  C: 0  396185  D:23  396186  D: 0  396187  D:21  396188


PFF Store
  D:23  396186  D: 0  396187  C: 0  396185  D:19  366342  D:20  396127  D:21  396188


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  426137  D:23  426138  D: 0  426139  D:21  426079  D:22  426140


LRU Store
  D:18  336390  D:19  366342  D:20  396294  D:21  426079  C: 0  426137  D:23  426138  D: 0  426139  D:22  426140


PFF Store
  D:23  426138  D: 0  426139  C: 0  426137  D:20  396294  D:21  426079  D:22  426140


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  485399  D:23  485400  D: 0  485401  D:15  485402


LRU Store
  D:19  366342  D:20  396294  D:21  426246  D:22  456198  C: 0  485399  D:23  485400  D: 0  485401  D:15  485402


PFF Store
  D:23  485400  D: 0  485401  C: 0  485399  D:21  426246  D:22  456198  D:15  485402


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  485942  D:23  485943  D: 0  485944  D:15  485892  D:16  485945


LRU Store
  D:20  396294  D:21  426246  D:22  456198  D:15  485892  C: 0  485942  D:23  485943  D: 0  485944  D:16  485945
```

```
******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  515894  D:23  515895  D: 0  515896  D:16  515844  D:17  515897


LRU Store
  D:21  426246  D:22  456198  D:15  485959  D:16  515844  C: 0  515894  D:23  515895  D: 0  515896  D:17  515897


PFF Store
  D:23  515895  D: 0  515896  C: 0  515894  D:15  485959  D:16  515844  D:17  515897


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  545846  D:23  545847  D: 0  545848  D:17  545796  D:18  545849


LRU Store
  D:22  456198  D:15  485959  D:16  515911  D:17  545796  C: 0  545846  D:23  545847  D: 0  545848  D:18  545849


PFF Store
  D:23  545847  D: 0  545848  C: 0  545846  D:16  515911  D:17  545796  D:18  545849


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  575798  D:23  575799  D: 0  575800  D:18  575748  D:19  575801


LRU Store
  D:15  485959  D:16  515911  D:17  545863  D:18  575748  C: 0  575798  D:23  575799  D: 0  575800  D:19  575801


PFF Store
  D:23  575799  D: 0  575800  C: 0  575798  D:17  545863  D:18  575748  D:19  575801


******************************
******************************
 LRU PFF Wor
LRU Store
  D:16   515911   D:17   545863   D:18   575815   D:19   605700   C: 0   605750   D:23   605751   D: 0   605752   D:20   605753


PFF Store
  D:23   605751   D: 0   605752   C: 0   605750   D:18   575815   D:19   605700   D:20   605753



******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0   635702   D:23   635703   D: 0   635704   D:20   635652   D:21   635705


LRU Store
  D:17   545863   D:18   575815   D:19   605767   D:20   635652   C: 0   635702   D:23   635703   D: 0   635704   D:21   635705


PFF Store
  D:23   635703   D: 0   635704   C: 0   635702   D:19   605767   D:20   635652   D:21   635705



******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0   665654   D:23   665655   D: 0   665656   D:21   665604   D:22   665657


LRU Store
  D:18   575815   D:19   605767   D:20   635719   D:21   665604   C: 0   665654   D:23   665655   D: 0   665656   D:22   665657


PFF Store
  D:23   665655   D: 0   665656   C: 0   665654   D:20   635719   D:21   665604   D:22   665657



******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0   724757   D:23   724758   D: 0   724759   D:15   724760


LRU Store
  D:19   605767   D:20   635719   D:21   665671   D:22   695623   C: 0   724757   D:23   724758   D: 0   724759   D:15   724760
```

```
******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  725216  D:23  725217  D: 0  725218  D:15  725158  D:16  725219


LRU Store
  D:20  635719  D:21  665671  D:22  695623  D:15  725158  C: 0  725216  D:23  725217  D: 0  725218  D:16  725219


PFF Store
  D:23  725217  D: 0  725218  C: 0  725216  D:15  725158  D:16  725219


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  755168  D:23  755169  D: 0  755170  D:16  755110  D:17  755171


LRU Store
  D:21  665671  D:22  695623  D:15  725325  D:16  755110  C: 0  755168  D:23  755169  D: 0  755170  D:17  755171


PFF Store
  D:23  755169  D: 0  755170  C: 0  755168  D:15  725325  D:16  755110  D:17  755171


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  785120  D:23  785121  D: 0  785122  D:17  785062  D:18  785123


LRU Store
  D:22  695623  D:15  725325  D:16  755277  D:17  785062  C: 0  785120  D:23  785121  D: 0  785122  D:18  785123


PFF Store
  D:23  785121  D: 0  785122  C: 0  785120  D:16  755277  D:17  785062  D:18  785123


******************************
******************************
```

```
LRU Store
  D:15  725325  D:16  755277  D:17  785229  D:18  815014  C: 0  815072  D:23  815073  D: 0  815074  D:19  815075

PFF Store
  D:23  815073  D: 0  815074  C: 0  815072  D:17  785229  D:18  815014  D:19  815075


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  845024  D:23  845025  D: 0  845026  D:19  844966  D:20  845027


LRU Store
  D:16  755277  D:17  785229  D:18  815181  D:19  844966  C: 0  845024  D:23  845025  D: 0  845026  D:20  845027


PFF Store
  D:23  845025  D: 0  845026  C: 0  845024  D:18  815181  D:19  844966  D:20  845027


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  874976  D:23  874977  D: 0  874978  D:20  874918  D:21  874979


LRU Store
  D:17  785229  D:18  815181  D:19  845133  D:20  874918  C: 0  874976  D:23  874977  D: 0  874978  D:21  874979


PFF Store
  D:23  874977  D: 0  874978  C: 0  874976  D:19  845133  D:20  874918  D:21  874979


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  904928  D:23  904929  D: 0  904930  D:21  904870  D:22  904931


LRU Store
  D:18  815181  D:19  845133  D:20  875085  D:21  904870  C: 0  904928  D:23  904929  D: 0  904930  D:22  904931
```

```
******************************
******************************
 LRU PFF Wor

WS . Store
  C: 0  963956  D:23  963957  D: 0  963958  D:15  963959


LRU Store
  D:19  845133  D:20  875085  D:21  905037  D:22  934989  C: 0  963956  D:23  963957  D: 0  963958  D:15  963959


PFF Store
  D:23  963957  D: 0  963958  C: 0  963956  D:21  905037  D:22  934989  D:15  963959


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  964331  D:23  964332  D: 0  964333  D:15  964273  D:16  964334


LRU Store
  D:20  875085  D:21  905037  D:22  934989  D:15  964273  C: 0  964331  D:23  964332  D: 0  964333  D:16  964334


PFF Store
  D:23  964332  D: 0  964333  C: 0  964331  D:15  964273  D:16  964334


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  983531  D:23  983532  D: 0  983533  D:16  983473  D:17  983534


LRU Store
  D:21  905037  D:22  934989  D:15  964348  D:16  983473  C: 0  983531  D:23  983532  D: 0  983533  D:17  983534


PFF Store
  D:23  983532  D: 0  983533  C: 0  983531  D:15  964348  D:16  983473  D:17  983534


******************************
```

```
LRU Store
  D:22  934989  D:15  964348  D:16  983548  D:17  1002673  C: 0  1002731  D:23  1002732  D: 0  1002733  D:18  1002734


PFF Store
  D:23  1002732  D: 0  1002733  C: 0  1002731  D:16  983548  D:17  1002673  D:18  1002734


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  1021931  D:23  1021932  D: 0  1021933  D:18  1021873  D:19  1021934


LRU Store
  D:15  964348  D:16  983548  D:17  1002748  D:18  1021873  C: 0  1021931  D:23  1021932  D: 0  1021933  D:19  1021934


PFF Store
  D:23  1021932  D: 0  1021933  C: 0  1021931  D:17  1002748  D:18  1021873  D:19  1021934


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  1041131  D:23  1041132  D: 0  1041133  D:19  1041073  D:20  1041134


LRU Store
  D:16  983548  D:17  1002748  D:18  1021948  D:19  1041073  C: 0  1041131  D:23  1041132  D: 0  1041133  D:20  1041134


PFF Store
  D:23  1041132  D: 0  1041133  C: 0  1041131  D:18  1021948  D:19  1041073  D:20  1041134


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  1060331  D:23  1060332  D: 0  1060333  D:20  1060273  D:21  1060334


LRU Store
```

```
  D:23  1060332   D: 0  1060333   C: 0  1060331   D:19  1041148   D:20  1060273   D:21  1060334


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  1079531   D:23  1079532   D: 0  1079533   D:21  1079473   D:22  1079534


LRU Store
  D:18  1021948   D:19  1041148   D:20  1060348   D:21  1079473   C: 0  1079531   D:23  1079532   D: 0  1079533   D:22  1079534


PFF Store
  D:23  1079532   D: 0  1079533   C: 0  1079531   D:20  1060348   D:21  1079473   D:22  1079534


******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  1117316   D:23  1117317   D: 0  1117318   D:15  1117319


LRU Store
  D:19  1041148   D:20  1060348   D:21  1079548   D:22  1098748   C: 0  1117316   D:23  1117317   D: 0  1117318   D:15  1117319
```

This manifestly does not happen in some situations. The reasons for this are varied. Firstly, with the large window size all pages referenced in the last 10,000 references remain in the working set. This will obviously include:-

i) the current code page

ii) the page containing the index variables

but depending on the complexity of the operations being carried out within a loop (in terms of the number of store references made) then an undetermined number of pages will be held in store.

Slightly modifying the current example could produce the following code:-

```
for i:= 1 to 2048 do
        element[i]:=0;
```

For each time round the loop there might be, say, five data storage accesses. Consequently, if memory accesses are estimated as roughly equivalent then as much as half of the array will be accumulated into the working set.

It would appear that the smaller window sizes are required for "techniques" of this type.

When the output for the smaller window size run is considered, an interesting anomaly appears. It is possible for the algorithm to swop out the current code page. This, theoretically, undesirable occurence stems from the particular implementation being considered, and the code page is removed when the array is being "initialised" to an internal "undefined" value. This is done by a single virtual machine instruction. Yet this machine instruction initiates 2048 storage references. After 500 of these, the current code page is no longer in the estimated locality.

This leads to the following results:-

PROPOSITION 4.13

> There exists a non-empty class of "real" programs for which the Working Set algorithm is a non-optimal estimator of locality.

PROOF

> See Examples 4.1.2 and 4.1.3.

Another drawback of this algorithm is its inability to determine quickly the cause of an increase in the working set size. Such an apparent expansion of the working set can be caused by:-

a) a change of locality, or

b) a true expansion of the working set.

No differentation can be made between these two cases until a time interval has elapsed. In fact, it is clear that, by the retrospective nature of this algorithm, it will always tend to over-estimate the working set size (See Example 4.1.4 and Diagram 4.3).

```
LRU Store
  C: 0  4  D:23  6  D: 0  7  D:15  13  D:16  269  D:17  525  D:18  526


PFF Store
  D:17  525  D:18  526
```

Example 4.1.2
Here the Working set algorithm swops out the current code page.

```
******************************
******************************
 LRU PFF Wor

WS  Store
  D:16  269  D:17  525  D:18  781  D:19  782
```

Example 4.1.3
The same will happen in any program with array initialisation, cf the programs of examples 4.2.4 and 4.3.5.

```
LRU Store
  C: 0  4  D:23  6  D: 0  7  D:15  13  D:16  269  D:17  525  D:18  781  D:19  782


PFF Store
  D:18  781  D:19  782



******************************
******************************
 LRU PFF Wor

WS  Store
  D:17  525  D:18  781  D:19  1037  D:20  1038


LRU Store
  D:23  6  D: 0  7  D:15  13  D:16  269  D:17  525  D:18  781  D:19  1037  D:20  1038


PFF Store
  D:19  1037  D:20  1038



******************************
******************************
 LRU PFF Wor

WS  Store
  D:18  781  D:19  1037  D:20  1293  D:21  1294
  D: 0   7  D:15   13  D:16   269  D:17   525  D:18   781  D:19   1037  D:20   1293  D:21   1294



PFF Store
  D:20   1293  D:21   1294



******************************
******************************
 LRU PFF Wor

WS  Store
  D:19   1037  D:20   1293  D:21   1549  D:22   1550



LRU Store
  D:15   13  D:16   269  D:17   525  D:18   781  D:19   1037  D:20   1293  D:21   1549  D:22   1550



PFF Store
  D:21   1549  D:22   1550



******************************
******************************
 LRU PFF Wor

WS  Store
  D:21   1549  D:22   1805  D:23   1806



LRU Store
  D:16   269  D:17   525  D:18   781  D:19   1037  D:20   1293  D:21   1549  D:22   1805  D:23   1806



PFF Store
  D:22   1805  D:23   1806



******************************
******************************
 LRU PFF Wor

WS  Store
  D:21   1549  D:22   1805  D:23   2058  D: 0   2059



LRU Store
  D:17   525  D:18   781  D:19   1037  D:20   1293  D:21   1549  D:22   1805  D:23   2058  D: 0   2059
```

```

    PFF Store
      D:23  2058  D: 0  2059



    ******************************
    ******************************
     LRU PFF Wor

    WS  Store
      D:21  1549  D:22  1805  D:23  2058  D: 0  2059  C: 0  2060



    LRU Store
      D:18  781  D:19  1037  D:20  1293  D:21  1549  D:22  1805  D:23  2058  D: 0  2059  C: 0  2060



    PFF Store
      D:23  2058  D: 0  2059  C: 0  2060



    ******************************
    ******************************
     PFF Wor

    WS  Store
      D:23  31932  D: 0  31941  C: 0  31939  D:22  31942



    LRU Store
      D:18  781  D:19  1037  D:20  1293  D:21  1549  D:23  31932  C: 0  31939  D: 0  31941  D:22  31942



    PFF Store
      D:23  31932  D: 0  31941  C: 0  31939  D:22  31942



    ******************************
    ******************************
     PFF Wor

    WS  Store
      D:23  62396  D: 0  62405  C: 0  62403  D:22  62287  D:21  62406



    LRU Store
      D:18  781  D:19  1037  D:20  1293  D:22  62287  D:23  62396  C: 0  62403  D: 0  62405  D:21  62406



```

```
******************************
******************************
 PFF Wor

WS  Store
  D:23  92860  D: 0  92869  C: 0  92867  D:21  92751  D:20  92870



LRU Store
  D:18  781  D:19  1037  D:22  62287  D:21  92751  D:23  92860  C: 0  92867  D: 0  92869  D:20  92870



PFF Store
  D:23  92860  D: 0  92869  C: 0  92867  D:21  92751  D:20  92870



******************************
******************************
 PFF Wor

WS  Store
  D:23  123324  D: 0  123333  C: 0  123331  D:20  123215  D:19  123334



LRU Store
  D:18  781  D:22  62287  D:21  92751  D:20  123215  D:23  123324  C: 0  123331  D: 0  123333  D:19  123334



PFF Store
  D:23  123324  D: 0  123333  C: 0  123331  D:20  123215  D:19  123334



******************************
******************************
 PFF Wor

WS  Store
  D:23  153788  D: 0  153797  C: 0  153795  D:19  153679  D:18  153798



LRU Store
  D:22  62287  D:21  92751  D:20  123215  D:19  153679  D:23  153788  C: 0  153795  D: 0  153797  D:18  153798



PFF Store
  D:23  153788  D: 0  153797  C: 0  153795  D:19  153679  D:18  153798

```

```
******************************
******************************
 LRU PFF Wor

WS  Store
  D:23  184252  D: 0  184261  C: 0  184259  D:18  184143  D:17  184262



LRU Store
  D:21  92751  D:20  123215  D:19  153679  D:18  184143  D:23  184252  C: 0  184259  D: 0  184261  D:17  184262



PFF Store
  D:23  184252  D: 0  184261  C: 0  184259  D:18  184143  D:17  184262



******************************
******************************
 LRU PFF Wor

WS  Store
  D:23  214716  D: 0  214725  C: 0  214723  D:17  214607  D:16  214726



LRU Store
  D:20  123215  D:19  153679  D:18  184143  D:17  214607  D:23  214716  C: 0  214723  D: 0  214725  D:16  214726



PFF Store
  D:23  214716  D: 0  214725  C: 0  214723  D:17  214607  D:16  214726



******************************
******************************
 LRU PFF Wor

WS  Store
  D:23  245180  D: 0  245189  C: 0  245187  D:16  245071  D:15  245190



LRU Store
  D:19  153679  D:18  184143  D:17  214607  D:16  245071  D:23  245180  C: 0  245187  D: 0  245189  D:15  245190



PFF Store
  D:23  245180  D: 0  245189  C: 0  245187  D:16  245071  D:15  245190



******************************
******************************
```

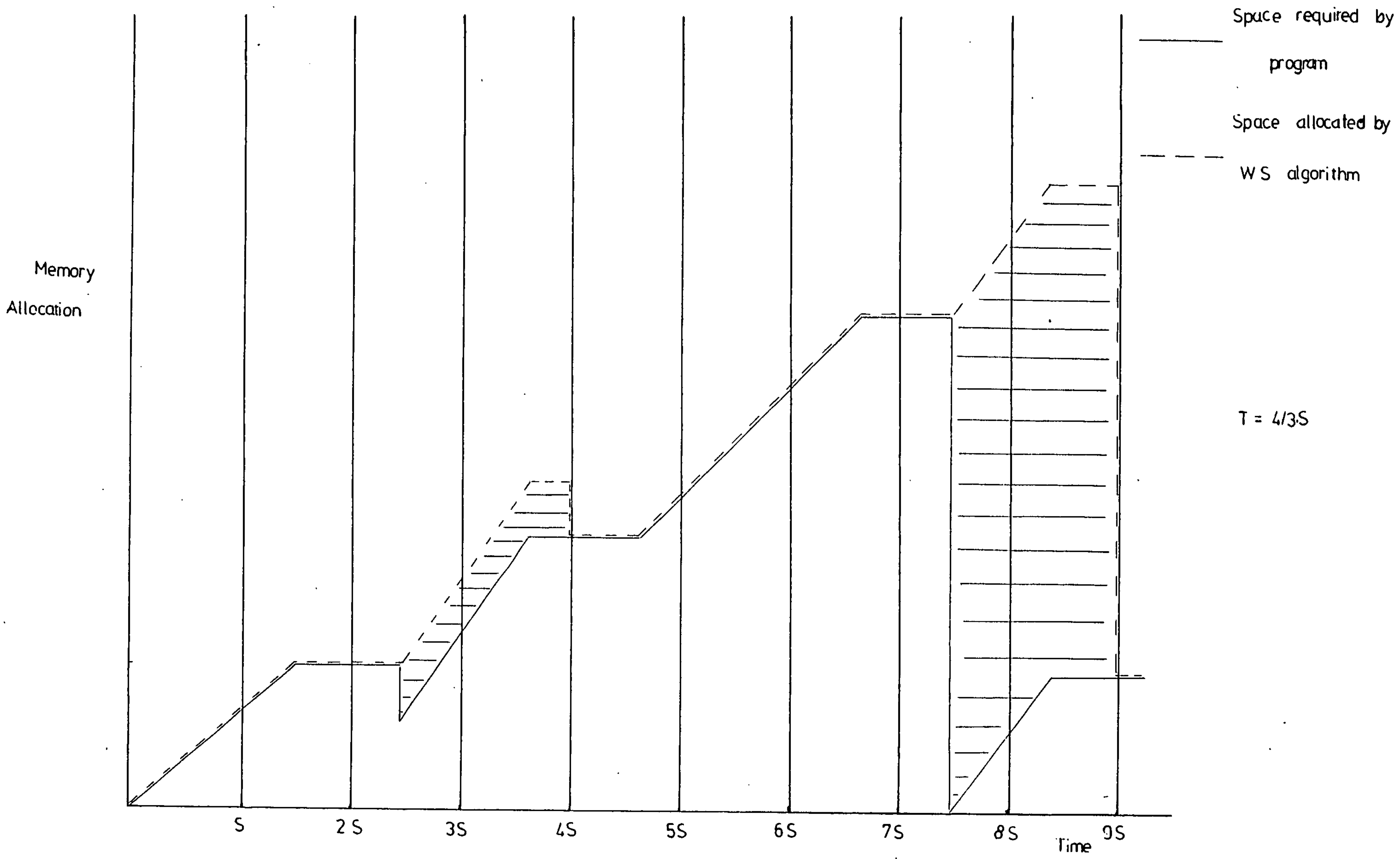
```

WS  Store
  D:23  246426  D: 0  246427  C: 0  246425  D:15  246367  D:16  246428



LRU Store
  D:19  153679  D:18  184143  D:17  214607  D:15  246367  C: 0  246425  D:23  246426  D: 0  246427  D:16  246428



PFF Store
  D:23  246426  D: 0  246427  C: 0  246425  D:16  246428  D:15  246367



******************************
******************************
 PFF Wor

WS  Store
  D:23  276378  D: 0  276379  C: 0  276377  D:16  276319  D:17  276380



LRU Store
  D:19  153679  D:18  184143  D:15  246534  D:16  276319  C: 0  276377  D:23  276378  D: 0  276379  D:17  276380



PFF Store
  D:23  276378  D: 0  276379  C: 0  276377  D:16  276319  D:15  246534  D:17  276380



******************************
******************************
 PFF Wor

WS  Store
  D:23  306330  D: 0  306331  C: 0  306329  D:17  306271  D:18  306332



LRU Store
  D:19  153679  D:15  246534  D:16  276486  D:17  306271  C: 0  306329  D:23  306330  D: 0  306331  D:18  306332



PFF Store
  D:23  306330  D: 0  306331  C: 0  306329  D:16  276486  D:17  306271  D:18  306332



******************************
******************************
 PFF Wor

WS  St
```

```



LRU Store
  D:15  246534  D:16  276486  D:17  306438  D:18  336223  C: 0  336281  D:23  336282  D: 0  336283  D:19  336284



PFF Store
  D:23  336282  D: 0  336283  C: 0  336281  D:17  306438  D:18  336223  D:19  336284



******************************
******************************
 LRU PFF Wor

WS  Store
  D:23  366234  D: 0  366235  C: 0  366233  D:19  366175  D:20  366236



LRU Store
  D:16  276486  D:17  306438  D:18  336390  D:19  366175  C: 0  366233  D:23  366234  D: 0  366235  D:20  366236



PFF Store
  D:23  366234  D: 0  366235  C: 0  366233  D:18  336390  D:19  366175  D:20  366236

```

EXAMPLE 4.1.4

Implementations of Working Set algorithms do not, in practice, follow exactly the theoretical model. It would be too expensive to check, after every reference, whether a given page remains in the working set or not. Consequently, the contents of the working set are only checked at intervals, usually known as strobe periods. This is also done after page faults. The size of the strobe interval is another arbitrary parameter that must be built into implementations of this algorithm. However there is a more significant disadvantage. This is most easily shown in a diagram (Diagram 4.3).

In this diagram is shown the memory requirements of a hypothetical program. At point (A) it is assumed that the program loses a number of pages from its current locality, but does not completely change it. Point (B) however represents a complete change of locality. The divisions along the horizontal axis are given in terms of strobe periods and T = 4/3 S. Note that it usually requires a strobe (or a page fault) at least T units after a locality change for the pages in that locality to be removed from the working set, if they are no longer required.



Space required by program
Space allocated by WS algorithm
Memory Allocation
T = 4/3.S
S
2 S
3S
4S
5S
6S
7S
8S
9S
Time

4.2 The LRU Replacement Algorithm

This is an example of what is known as a fixed-space policy (Den 75) in that the size of the set of pages in memory belonging to a program is kept fixed. In the theoretical model of the Working Set policy, this is not the case. In practical systems, more or less based on the Working Set philosophy, this is not always true however (Whi 73).

Coffman and Ryan (Cof 72) using a mathematical model of locality, showed, as might be anticipated, that variable space policies are always better than fixed. However the implementation of a pure Working Set strategy is very expensive. This is due to the fact that pages are freed whenever they leave the window and not simply at page fault times. As mentioned above this would involve testing each page in the working set after each reference, or decrementing some kind of counter associated with each page to see if that page was still eligible for membership of the working set. The implementation of a strobing technique to remove this large overhead widens the gap between theoretically achievable performance and the best practical implementations.

As a result of this cost, LRU algorithms, which are much cheaper to implement, have achieved considerable popularity.

With regard to the comparison of local LRU strategies and global LRU strategies, Oliver (Oli 74) has shown that the global LRU strategy performs better than the local LRU strategy where thrashing does not arise.

This is to some extent surprising and perplexing. Surprising because the most obvious criticism of the global LRU strategy is that those pages which have been, globally, unreferenced for the longest period of time are those belonging to the program which has not been running for the longest time. So, if the program, scheduling algorithm is to any extent "fair", then this program will have a high probability of being the next allowed to run. As a result of this, the global LRU algorithm would appear to tend to remove pages which might be referenced in the near future. Oliver states that, although evidence of this was found in his studies, it turned out that any such space could be more effectively used by the current program than by reserving it for future programs. These results are perplexing because the two other major algorithms are <u>local</u> algorithms, that is to say, they concern themselves only with pages belonging to the current program. To compare a global strategy with such local strategies is an extremely complex business. Not only does the mix of programs have to be considered for global strategies, but also the scheduling algorithm for the programs themselves has a significant effect. Both these factors concern the observed behaviour of a program as far as the user is concerned, in that a change of program mix over a number of runs of a program or a change of the scheduling algorithm (or its parameters) could affect the paging behaviour (and, consequently, on some systems the cost) of a running program.

For these reasons, it is proposed that a global algorithm is not a good idea <u>on principle</u>, and the local LRU only will be considered below.

With the local strategy the algorithm can be formulated as follows:-

> A program will be allocated a fixed amount of space, L pages. Initially, it will be allowed to acquire pages, if it requires them, up to this limit. The pages are conceived of as being ordered on a stack with the most recently used at the top and the least recently used page at the foot. If, when the program has acquired its L pages, it requests another page not already in store, then the page at the foot of the stack is freed and the new page will be brought in and placed at the top of the stack. Thus the memory allocation stays constant at L pages.

Theoretically some of the limitations of this approach are immediately apparent. (That these limitations can occur in practice will also be shown below.

Firstly, the store set size for a program remains fixed once it has acquired L pages. This tends to imply that programs whose locality sizes do not match this size behave poorly. This can manifest itself in two ways. Firstly, a program which requires more space than it has been allocated will thrash. That is to say, it will spend

EXAMPLE 4.2.1

```
Let N = {0,1,2,3,4,5,6,7}
    L = 3
and R =            0,0,1,2,7,0,1,2,7,0,1,2,7,....
then Store Set =   0 0 1 2 7 0 1 2 7 0 1 2 7
                   - - 0 1 2 7 0 1 2 7 0 1 2
                   - - - 0 1 2 7 0 1 2 7 0 1
                   *   * * * * * * * * * * *
```

where * implies the occurrence of a page fault.

Whereas, in the same situation, if L = 4 the following takes place:-

```
   R =             0,0,1,2,7,0,1,2,7,0,1,2,7,....
then Store Set =   0 0 1 2 7 0 1 2 7 0 1 2 7
                   - - 0 1 2 7 0 1 2 7 0 1 2
                   - - - 0 1 2 7 0 1 2 7 0 1
                   - - - - 0 1 2 7 0 1 2 7 0
                   *   * * *
```

with a significant reduction in the number of page faults.

<u>EXAMPLE 4.2.2</u>

```
Let N =  {0,1,2,3,4,5,6,7}
    L = 3
and R =             0,0,1,2,7,0,1,2,7,0,1,2,7,3,4,5,3,5,4,4,3,..
then Store Set =    0 0 1 2 7 0 1 2 7 0 1 2 7 3 4 5 3 5 4 4 3
                    - - 0 1 2 7 0 1 2 7 0 1 2 7 3 4 5 3 5 5 4
                    - - - 0 1 2 7 0 1 2 7 0 1 2 7 3 4 4 3 3 5
                    *   * * * * * * * * * * * * * * *
```

showing poor behaviour in the first part of the reference string but impeccable behaviour in the second part.

Again, with the same reference string and now L = 4 the following occurs:-

```
  R =               0,0,1,2,7,0,1,2,7,0,1,2,7,3,4,5,3,5,4,4,3
then Store Set =    0 0 1 2 7 0 1 2 7 0 1 2 7 3 4 5 3 5 4 4 3
                    - - 0 1 2 7 0 1 2 7 0 1 2 7 3 4 5 3 5 5 4
                    - - - 0 1 2 7 0 1 2 7 0 1 2 7 3 4 4 3 3 5
                    - - - - 0 1 2 7 0 1 2 7 0 1 2 7 7 7 7 7 7
                    *   * * *                     * * *
```

Although the paging behaviour has been improved in the first part of the string, in the second part the algorithm consistently over-estimates the locality size.

EXAMPLE 4.2.3

```
  Let N =  {0,1,2,3,4,5,6,7}
      L =  3
  and R =           1,2,3,1,2,3,1,3,2,4,2,3,4,4,3,2,....
  then Store Set  = 1 2 3 1 2 3 1 3 2 4 2 3 4 4 3 2
                    - 1 2 3 1 2 3 1 3 2 4 2 3 3 4 3
                    - - 1 2 3 1 2 2 1 3 3 4 2 2 2 4
                    * * *             *
```

As can be seen a "heavily" used page which is totally discarded can be readily handled.

However, an almost identical situation produces a different result:-

```
      R =           1,2,3,1,2,3,1,3,2,1,4,3,2,2,3,4,4,3,..
  then Store Set  = 1 2 3 1 2 3 1 3 2 1 4 3 2 2 3 4 4 3
                    - 1 2 3 1 2 3 1 3 2 1 4 3 3 2 3 3 4
                    - - 1 2 3 1 2 2 1 3 2 1 4 4 4 2 2 2
                    * * *               * * *
```

A "hiccup" has occurred due to the exact timing of the reference to the new page. This is equivalent to the disruption caused by a "casual" reference to one page:-

```
       R =          1,2,3,1,2,3,4,1,2,3,1,2,3,..
  then Store Set  = 1 2 3 1 2 3 4 1 2 3 1 2 3
                    - 1 2 3 1 2 3 4 1 2 3 1 2
                    - - 1 2 3 1 2 3 4 1 2 3 1
                    * * *       * * * *
```

more time paging than doing useful work (see Example 4.2.1). Secondly, a program whose locality size varies will alternate between a thrashing state (or a reasonably satisfactory state if L has been well chosen) and a state in which the memory in the system is poorly utilised (See Example 4.2.2).

Secondly, such a LRU strategy tends to favour programs which heavily use sets of pages and then discard them (see Example 4.2.3).

That these examples can be generated is not sufficient. It must be true that similar observations can be made in practice before the represesent a significant criticism of the algorithm itself.

The following examples (4.2.4, 4.2.5, 4.2.6) again in PASCAL show that these situations do indeed occur.

4.3 The Page Fault Frequency Algorithm

The underlying assumption of this algorithm is that a high page fault frequency indicates that a program is running inefficiently due to the fact that it has too little space allocated to it. Consequently, a page

```
program ex424(output);


(*****************************************************)
(* This program should show the effect of running     *)
(* LRU with too small a stacksize, if the chosen      *)
(* stacksize=4.                                       *)
(*****************************************************)

var
element:array[1..2048] of integer;
i:integer;
begin
   for i:=1 to 512 do
   begin
      element[i]:=1;
      element[i+512]:=2;
      element[i+1024]:=3;
      element[i+1536]:=4;
   end;
end.
```

Example 4.2.5

This is clearly shown in the output from Example 4.1.1, where if the LRU store is considered, the stack size of eight pages is a large overestimate for this problem.

```

WS  Store
  D:23  29953  D: 0  29960  C: 0  29958  D:15  29940  D:21  29937  D:19  29961  D:17  29863



LRU Store
  D:23  29953  C: 0  29958  D: 0  29960  D:19  29961



PFF Store
  D:23  29953  D: 0  29960  C: 0  29958  D:15  29940  D:21  29937  D:19  29961  D:17  29863



******************************
******************************
 LRU                                        Note the frequency of LRU faults.

WS  Store
  D:23  29963  D: 0  29960  C: 0  29962  D:15  29964  D:21  29937  D:19  29961  D:17  29863



LRU Store
  D:19  29961  C: 0  29962  D:23  29963  D:15  29964



PFF Store
  D:23  29963  D: 0  29960  C: 0  29962  D:15  29964  D:21  29937  D:19  29961  D:17  29863



******************************
******************************
 LRU

WS  Store
  D:23  29963  D: 0  29965  C: 0  29962  D:15  29964  D:21  29937  D:19  29961  D:17  29863



LRU Store
  C: 0  29962  D:23  29963  D:15  29964  D: 0  29965



PFF Store
  D:23  29963  D: 0  29965  C: 0  29962  D:15  29964  D:21  29937  D:19  29961  D:17  29863



******************************
******************************
 LRU

```

```



LRU Store
  D:23  29977  C: 0  29982  D: 0  29984  D:17  29985



FFF Store
  D:23  29977  D: 0  29984  C: 0  29982  D:15  29964  D:21  29937  D:19  29961  D:17  29985



******************************
******************************
 LRU

WS  Store
  D:23  29990  D: 0  29987  C: 0  29989  D:15  29991  D:21  29937  D:19  29961  D:17  29985



LRU Store
  D: 0  29987  C: 0  29989  D:23  29990  D:15  29991



FFF Store
  D:23  29990  D: 0  29987  C: 0  29989  D:15  29991  D:21  29937  D:19  29961  D:17  29985



******************************
******************************
 LRU

WS  Store
  D:23  30051  D: 0  30058  C: 0  30056  D:15  30038  D:21  30059  D:19  29961  D:17  29985



LRU Store
  D:23  30051  C: 0  30056  D: 0  30058  D:21  30059



FFF Store
  D:23  30051  D: 0  30058  C: 0  30056  D:15  30038  D:21  30059  D:19  29961  D:17  29985



******************************
******************************
 LRU

WS  Store
  D:23  30061  D: 0  30058  C: 0  30060  D:15  30062  D:21  30059  D:19  29961  D:17  29985


```

```
LRU Store
  D:21  30059  C: 0  30060  D:23  30061  D:15  30062



PFF Store
  D:23  30061  D: 0  30058  C: 0  30060  D:15  30062  D:21  30059  D:19  29961  D:17  29985



******************************
******************************
 LRU

WS  Store
  D:23  30061  D: 0  30063  C: 0  30060  D:15  30062  D:21  30059  D:19  29961  D:17  29985



LRU Store
  C: 0  30060  D:23  30061  D:15  30062  D: 0  30063



PFF Store
  D:23  30061  D: 0  30063  C: 0  30060  D:15  30062  D:21  30059  D:19  29961  D:17  29985



******************************
******************************
 LRU

WS  Store
  D:23  30075  D: 0  30082  C: 0  30080  D:15  30062  D:21  30059  D:19  30083  D:17  29985



LRU Store
  D:23  30075  C: 0  30080  D: 0  30082  D:19  30083



PFF Store
  D:23  30075  D: 0  30082  C: 0  30080  D:15  30062  D:21  30059  D:19  30083  D:17  29985



******************************
******************************
 LRU

WS  Store
  D:23  30085  D: 0  30082  C: 0  30084  D:15  30086  D:21  30059  D:19  30083  D:17  29985



LRU Store
  D:19  30083  C: 0  30084  D:23  30085  D:15  30086
```

```


PFF Store
  D:23  30085  D: 0  30082  C: 0  30084  D:15  30086  D:21  30059  D:19  30083  D:17  29985



******************************
******************************
 LRU

WS  Store
  D:23  30085  D: 0  30087  C: 0  30084  D:15  30086  D:21  30059  D:19  30083  D:17  29985



LRU Store
  C: 0  30084  D:23  30085  D:15  30086  D: 0  30087



PFF Store
  D:23  30085  D: 0  30087  C: 0  30084  D:15  30086  D:21  30059  D:19  30083  D:17  29985



******************************
******************************
 LRU

WS  Store
  D:23  30099  D: 0  30106  C: 0  30104  D:15  30086  D:21  30059  D:19  30083  D:17  30107



LRU Store
  D:23  30099  C: 0  30104  D: 0  30106  D:17  30107



PFF Store
  D:23  30099  D: 0  30106  C: 0  30104  D:15  30086  D:21  30059  D:19  30083  D:17  30107



******************************
******************************
 LRU

WS  Store
  D:23  30112  D: 0  30109  C: 0  30111  D:15  30113  D:21  30059  D:19  30083  D:17  30107



LRU Store
  D: 0  30109  C: 0  30111  D:23  30112  D:15  30113


```

```
       D:23  30112  D: 0  30109  C: 0  30111  D:15  30113  D:21  30059  D:19  30083  D:17  30107



     ******************************
     ******************************
      LRU

     WS  Store
       D:23  30173  D: 0  30180  C: 0  30178  D:15  30160  D:21  30181  D:19  30083  D:17  30107



     LRU Store
       D:23  30173  C: 0  30178  D: 0  30180  D:21  30181



     FFF Store
       D:23  30173  D: 0  30180  C: 0  30178  D:15  30160  D:21  30181  D:19  30083  D:17  30107



     ******************************
     ******************************
      LRU

     WS  Store
       D:23  30183  D: 0  30180  C: 0  30182  D:15  30184  D:21  30181  D:19  30083  D:17  30107



     LRU Store
       D:21  30181  C: 0  30182  D:23  30183  D:15  30184



     FFF Store
       D:23  30183  D: 0  30180  C: 0  30182  D:15  30184  D:21  30181  D:19  30083  D:17  30107



     ******************************
     ******************************
      LRU

     WS  Store
       D:23  30183  D: 0  30185  C: 0  30182  D:15  30184  D:21  30181  D:19  30083  D:17  30107



     LRU Store
       C: 0  30182  D:23  30183  D:15  30184  D: 0  30185



     FFF Store
       D:23  30183  D: 0  30185  C: 0  30182  D:15  30184  D:21  30181  D:19  30083  D:17  30107
```

Example 4.2.6

The LRU algorithm along with the others handles well a program with a small set of heavily used pages.

```
******************************
 LRU PFF Wor

WS  Store
  C: 0  1




LRU Store
  C: 0  1




PFF Store
  C: 0  1




******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  1  D:23  2




LRU Store
  C: 0  1  D:23  2




PFF Store
  C: 0  1  D:23  2




******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  1  D:23  2  D: 0  3




LRU Store
  C: 0  1  D:23  2  D: 0  3




PFF Store
  C: 0  1  D:23  2  D: 0  3




******************************
******************************
```

```
WS  Store
  C: 0  999  D:23  1000  D: 0  997



LRU Store
  D: 0  997  C: 0  999  D:23  1000



PFF Store
  C: 0  999  D:23  1000  D: 0  997



******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  1177  D:23  1172  D: 0  1178  C: 1  1179



LRU Store
  D:23  1172  C: 0  1177  D: 0  1178  C: 1  1179



PFF Store
  C: 0  1177  D:23  1172  D: 0  1178  C: 1  1179



******************************
     2     1     1
     3     3     2
******************************
 Str

WS  Store
  C: 0  1998  D:23  1993  D: 0  2000  C: 1  1179



LRU Store
  C: 1  1179  D:23  1993  C: 0  1998  D: 0  2000



PFF Store
  C: 0  1998  D:23  1993  D: 0  2000  C: 1  1179



******************************
     4     1     3
******************************
```

```
WS  Store
  C: 0  2996  D:23  2998  D: 0  3000



LRU Store
  C: 1  1179  C: 0  2996  D:23  2998  D: 0  3000



PFF Store
  C: 0  2996  D:23  2998  D: 0  3000  C: 1  1179



******************************
     5     2     1
******************************
 Str

WS  Store
  C: 0  3998  D:23  4000  D: 0  3999



LRU Store
  C: 1  1179  C: 0  3998  D: 0  3999  D:23  4000



PFF Store
  C: 0  3998  D:23  4000  D: 0  3999  C: 1  1179



******************************
     6     3     3
     7     1     2
******************************
 Str

WS  Store
  C: 0  5000  D:23  4994  D: 0  4999



LRU Store
  C: 1  1179  D:23  4994  D: 0  4999  C: 0  5000



PFF Store
  C: 0  5000  D:23  4994  D: 0  4999  C: 1  1179



******************************
******************************
```

```
WS  Store
  C: 0  5998  D:23  5991  D: 0  6000



LRU Store
  C: 1  1179  D:23  5991  C: 0  5998  D: 0  6000



PFF Store
  C: 0  5998  D:23  5991  D: 0  6000  C: 1  1179



******************************
     8     3     1
******************************
 Str

WS  Store
  C: 0  6991  D:23  6999  D: 0  7000



LRU Store
  C: 1  1179  C: 0  6991  D:23  6999  D: 0  7000



PFF Store
  C: 0  6991  D:23  6999  D: 0  7000  C: 1  1179



******************************
     9     2     3
******************************
 Str

WS  Store
  C: 0  7998  D:23  7993  D: 0  8000



LRU Store
  C: 1  1179  D:23  7993  C: 0  7998  D: 0  8000



PFF Store
  C: 0  7998  D:23  7993  D: 0  8000  C: 1  1179



******************************
******************************
 Str

```

```
  C: 0  8998  D:23  8995  D: 0  9000



LRU Store
  C: 1  1179  D:23  8995  C: 0  8998  D: 0  9000



PFF Store
  C: 0  8998  D:23  8995  D: 0  9000  C: 1  1179



******************************
******************************
 Str

WS  Store
  C: 0  10000  D:23  9995  D: 0  9999



LRU Store
  C: 1  1179  D:23  9995  D: 0  9999  C: 0  10000



PFF Store
  C: 0  10000  D:23  9995  D: 0  9999  C: 1  1179



******************************
******************************
 Str

WS  Store
  C: 0  10998  D:23  11000  D: 0  10997



LRU Store
  C: 1  1179  D: 0  10997  C: 0  10998  D:23  11000



PFF Store
  C: 0  10998  D:23  11000  D: 0  10997  C: 1  1179



******************************
******************************
 Str

WS  Store
  C: 0  12000  D:23  11995  D: 0  11999
```

```

LRU Store
  C: 1  1179  D:23  11995  D: 0  11999  C: 0  12000


PFF Store
  C: 0  12000  D:23  11995  D: 0  11999  C: 1  1179



******************************
     3     2     3
******************************
 Str

WS  Store
  C: 0  12996  D:23  12998  D: 0  13000



LRU Store
  C: 1  1179  C: 0  12996  D:23  12998  D: 0  13000



PFF Store
  C: 0  12996  D:23  12998  D: 0  13000  C: 1  1179



******************************
     4     3     1
******************************
 Str

WS  Store
  C: 0  13998  D:23  14000  D: 0  13996



LRU Store
  C: 1  1179  D: 0  13996  C: 0  13998  D:23  14000



PFF Store
  C: 0  13998  D:23  14000  D: 0  13996  C: 1  1179



******************************
     5     1     2
     6     3     3
******************************
 Str

WS  Store
  C: 0  14995  D:23  14998  D: 0  15000
```

```


LRU Store
  C: 1  1179  C: 0  14995  D:23  14998  D: 0  15000



PFF Store
  C: 0  14995  D:23  14998  D: 0  15000  C: 1  1179



******************************
     7     2     1
******************************
 Str

WS  Store
  C: 0  15999  D:23  15996  D: 0  16000



LRU Store
  C: 1  1179  D:23  15996  C: 0  15999  D: 0  16000



PFF Store
  C: 0  15999  D:23  15996  D: 0  16000  C: 1  1179



******************************
     8     1     3
******************************
 Str

WS  Store
  C: 0  17000  D:23  16995  D: 0  16999



LRU Store
  C: 1  1179  D:23  16995  D: 0  16999  C: 0  17000



PFF Store
  C: 0  17000  D:23  16995  D: 0  16999  C: 1  1179



******************************
******************************
 Str

WS  Store
  C: 0  18000  D:23  17997  D: 0  17999
```

```

LRU Store
  C: 1  1179  D:23  17997  D: 0  17999  C: 0  18000



PFF Store
  C: 0  18000  D:23  17997  D: 0  17999  C: 1  1179



******************************
     9     3     2
******************************
 Str

WS  Store
  C: 0  18998  D:23  19000  D: 0  18996



LRU Store
  C: 1  1179  D: 0  18996  C: 0  18998  D:23  19000



PFF Store
  C: 0  18998  D:23  19000  D: 0  18996  C: 1  1179



******************************
******************************
 Str

WS  Store
  C: 0  20000  D:23  19995  D: 0  19999



LRU Store
  C: 1  1179  D:23  19995  D: 0  19999  C: 0  20000



PFF Store
  C: 0  20000  D:23  19995  D: 0  19999  C: 1  1179



******************************
******************************
 Str

WS  Store
  C: 0  20998  D:23  21000  D: 0  20997



```

```
     C: 1  1179  D: 0  20997  C: 0  20998  D:23  21000



    PFF Store
     C: 0  20998  D:23  21000  D: 0  20997  C: 1  1179



    ******************************
    ******************************
     Str

    WS  Store
     C: 0  21999  D:23  21986  D: 0  22000



    LRU Store
     C: 1  1179  D:23  21986  C: 0  21999  D: 0  22000



    PFF Store
     C: 0  21999  D:23  21986  D: 0  22000  C: 1  1179



    ******************************
    ******************************
     Str

    WS  Store
     C: 0  22997  D:23  23000  D: 0  22999



    LRU Store
     C: 1  1179  C: 0  22997  D: 0  22999  D:23  23000



    PFF Store
     C: 0  22997  D:23  23000  D: 0  22999  C: 1  1179



    ******************************
    ******************************
     Wor

    WS  Store
     C: 0  23881  D:23  23883  D: 0  23880  C: 1  23884



    LRU Store
     D: 0  23880  C: 0  23881  D:23  23883  C: 1  23884
```

```

      PFF Store
        C: 0  23881  D:23  23883  D: 0  23880  C: 1  23884



      ******************************
      no solution

```

fault frequency, P, is defined as:-

$$P = 1/T_o$$

So that if the time between two consecutive page faults is less than T_o then the new page is added to the store set, otherwise pages are removed from the store set according to the Working Set policy, and the new page is added to the remaining set.

The authors of this algorithm, Chu and Opderbeck (Chu 72) present the following drawbacks of LRU and Working Set:-

> "...the major disadvantage of the LRU replacement algorithm is that it is not at all clear how many pages have to be allocated for different programs in order to assure efficient running without wasting space. In addition, this number is usually data dependent and may vary during execution. The Working Set algorithm constitutes a possible solution to this problem"

"In general, the Working Set algorithm can be considered as an LRU algorithm with variable size memory allocation. There is, however, a crucial difference. Using LRU pages are always replaced when a fault occurs. This does not apply to the Working Set algorithm. Here, page frames are freed whenever they have not been referenced for the last T msec. .... it appears to be rather expensive to implement the Working Set algorithm."

In support of their own algorithm, the authors state:-

"An "ideal" replacement algorithm should be independent of prior knowledge about program behaviour; instead, all of the information needed to assure efficient memory allocation should be gathered during program execution."

These authors consider their own algorithm to be roughly a Working Set algorithm with a variable T.

A study of the PFF algorithm by Sadeh (Sad 75) using a mathematical model has been carried out. This study is important in that it draws attention to the limitations of mathematical models of program behaviour:-

"no presently available satisfactory model of program behaviour incorporates localities of different sizes and the transitions between them"

A criticism of the simple LRU model, supported by Denning (Spi 72) is also made:-

"The main drawback of the simple LRU stack model is that it generates reference strings that do not reflect transitions between localities."

These limitations reduce the applicability of the results presented. However it is possible both theoretically and practically to demonstrate the drawbacks of this algorithm.

As might be expected, major difficulties arise with the choice of frequency threshold (see Examples 4.3.1 and 4.3.2). The practical realisation of this problem is shown in Example 4.3.3.

Another limitation, only partly alleviated by the Sadeh amendment (2.4.1.5) concerns locality changes. It is also to be expected that at a locality change programs may refer to pages in both localities for a short period. If this period coincides with the acquisition of all the pages of the new locality, then it is possible, due to the fact that pages are only removed at the time of a page fault, that this algorithm will over-estimate the page requirements of a program for a considerable period after a locality change (see Examples 4.3.4 and 4.3.5).

4.4 Improving the Behaviour of Current Algorithms

A number of authors have appreciated some of the drawbacks presented above, and have attempted to improve the behaviour of the algorithms. The techniques that will be considered

EXAMPLE 4.3.1

Let $N = \{0,1,2,3,4,5,6,7,8,9\}$

and $R = 1,2,3,4,5,6,\ldots\ldots$

If the inter-page fault time is less than T_o then the

$$\text{Store Set} = \begin{array}{cccccc} 1 & 1 & 1 & 1 & 1 & 1 \\ & 2 & 2 & 2 & 2 & 2 \\ & & 3 & 3 & 3 & 3 \\ & & & 4 & 4 & 4 \\ & & & & 5 & 5 \\ & & & & & 6 \end{array}$$

Some other criterion must be applied to prevent this store set expanding until all pages have been acquired. Since, if all pages are acquired then no page faults occur and no pages will be removed. It is to remove this problem that the Sadeh amendment was proposed.

EXAMPLE 4.3.2

In Example 4.3.1 if the inter-page fault time had been greater than T_o then the

$$\text{Store Set} = \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \\ & 1 & 2 & 3 & 4 & 5 \end{array}$$

```
 ******************************
  LRU PFF Wor

 WS  Store
   C: 0  1



 LRU Store
   C: 0  1



 PFF Store
   C: 0  1



 ******************************
 ******************************
  LRU PFF Wor

 WS  Store
   C: 0  1  D:23  2



 LRU Store
   C: 0  1  D:23  2



 PFF Store
   C: 0  1  D:23  2



 ******************************
 ******************************
  LRU PFF Wor

 WS  Store
   C: 0  1  D:23  2  D: 0  3



 LRU Store
   C: 0  1  D:23  2  D: 0  3



 PFF Store
   C: 0  1  D:23  2  D: 0  3



 ******************************
 ******************************
```

The trace for the PFF algorithm is identical for a critical frequency of both 100 and 500 references.

```
WS  Store
  C: 0  500  D:23  495  D: 0  499



LRU Store
  D:23  495  D: 0  499  C: 0  500



PFF Store
  C: 0  500  D:23  495  D: 0  499



******************************
******************************
 Str

WS  Store
  C: 0  999  D:23  1000  D: 0  997



LRU Store
  D: 0  997  C: 0  999  D:23  1000



PFF Store
  C: 0  999  D:23  1000  D: 0  997



******************************
******************************
 LRU PFF Wor

WS  Store
  C: 0  1177  D:23  1172  D: 0  1178  C: 1  1179



LRU Store
  D:23  1172  C: 0  1177  D: 0  1178  C: 1  1179



PFF Store
  C: 0  1177  D:23  1172  D: 0  1178  C: 1  1179



******************************
     2     1     1
******************************
 Str

WS  Store
```

```



LRU Store
  C: 1  1179  C: 0  1487  D: 0  1499  D:23  1500


PFF Store
  C: 0  1487  D:23  1500  D: 0  1499  C: 1  1179



******************************
    3     3     2
******************************
 Str

WS  Store
  C: 0  1998  D:23  1993  D: 0  2000  C: 1  1179


LRU Store
  C: 1  1179  D:23  1993  C: 0  1998  D: 0  2000


PFF Store
  C: 0  1998  D:23  1993  D: 0  2000  C: 1  1179



******************************
    4     1     3
******************************
 Str

WS  Store
  C: 0  2497  D:23  2499  D: 0  2500


LRU Store
  C: 1  1179  C: 0  2497  D:23  2499  D: 0  2500


PFF Store
  C: 0  2497  D:23  2499  D: 0  2500  C: 1  1179



******************************
******************************
 Str

WS  Store
  C: 0  2996  D:23  2999  D: 0  3000
```

```


LRU Store
  C: 1   1179   C: 0   2996   D:23   2998   D: 0   3000



PFF Store
  C: 0   2996   D:23   2998   D: 0   3000   C: 1   1179



******************************
******************************
 Str

WS  Store
  C: 0   3500   D:23   3497   D: 0   3499



LRU Store
  C: 1   1179   D:23   3497   D: 0   3499   C: 0   3500



PFF Store
  C: 0   3500   D:23   3497   D: 0   3499   C: 1   1179



******************************
     5     2     1
******************************
 Str

WS  Store
  C: 0   3998   D:23   4000   D: 0   3999



LRU Store
  C: 1   1179   C: 0   3998   D: 0   3999   D:23   4000



PFF Store
  C: 0   3998   D:23   4000   D: 0   3999   C: 1   1179



******************************
     6     3     3
******************************
 Str

WS  Store
  C: 0   4497   D:23   4488   D: 0   4500
```

```

        LRU Store
          C: 1   1179   D:23   4488   C: 0   4497   D: 0   4500



        PFF Store
          C: 0   4497   D:23   4488   D: 0   4500   C: 1   1179



        ******************************
             7      1      2
        ******************************
         Str

        WS  Store
          C: 0   5000   D:23   4994   D: 0   4999



        LRU Store
          C: 1   1179   D:23   4994   D: 0   4999   C: 0   5000



        PFF Store
          C: 0   5000   D:23   4994   D: 0   4999   C: 1   1179



        ******************************
        ******************************
         Str

        WS  Store
          C: 0   5498   D:23   5500   D: 0   5499



        LRU Store
          C: 1   1179   C: 0   5498   D: 0   5499   D:23   5500



        PFF Store
          C: 0   5498   D:23   5500   D: 0   5499   C: 1   1179



        ******************************
        ******************************
         Str

        WS  Store
          C: 0   5998   D:23   5991   D: 0   6000


```

```
    C: 1   1179   D:23   5991   C: 0   5998   D: 0   6000



  PFF Store
    C: 0   5998   D:23   5991   D: 0   6000   C: 1   1179




  ******************************
       8     3     1
  ******************************
   Str

  WS  Store
    C: 0   6500   D:23   6497   D: 0   6499



  LRU Store
    C: 1   1179   D:23   6497   D: 0   6499   C: 0   6500



  PFF Store
    C: 0   6500   D:23   6497   D: 0   6499   C: 1   1179



  ******************************
  ******************************
   Str

  WS  Store
    C: 0   6991   D:23   6999   D: 0   7000



  LRU Store
    C: 1   1179   C: 0   6991   D:23   6999   D: 0   7000



  PFF Store
    C: 0   6991   D:23   6999   D: 0   7000   C: 1   1179




  ******************************
       9     2     3
  ******************************
   Str

  WS  Store
    C: 0   7500   D:23   7497   D: 0   7499



```

```



      PFF Store
        C: 0  7500  D:23  7497  D: 0  7499  C: 1  1179



      ******************************
      ******************************
       Str

      WS  Store
        C: 0  7998  D:23  7993  D: 0  8000



      LRU Store
        C: 1  1179  D:23  7993  C: 0  7998  D: 0  8000



      PFF Store
        C: 0  7998  D:23  7993  D: 0  8000  C: 1  1179



      ******************************
      ******************************
       Str

      WS  Store
        C: 0  8497  D:23  8500  D: 0  8499



      LRU Store
        C: 1  1179  C: 0  8497  D: 0  8499  D:23  8500



      PFF Store
        C: 0  8497  D:23  8500  D: 0  8499  C: 1  1179



      ******************************
      ******************************
       Str

      WS  Store
        C: 0  8998  D:23  8995  D: 0  9000



      LRU Store
        C: 1  1179  D:23  8995  C: 0  8998  D: 0  9000

```

```
PFF Store
  C: 0  8998  D:23  8995  D: 0  9000  C: 1  1179



******************************
******************************
 Str

WS  Store
  C: 0  9499  D:23  9492  D: 0  9500



LRU Store
  C: 1  1179  D:23  9492  C: 0  9499  D: 0  9500



PFF Store
  C: 0  9499  D:23  9492  D: 0  9500  C: 1  1179



******************************
******************************
 Str

WS  Store
  C: 0  10000  D:23  9995  D: 0  9999



LRU Store
  C: 1  1179  D:23  9995  D: 0  9999  C: 0  10000



PFF Store
  C: 0  10000  D:23  9995  D: 0  9999  C: 1  1179



******************************
******************************
 Str

WS  Store
  C: 0  10500  D:23  10487  D: 0  10499



LRU Store
  C: 1  1179  D:23  10487  D: 0  10499  C: 0  10500



PFF Store
  C: 0  10500  D:23  10487  D: 0  10499  C: 1  1179
```

```


      ******************************
      ******************************
       Str

      WS  Store
        C: 0  10998  D:23  11000  D: 0  10997



      LRU Store
        C: 1  1179  D: 0  10997  C: 0  10998  D:23  11000



      PFF Store
        C: 0  10998  D:23  11000  D: 0  10997  C: 1  1179



      ******************************
      ******************************
       Str

      WS  Store
        C: 0  11500  D:23  11490  D: 0  11499



      LRU Store
        C: 1  1179  D:23  11490  D: 0  11499  C: 0  11500



      PFF Store
        C: 0  11500  D:23  11490  D: 0  11499  C: 1  1179



      ******************************
      ******************************
       Str

      WS  Store
        C: 0  12000  D:23  11995  D: 0  11999



      LRU Store
        C: 1  1179  D:23  11995  D: 0  11999  C: 0  12000



      PFF Store
        C: 0  12000  D:23  11995  D: 0  11999  C: 1  1179


      3     2     3
******************************
 Str

WS  Store
  C: 0  12497  D:23  12492  D: 0  12500



LRU Store
  C: 1  1179  D:23  12492  C: 0  12497  D: 0  12500



PFF Store
  C: 0  12497  D:23  12492  D: 0  12500  C: 1  1179



******************************
******************************
 Str

WS  Store
  C: 0  12996  D:23  12998  D: 0  13000



LRU Store
  C: 1  1179  C: 0  12996  D:23  12998  D: 0  13000



PFF Store
  C: 0  12996  D:23  12998  D: 0  13000  C: 1  1179



******************************
******************************
 Str

WS  Store
  C: 0  13500  D:23  13495  D: 0  13499



LRU Store
  C: 1  1179  D:23  13495  D: 0  13499  C: 0  13500



PFF Store
  C: 0  13500  D:23  13495  D: 0  13499  C: 1  1179



******************************
```

```
 Str

WS  Store
  C: 0  13998  D:23  14000  D: 0  13996



LRU Store
  C: 1  1179  D: 0  13996  C: 0  13998  D:23  14000



PFF Store
  C: 0  13998  D:23  14000  D: 0  13996  C: 1  1179



******************************
     5     1     2
******************************
 Str

WS  Store
  C: 0  14498  D:23  14500  D: 0  14499



LRU Store
  C: 1  1179  C: 0  14498  D: 0  14499  D:23  14500



PFF Store
  C: 0  14498  D:23  14500  D: 0  14499  C: 1  1179



******************************
     6     3     3
******************************
 Str

WS  Store
  C: 0  14995  D:23  14998  D: 0  15000



LRU Store
  C: 1  1179  C: 0  14995  D:23  14998  D: 0  15000



PFF Store
  C: 0  14995  D:23  14998  D: 0  15000  C: 1  1179



******************************
******************************
```

```

WS  Store
  C: 0  15499  D:23  15500  D: 0  15498



LRU Store
  C: 1  1179  D: 0  15498  C: 0  15499  D:23  15500



PFF Store
  C: 0  15499  D:23  15500  D: 0  15498  C: 1  1179



******************************
     7     2     1
******************************
 Str

WS  Store
  C: 0  15999  D:23  15996  D: 0  16000



LRU Store
  C: 1  1179  D:23  15996  C: 0  15999  D: 0  16000



PFF Store
  C: 0  15999  D:23  15996  D: 0  16000  C: 1  1179



******************************
******************************
 Str

WS  Store
  C: 0  16500  D:23  16497  D: 0  16499



LRU Store
  C: 1  1179  D:23  16497  D: 0  16499  C: 0  16500



PFF Store
  C: 0  16500  D:23  16497  D: 0  16499  C: 1  1179



******************************
     8     1     3
******************************
```

```
WS  Store
  C: 0  17000  D:23  16995  D: 0  16999



LRU Store
  C: 1  1179  D:23  16995  D: 0  16999  C: 0  17000



PFF Store
  C: 0  17000  D:23  16995  D: 0  16999  C: 1  1179



******************************
******************************
 Str

WS  Store
  C: 0  17500  D:23  17495  D: 0  17499



LRU Store
  C: 1  1179  D:23  17495  D: 0  17499  C: 0  17500



PFF Store
  C: 0  17500  D:23  17495  D: 0  17499  C: 1  1179



******************************
******************************
 Str

WS  Store
  C: 0  18000  D:23  17997  D: 0  17999



LRU Store
  C: 1  1179  D:23  17997  D: 0  17999  C: 0  18000



PFF Store
  C: 0  18000  D:23  17997  D: 0  17999  C: 1  1179



******************************
     9     3     2
******************************
 Str

WS  Store
```

```



LRU Store
  C: 1  1179  D:23  18487  D: 0  18499  C: 0  18500



PFF Store
  C: 0  18500  D:23  18487  D: 0  18499  C: 1  1179



******************************
******************************
 Str

WS  Store
  C: 0  18998  D:23  19000  D: 0  18996



LRU Store
  C: 1  1179  D: 0  18996  C: 0  18998  D:23  19000



PFF Store
  C: 0  18998  D:23  19000  D: 0  18996  C: 1  1179



******************************
******************************
 Str

WS  Store
  C: 0  19500  D:23  19494  D: 0  19499



LRU Store
  C: 1  1179  D:23  19494  D: 0  19499  C: 0  19500



PFF Store
  C: 0  19500  D:23  19494  D: 0  19499  C: 1  1179



******************************
******************************
 Str

WS  Store
  C: 0  20000  D:23  19995  D: 0  19999

```

```
LRU Store
  C: 1   1179  D:23   19995  D: 0   19999  C: 0   20000



PFF Store
  C: 0   20000  D:23   19995  D: 0   19999  C: 1   1179



*****************************
*****************************
 Str

WS  Store
  C: 0   20497  D:23   20488  D: 0   20500



LRU Store
  C: 1   1179  D:23   20488  C: 0   20497  D: 0   20500



PFF Store
  C: 0   20497  D:23   20488  D: 0   20500  C: 1   1179



*****************************
*****************************
 Str

WS  Store
  C: 0   20998  D:23   21000  D: 0   20997



LRU Store
  C: 1   1179  D: 0   20997  C: 0   20998  D:23   21000



PFF Store
  C: 0   20998  D:23   21000  D: 0   20997  C: 1   1179



*****************************
*****************************
 Str

WS  Store
  C: 0   21499  D:23   21496  D: 0   21500



LRU Store
  C: 1   1179  D:23   21496  C: 0   21499  D: 0   21500
```

```


      PFF Store
        C: 0  21499  D:23  21496  D: 0  21500  C: 1  1179



      ******************************
      ******************************
       Str

      WS  Store
        C: 0  21999  D:23  21986  D: 0  22000



      LRU Store
        C: 1  1179  D:23  21986  C: 0  21999  D: 0  22000



      PFF Store
        C: 0  21999  D:23  21986  D: 0  22000  C: 1  1179



      ******************************
      ******************************
       Str

      WS  Store
        C: 0  22496  D:23  22498  D: 0  22500



      LRU Store
        C: 1  1179  C: 0  22496  D:23  22498  D: 0  22500



      PFF Store
        C: 0  22496  D:23  22498  D: 0  22500  C: 1  1179



      ******************************
      ******************************
       Str

      WS  Store
        C: 0  22997  D:23  23000  D: 0  22999



      LRU Store
        C: 1  1179  C: 0  22997  D: 0  22999  D:23  23000


      C: 0  22997  D:23  23000  D: 0  22999  C: 1  1179



    ******************************
    ******************************
     Str

    WS  Store
      C: 0  23496  D:23  23498  D: 0  23500



    LRU Store
      C: 1  1179  C: 0  23496  D:23  23498  D: 0  23500



    PFF Store
      C: 0  23496  D:23  23498  D: 0  23500  C: 1  1179



    ******************************
    ******************************
     Wor

    WS  Store
      C: 0  23881  D:23  23883  D: 0  23880  C: 1  23884



    LRU Store
      D: 0  23880  C: 0  23881  D:23  23883  C: 1  23884



    PFF Store
      C: 0  23881  D:23  23883  D: 0  23880  C: 1  23884



    ******************************
    no solution
```

EXAMPLE 4.3.4

This is a necessarily simplified example.

Let N = $\{0,1,2,3,4,5,6,7\}$

Critical Frequency = 1/3

```
    R =          1,2,3,4,1,4,3,2,1,4,3,4,3,2,1,7,5,6,5,7,6
then Store Set = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                   2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
                     3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
                       4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
                                               7 7 7 7 7 7
                                                 5 5 5 5 5
                                                   6 6 6 6
                 * * * *                       * * *
                                               A B B
```

A - Although the page fault frequency is less than the critical frequency, no page is removed since all have been used since the last page fault.

B - The page fault frequency is higher than the critical frequency, consequently pages are added <u>without</u> replacement.

In this situation, the over-estimate will exist until the first page fault that creates a lower frequency than the critical frequency. (Whether this be a natural fault or a "pseudo-fault".)

```
program ex435(output)


(*************************************************************)
(* This program should show the effect of the PFF algorithm   *)
(* holding onto pages after the indicated locality change     *)
(*************************************************************)


   element:array[1..2048] of integer;
   i:integer;
begin
   for i:=1 to 512 do
   begin

      element[i]:=1;
      element[i+512]:=2;
      element[i+1024]:=3;
   end;
(*************************************************************)
(*            LOCALITY CHANGE                                 *)
(*************************************************************)
   for i:=1537 to 2048 do element[i]:=4;
end.
```

```
******************************
******************************
 Str

WS  Store
  D:23  6295  D: 0  6300  C: 0  6299  D:15  6296  D:21  6237  D:19  6261



LRU Store
  D:21  6237  D:19  6261  D:23  6295  D:15  6296  C: 0  6299  D: 0  6300



PFF Store
  D:23  6295  D: 0  6300  C: 0  6299  D:15  6296  D:21  6237  D:19  6261



******************************
******************************
 Str

WS  Store
  D:23  6393  D: 0  6399  C: 0  6400  D:15  6394  D:21  6335  D:19  6359



LRU Store
  D:21  6335  D:19  6359  D:23  6393  D:15  6394  D: 0  6399  C: 0  6400



PFF Store
  D:23  6393  D: 0  6399  C: 0  6400  D:15  6394  D:21  6335  D:19  6359



******************************
******************************
 Str

WS  Store
  D:23  6499  D: 0  6500  C: 0  6498  D:15  6492  D:21  6433  D:19  6457



LRU Store
  D:21  6433  D:19  6457  D:15  6492  C: 0  6498  D:23  6499  D: 0  6500



PFF Store
  D:23  6499  D: 0  6500  C: 0  6498  D:15  6492  D:21  6433  D:19  6457



******************************
```

```
         WS  Store
           D:23  6597  D: 0  6599  C: 0  6600  D:15  6590  D:21  6531  D:19  6555



         LRU Store
           D:21  6531  D:19  6555  D:15  6590  D:23  6597  D: 0  6599  C: 0  6600



         PFF Store
           D:23  6597  D: 0  6599  C: 0  6600  D:15  6590  D:21  6531  D:19  6555



         ******************************
         ******************************
          Str

         WS  Store
           D:23  6695  D: 0  6699  C: 0  6700  D:15  6688  D:21  6629  D:19  6653



         LRU Store
           D:21  6629  D:19  6653  D:15  6688  D:23  6695  D: 0  6699  C: 0  6700



         PFF Store
           D:23  6695  D: 0  6699  C: 0  6700  D:15  6688  D:21  6629  D:19  6653



         ******************************
         ******************************
          Str

         WS  Store
           D:23  6793  D: 0  6800  C: 0  6798  D:15  6786  D:21  6727  D:19  6751



         LRU Store
           D:21  6727  D:19  6751  D:15  6786  D:23  6793  C: 0  6798  D: 0  6800



         PFF Store
           D:23  6793  D: 0  6800  C: 0  6798  D:15  6786  D:21  6727  D:19  6751



         ******************************
         ******************************
          Str

         WS  Store
```

```



LRU Store
  D:21  6825  D:19  6849  D:15  6884  D: 0  6898  D:23  6899  C: 0  6900



PFF Store
  D:23  6899  D: 0  6898  C: 0  6900  D:15  6884  D:21  6825  D:19  6849



******************************
******************************
 Str

WS  Store
  D:23  6993  D: 0  7000  C: 0  6999  D:15  6994  D:21  6923  D:19  6947



LRU Store
  D:21  6923  D:19  6947  D:23  6993  D:15  6994  C: 0  6999  D: 0  7000



PFF Store
  D:23  6993  D: 0  7000  C: 0  6999  D:15  6994  D:21  6923  D:19  6947



******************************
******************************
 LRU PFF Wor
```

This is the last actual page fault, and is comparable to the locality change indicated.

```

WS  Store
  D:23  7014  D: 0  7021  C: 0  7019  D:15  7007  D:21  6923  D:19  6947  D:17  7022



LRU Store
  D:19  6947  D:15  7007  D:23  7014  C: 0  7019  D: 0  7021  D:17  7022



PFF Store
  D:23  7014  D: 0  7021  C: 0  7019  D:15  7007  D:21  6923  D:19  6947  D:17  7022



******************************
******************************
 Str

WS  Store
  D:23  7093  D: 0  7100  C: 0  7099  D:15  7094  D:21  6923  D:19  6947  D:17  7072

```

```
LRU Store
  D:19  6947  D:17  7072  D:23  7093  D:15  7094  C: 0  7099  D: 0  7100



PFF Store
  D:23  7093  D: 0  7100  C: 0  7099  D:15  7094  D:21  6923  D:19  6947  D:17  7072



******************************
******************************
 Str
```

The effect of the Working set "strobe" is shown in this output to indicate where the Working set algorithm removes the unnecessary pages.

```
WS  Store
  D:23  7193  D: 0  7200  C: 0  7199  D:15  7194  D:21  6923  D:19  6947  D:17  7172



LRU Store
  D:19  6947  D:17  7172  D:23  7193  D:15  7194  C: 0  7199  D: 0  7200



PFF Store
  D:23  7193  D: 0  7200  C: 0  7199  D:15  7194  D:21  6923  D:19  6947  D:17  7172



******************************
******************************
 Str

WS  Store
  D:23  7293  D: 0  7300  C: 0  7299  D:15  7294  D:21  6923  D:19  6947  D:17  7272



LRU Store
  D:19  6947  D:17  7272  D:23  7293  D:15  7294  C: 0  7299  D: 0  7300



PFF Store
  D:23  7293  D: 0  7300  C: 0  7299  D:15  7294  D:21  6923  D:19  6947  D:17  7272



******************************
******************************
 Str

WS  Store
  D:23  7393  D: 0  7400  C: 0  7399  D:15  7394  D:21  6923  D:19  6947  D:17  7372



LRU Store
  D:19  6947  D:17  7372  D:23  7393  D:15  7394  C: 0  7399  D: 0  7400
```

```


PFF Store
  D:23  7393  D: 0  7400  C: 0  7399  D:15  7394  D:21  6923  D:19  6947  D:17  7372



******************************
******************************
 Str

WS  Store
  D:23  7493  D: 0  7500  C: 0  7499  D:15  7494  D:17  7472



LRU Store
  D:19  6947  D:17  7472  D:23  7493  D:15  7494  C: 0  7499  D: 0  7500



PFF Store
  D:23  7493  D: 0  7500  C: 0  7499  D:15  7494  D:21  6923  D:19  6947  D:17  7472



******************************
******************************
 Str

WS  Store
  D:23  7593  D: 0  7600  C: 0  7599  D:15  7594  D:17  7572



LRU Store
  D:19  6947  D:17  7572  D:23  7593  D:15  7594  C: 0  7599  D: 0  7600



PFF Store
  D:23  7593  D: 0  7600  C: 0  7599  D:15  7594  D:21  6923  D:19  6947  D:17  7572



******************************
******************************
 Str

WS  Store
  D:23  7693  D: 0  7700  C: 0  7699  D:15  7694  D:17  7672



LRU Store
  D:19  6947  D:17  7672  D:23  7693  D:15  7694  C: 0  7699  D: 0  7700



```

```
            D:23  7693  D: 0  7700  C: 0  7699  D:15  7694  D:21  6923  D:19  6947  D:17  7672



          ******************************
          ******************************
           Str

          WS  Store
            D:23  7793  D: 0  7800  C: 0  7799  D:15  7794  D:17  7772



          LRU Store
            D:19  6947  D:17  7772  D:23  7793  D:15  7794  C: 0  7799  D: 0  7800



          PFF Store
            D:23  7793  D: 0  7800  C: 0  7799  D:15  7794  D:21  6923  D:19  6947  D:17  7772



          ******************************
          ******************************
           Str

          WS  Store
            D:23  7893  D: 0  7900  C: 0  7899  D:15  7894  D:17  7872



          LRU Store
            D:19  6947  D:17  7872  D:23  7893  D:15  7894  C: 0  7899  D: 0  7900



          PFF Store
            D:23  7893  D: 0  7900  C: 0  7899  D:15  7894  D:21  6923  D:19  6947  D:17  7872



          ******************************
          ******************************
           Str

          WS  Store
            D:23  7993  D: 0  8000  C: 0  7999  D:15  7994  D:17  7972



          LRU Store
            D:19  6947  D:17  7972  D:23  7993  D:15  7994  C: 0  7999  D: 0  8000



          PFF Store
            D:23  7993  D: 0  8000  C: 0  7999  D:15  7994  D:21  6923  D:19  6947  D:17  7972
```

```

******************************
******************************
 Str

WS  Store
  D:23  8093  D: 0  8100  C: 0  8099  D:15  8094  D:17  8072



LRU Store
  D:19  6947  D:17  8072  D:23  8093  D:15  8094  C: 0  8099  D: 0  8100



PFF Store
  D:23  8093  D: 0  8100  C: 0  8099  D:15  8094  D:21  6923  D:19  6947  D:17  8072



******************************
******************************
 Str

WS  Store
  D:23  8193  D: 0  8200  C: 0  8199  D:15  8194  D:17  8172



LRU Store
  D:19  6947  D:17  8172  D:23  8193  D:15  8194  C: 0  8199  D: 0  8200



PFF Store
  D:23  8193  D: 0  8200  C: 0  8199  D:15  8194  D:21  6923  D:19  6947  D:17  8172



******************************
******************************
 Str

WS  Store
  D:23  8293  D: 0  8300  C: 0  8299  D:15  8294  D:17  8272



LRU Store
  D:19  6947  D:17  8272  D:23  8293  D:15  8294  C: 0  8299  D: 0  8300



PFF Store
  D:23  8293  D: 0  8300  C: 0  8299  D:15  8294  D:21  6923  D:19  6947  D:17  8272



******************************
```

here are:-

1. Program Restructuring
2. Swapped Working Sets
3. Critical Working Sets

4.4.1 Program Restructuring

Hatfield and Gerald (Hat 71) and Hatfield (Hat 72) developed techniques for examining programs that are to be run in virtual memory systems, and for reducing their physical memory requirements with little or no recoding. A program is divided into sectors, which represent contiguous locations which are logically associated one with another. A "nearness matrix" is constructed during a pre-run of the program wherein the numbers of references from each sector to another are filled into the relevant position in this matrix.

Different sector orderings can be selected where references out of blocks of sectors are reduced to a minimum. This can be represented by a clustering around the diagonals of the matrix (see Example 4.4.1.1).

The authors themselves present some criticisms of this technique:-

1. The matrix only presents global nearness and does not show any time dependent behaviour.
2. A new nearness matrix might have to be generated every time a program is run with new data.

EXAMPLE 4.4.1.1

Sectors = $\{1,2,3,4,5,6\}$

Nearness matrix = C_{ij} where

$$C_{ij} = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 213 & & 43 & & 612 & \\ 2 & & 769 & & & & \\ 3 & & 3 & 416 & & & 541 \\ 4 & & & & 317 & & \\ 5 & & & & & 684 & \\ 6 & 39 & & 297 & & & 291 \end{array}$$

which can be restructured to give

$$C_{ij} = \begin{array}{c|cccccc} & 1 & 5 & 3 & 6 & 4 & 2 \\ 1 & 213 & 612 & 43 & & & \\ 5 & & 684 & & & & \\ 3 & & & 416 & 541 & & 3 \\ 6 & 39 & & 297 & 291 & & \\ 4 & & & & & 317 & \\ 2 & & & & & & 769 \end{array}$$

Their own experiments have tended to show that:-

a) program behaviour improves with restructuring based on this technique

b) programs which are commonly used tend to show data independent behaviour.

For the actual running of programs, the blocks of sectors achieved by the restructuring are allocated to pages in a minimal fashion. That is to say, in a manner which will minimise the number of inter-page references.

The techniques described have been shown to produce a reduction in paging of between two-to-one and ten-to-one.

The major problem with this technique is that the tracing program takes about 30 to 60 times as long as the traced program to run. This seems to be a prohibitively long time for all but the most frequently used programs.

4.4.2 Swapped Working Sets

If when a program starts a period of execution its complete working set is not in store, a considerable amount of page traffic occurs while it builds up its working set. Experimental evidence reported by Adams (Ada 76) indicates that more than fifty per cent of page traffic comes from these faults.

A solution to this problem is to pre-load the working set of a program when it is re-activated. This is, in effect, an exact implementation in this respect of the Working Set policy, since that technique requires that a process have its complete current working set in store before it is allowed to run. In practice, this was often ignored for implementation reasons.

This technique has two effects:-

1. The number of individual page faults is reduced. (Although the volume of page traffic is not reduced)

2. A bulk request is made to secondary memory for the absent working set pages, and this allows optimisation at the level of secondary memory.

This technique, reported by Potier (Pot 77) was implemented on the Edinburgh Multi-Access System (EMAS) (Ada 75).

The only drawback of this technique occurs if a program is re-activated at the time of a locality transition. The working set that is pre-loaded will be out-of-date and will not refer to the new locality. Consequently, the program will demand-page up to its new working set and will remain with an over-large store allocation for some period which depends on the exact implementation of the Working Set principle.

4.4.3 Critical Working Sets

This technique also aims at program locality improvement by means of restructuring (Fer 74).

In this technique, a working set, W(t,T), is said to be a critical working set if

$$r(t + 1) \notin W(t,T).$$

That is to say, a page fault occurs at r(t + 1).

The idea behind this algorithm is to consider two reference strings S_b and S_p. S_b is the reference string with respect to the logical blocks of a program, and S_p is the reference string with respect to the pages of the program. Any mapping of the program blocks into pages transforms the block reference string into a page reference string. It is shown that a page fault in S_p always corresponds to a block fault in S_b, although all block faults do not have corresponding page fault. The aim of the Critical Working Set algorithms is to minimise the number of critical working sets in S_p.

A critical working set matrix is created which is an n x n matrix whose entry c_{ij} is the number of critical working sets having i as their critical reference and containing j. Consequently, $c_{ij} + c_{ji}$ is the number of critical working sets which disappear if i and j are mapped onto the same page.

This matrix is then used to produce an optimal allocation of blocks to pages.

It is reported that this technique is as successful as that reported by Hatfield and Gerald (see 4.4.1) in practical situations.

Again the major drawback of such a method is the considerable amount of processing time required to obtain all the necessary information for restructuring.

4.5 Conclusions

A number of conclusions can be drawn from the results and comments above:-

1. No real measure of how well a program behaves with respect to a particular page replacement

algorithm has been established. Mathematical models of program behaviour only extend to some types of reference strings and do not, in general, deal with locality transitions. Equally, no "standard" reference string has been produced against which the behaviour of algorithms can be measured.

2. Given a replacement algorithm, it is all too easy to find "real" programs which will behave badly under that algorithm. Indeed, even the "near-optimal" Working Set algorithm is shown to have unexpected far-from-optimal behaviour in simple cases.

3. If a process behaves badly with respect to a page replacement algorithm, it will always behave badly with respect to that algorithm no matter how many times it runs.

4. Replacement algorithms tend to make assumptions about the reference behaviour of programs, whether they display this behaviour or not.

5. A program is a deterministic entity, yet most algorithms are based on a probabilistic approach to program behaviour which completely ignores any prior information that may be available about the program's behaviour.

6. Attempts to improve locality by restructuring the address space of programs to fit the

replacement algorithms have met with some success but are extremely costly.

In the next chapter an alternative approach which attempts to mould a flexible algorithm to each individual program is proposed and subsequently developed.

CHAPTER 5 A PROPOSAL FOR MEMORY MANAGEMENT SYSTEMS BASED ON A KNOWLEDGE OF PROGRAM STRUCTURE

5.1 Introduction

In the previous chapter, it has been shown that each of the major replacement algorithms suffers from major practical deficiencies. In this chapter the background to a somewhat different approach to storage management is presented.

5.2 Program Structure and Program Behaviour

With the development of high-level languages into complex software tools, it is only natural that programs have, themselves, grown more complex and more structured.

Due to the timing of the work done on replacement algorithms, much of the work related to reference strings produced by FORTRAN or Assembler programs. It is one of the contentions of this thesis that analysis of the localities in such strings has produced algorithms that are appropriate *only* to such strings.

This, in itself, would be no great disadvantage if it were not for the fact that reference behaviour within languages which maintain a very linear and static address space is radically different from that type of behaviour displayed in the dynamically changing block-based address space in wide-spread use today.

At this point, then, it is useful to examine some of the language features available and to consider how these features affect the notion of program locality.

5.2.1 Block Structure

This feature, in itself, gives good support to the notion of a program going through a series of localities during its execution. This is a significant diversion from the early static languages. Dijkstra (Dij 76) examines the idea of accessibility of variables and states:-

> "From the point-of-view of flexibility and general applicability, the random access of store is, of course, a splendid invention, but comes the moment that we must realise that each flexibility, each generality of our tools requires a discipline for its exploitation. That moment has come."

Dijkstra identifies first the notion of a declaration as a useful form of redundancy, not present in the original version of FORTRAN. Declaring variables meant that data items could not deliberately (or accidentally) be created at run-time in a haphazard manner by simply placing the new name in the text of the program. Block structure itself was a great departure from the FORTRAN background. The idea of being able to nest blocks and their associated variable declarations has led to the idea of global and local variables.

When a program is executing in a given block, variables declared in an inner block are protected by the scope rules and are inaccessible. Thus a program can access only a subset of its total address space at a given time, thereby supporting the idea of locality. However in a given block of program, everything outside that block is accessible (except for those identifiers which have been re-declared in some of the nested blocks). It has proved to be the case that fledgling programmers have been encouraged to use local variables widely and global variables sparingly because of "good style". It is also true that the use of local variables improves the locality of a program, whereas reference to variables global to a given block increases the size of the locality.

It is interesting to note that current notions of departing from this extensive block context will tend to improve locality. The idea presented by Dijkstra and others of maintaining textual context but explicitly enumerating the names that make up this context at block entry, further restricts the address space accessible to a program at a given time and effectively defines the data locality of the program at that instant (cf, for example (Lam 77)).

5.2.2 Procedures and Functions

Even early implementations of FORTRAN and some assemblers allowed the idea of procedures and functions (or sub-programs). These were the first occurrences of explicit locality in a program text. Statements performing a logically distinct function were physically gathered into a distinct textual unit. During the execution of this distinct function, the program maintained a distinct locality (up to the restrictions mentioned above). It is useful to note that procedures and functions used the first primitive import and export list for their parameter lists.

Again it is useful to note that "good programming style" tends towards good locality. Within the context of procedures and functions (particularly the latter) side-effects are frowned upon. Such entities, it is recommended, should only affect their environment through their parameters or result. This is another way of restricting the accessible address space of a program at a given time.

However for procedures and functions, the most interesting aspects of locality behaviour are displayed by the use and implementation of parameters. Three types of parameter are identifiable:-

1. Name-type parameters
2. Reference-type parameters
3. Value-type parameters.

These have different effects on locality;-

1. Name-type parameters:- Such parameters, have addresses calculated at <u>each</u> time of use within the body of the procedure. This is unfortunate for locality, as it is impossible at the point of locality transition to determine exactly the

extent of the locality. It is fortunate, from a locality point-of-view, that this method of parameter passing has gone out of favour. In fact, the reasons that this technique has gone out of favour are similar to those used from arguing the locality viewpoint.

2. Reference-type parameters:- This type of parameter passing mechanism is widely used to enable the effect of procedures and functions to be passed out to the environment. However two implementation techniques result in different locality behaviour:-

   a) Reference: the address of the parameter is worked out at the call of the procedure or function and this address is used throughout the body of the procedure wherever the parameter name occurs.

   b) Value-result: a variable local to the procedure or function is set up with the same type and same value as the actual parameter at the time of the call. At the end of the procedure or function the value in this local variable is copied back to the actual parameter.

   Although these techniques are often used interchangeably in the implementations of a programming language, the reference behaviour produced is radically different. In the case of "pure" reference, this technique will almost certainly add to the locality size of the program at this point. A (not necessarily distinct) page (or pages) will be added for each reference parameter used. This addition could be critical in the case of LRU where the stack length is exceeded.

On the other hand, value-result packs the parameter-inspired variables in the local data space at the cost of an extra page fault for each parameter (approximately) at procedure exit. This could be a poor technique, in that retrospective algorithms will maintain the pages containing the actual parameters anyway for a strobe period. However from an aesthetic viewpoint the purely internal locality (in the absence of explicit globals) has its attractions. Value-result is not very popular for dealing with array parameters due to the copying involved.

3. Value-type parameters: these are essentially implemented as the first half of value-result parameters, i.e. the result is not passed back at exit time. The same comments can be made as of value-result above.

Another feature of procedures and functions is their relationship to the Page-Fault Frequency algorithm mentioned above. It is clear that at procedure/function entry a program will attempt to acquire pages relating to the new locality quickly. This is the point at which PFF is likely to behave badly, since the previous locality is only removed at the first non-critical page fault. A program which displays good locality in its procedures using value-result parameters etc might well have a small locality over-estimated by PFF in this situation.

5.2.3 Arrays and Records

In that these structures are inherently similar - arrays being named collections of objects of identical type and records being named collections of objects of not necessarily the same type - it is useful to consider them together. However it is important to note that in typical applications arrays are significantly larger objects than records.

The important feature of these structures is that the whole structure may be referenced with a single textual reference. (It is assumed that "reasonable" languages allow record/array assignment and, at least, comparison for equality and inequality on records and some arrays, e.g. strings). As far as arrays are concerned, single items within the structure are identified by indices that may themselves be program variables. This implies that the actual items being dealt with cannot be identified, in many cases, by simply examining the program text. This is not usually true of records whose fields are usually identified by fixed names which are extensions of the record name and cannot be variables.

The implications of the above are two-fold. Firstly, as was shown earlier single instructions at the high-level language level and even at the machine-level language level can generate large numbers of references. If this number is large enough it can tend to saturate a retrospective page replacement algorithm. In fact this saturation can occur without the array/record assignment statement, consider:-

```
    for i: = 1 to 2048 do a[i]:=0;
```

If the system on which this little program runs has a page size of 256 words and utilises, say, an LRU algorithm with a stack size of 8 pages, then the stack is rapidly filled up with the pages of the array being initialised. Only the context of the program after this statement will say if this is reasonable or not.

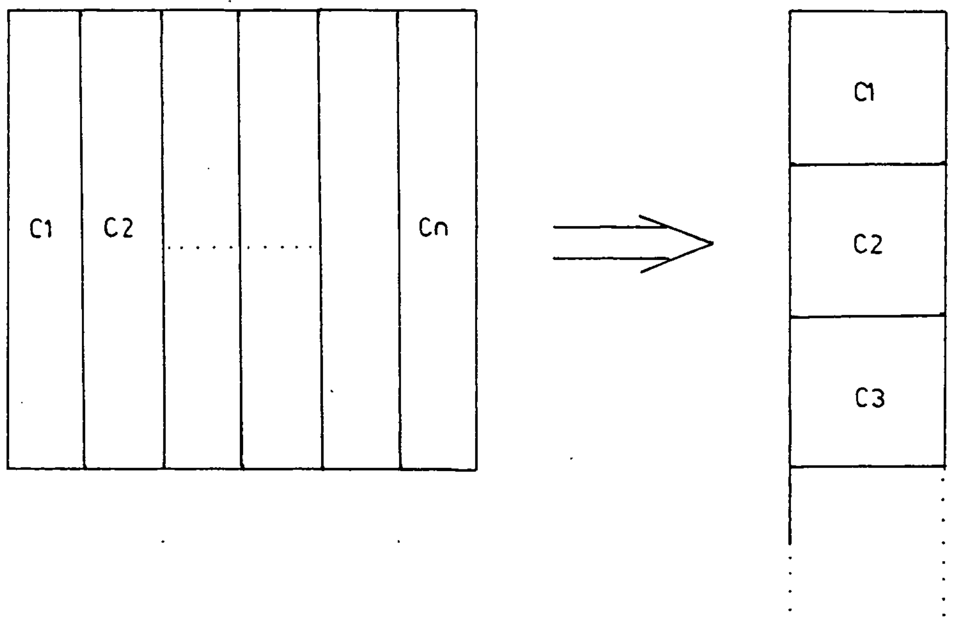
It is interesting to note that Dijkstra (Dij 76) hesitates to allow array assignments in languages because they are not "nice". That is to say, their implications are not really clear at the language level.

The second point that arises, chiefly from arrays, is the notion of access to the mapped linear form of the array in store. In the case of two dimensional arrays, two possibilities exist, namely to store by rows or to store by columns (see Diagram 5.1 a) and Diagram 5.1 b)).

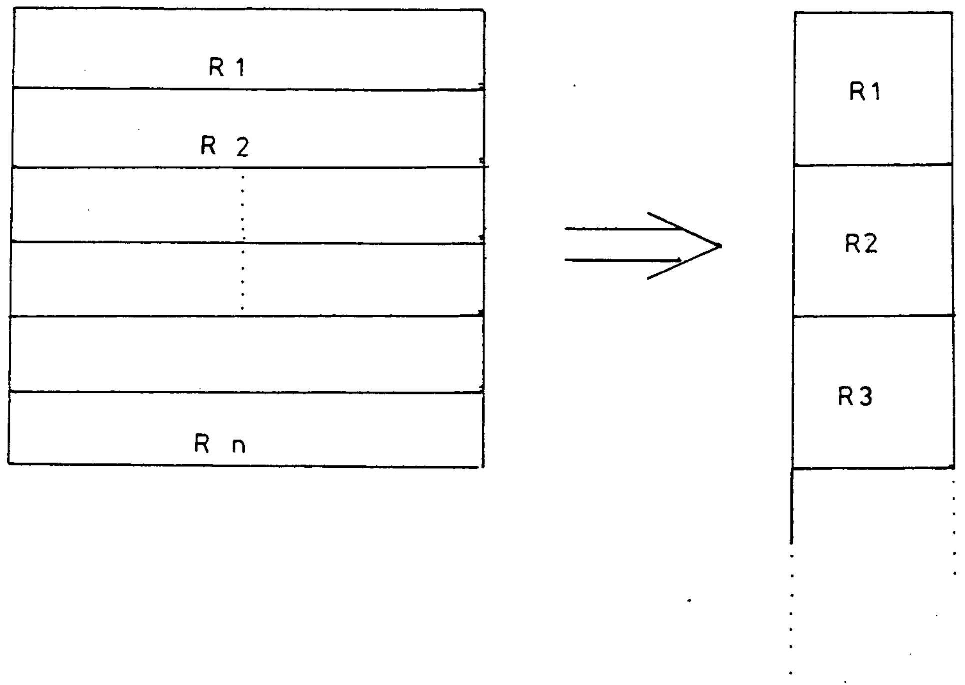
To illustrate the point a worst-case can be constructed. Assume a system with a page size of 256 words, and assume the following (PASCAL) array definition:-

```
      A : array[1..256,1..256] of integer;
```

Assume further (though this is not necessary) that A is aligned



a) Storing by columns



b) Storing by rows

Diagram 5.1

to a page boundary . Consider the following two pieces of code:-

```
for  i: = 1 to 256 do
                 for j: = 1 to 256 do A[i,j]  := 0; (5.1)
for  i: = 1 to 256 do
                 for j: = 1 to 256 do A[j,i]  := 0; (5.2)
```

The effect of these two pieces of code is identical, namely the elements of A are set to zero.

However, (5.1) assigns zeros on a row by row basis and (5.2) assigns zeros on a column by column basis. (Assuming the convention of row index followed by column index in the ordering of array indices).

In the case of arrays stored by row (Diagram 5.1 b)) then (5.1) will generate a page fault every 256 references. Similarly in the case of arrays stored by column (Diagram 5.1 a)) then (5.2) will generate a page fault every 256 references. However if the code of (5.1) is used in the situation where arrays are stored by column or vice versa, unless the store set size is allowed to reach 256 pages then this code will generate a page fault on <u>every</u> reference to the array.

There is nothing that a retrospective algorithm looking at the reference string can do about this. It is unfortunate that the behaviour penalty for not knowing about how the arrays have benn implemented is so severe (256 <u>times</u> more page faults in this phase!).

5.2.4 Complex Data Structures and Pointers

In this category are considered the so-called dynamic and recursive data structures e.g. lists, queues, trees, etc. Although such structures can be implemented using the static data structures the tendency has been to implement them using pointers. Both techniques have their disadvantages.

If an array is used to simulate any dynamic structure, two deficiencies are apparent. Firstly, over-estimation of the space required is necessary in many problems to deal with all the contingencies. Secondly, the penalty of random access to a linear store is incurred. For example, in a list structure it is common to have an array of list-heads (depending on how many lists are required) and an available space list which initially links all the items in the data array (see Diagram 5.2).

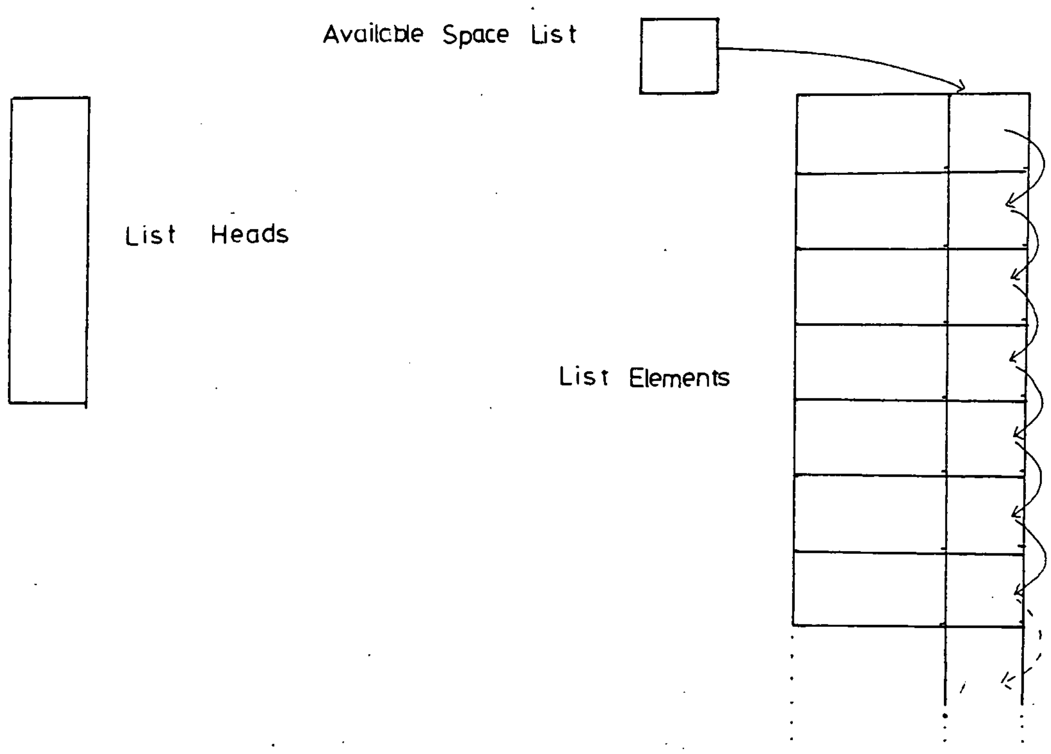


Diagram 5.2

However as these items are added, in whatever way the problem requires, to the individual lists and possibly transferred from list to list, the overall structure becomes less orderly. As can be seen in Diagram 5.3, references to logically associated items, i.e. they are currently on the same list, can lead to accesses to physically distant areas of store.

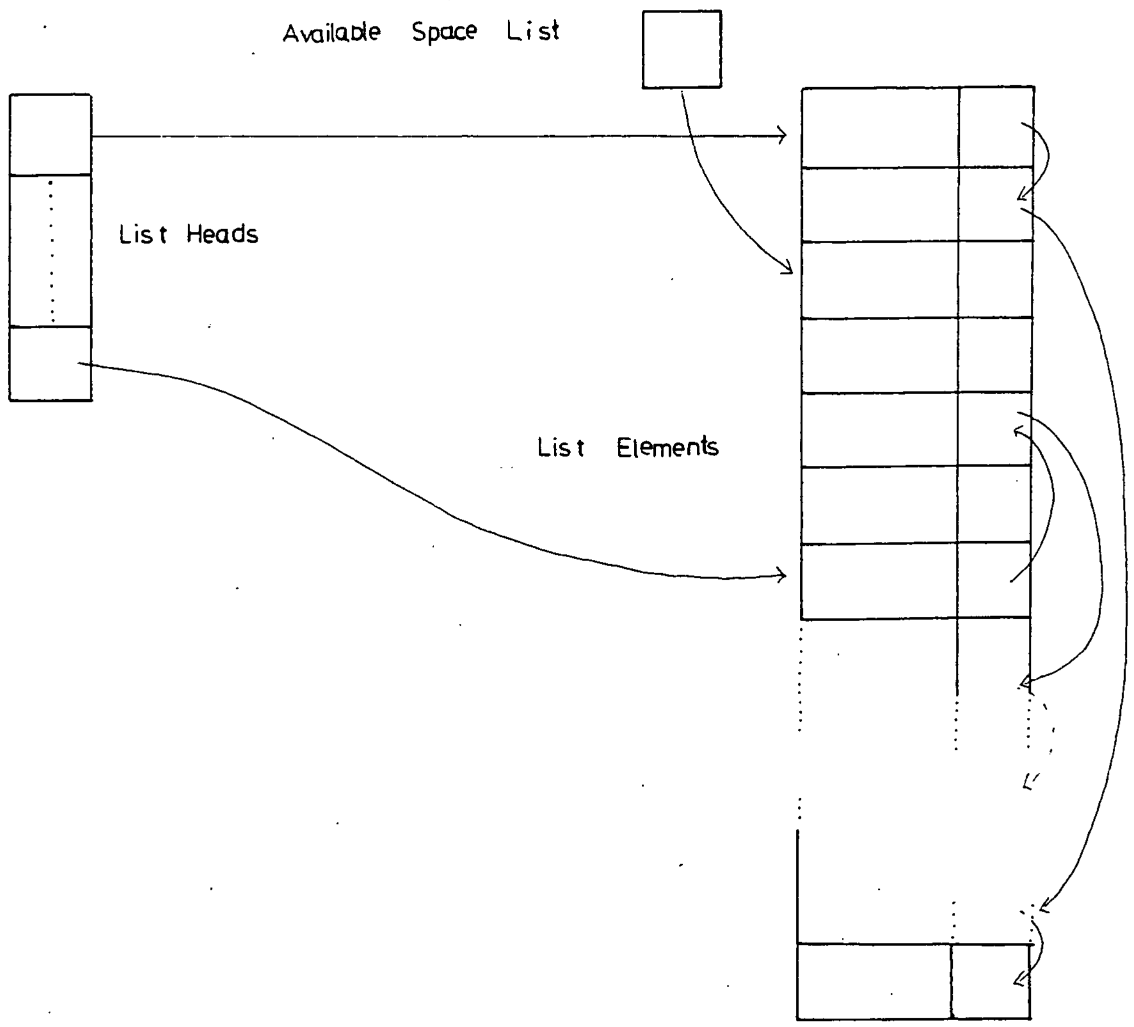


Diagram 5. 3

Implementation of dynamic data structures using pointers and the ability to create and destroy elements of a given type during the run of a program implies more complex storage management structures for the program itself (e.g. the "heap" construct) and tends to require effective garbage collection to tidy up disposed-of items.

Apart from this overhead, the problem of less disciplined access to store is not resolved. If list items are created as they are needed and then used in any order than the one in which they were created, the tendency towards random access is just as strong as in the first case.

It is not fair to use this as a criticism of page replacement algorithms alone, because it is difficult to see how <u>any</u> storage management technique could accommodate such potentially undisciplined behaviour. This section has been included rather to show what potential there is for poor behaviour in even relatively simple problems.

In the above sections, some aspects of data representations have been related to data locality behaviour, in the following sections program structures will be related to program locality behaviour.

5.2.5 Procedures and Functions

Such program elements as these represent the support for an intuitive belief in program locality. As well as being able (if the suggestions in 5.2.2 are implemented) to completely define the data locality, the program (or code) locality is restricted to the code of the procedure and that of any other procedures or functions it calls. Under these constraints it is almost possible to completely define the locality and the locality transitions during the lifetime of a program.

For a procedure or function can be defined the following objects:-

a) IMPORTS - Values imported from its environment. These represent the values of parameters etc defined outside the procedure and used within the procedure.

b) EXPORTS - Values exported from the procedure at its exit. Again, these represent the variables defined in its environment changed by the procedure.

c) ASSOCIATES - The procedures and their environments that are possibly used by the procedure during its lifetime.

The relationship between these objects is shown graphically in Diagram 5.4.

It is not sufficient to look only at this macro-structure. The program constructs described below have an effect on the duration of time spent in particular localities and in the choice of possible localities used by a program during its lifetime.

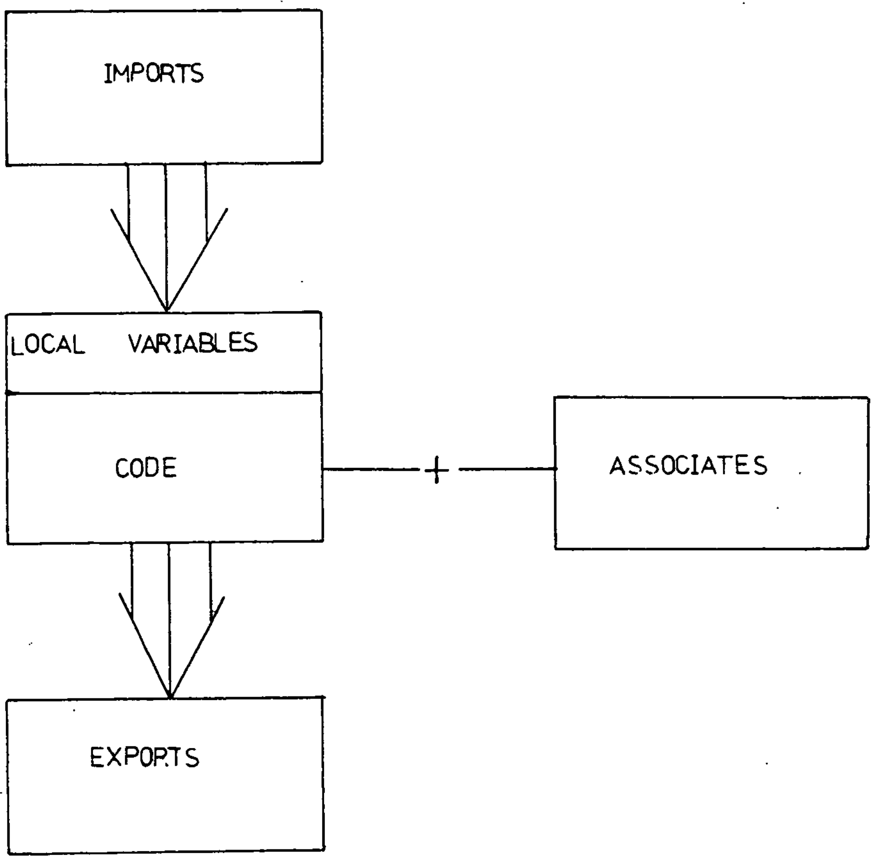


Diagram 5.4

5.2.6 Loop Structures

There are three common loop structures:-

a) the for loop - for i:= 1 to n do ....;

b) the while loop - while ‹ cond › do ....;

c) the repeat loop - repeat .... until ‹ cond› ;

As far as program reference behaviour is concerned, the body of a loop represents a locality which consists of a section of code that is repeated a (not necessarily pre-determinable) number of times. This is further complicated by the fact that, if a loop is considered as a locality, it would be desirable to standardise the treatment of localities. Consequently, an import and export list is required. This is not as easy to handle as the procedure/function case where the import and export lists can legitimately be prespecified. However, as will be shown later, the contents of the import and export lists can be predetermined with little extra cost.

The associates of a loop locality can be identified in a similar fashion to those of a procedure or function.

The two remaining sections deal with two constructs that can control the particular localities chosen during the execution of a program.

5.2.7 Conditions

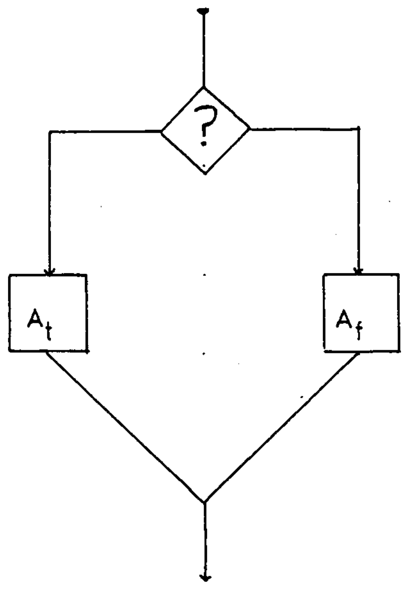
Here are considered the two conditional constructs:-

a) if ‹ cond › then ......;

and the more general form of the condition

b) case .... of ....

Each of these constructs represents the selection of one out of one or more actions depending on the value of some expression. Diagram 5.5 below shows the relationship between the if statement and the case statement.



If the condition is true then A_t is carried out, otherwise A_f is performed (A_f may be null).

The selection is performed with more components than the boolean in the if statement. Any of the A_i may be null.

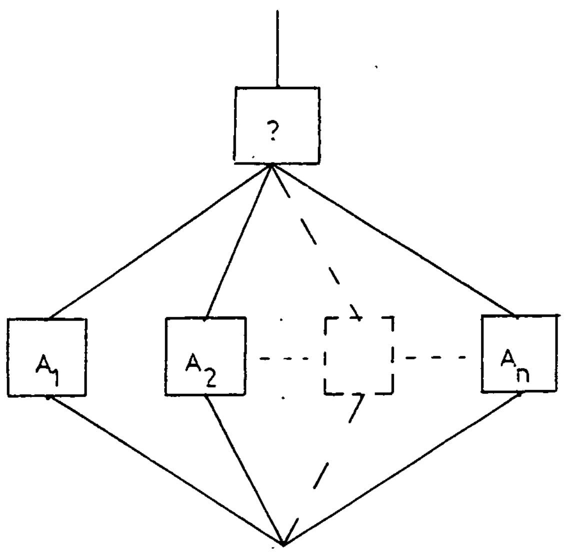


Diagram 5.5

Obviously, if the actions selected represent a significant amount of code then each action could be considered to be a locality. This implies that the direction, in terms of locality, which is taken by a program cannot be determined until the condition is tested.

5.2.8 The goto Statement

The goto statement, often combined with a conditional statement, represents an almost arbitrary selection of the next locality. This can cause a complete change of locality, "at a stroke". Similar criticisms are made of the goto statement from the program structure point-of-view. Such a statement which can cause control to be moved around a program structure in an unrestrained fashion represents bad style and a positive hindrance to reliable program development.

However the issue that the goto statement raises in the terms of program locality is that of the successor to a given locality. For a given locality, there can be identified two associated sets of localities:-

a) Predecessors: these are the localities which have the given locality as a successor

b) Successors: these are the localities which may be entered on exit from this locality.

It is the successor relationship that is the most important. The problem that exists is to determine which of the possible successors (if there are more than one) will be chosen when the program is executed.

5.3 The Formalisation of the Program Structure Approach

It is not the intention of this thesis to present a theoretical description of the basis of the program structure approach to storage management. It is all too often the case that a theoretical approach to any topic is forced to make concessions to the tractability of theoretical analysis which ultimately reduces the applicability of the results. This criticism can be made of the theoretical approach to the state-of-the-art presented in the earlier part of this thesis. A fundamental flaw with such approaches is that, although the final algorithm may match the theory well, practical programs have a habit of diverging from the theory at a critical point. The particular problem area for previous algorithms has been at locality changes. It is intuitively obvious and easy to demonstrate that any retrospective algorithm will fail when the past and future diverge.

Consequently, the approach of this thesis has been to identify localities and to determine the constituents of all future localities for a given locality.

A locality is defined as follows:-

DEFINITION 5.1 Locality

A program locality is a 7 - tuple

$\langle C, L, I, E, P, S, A \rangle$

where

C - the code executed in this locality

L - the variables local to this locality (if any) that are accessed in this locality

I - the values of variables global to this locality that are accessed in this locality

E - the variables global to this locality whose values may be changed in this locality

P - that set of localities that have the given locality as a successor

S - that set of localities which may be entered on exit from this locality

A - that set of localities which may be entered from this locality but return control to this locality.

NOTE: Thus, as was introduced above, for each locality is identified the local code and variables, import and export lists, predecessor, successor, and associate locality sets.

Two particular locality types can be identified:-

<u>DEFINITION 5.2   Initial Locality</u>

An initial locality is a locality with no predecessor in the current program context. This will correspond to the locality entered at the beginning of a program.

<u>DEFINITION 5.3   Final Locality</u>

A final locality is a locality with no successor in the current program context. Such a program locality will represent a locality in which the program can halt under program control.

For a program, there will be a unique locality with no predecessor. However if the system allows pre-compiled procedures to be available with multiple entry points, then these may have multiple initial localities.

If a programming language provides halt/stop instructions then there may be multiple localities in which the program can halt under program control. However if no such instruction exists then there will be only one such locality.

As a result, if normal execution of a program is considered to be running a program until it halts under program control, then the following formal definition can be made:-

<u>DEFINITION 5.4 Normal Program Execution</u>

Normal program execution is described as a path from an initial locality to a final locality. At the exit from a locality the next locality is chosen from among the successors of the current locality.

If a program is assumed to have only one initial locality and one final locality then the possible execution paths can be represented as shown in Diagram 5.6.

At this point it should be noted that the structure in Diagram 5.6 implies that there might be some potential for lattice structure analysis of programs.

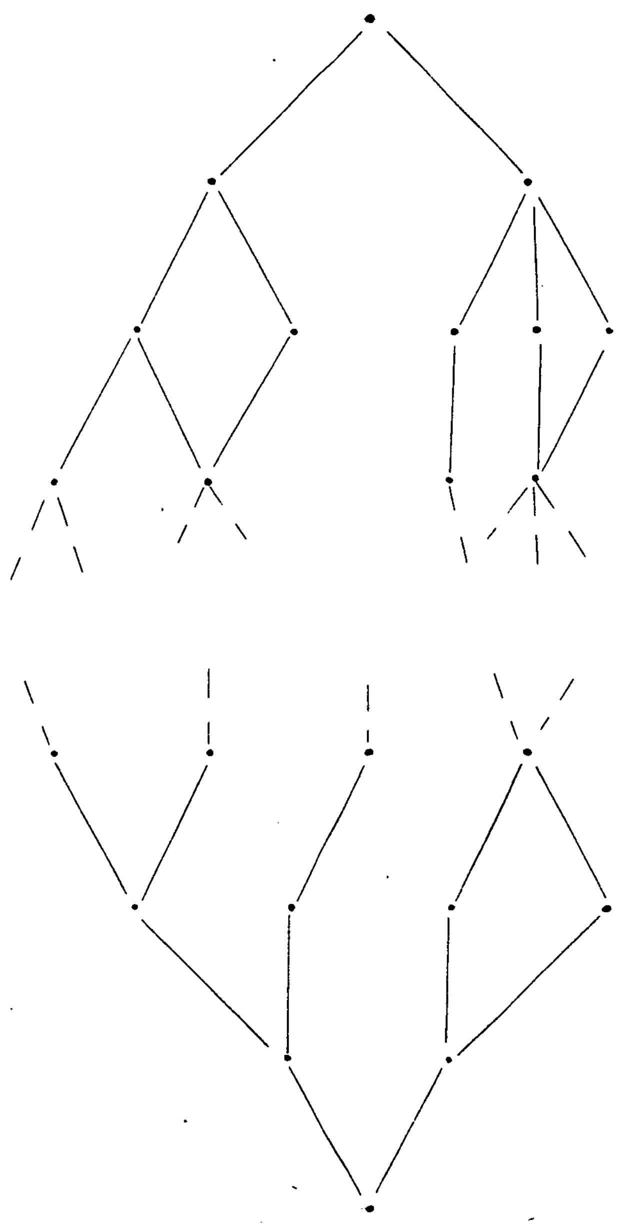


Diagram 5.6

5.4 Conclusion

In this chapter, the aim has been to show how program structure relates to program locality, and further to show that in many instances good program structure and potentially good program behaviour from a memory management point-of-view go hand-in-hand. From the point-of-view of program structure, components of program localities can be identified and this led to a formal definition of a program locality.

In the next chapter, it will be shown how program localities can be identified at the time of compilation of a program, thereby providing the run-time environment with a behavioural description of the complete program.

CHAPTER 6 AN EXPERIMENTAL IMPLEMENTATION

6.1 Introduction

In this chapter a description of a practical implementation of a program structure oriented approach to storage management is given. One of the main aims was to show that this approach could be implemented _without_ significant changes to existing systems and _without_ significant reductions in their efficiency. The implementation described is of modifications made to a PASCAL (Jen 74) compiler running on a PDP-11 computer under the UNIX operating system (Rit 78) at the University of Stirling.

6.2 Implementation Aims

The aim of the work was to extract at _compile time_ information sufficient to identify and describe program segments in the manner introduced above (Chapter 5). This information would then be made available to the run-time system and consequently the complete nature of localities and locality changes would be known at run-time.

This approach is in direct contrast to that of conventional paged systems. The major practical attractions of paging are worth recounting at this point. Firstly, the technique is inherently simple. Programs are all divided into the same fixed size units and these units become the units of primary and secondary memory allocation as well as being the unit of transfer between main and secondary memory. Secondly, the programmer need know nothing about how the system works. This technique does not hinder program portability in any way (cf overlaying).

On the other hand, paging has brought its own problems. Firstly, the choice of page size is critical and difficult. Secondly, large amounts of system storage space can be occupied by tables. Thirdly, the use of demand paging systems has also tended to mean the use of retrospective page replacement algorithms whose drawbacks have been outlined above.

The proposed system returns to the idea of segmentation thereby removing the problems associated with page size. Equally, the proposed system directly identifies localities and locality changes thereby overcoming the uncertainty and capacity for error inherent in retrospective systems.

Another aim of the system described here was to implement the proposals starting from an existing compiler. Apart from reducing the amount of ancillary work to be carried out, this approach has two advantages. Firstly, as no modifications were to be made to the compilation process itself, it would be impossible to lay the criticism that such a technique would only work in a controlled test situation with the desired end always in view. Secondly, a "normal" programming language could be considered and not, again, a limited test vehicle.

The programming language, ultimately, chosen was PASCAL. The PASCAL system used was written by R G Clark at the

University of Stirling and the compiler produces code for a simple stack machine which is subsequently interpreted. In the approach described below, this simple machine language is considered to be the machine language of a "real" machine. This is consistent with either micro-code interpretation or with the design of language-oriented hardware, both of which techniques seem to be gaining an increasing number of adherents.

To return to the choice of PASCAL, a number of reasons can be identified. Firstly, the compiler was available and access to, and modification of, the code of the compiler was possible. Secondly, PASCAL seemed to be an important language. While it is still not clear that PASCAL itself will be of the utmost importance, it is clear that PASCAL embodies many of the current ideas concerning structured programming and it is likely that PASCAL will form the basis of a number of future languages. Consequently, it was decided that, if in the limited context of this thesis only one language could be studied, PASCAL should be that language. It is fair to say that PASCAL is not totally suited to this research due to the fact that it lacks suitable constructs to describe data access and locality. Such facilities are only recently becoming available in languages like LIS (Ich 76) and EUCLID (Lam 77)

The technique employed was to divide the program being compiled, during the compilation process, into units of program or data space (subsequently called segments for want of a better word) where the constituents of each unit were logically associated. For these segments the features described in Chapter 5 would be identified.

The units identified fall into two categories:

a) Data segments

b) Program segments

These will be considered in turn.

6.2.1 Data Segments

As has been shown above, it is unwise to consider program and data locality together. In order that one does not swamp the other, the locality in each area should be considered separately.

As was mentioned earlier it would be ideal if the data structure itself could mirror the access that will be made to it, much as, say, the structure of a rooted tree mirrors access made to it via the root. In such an example the notion of the data locality for a given element could be identified as its parent and children (see Diagram 6.1)

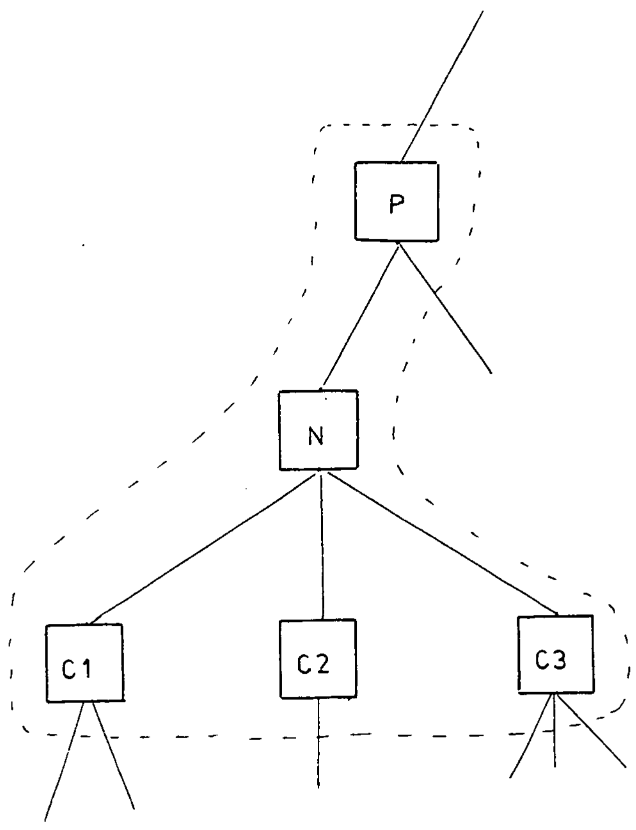


Diagram 6.1

That is to say that having accessed node N it is likely that the next node to be accessed will be in the set $\{P, C1, C2, C3\}$

Languages exist now in which the access methods for a data structure can be matched to the structure itself at the implementation level. PASCAL is not such a language.

Consequently only the most rudimentary data localities are identified within PASCAL programs. Within any given block can be identified:-

a) constants: this is not strictly (or at all) necessary since such identifiers are changed when generating code into the equivalent numerical constant. However it seemed that it might be a good idea to consider these as initialised variables (own variables) - a feature unavailable in PASCAL.

b) local variables: the variables local to this block.

c) arrays: each array is a separate data segment. This can be done at compiler time due to the fact that PASCAL allows only static arrays. Consequently the size of such a segment is known at compile time.

Under b) more than one segment can be created if array declarations are mixed lexically with scalar declarations. This is a relatively trivial point but it is possible to support such an action in that the layout of variable declarations ought to have some logical significance. Equally a minor modification could produce all local scalars in a single segment.

6.2.2 Program Segments

Each block (program, procedure/function) is considered to be a separate unit, although the nesting structure of the original program is retained for convenience.

Such program segments come under the general heading of code segments and the following types can be identified:-

1. Compound statements:- Loops, "if" statements, and "case" statements.
2. Simple statements.

Each compound statement represents a segment and the nesting of compound statements particularly loops is significant. Similarly a sequence of simple statements represents a segment. In this way it is hoped that logically associated statements can be grouped together.

For each program segment, its associates (which includes variables used by this segment) are identifiable and retained with the segment. Since a complete division has been made between data and program segments all variables are, strictly, imported. However it is probably useful to distinguish between variables local to the block to which this segment belongs and those local to other blocks. In other words some variables are more important than others.

It is simple to identify successors since, in the worst case, the "go to", all labels must be declared and consequently the segments to which a label refers can be extracted from the symbol table. Another bad case is that of a procedure call and return. The successor of a procedure call is the segment containing the call.

Procedure parameters are specified at call and these are treated as associates to that particular instance of the procedure segment.

In the demonstration case the compiler produces output which divides the object code into segments. Each of which has its own associates identified (see Example 6.1). The implementation is described fully in the next section.

6.3 Implementation

The first modification to the compiler is to extend the symbol table entry for all named items. This extension contains all the segment information for that given item.

A segment has the following information held on it:-

no:	its number
block:	its block number
actualseg:	whether it is, for code segments, a real segment (i.e. has code in it)
unit:	its unit number
kind:	the type of the segment
start:	starting address of the segment
finish:	finish address of the segment
assocst:	the starting position in the associate table (q.v.) for the associates of this segment
assocend:	the finish position in the associate table for the associates of this segment.

Each segment has associates, other segments referred to during the lifetime of that segment. These associates are held in an array which is passed to the run-time system, after the code and other information. Each segment has itself as an associate. This is to enable the code for loading associates to be dumped at the start of a segment before the ultimate extent of the segment or its associates is known.

The other major modification is to extend the instruction set of the virtual machine by one instruction. This instruction is the "fldctxt" instruction.

The function of this instruction is to load the context of the segment about to be entered. That is to say it causes the loading of the segments containing the associates of the current segment. It has a second, subsidiary, function and that is the marking as free any areas of store occupied by segments no longer required by the system.

These, then, are the only major modifications to the operation of an existing compiler to support this approach.

The system identifies two basic segment types:-

1. Data segments
2. Code segments

These are discussed more fully in the following sections.

6.3.1 Data Segments

As was mentioned in Chapter 5, data segments in PASCAL are easy to identify and delineate. The reasons for this are as follows:-

1. No dynamic structures (other than those created by the facility "now") exist.
2. The data element descriptions are constructed in such a way that only one pass is required of the compiler. In other words, when the compiler encounters a declaration of a data element, it has all the necessary information to compute the size of the element available to it.

Within the area of data segments, these types can be identified:-

1. Constants
2. Arrays
3. Others

These are considered below:

1. Constants: Strictly these should not be data segments at all, since they are implemented by substituting the value whenever a constant identifier is encountered in the text. However, since it seems possible that initialised variables might be incorporated in PASCAL programs in a similar textual fashion it was thought a useful experiment to examine such segments.

2. Arrays: Each array is a segment. This is the first step towards data structures with accessing methods as segments.

3. Others: The number of other data segments for variables depends on the layout of the declaration of arrays, but it was thought that, at least initially, the total number of segments was not important.

Each data element has its segment information associates with it in its symbol table entry.

6.3.2 Code Segments

These are somewhat more complex entities. The major features identified in PASCAL were:-

1. Procedures and functions
2. Repeat / White / For loops
3. Other compound statements

1. Procedures and functions

These represent the main block structure of PASCAL. A check of the nesting level of the current block is kept in the segment information (module.block). An index of the sub-units within each block is also kept (module.unit) as well as a simple number to identify the segment (module.no).

The procedure/function has a unique segment id associated with its total extent, so that the complete environment of a procedure/function may be loaded when that procedure/ function is called.

At procedure/function call any variable parameters cause their segment to be loaded as well. Consequently the total environment of a procedure/function when called is:-

1. All associates created by the procedure/function body
2. All associates created by the parameters.

2. Repeat / While / For loops

Loops represent localities in which programs can reside for indefinite amounts of time, consequently they are allocated segments of their own. The problem of nested loops is considered fully in Chapter 7, but each loop with code of its own is considered to be a separate loop. If two nested loops exist as in the example below:-

```
for  i:=  1 to n do
     for j:= 1 to m do
         begin
         ....
         ....
          end;
```

then this can be considered to be a single segment.

3. Other Compound Statements

Here can be considered

a) if .... then ‹Compound Statement› else‹Compound Statement›

b) case .... of ...

Each of these constructs is in effect made out of a number of sub-segments representing the compound statements. They can be grouped together into an encompassing segment since it is normal for only one of these sub-segments to be executed.

Program statements not covered in the above categories are grouped into segments of an indeterminable nature, but it is hoped that their close proximity would make this a defensible action.

It should be mentioned at this point that labels should indicate the start of a new segment since it is possible to jump to a label from a distant point, and consequently the context must be loaded when the jump is completed. Alternatively since Pascal requires labels to be declared it is possible to load the correct context immediately before the jump takes place.

Since labels were not available in the reduced compiler used, and since the use of "goto"s is currently considered bad practice, it was not thought to be sufficiently important to implement this feature.

6.4 Conclusion

This chapter has presented a brief description of the implementation of the proposed program structure approach. Examples of the operation of the system are presented in detail in Chapter 7.

CHAPTER 7 RESULTS

7.1 Introduction

The results presented arise from the running of the compiler and interpreter mentioned in the previous chapter. The results show that it may be possible to implement systems which make no storage allocation decisions other than at locality transitions. The task of directly comparing this system with current paging systems is difficult. To do this effectively, complete operating systems must be built assuming the use of one of the techniques. In this way it would be possible to do something which is significantly lacking in the field at present. That is, obtain information on how well given systems run. Up to this point, there has been a tendency to obtain only crude qualitative assessment of the behaviour of paging systems. What comparisons have been made between systems have not attempted to ascertain what can ultimately be done with memory management systems but rather have attempted to find which system is better than the others.

What is not shown in the results below therefore, is how a complete system can be built round the proposed technique. What has been done, however, is to show how such a system might be expected to behave, where its strengths and weaknesses lie, and how it compares on a number of counts with existing systems. A side-effect of these measurements is that the sensitivity of some of the existing techniques to variations in their operational parameters is amply demonstrated.

7.2 Quantitative Assessment

As has been indicated earlier, to measure the behaviour of the memory management techniques interpreters were written which simulated the behaviour of the compiled programs under different management strategies and under similar strategies with different parameters. The four strategies used were:-

a) the proposed segmented approach

b) Working Set

c) Page Fault Frequency

d) Least Recently Used.

At this point it is necessary to indicate the significant limitations placed on the experiments by this implementation.

Firstly, the relatively small available address space of the PDP 11/34 significantly constrained the size of program that could be compiled and run.

Secondly, the speed of the 11/34 meant that relatively long programs could not be simulated readily (e.g. full bubblesort).

Finally, no assessment could be made of the behaviour of each strategy in a multi-programming environment.

As a consequence, many of the measurements made below are scaled-down to this environment, but it is suggested that extrapolation to "real" systems is both reasonable and valid.

In this context then the following measurements could be made:-

a) for all approaches:

  (i) the number of allocation decisions made

  (ii) the amount of program plus data space occupied during the execution of the program

  (iii) the traffic between backing store and main store

  (iv) the number of entries in the page/segment tables

  (v) the time (in number of references) between allocation decisions.

b) for the segmented approach:

  (i) the number of segments moved in.

c) for the paged approaches:

  (i) the effect of variations in page size

  (ii) the effect of variations in LRU stack size

  (iii) the effect of variations in PFF critical frequency

  (iv) the effect of variations in WS strobe interval and window size.

These measurements were carried out during the execution of four programs:-

a) /

a) Permutation generation program	(Example 7.2.1)
b) Knight's tour program	(Example 7.2.2)
c) Stable marriage program	(Example 7.2.3)
d) Bubblesort (reduced) program	(Example 7.2.4)

These programs were chosen for a number of reasons but it was hoped that they would show up different types of program and data locality thereby enabling the algorithms to be tested satisfactorily.

EX 7.21

```
program permute(input,output);
const limit=20;
var perm:array[1..limit] of integer;
    line:array[1..limit] of integer;
    eqn:array[1..19] of integer;
    a,i,startval,upin,n,count:integer;
    up:boolean;

function min(lastin:integer):integer;
var i:integer;
begin
    i:=1;
    while((i<=n)and(line[i]<>0)) do
 i:=i+1;
    if i<lastin then
    begin
 line[i]:=1;
 min:=i;
    end
    else min:=0;
end;

function max(lastin:integer):integer;
var i:integer;
begin
    i:=n;
    while((i>=1)and(line[i]<>0)) do
        i:=i-1;
    if i>lastin then
    begin
 line[i]:=1;
 max:=i;
    end
    else max:=0;
end;

function makeq(x:integer):integer;
var i,j,temp:integer;
    illegal:boolean;
begin
    perm[1]:=startval;
    line[startval]:=1;
    illegal:=false;
    for i:= 1 to x do
    begin
 if line[perm[i]+eqn[i]]=0 then
        begin
     perm[i+1]:=perm[i]+eqn[i];
            line[perm[i+1]]:=1;
 end
 else illegal:=true;
    end;
    if not illegal then
 if ((x=0)or(x<>n-3)or ((x= n-3) and(abs(eqn[n-2])<=abs(eqn[n-1]))))th
       begin
        i:= x+1;
        while((i<=n-1)and not illegal) do
        begin
 j:=i+1;
        if((up and (j div 2 *2=i+1))or(not up and(j div 2*2<>i+1))) t
```

```
                  j:=1
  else j:=0;
         case j of
    0:perm[i+1]:=min(perm[i]);
           1:perm[i+1]:=max(perm[i]);
         end;
         if perm[i+1]=0 then
             illegal:=true
                else eqn[i]:=perm[i+1] -perm[i];
                i:=i+1;
            end;
 end
        else
     if x=n-3 then
     begin
         temp:= -1*eqn[n-1];
         eqn[n-1]:= -1 * eqn[n-2];
         eqn[n-2]:= temp;
         perm[n-1]:=perm[n-2]+eqn[n-2];
         perm[n]:= perm[n-1]+eqn[n-1];
     end;
    for i:= 1 to n do line[i]:=0;
    if not illegal then
 makeq:=1
    else makeq:=0;
end;

procedure permprint;
var i:integer;
begin
     for i:= 1 to n do
         write( perm[i]);
     writeln;
    count:= count+1;
end;

function search(x:integer):integer;
var i,eqx:integer;
begin
    eqx:=eqn[x];
    if x>0 then
 if eqx>0 then
     eqn[x]:=eqx-1
 else eqn[x]:=eqx+1;
        eqx:=eqn[x];
        if x>1 then
 if abs(eqx)=abs(eqn[x-1]) then
     if eqx>0 then
  eqn[x]:=eqx-1
     else eqn[x]:= eqx+1;
 if eqn[x]<>0 then
     if makeq(x)=1 then
     begin
  permprint;
  search:=n-3;
     end
     else search:=x
 else search:=x-1;
end;
```

```
egin(*main*)
   writeln('Input number of integers to be permuted');
   writeln('1<n<=20');
   read(n);
   while n<>0 do
   begin
       count:=0;
       while ((n<0) or (n>20)) do
begin
    writeln('Range is 1..20');
    writeln('Type 0 to finish');
    read(n);
end;
       for i:= 1 to n-1 do eqn[i]:=0;
       for i:= 1 to n do line[i]:=0;
writeln('Input direction of permutation');
writeln('UP-DOWN = 1,DOWN-UP = 0');
read(upin);
while ((upin<>1) and (upin<>0)) do
begin
    writeln('Direction is either 1 -> UP-DOWN or 0-> DOWN-UP.');
    read(upin);
end;
if upin=1 then
  up:=true
else up:=false;
writeln;writeln;writeln('Permutations are:');
if up then
begin
    for startval:= 1 to n-1 do
 if makeq(0)=1 then
 begin
     permprint;
     if n>3 then
     begin
         a:=n-3;
         repeat
      a:=search(a)
         until a= 0;
     end;
 end
end
else for startval:= n downto 1 do
 if makeq(0)=1 then
 begin
     permprint;
     if n>3 then
     begin
         a:=n-3;
         repeat
      a:=search(a)
                 until a= 0;
     end;
 end;
      writeln; writeln('Number of permutations is:',count);
      writeln;
writeln('Input No. of numbers to be permuted');
writeln('Type 0 to finish');
read(n);
```

EX 7.2.2

```
program knightstour(output);
const n=3;
var i,j:integer;
    success:boolean;
    a,b:array[1..8] of integer;
    table:array[1..n,1..n] of integer;

procedure try(i,x,y:integer);
  (*i is the number of moves made. x and y give the current position*
var k,u,v:integer;
begin
  k:=0;
  writeln(i,x,y);
  repeat k:=k+1;
    (*set up next move*)
    u:=x+a[k];v:=y+b[k];
    (*test if the move is acceptable*)
    if (u>=1)and(u<=n)and(v>=1)and(v<=n) then
    if table[u,v]=0 then
    begin
      table[u,v]:=i;(*record move*)
      (*test if board is full*)
      if i=n*n then success:=true else
      begin
        try(i+1,u,v);(*try next move*)
        (*if the move is unsuccessful then erase move*)
        if not success then table[u,v]:=0
      end
    end
  until success or (k=8);
end(*try*);

begin
  (*initialise difference arrays and board squares*)
  a[1]:=2;a[2]:=1;a[3]:=-1;a[4]:=-2;a[5]:=-2;a[6]:=-1;a[7]:=1;a[8]:=
  b[1]:=1;b[2]:=2;b[3]:=2;b[4]:=1;b[5]:=-1;b[6]:=-2;b[7]:=-2;b[8]:=-
  for i:=1 to n do
  for j:=1 to n do table[i,j]:=0;
  (*initialise tour from position 1,1*)
  success:=false;table[1,1]:=1;
  try(2,1,1);
  if success then
  begin
    (*write out table*)
    for i:=1 to n do
    begin
      for j:=1 to n do write(table[i,j]);
      writeln(output)
    end
  end else writeln('no solution')
end.
```

EX 7.23

```
program marriage(input,output);
const
  n=8;
var
  m:integer;
  w:integer;
  r:integer;
  wmr:array[1..n,1..n] of integer;
  mwr:array[1..n,1..n] of integer;
  rmw:array[1..n,1..n] of integer;
  rwm:array[1..n,1..n] of integer;
  x  :array[1..n] of integer;
  y  :array[1..n] of integer;
  single: array[1..n] of boolean;


procedure print;
var
  m:integer;
  rm,rw:integer;
begin
  rm:=0;
  rw:=0;
  for m:=1 to n do
  begin
    write(x[m]);
    rm:=rm+rmw[m,x[m]];
    rw:=rw+rwm[x[m],m];
  end;
  writeln(rm,rw);
end;

procedure try(m:integer);
var
  r:integer;
  w:integer;

function stable:boolean;
var
  pm:integer;
  pw:integer;
  i,lim:integer;
  s:boolean;
begin
  s:=true;
  i:=1;
  while (i<r) and s do
  begin
    pw:=wmr[m,i];
    i:=i+1;
    if not single[pw] then s:=rwm[pw,m]<rwm[pw,y[pw]];
  end;
  i:=1;
  lim:=rwm[w,m];
  while (i<lim) and s do
  begin
    pm:=mwr[w,i];
    i:=i+1;
    if pm <m then s:=rmw[pm,w]>rmw[pm,x[pm]];
  end;
```

```
      stable:=s;
    end;

    begin
      for r:=1 to n do
      begin
        w:=wmr[m,r];
        if single[w] then
          if stable then
          begin
            x[m]:=w;
            y[w]:=m;
            single[w]:=false;
            if m<n then try(m+1) else print;
            single[w]:=true;
          end;
      end;
    end;


    begin
      for m:=1 to n do
        for r:=1 to n do
        begin
          read(wmr[m,r]);
          rmw[m,wmr[m,r]]:=r;
        end;
      for w:=1 to n do
        for r:=1 to n do
        begin
          read(mwr[w,r]);
          rwm[w,mwr[w,r]]:=r;
        end;
      for w:=1 to n do single[w]:=true;
      try(1);
    end.
```

Example 7.2.4

This is the same bubblesort algorithm that was used in Example 4.1.1. This example will be used to describe in detail the output generated by the experimental system.

Two sets of output are produced for each program:-

i) a descriptive listing produced by the compiler

ii) the object code generated by the compiler.

These are considered below.

i) Compiler Listing:-

The output is divided into segments, the start of each segment being indicated by the "New Segment" message. Along with this message three numbers are produced. These represent a) the block level of this segment, b) the unit number of this segment (i.e. the number of the segment with respect to this level) and c) the number of this segment. In data segments, only the size of the segment is produced as any further information. In this example can be seen the division of the program variable area into two segments. The first segment corresponds to the three scalars "i,j,k" and the second to the array "element".

Code segments have the code displayed before the corresponding source line. Here also can be seen the listing of the associates of the current segment. Each segment has itself as an associate, this is an operational convenience with no special significance.

At the head of each segment can be seen the "fldctxt" instruction, which loads the context for this given segment. The second operand field for this segment indicates where the contextual information for this segment may be found in an array that is passed to the run-time system. (This array is described in part ii) below). Two further points can be observed in this example. Firstly, the problem of jumps (occurring here in loops) which may go to the middle of segments, thereby avoiding the loading of that segment's context. This has been removed by identifying the segment containing the jump address and passing this segment identifier to the run-time system as the second operand of the jump instruction. Secondly, due to the time of the generation of the code produced with the listing, some jumps have not had their addresses and segment identifiers determined, the full code output in section ii) has the complete correct code.

ii) Object Code:-

The object code output consists of a complete listing of the object code in a numerical form, followed by the symbol table, string constants, the segment context information, and error information.

The significant part is the segment context information. This has the following components:-

1) the line number in this table
2) the actual segment number
3) the start of the associates of this segment
4) the finish of the associates of this segment (both of these are line indexes for this table)
5) the start of the area reserved for this segment
6) the end of the area reserved for this segment (for data segments the end is the size of the segment)

It is this table that is referred to by the second operand of all jump instructions and the "fldctxt" instruction.

EX 7.24

```
program bubblesort(input,output);
var
  i,j,k:integer;
  element:array[1..2048]of integer;
begin
  j:=1;k:=-1;
  for i:=1 to 2048 do
  begin
      j:=-(j+1);
      k:=-(k-1);
      element[i]:=i+j-k;
  end;
  for i:=2 to 5 do
      for j:=2048 downto i do
      begin
          if element[j-1]>element[j] then
          begin
           k:=element[j-1];
           element[j-1]:=element[j];
           element[j]:=k;
         end;
      end;
end.
```

```

                        ********** New Segment =    1   0   0 **********
0001  program bubblesort(input,output);

                        ********** New Segment =    1   1   1 **********
0002  var
0003    i,j,k:integer;
                        Size=     3 words

                        ********** New Segment =    1   2   2 **********
0004    element:array[1..2048]of integer;
                        Size=  2048 words

                        ********** New Segment =    1   3   3 **********



   0                                    fldctxt      0    4
   1                                    fjsub        2    0
   2                                    fsave     2051    1

                        Associates:
                        Segment No    3
                        ----------------
                        seg:      3
                        ==========
                        Size=     3 words
0005  begin

                        ********** New Segment =    1   4   4 **********



   3                                    fldctxt      0    5
   4                                    fldlit       1    1
   5                                    fldlit       0    1
   6                                    fass         0    0



   7                                    fldlit       1    2
   8                                    fldlit       0    0
   9                                    fldlit       0    1
  10                                    fsub         0    0
  11                                    fass         0    0
0006     j:=1;k:=-1;

                        Associates:
                        Segment No    4
                        ----------------
                        seg:      4
                        seg:      1
                        ==========
                        Size=     9 words

                        ********** New Segment =    1   5   5 **********


```

```
    13                                  fldlit      1    0
    14                                  fldlit      0    1
    15                                  fass        0    0
    16                                  fload       1    0
    17                                  fldlit      0 2048
    18                                  fle         0    0
    19                                  fjfalse     0    0
  0007      for i:=1 to 2048 do
  0008      begin



    20                                  fldlit      1    1
    21                                  fldlit      0    0
    22                                  fload       1    1
    23                                  fldlit      0    1
    24                                  fadd        0    0
    25                                  fsub        0    0
    26                                  fass        0    0
  0009         j:=-(j+1);



    27                                  fldlit      1    2
    28                                  fldlit      0    0
    29                                  fload       1    2
    30                                  fldlit      0    1
    31                                  fsub        0    0
    32                                  fsub        0    0
    33                                  fass        0    0
  0010         k:=-(k-1);



    34                                  fload       1    0
    35                                  fin         1 2048
    36                                  flmod       1    2
    37                                  fload       1    0
    38                                  fload       1    1
    39                                  fadd        0    0
    40                                  fload       1    2
    41                                  fsub        0    0
    42                                  fass        0    0
  0011         element[i]:=i+j-k;



    43                                  fldlit      1    0
    44                                  fload       1    0
    45                                  fldlit      0    1
    46                                  fadd        0    0
    47                                  fass        0    0
    48                                  fjump      16    0
    49                                  fundef      1    0
  0012      end;

                      Associates:
                      Segment No    5
```

```
                                 seg:      1
                                 seg:      2
                                 ==========
                                 Size=    38 words

                                 ********** New Segment =    1   6   6 **********



           50                                  fldctxt       0   10
           51                                  fldlit        1    0
           52                                  fldlit        0    2
           53                                  fass          0    0
           54                                  fload         1    0
           55                                  fldlit        0 2048
           56                                  fle           0    0
           57                                  fjfalse       0    0
         0013    for i:=2 to 2048 do

                                 Associates:
                                 Segment No    6
                                 ----------------
                                 seg:      6
                                 seg:      1
                                 ==========
                                 Size=     8 words

                                 ********** New Segment =    1   7   7 **********



           58                                  fldctxt       0   12
           59                                  fldlit        1    1
           60                                  fldlit        0 2048
           61                                  fass          0    0
           62                                  fload         1    1
           63                                  fload         1    0
           64                                  fge           0    0
           65                                  fjfalse       0    0
         0014       for j:=2048 downto i do
         0015       begin

                                 Associates:
                                 Segment No    7
                                 ----------------
                                 seg:      7
                                 seg:      1
                                 ==========
                                 Size=     8 words

                                 ********** New Segment =    1   8   8 **********



           66                                  fldctxt       0   14
           67                                  fload         1    1
           68                                  fldlit        0    1
           69                                  fsub          0    0
           70                                  fiq           1 2048
```

```
     72                               fload         1    1
     73                               fin           1 2048
     74                               flindmod      1    2
     75                               fst           0    0
     76                               fjfalse       0    0
   0016             if element[j-1]>element[j] then
   0017             begin


     77                               fldlit        1    2
     78                               fload         1    1
     79                               fldlit        0    1
     80                               fsub          0    0
     81                               fin           1 2048
     82                               flindmod      1    2
     83                               fass          0    0
   0018                k:=element[j-1];


     84                               fload         1    1
     85                               fldlit        0    1
     86                               fsub          0    0
     87                               fin           1 2048
     88                               flmod         1    2
     89                               fload         1    1
     90                               fin           1 2048
     91                               flindmod      1    2
     92                               fass          0    0
   0019                element[j-1]:=element[j];


     93                               fload         1    1
     94                               fin           1 2048
     95                               flmod         1    2
     96                               fload         1    2
     97                               fass          0    0
   0020                element[j]:=k;

                         Associates:
                         Segment No    8
                         ----------------
                         seg:      8
                         seg:      2
                         seg:      1
                         ==========
                         Size=    32 words
   0021             end;

                         ********** New Segment =    1    9    9 **********


     98                               fldctxt       0   17
     99                               fldlit        1    1
    100                               fload         1    1
    101                               fldlit        0    1
```

```
103                                 fass        0    0
104                                 fjump      62   12
105                                 fundef      1    1



106                                 fldlit      1    0
107                                 fload       1    0
108                                 fldlit      0    1
109                                 fadd        0    0
110                                 fass        0    0
111                                 fjump      54   10
112                                 fundef      1    0
0022        end;



113                                 fstop       0    0
0023  end.

                      Associates:
                      Segment No    9
                      ----------------
                      seg:      9
                      ==========
                      Size=    16 words


```

```
 1          s   113    18
 2               34     0     4
 3               27     2     0
 4               21  2051     1
 5               34     0     5
 6                1     1     1
 7                1     0     1
 8                4     0     0
 9                1     1     2
10                1     0     0
11                1     0     1
12               14     0     0
13                4     0     0
14               34     0     7
15                1     1     0
16                1     0     1
17                4     0     0
18                2     1     0
19                1     0  2048
20                5     0     0
21               18    49     7
22                1     1     1
23                1     0     0
24                2     1     1
25                1     0     1
26               11     0     0
27               14     0     0
28                4     0     0
29                1     1     2
30                1     0     0
31                2     1     2
32                1     0     1
33               14     0     0
34               14     0     0
35                4     0     0
36                2     1     0
37               30     1  2048
38               28     1     2
39                2     1     0
40                2     1     1
41               11     0     0
42                2     1     2
43               14     0     0
44                4     0     0
45                1     1     0
46                2     1     0
47                1     0     1
48               11     0     0
49                4     0     0
50               17    16     7
51               32     1     0
52               34     0    10
53                1     1     0
54                1     0     2
55                4     0     0
56                2     1     0
57                1     0  2048
58                5     0     0
59               18   112    17
```

```
 61            1     1     1
 62            1     0  2048
 63            4     0     0
 64            2     1     1
 65            2     1     0
 66            7     0     0
 67           18   105    17
 68           34     0    14
 69            2     1     1
 70            1     0     1
 71           14     0     0
 72           30     1  2048
 73           29     1     2
 74            2     1     1
 75           30     1  2048
 76           29     1     2
 77           10     0     0
 78           18    98    17
 79            1     1     2
 80            2     1     1
 81            1     0     1
 82           14     0     0
 83           30     1  2048
 84           29     1     2
 85            4     0     0
 86            2     1     1
 87            1     0     1
 88           14     0     0
 89           30     1  2048
 90           28     1     2
 91            2     1     1
 92           30     1  2048
 93           29     1     2
 94            4     0     0
 95            2     1     1
 96           30     1  2048
 97           28     1     2
 98            2     1     2
 99            4     0     0
100           34     0    17
101            1     1     1
102            2     1     1
103            1     0     1
104           14     0     0
105            4     0     0
106           17    62    12
107           32     1     1
108            1     1     0
109            2     1     0
110            1     0     1
111           11     0     0
112            4     0     0
113           17    54    10
114           32     1     0
115            0     0     0
116           16
117       sqr          1     1     0     8     1
118       abs          2     1     0     8     1
```

```
121          chr          5     2     0     8     1
122          pred         6     2     0     8     2
123          succ         7     2     0     8     2
124          false        0     3     0     1     0
125          true         1     3     0     1     0
126          input        1    60     1     0     0
127          output       2    60     1     0     0
128          i            0     1     1     0     0
129          j            1     1     1     0     0
130          k            2     1     1     0     0
131          element      2     1     1     5     1
132          ********     1     1  2048     6     1
133              0:
134             1    0    1    0    0    0
135             2    1    2    2    0    3
136             3    2    3    3    0 2048
137             4    3    4    4    0    2
138             5    4    5    6    3   11
139             6    1    2    2    0    3
140             7    5    7    9   12   49
141             8    1    2    2    0    3
142             9    2    3    3    0 2048
143            10    6   10   11   50   57
144            11    1    2    2    0    3
145            12    7   12   13   58   65
146            13    1    2    2    0    3
147            14    8   14   16   66   97
148            15    2    3    3    0 2048
149            16    1    2    2    0    3
150            17    9   17   17   98  113
151            18   10   18   12   50   57
152            19    0    0    0    0    0
153               1     0     2     0     3     0     4     0     5     3     6    12     7    20     8    20     9    27
cont         10    34    11    43    12    50    13    58    14    66    15    66    16    77    17    77    18    84    19
cont            93    20    98    21    98    22   113    23   114    -1    -1
```

7.3 Experimental Results

The results presented in this section are given both in tabular and graphical form with comments being made on the appropriate graph or table where necessary.

Table 7.3.1 Variations in operational parameters used in simulation runs.

Page sizes: 64, 128, 256, 512 words

LRU stack size: 6, 8 pages

PFF critical frequency: 500, 1000 references

Working Set window: 1000, 10000 references

Working Set strobe: 1000 references

The results presented on the following tables and graphs have the following layout:-

Segmentation result		
PFF (page size 64)	Cr freq 500	Cr freq 1000
WS (page size 64)	Window 1000	Window 10000
LRU (page size 64)	Stack 6	Stack 8
PFF (page sz 128)	Cr freq 500	Cr freq 1000
WS (page sz 128)	Window 1000	Window 10000
LRU (page sz 128)	Stack 6	Stack 8
PFF (page sz 256)	Cr freq 500	Cr freq 1000
WS (page sz 256)	Window 1000	Window 10000
LRU (page sz 256)	Stack 6	Stack 8
PFF (page sz 512)	Cr freq 500	Cr freq 1000
WS (page sz 512)	Window 1000	Window 10000
LRU (page sz 512)	Stack 6	Stack 8

Table 7.3.2.1   *Average memory space allocated for program 7.2.1*

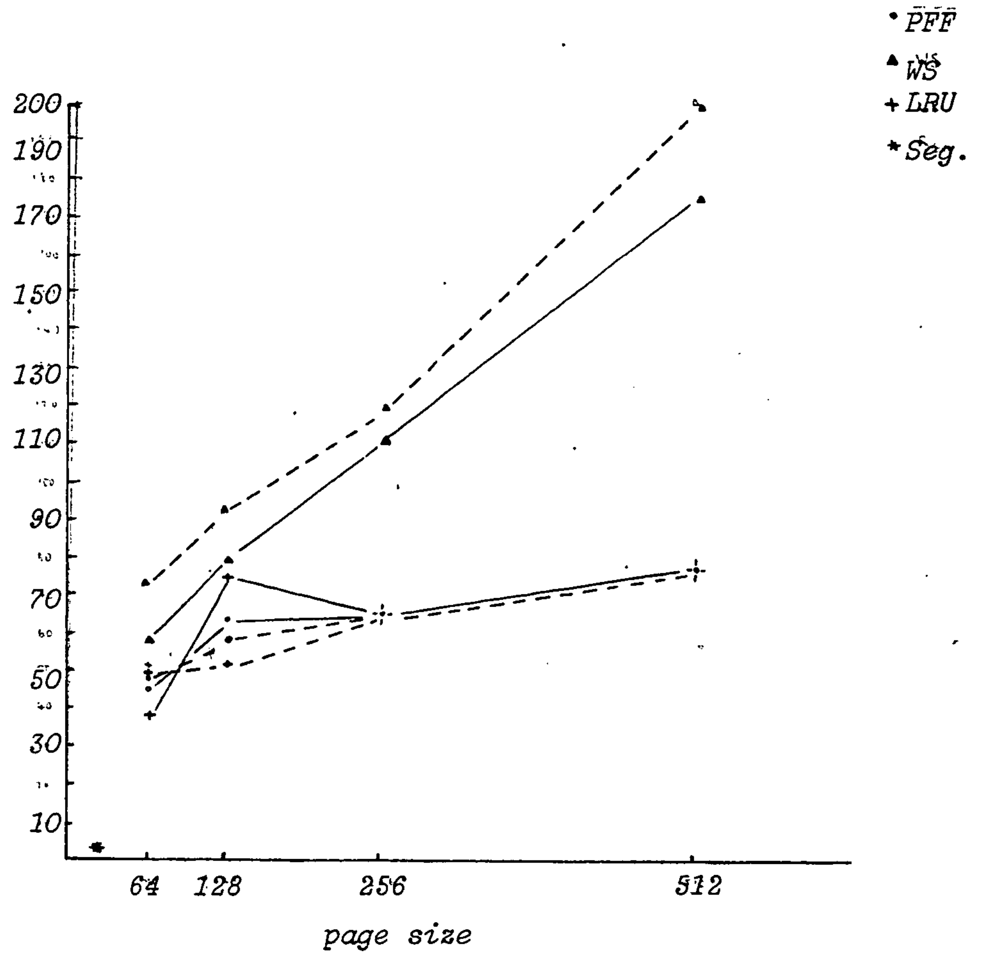
Segmentation	34	34
PFF (ps 64)	457	481
WS (ps 64)	581	726
LRU (ps 64)	378	497
PFF (128)	632	570
WS (128)	787	919
LRU (128)	742	512
PFF (256)	640	640
WS (256)	1110	1192
LRU (256)	640	640
PFF (512)	768	768
WS (512)	1756	1934
LRU (512)	768	768

Comments

Note how the same variation of parameters for PFF and LRU with page sizes 64 and 128 produce in the first case an increase in the amount of memory allocated but in the second case produces a reduction in the allocated space.

Note generally the very considerable difference between the allocated memory in the segmentation approach and the amount allocated by the paged approaches.

Graph 7.3.2.1   Graphical representation of Table 7.3.2.1



Comments

In this and all subsequent graphs the values from the left-hand column of the table are joined by solid lines, whereas those from the right-hand column are joined by dashed lines.

Note the significantly poorer performance of the WS algorithm.

Table 7.3.2.2 *Average memory space allocated for program 7.2.2*

Segmentation	27	27
PFF (64)	250	241
WS (64)	285	352
LRU (64)	261	256
PFF (128)	298	298
WS (128)	375	452
LRU (128)	256	256
PFF (256)	384	384
WS (256)	723	803
LRU (256)	384	384
PFF (512)	512	512
WS (512)	1422	1422
LRU (512)	512	512

Comments

Note that the same variation of parameters as in Table 7.3.2.1 produces the opposite effect, namely a reduction of allocated memory, for page size 64, for the PFF and LRU algorithms.

Graph 7.3.2.2   *Graphical representation of Table 7.3.2.2*

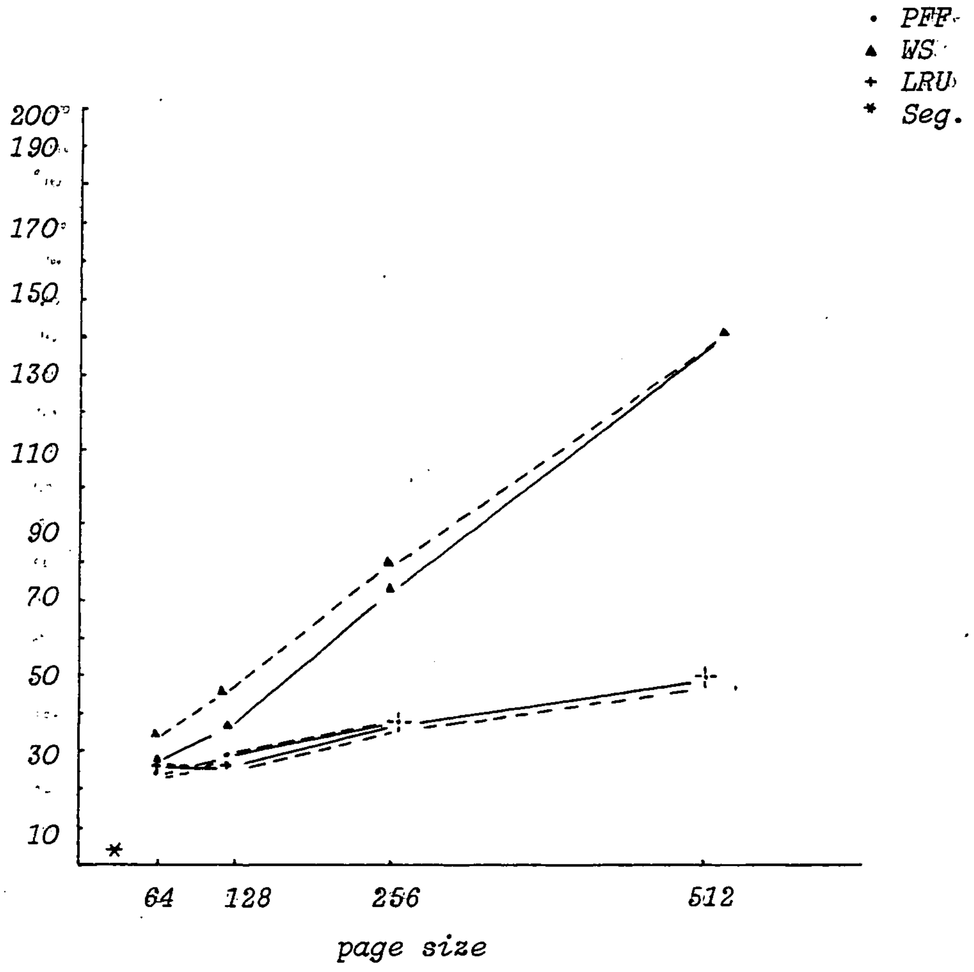


Table 7.3.2.3 Average memory space allocated for program 7.2.3

Segmentation	50	50
PFF (64)	436	430
WS (64)	595	697
LRU (64)	383	511
PFF (128)	565	546
WS (128)	832	889
LRU (128)	767	512
PFF (256)	512	512
WS (256)	1201	1268
LRU (256)	512	512
PFF (512)	512	512
WS (512)	1667	1531
LRU (512)	512	512

Comments

Note how for both LRU and PFF the average amount of memory allocated has a tendency to level out, whereas for WS this is not so.

Graph 7.3.2.3 Graphical representation of Table 7.3.2.3

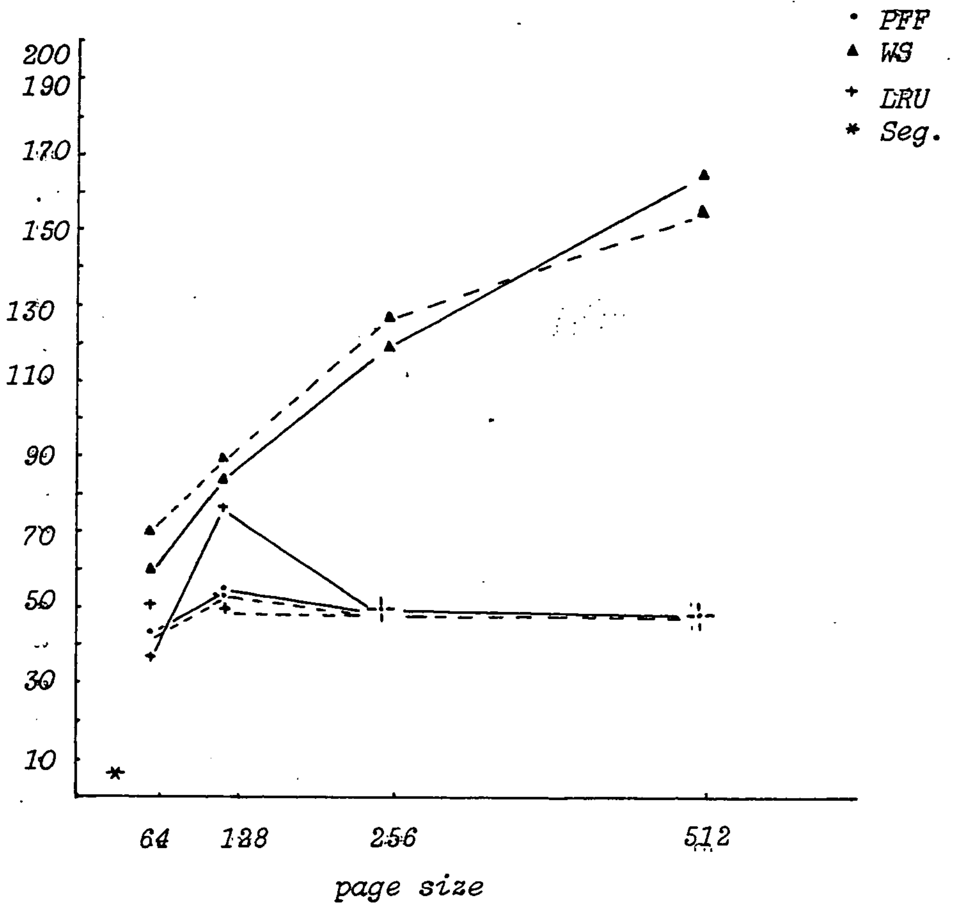


Table 7.3.2.4 Average memory allocated for program 7.2.4

Segmentation	703	703
PFF (64)	1251	1120
WS (64)	384	426
LRU (64)	383	502
PFF (128)	1280	1152
WS (128)	557	613
LRU (128)	767	970
PFF (256)	1280	1280
WS (256)	1006	1095
LRU (256)	1424	1838
PFF (512)	2001	1536
WS (512)	1928	2017
LRU (512)	2583	1536

Comments

It is interesting to note again how LRU and PFF seem to come together in the right-hand column. In this example the poor data locality measure in the current segmented approach causes the average store set size to increase significantly over the previous examples.

Graph 7.3.2.4 Graphical representation of Table 7.3.2.4

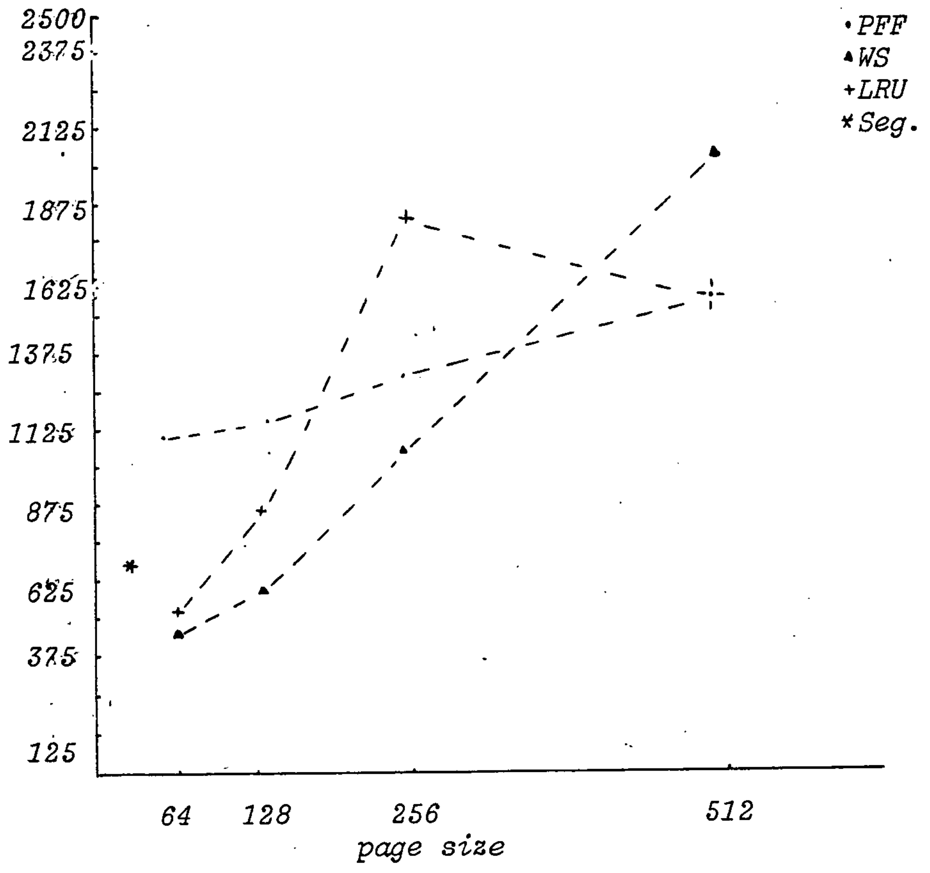


Table 7.3.3.1 *Total amount of program and data traffic into memory for program 7.2.1*

Segmentation	22028	22028
PFF (64)	1280	1216
WS (64)	6016	6016
LRU (64)	14912	10176
PFF (128)	2304	1408
WS (128)	6272	1152
LRU (128)	13312	1152
PFF (256)	1536	1536
WS (256)	3072	1536
LRU (256)	1536	1536
PFF (512)	2048	2048
WS (512)	5120	2048
LRU (512)	2048	2048

Comments

Note the fact that the segmentation approach moves a considerably greater amount of information into memory during program execution than the paged approaches. This is the penalty for small memory allocations.

Note also that LRU tends to "blow-up" as the page sizes reduce.

Equally note that as page sizes get larger the amount of information moved in tends to become the same for all the paged approaches.

Graph 7.3.3.1 Graphical representation of Table 7.3.3.1

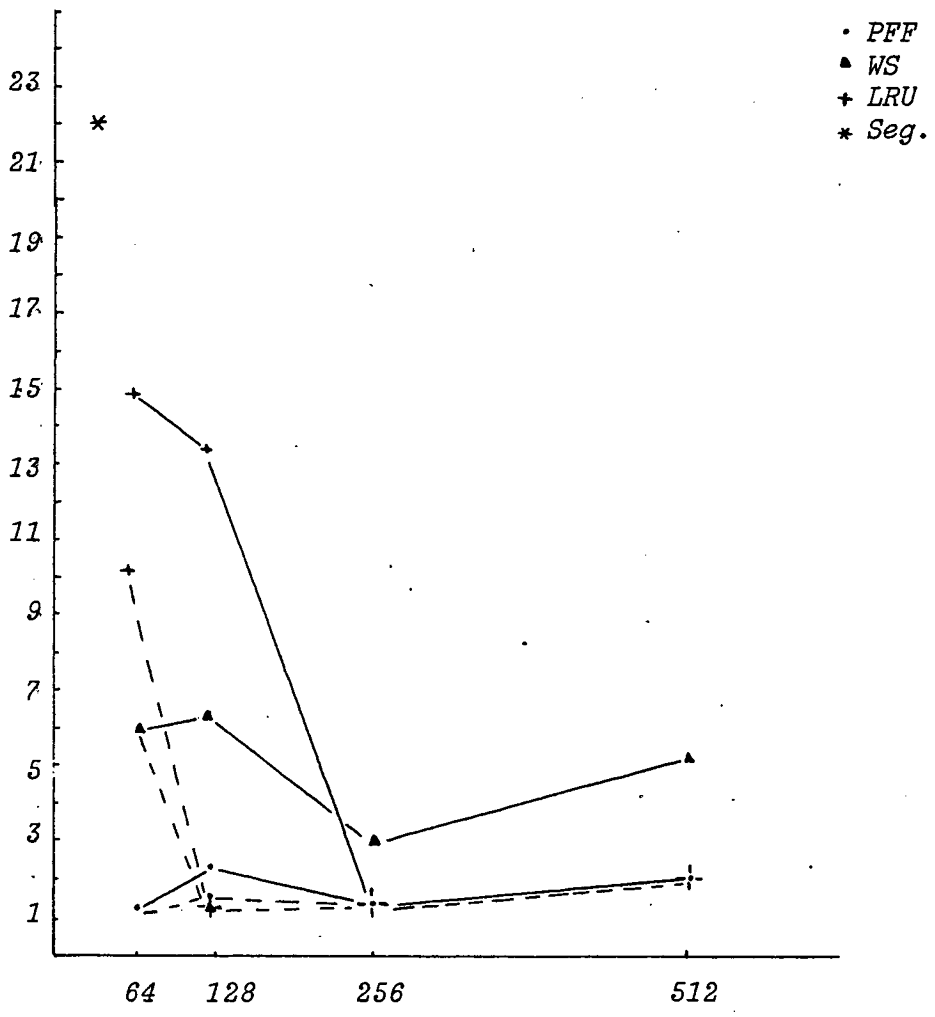


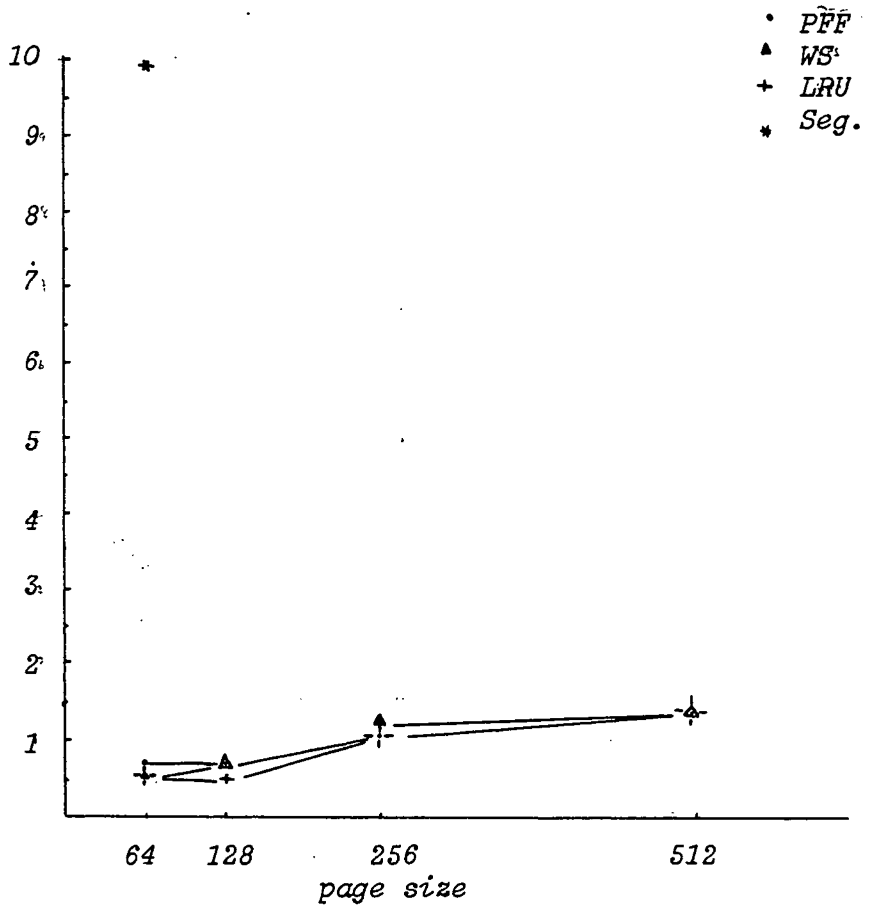
Table 7.3.3.2 Total amount of program and data traffic into memory for program 7.2.2

Segmentation	9993	9993
PFF (64)	768	576
WS (64)	704	640
LRU (64)	704	576
PFF (128)	768	768
WS (128)	768	768
LRU (128)	640	640
PFF (256)	1024	1024
WS (256)	1280	1280
LRU (256)	1024	1024
PFF (512)	1536	1536
WS (512)	1536	1536
LRU (512)	1536	1536

Comments

All the data movement tables assume that only the required amount of data need be moved from backing store when required and that there is no need to access or transfer any encapsulating block. This is reasonable for the larger page sizes but will tend to favour the smaller page sizes and particularly the segmented approach.

Graph 7.3.3.2   Graphical representation of Table 7.3.3.2



Comments -

Only one set of results is shown in the interests of clarity.

Note generally how badly the segmented system behaves

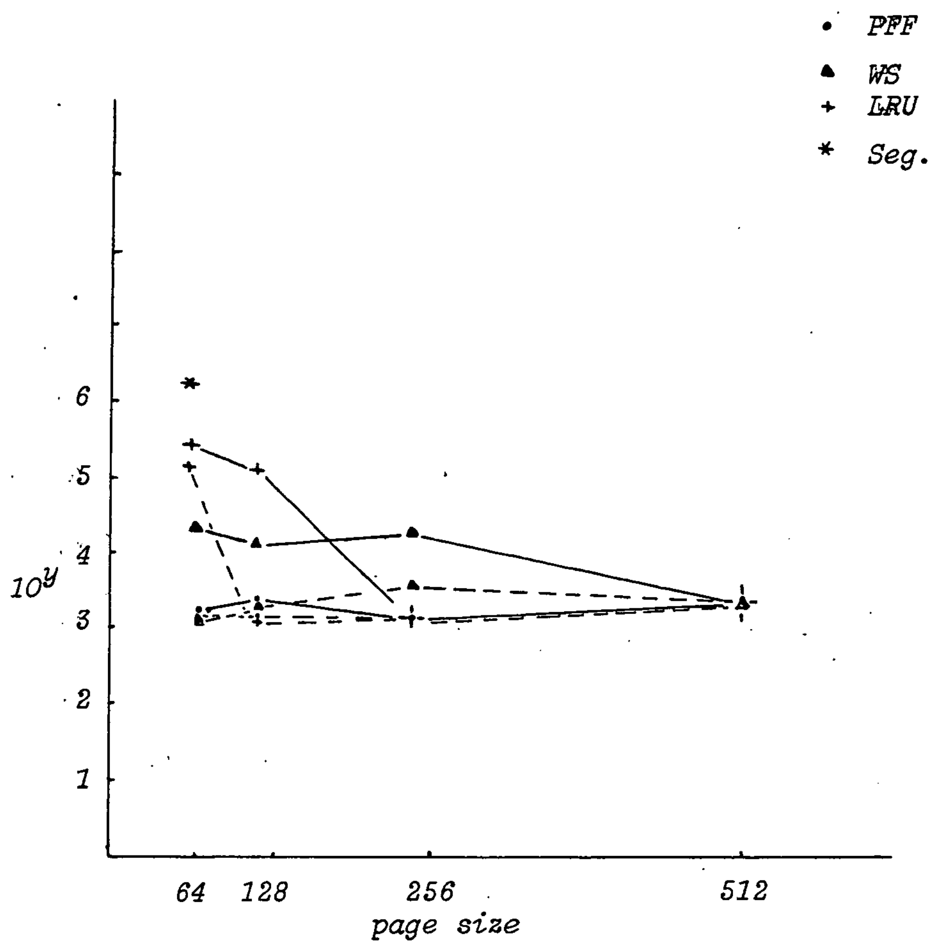
<u>Table 7.3.3.3</u> Total amount of program and data traffic into memory for program 7.2.3

Segmentation	1057007	1057007
PFF (64)	1472	1408
WS (64)	32704	1344
LRU (64)	477248	227136
PFF (128)	1536	1408
WS (128)	14336	1536
LRU (128)	344416	1152
PFF (256)	1280	1280
WS (256)	26312	1792
LRU (256)	1280	1280
PFF (512)	1536	1536
WS (512)	1536	1536
LRU (512)	1536	1536

<u>Comments</u>

In this example the segmented approach fares particularly badly. Again note the tendency for the LRU algorithm's performance to detiorate rapidly as the page size decreases and that, in this case, the WS algorithm shows similar but less extreme behaviour.

Graph 7.3.3.3 *Graphical representation of Table 7.3.3.3*



Comments -

Note the change of scale on the y-axis

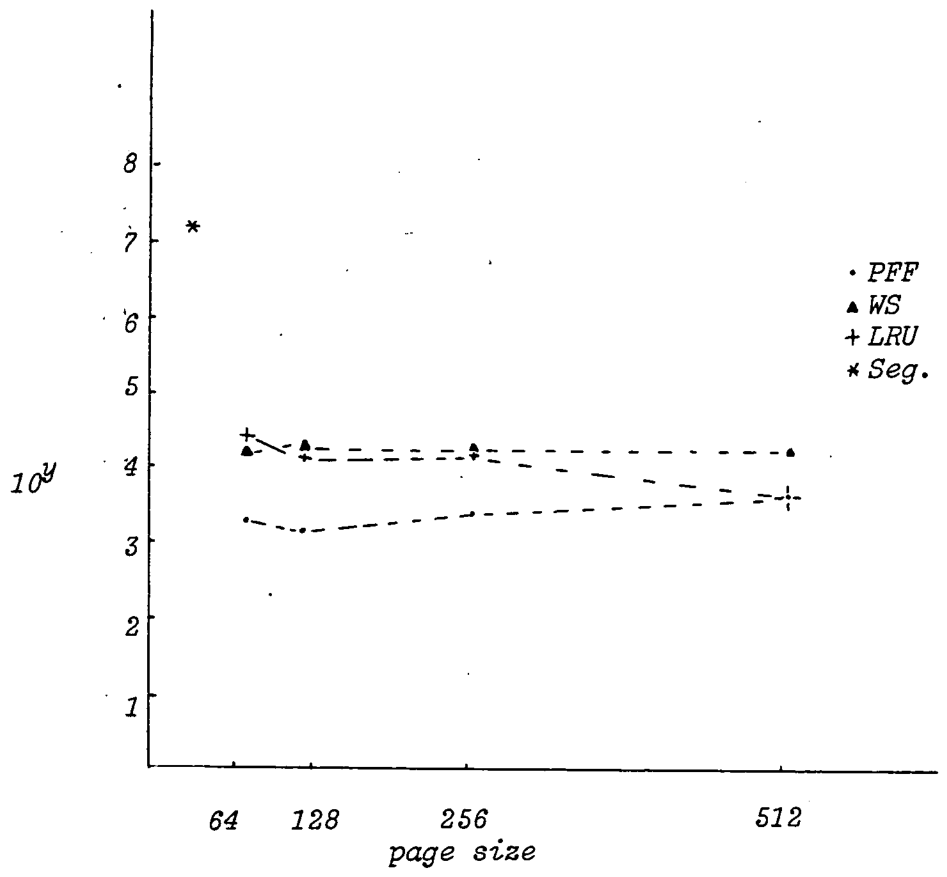
Table 7.3.3.4 Total amount of program and data traffic into memory for program 7.2.4

Segmentation	17217176	17217176
PFF (64)	2816	2304
WS (64)	49252	12352
LRU (64)	539392	13960
PFF (128)	2304	2048
WS (128)	32640	12416
LRU (128)	16384	11264
PFF (256)	2816	2816
WS (256)	13056	12544
LRU (256)	12288	11264
PFF (512)	11264	3584
WS (512)	13824	12800
LRU (512)	11264	3584

Comments

It is again interesting to note that after an initial blow up LRU comes very close to PFF for large page sizes.
Note also just how large the amount of information moved by the segmented approach is. This is due to moving the whole data array in and out of store.

Graph 7.3.3.4   Graphical representation of Table 7.3.3.4



Comments

Only one set of data has been shown for clarity.

It is interesting to see how under these conditions, the data traffic for each algorithm remain relatively constant.

Table 7.3.4.1 *Frequency of memory allocation decisions for program 7.2.1*

Segmentation	*47* refs	*47*
PFF (64)	*2068*	*2176*
WS (64)	*306*	*725*
LRU (64)	*177*	*260*
PFF (128)	*2297*	*3760*
WS (128)	*459*	*827*
LRU (128)	*397*	*4595*
PFF (256)	*6893*	*6893*
WS (256)	*780*	*880*
LRU (256)	*6893*	*6893*
PFF (512)	*10340*	*10340*
WS (512)	*811*	*919*
LRU (512)	*10340*	*10340*

Comments

Here again the penalty of relativel small locality sizes is shown for the segmented approach.

However it is interesting to note the relatively poor performance of both LRU and WS with small page sizes.

Equally it is interesting to see just how similarly PFF and LRU behave under favourable circumstances.

<u>Graph 7.3.4.1</u> *Graphical representation of Table 7.3.4.1*

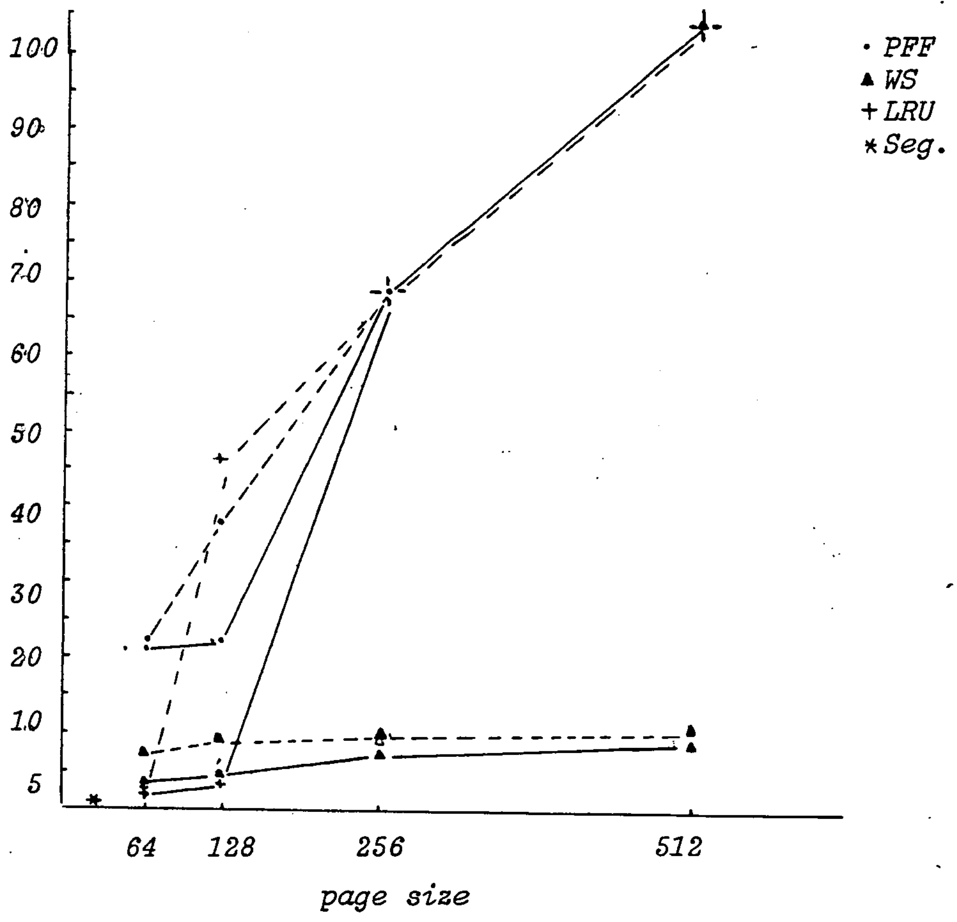


Table 7.3.4.2 Frequency of memory allocation decisions for program 7.2.2

Segmentation	49	49
PFF (64)	2031	2708
WS (64)	696	717
LRU (64)	2216	2708
PFF (128)	4063	4063
WS (128)	812	812
LRU (128)	4876	4876
PFF (256)	6095	6095
WS (256)	4876	4876
LRU (256)	6095	6095
PFF (512)	8126	8126
WS (512)	8126	8126
LRU (512)	8126	8126

Comments

Note here the tendency for the behaviour of all the algorithms to come together. This implies that here can be seen some of the few cases so far where the WS algorithm has not been "worse" than PFF and LRU.

Again the segmented approach behaves poorly.

Graph 7.3.4.2   *Graphical representation of Table 7.3.4.2*

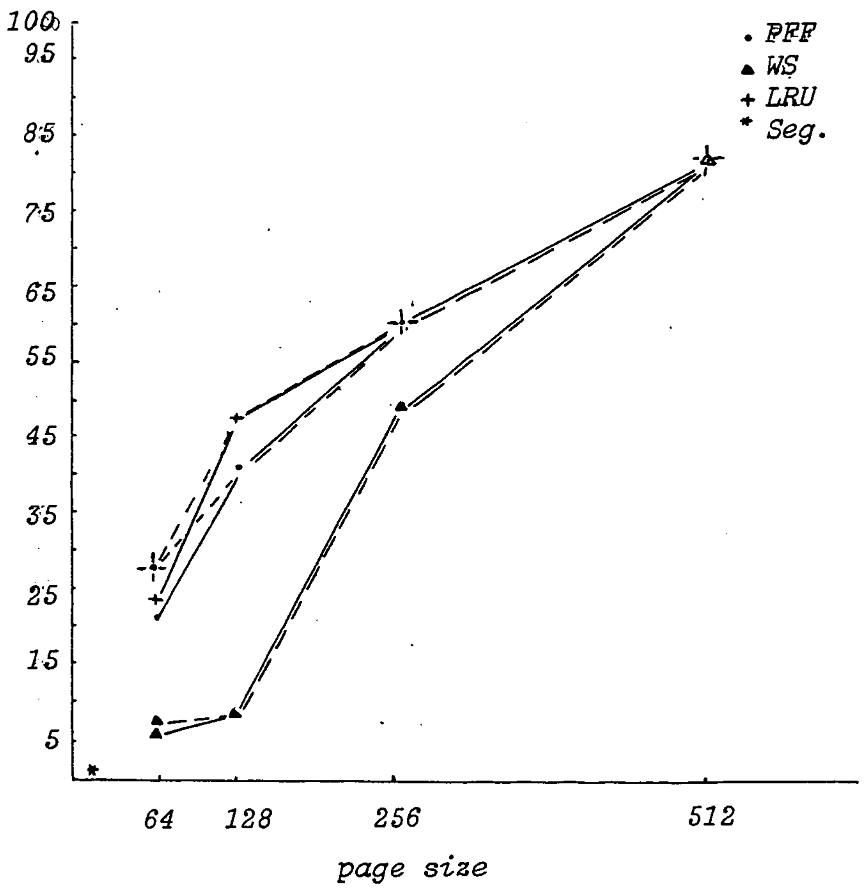


Table 7.3.4.3 Frequency of memory allocation decisions for program 7.2.3

Segmentation	34	34
PFF (64)	31937	33389
WS (64)	590	972
LRU (64)	98	206
PFF (128)	61213	66778
WS (128)	776	984
LRU (128)	270	81618
PFF (256)	146912	146912
WS (256)	876	991
LRU (256)	146912	146912
PFF (512)	244584	244584
WS (512)	996	996
LRU (512)	244584	244584

Comments

Note again the sensitivity of the LRU algorithm to the page size variations and how this can be alleviated by increasing the stack length.

Graph 7.3.4.3 Graphical representation of Table 7.3.4.3

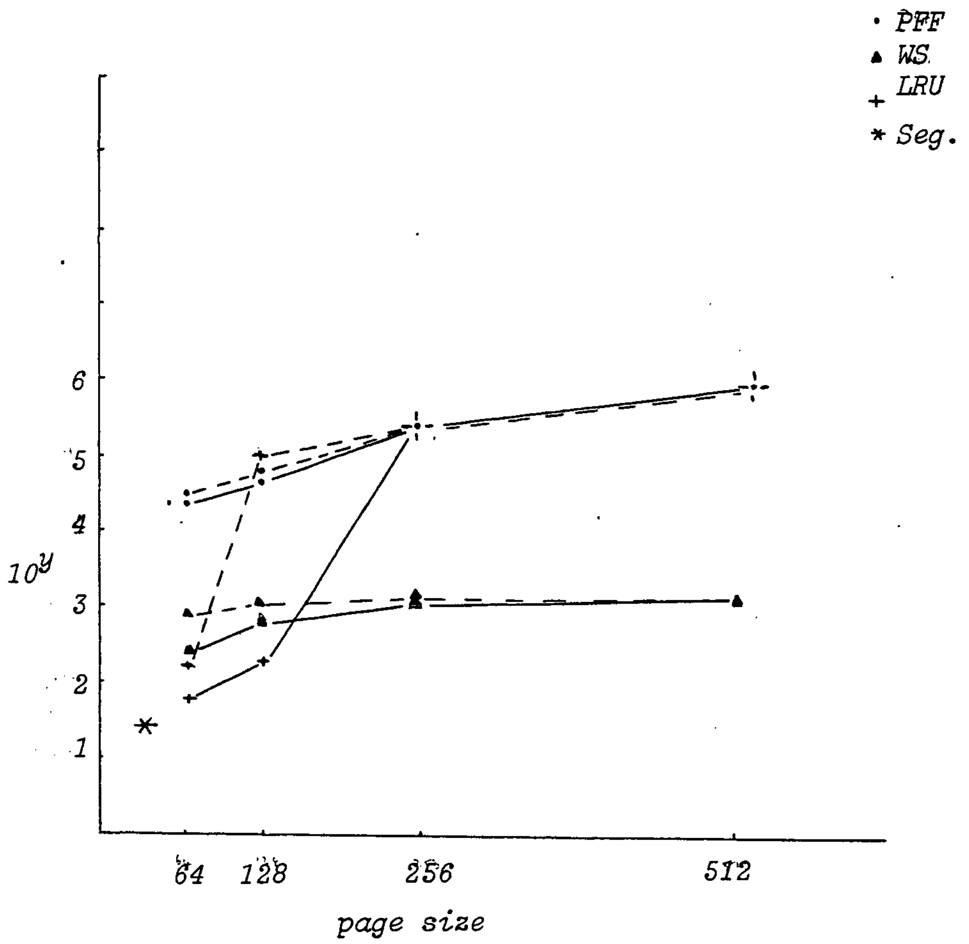


Table 7.3.4.4 Frequency of memory allocation decisions for program 7.2.4

Segmentation	46	46
PFF (64)	25765	31490
WS (64)	590	857
LRU (64)	134	5966
PFF (128)	62925	59665
WS (128)	699	921
LRU (128)	4429	12322
PFF (256)	103059	103059
WS (256)	957	959
LRU (256)	23617	25764
PFF (512)	51529	161950
WS (512)	977	979
LRU (512)	51529	161950

Comments

PFF shows some sensitivity here particularly with the large page size.

<u>*Graph 7.3.4.4*</u>   *Graphical representation of Table 7.3.4.4*

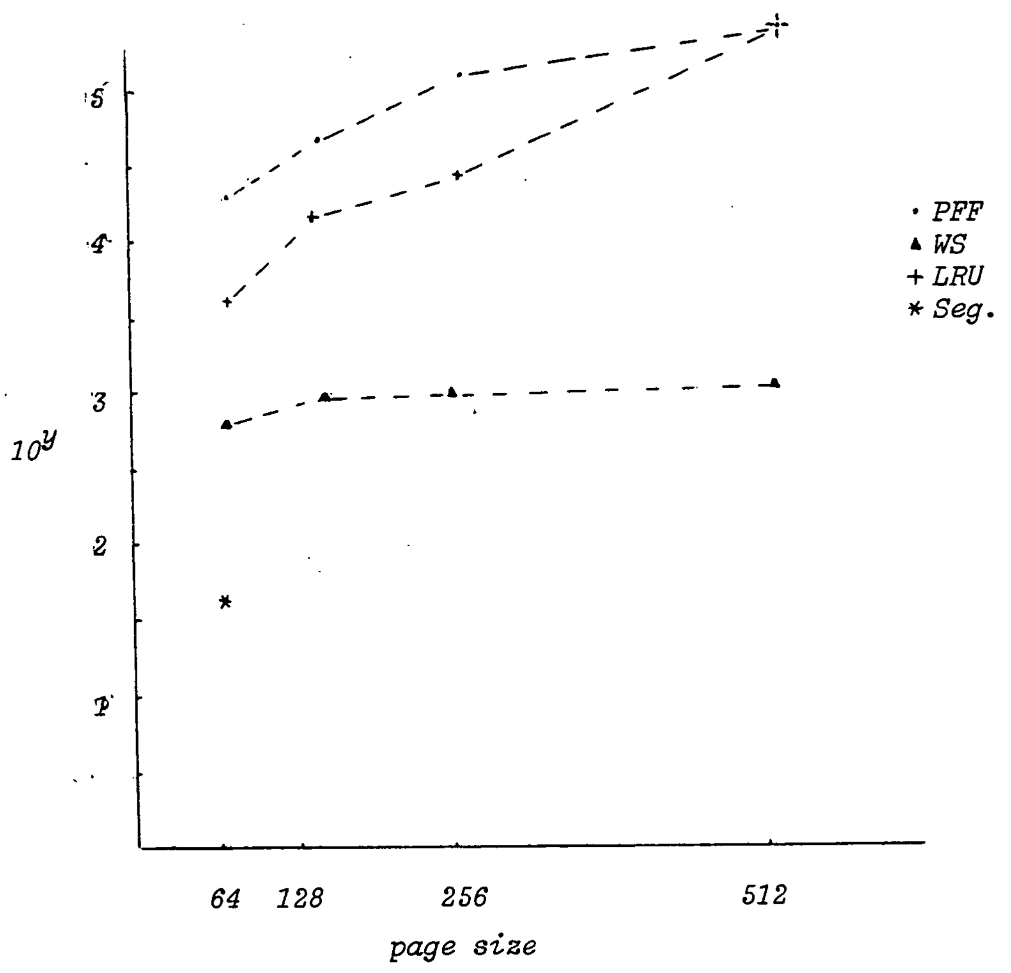


Table 7.3.5.1 Reference density for program 7.2.1

Segmentation	1.38	1.38
PFF (64)	4.52	4.52
WS (64)	.52	.99
LRU (64)	.46	.52
PFF (128)	3.63	6.59
WS (128)	.58	.89
LRU (128)	.54	8.97
PFF (256)	10.7	10.7
WS (256)	.70	.74
LRU (256)	10.7	10.7
PFF (512)	13.4	13.4
WS (512)	.46	.48
LRU (512)	13.4	13.4

Comments

The reference density is estimated by dividing the average number of references between reference decisions by the average amount of memory allocated. This should give an idication of how successful the algorithm has been at estimating locality.

It is interesting to note here how poorly the WS algorithm behaves, this is perhaps due to counting strobe decisions as actual memory decisions.

Graph 7.3.5.1 Graphical representation of table 7.3.5.1

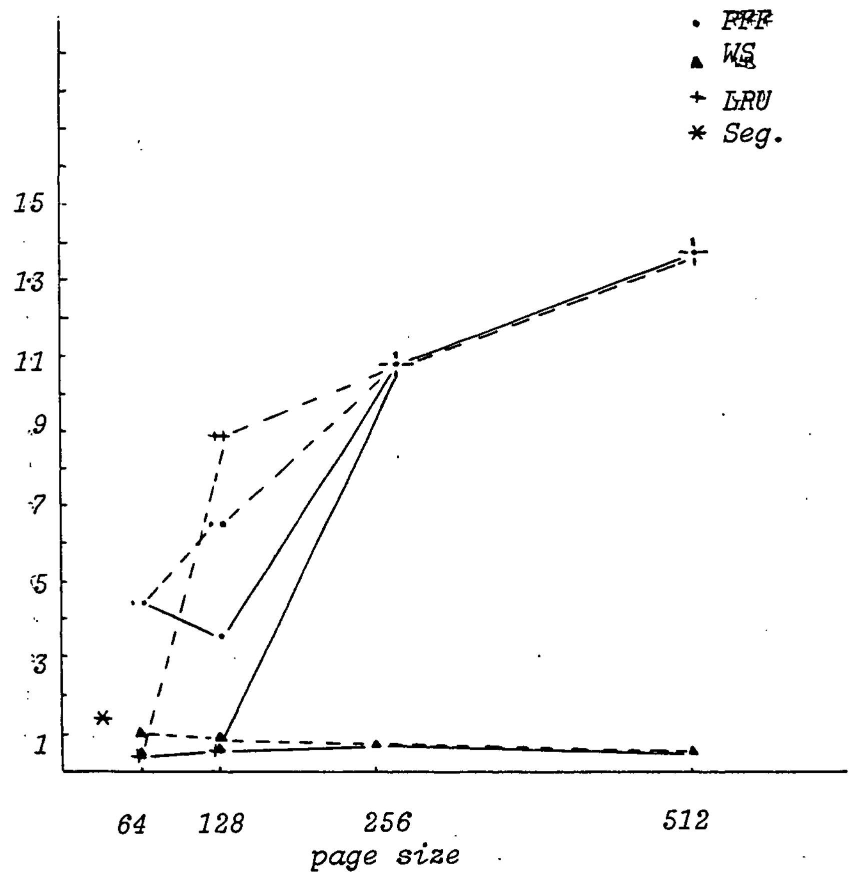


Table 7.3.5.2 Reference density for program 7.2.2

Segmentation	1.81	1.81
PFF (64)	9.2	11.2
WS (64)	2.4	2.03
LRU (64)	8.4	10.5
PFF (128)	13.6	13.6
WS (128)	2.16	1.79
LRU (128)	19.0	19.0
PFF (256)	15.8	15.8
WS (256)	6.74	6.07
LRU (256)	15.8	15.8
PFF (512)	15.8	15.8
WS (512)	5.71	5.71
LRU (512)	15.8	15.8

Comments

In this example the segmented approach does not compare as well as in the previous case.

Graph 7.3.5.2 Graphical representation of Table 7.3.5.2

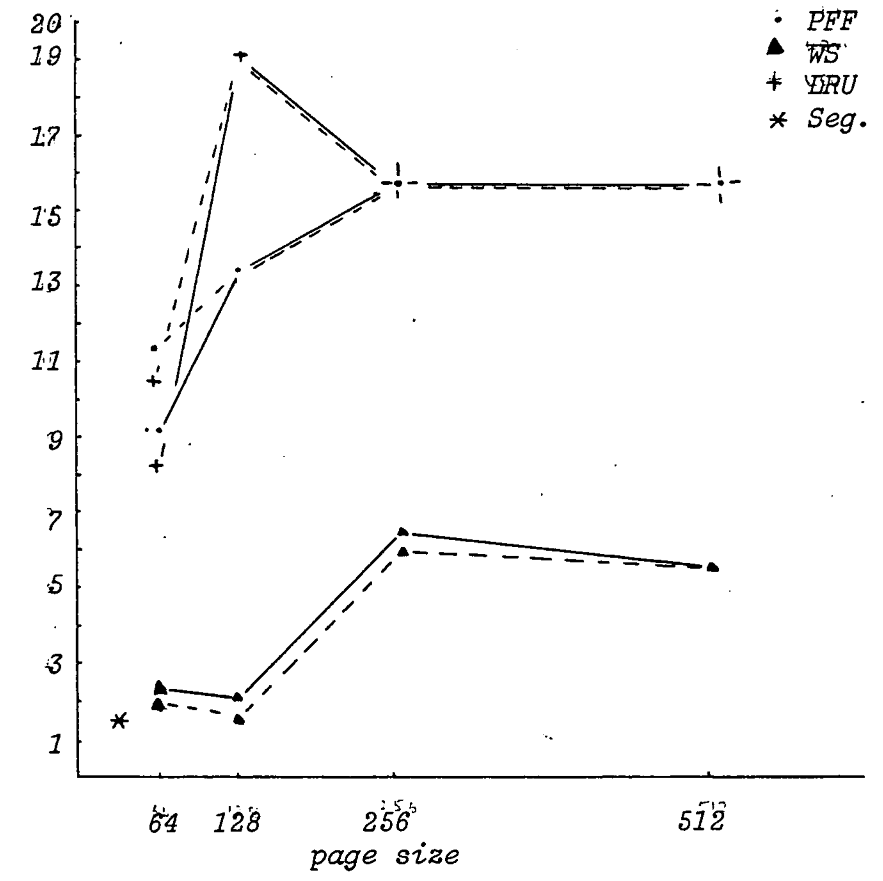


Table 7.3.5.3 Reference density for program 7.2.3

Segmentation	.68	.68
PFF (64)	73.	78.
WS (64)	.99	1.39
LRU (64)	.25	.40
PFF (128)	109.	122.
WS (128)	.93	1.1
LRU (128)	.35	159.
PFF (256)	287.	287.
WS (256)	.70	.78
LRU (256)	287.	287.
PFF (512)	478.	478.
WS (512)	.55	.55
LRU (512)	478.	478.

Comments

Again PFF seems to be clearly the best approach.

LRU graphically displays the effects of its "blow-up" in its reference densities.

WS also displays what happens to the reference density when memory is over-allocated when the page size is 512.

Graph 7.3.5.3 *Graphical representation of Table 7.3.5.3*

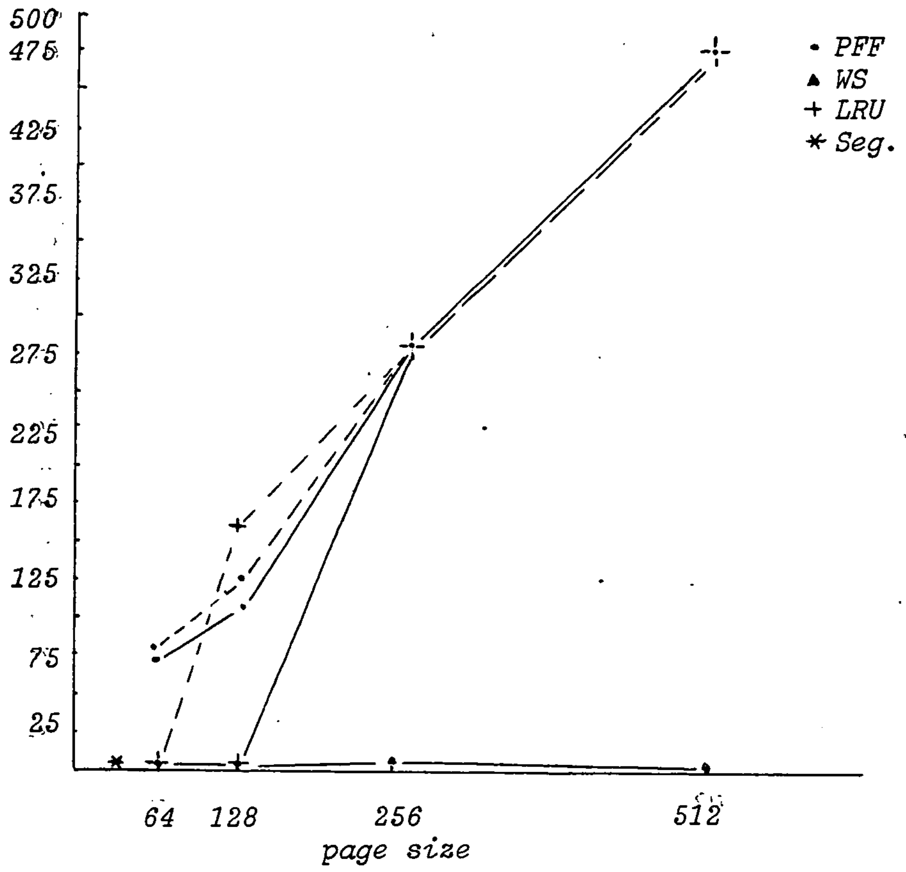


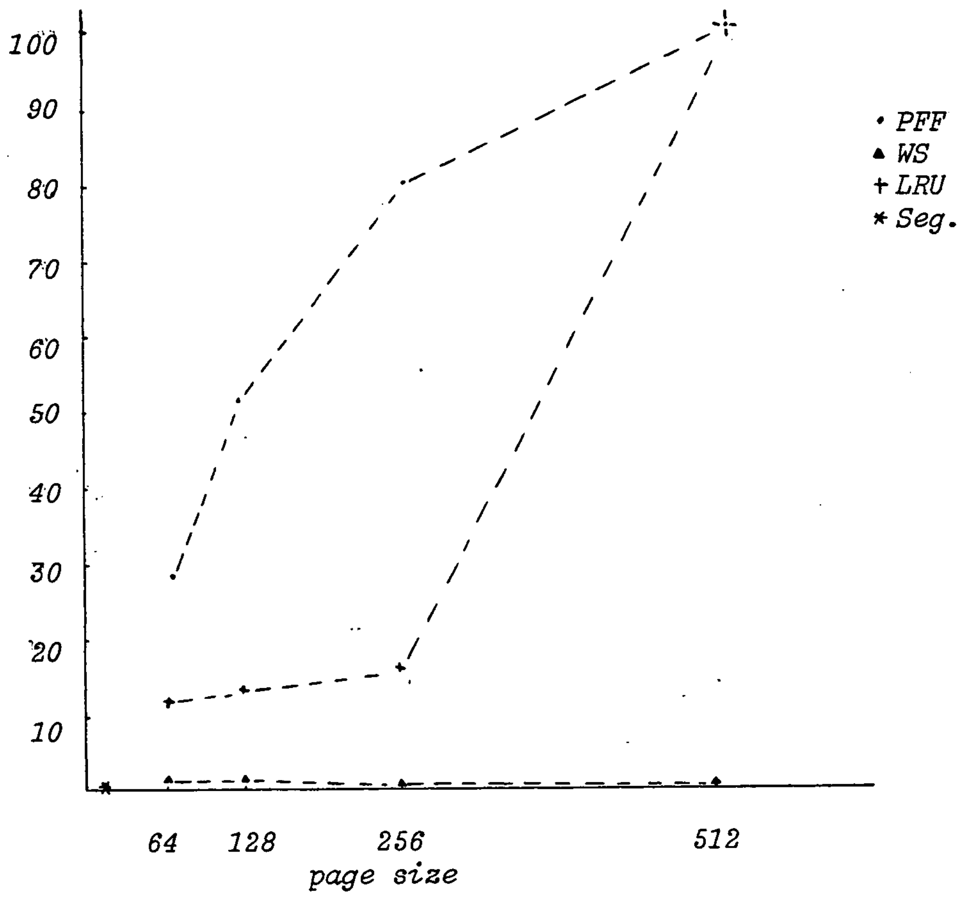
Table 7.3.5.4 *Reference density for program 7.2.4*

Segmentation	.06	.06
PFF (64)	20.6	28.
WS (64)	1.6	1.89
LRU (64)	2.9	11.8
PFF (128)	49.	51.7
WS (128)	1.3	1.5
LRU (128)	5.8	12.6
PFF (256)	80.5	80.5
WS (256)	.95	.88
LRU (256)	16.58	14.01
PFF (512)	25.75	105.43
WS (512)	.50	.48
LRU (512)	19.9	105.43

Comments

It is possible to see here how an increase in the page size causes even PFF (which has up to now appeared to be a fairly stable algorithm) to have its reference density reduced when changing the page size from 256 to 512 and increased in the same cicumstances under different operational parameters.

<u>Graph 7.3.5.4</u> Graphical representation of Table 7.3.5.4



<u>Comments</u>

Again only one set of results has been shown for clarity.

7.4 Conclusions

A number of conclusions can be drawn from the above results.

Firstly, the segmented approach significantly reduces the amount of space occupied by a program during its execution. This can be explained by the relatively small size of, particularly, code localities established by the segmentation approach. This space reduction is not, however, achieved without cost. The cost is first of all shown by the high number of allocation decisions made during the execution time of the program. This would cause a significant increase in the run-time overheads of a program, particularly when coupled with the second high-cost factor - data transfers. The segmented approach causes a very significant increase in traffic between main store and backing store. This is a severe limitation of the proposed approach.

Secondly, the segmented approach does not, in general, give rise to improved locality behaviour when compared particularly with favourable versions of PFF and LRU. This is shown in the reference density figures of the previous section.

Thirdly, compaction overheads would appear to be almost negligible with this segmented approach due to the fact that in most cases relatively small amounts of memory are allocated to programs.

Fourthly, it is clear that multiprogramming systems would not fare very well with the segmented system since the small amount of main store occupied by a program would imply an increase in the multi-programming level which would give rise to severe congestion on the main store to backing store data pathway.
Fifthly, without further investigation of data locality it is not possible to estimate the performance of the segmentation approach for large unstructured data areas. Currently the system would require that, say, a large array would have to be loaded in its entirety to satisfy the context requirements. As well as being potentially wasteful this might even be physically impossible on some system configurations. This physical limitation however, tends not to arise on PDP-11 configurations where the maximum addressable space of a program is usually less than the available memory.
Sixthly, the investigations have tended to show that the paged systems are, as was hypothesised earlier, extremely sensitive to variations in their operational parameters and that the degree of sensitivity is not the same from program to program. Equally it has been shown that if strobes are taken as being allocation decisions for the Working Set algorithm then this algorithm behaves relatively poorly compared with PFF and LRU. This is perhaps unfair but it does show that if pure WS is not

used then practical performance may differ significantly from theoretical predictions.

Finally, it may be concluded that paged systems, when behaving optimally, will easily outperform the proposed approach. However, it is hard to be convinced that paged systems always or frequently behave optimally. Whether an improved segmented approach or some combination of the paged and segmentation strategies could produce stable and satisfactory performance figures in most cases must remain an open question. Any future developments must therefore critically depend on an answer to this question.

CHAPTER 8 FUTURE DEVELOPMENTS

The techniques described above relate strongly the ideas of program structure and program behaviour. It is tempting, therefore, to associate these ideas with the capability concept of Dennis and Van Horn (Den 65). If this were done, it would be possible to implement program modules with protection, locality and behaviour information built-in to them. This, in effect, creates totally self-sufficient program modules, and as such, would present a totally unifying construct for all aspects of program behaviour.

To implement such a system requires an implementation of a language such as EUCLID with a segment-based operating system. Such a system would require a compiler which would extract locality information as well as access information and divide programs into distinct modules with their associated environments. All this information would then be passed to the run-time system.

CHAPTER 9 REFERENCES

This chapter contains those papers referenced in the text and a handful of others which although not explicitly referenced do contribute slightly to the development.

(Ada 75) Adams, J C, and Milliard, G E.
Performance Measurement of the Edinburgh Multi-Access System
Proc. International Computing Symposium 1975

(Ada 76) Adams, J C.
Evaluation of Performance of the EMAS System
Seminaires Modelisation et Mesures
IRIA - LABORIA (1976)

(Aho 71) Aho, A V, Denning, P J, and Ullman, J D.
Principles of Optimal Page Replacement
JACM 18,1 (Jan 1971) pp 80 - 83

(Ale 69) Alexander, M T.
Time-Sharing Supervisor Program
Univ. of Michigan Computer Centre (May 1969)

(Ard 66) Arden, B, Galler, B, et al
Program and Addressing Structure in a Time-Sharing Environment
JACM 13,1 (Jan 1966) pp 1 - 16

(Bar 73) Bard, Y.
Characterisation of Program Paging in a Time-Sharing Environment
IBM Jour. of Res. and Dev. 17, 5 (Sept 1973) pp 387 - 393

(Bar 75) Bard, Y.
Application of the Page Survival Index (PSI) to Virtual Memory System Performance
IBM Jour. of Res. and Dev. 19, 3 (May 1975) pp 212 - 220

(Bat 70) Batson, A, Ju, S, and Wood, D.
Measurements of Segment Size
CACM 13, 3 (March 1970) pp 155 - 159

(Bat 77) Batson, A, Blatt, W E, and Kearns, J P.
Structure within Locality Intervals
Proc. Symp. on modelling and Performance Evaluation ed. Beilner and Gelenbe (Oct 1977) pp 221 - 232

(Bay 68) Bayels, R A et al.
Control Program - 67/Cambridge Monitor System
Program No 360D 05.2.005
Cambridge, Mass. 1968

(Bay 68b) Bayliss, M H J, Fletcher, D G, and Howarth, D J.
Paging Studies Made on the ICT Atlas Computer
IFIP 1968 pp 835 - 836

(Bel 66) Belady, L A.
A Study of Replacement Algorithms for a Virtual Store Computer
IBM Systems Journal 5, 2 (1966) pp 78 - 101

(Bel 69) Belady, L A. and Kuehner, C J.
Dynamic Space Sharing in Computer Systems
CACM 12, 5 (May 1969) pp 282 - 288

(Bel 69b) Belady, L A, Nelson R A, and Shedler, G S.
An anomaly in the Space-Time Characteristics of Certain Programs Running in a Paging Machine
CACM 12, 6 (June 1969) pp 349 - 353

(Bel 71) Bell, C G, and Newell, A.
Computer Structures: Readings and Examples
McGraw-Hill 1971

(Bra 68) Brawn, B, and Gustavson, F.
Program Behaviour in a Paging Environment
AFIPS FJCC (1968) pp 1019 - 1032

(Bry 75) Bryant, P.
Predicting Working Set Sizes
IBM Jour. of Res. and Dev. 19, 3 (May 1975) pp 221 - 229

(Bur 76) Burgevin, P, and Leroudier, J.
Characteristics and Models of Program Behaviour
Proc. 1976 ACM Nat. Conf. pp 344 - 350

(Chu 72) Chu, W W, and Opderbeck, H.
The Page Fault Frequency Algorithm
AFIPS FJCC (1972) pp 597 - 609

(Chu 76) Chu, W W, and Opderbeck, H.
Program Behaviour and the Page Fault Frequency Replacement Algorithm
Computer, 9, 11 (Nov 1976) pp 29 - 38

(Cof 68) Coffman, E G, and Varian, L C.
Further Experimental Data on the Behaviour of Programs in a Paging Environment
CACM 11, 7 (July 1968) pp 471 - 474

(Cof 72) Coffman, E G, and Ryan, T.
A Study of Storage Partitioning Using a Mathematical Model of Locality
CACM 15, 3 (March 1972) pp 185 - 190

(Cof 73) Coffman, E G, and Denning, P J.
Operating Systems Theory
Prentice- Hall 1973

(Cor 65) Corbato, F J, and Vyssotsky, V A.
Introduction and Overview of the Multics System
AFIPS FJCC (1965) Pt I pp 185 - 196

(Den 68a) Denning, P J.
Resource Allocation in Multi-Process Computer Systems
MIT Project MAC Technical Report MAC-TR 50 (May 1968)

(Den 68b) Denning, P J.
The Working Set Model for Program Behaviour
CACM 11, 5 (May 1968) pp 323 - 333

(Den 70) Denning, P J.
Virtual Memory
Computing Surveys, 2, 3 (Sept 1970) pp 153 - 189

(Den 72) Denning, P J, and Schwartz, S C.
Properties of the Working Set Model
CACM 15, 3 (March 1972) pp 191 - 198

(Den 75) Denning, P J, and Kahn, K C.
A Study of Program Locality and Lifetime Functions
Proc 5th ACM SIGOPS Symposium (Nov 1975) pp 207 - 216

(Den 78) Denning, P J, and Slutz, D R.
Generalised Working Sets for Segment Reference Strings
CACM 21, 9 (1978) pp 750 - 759

(Den 65) Dennis, J B.
Segmentation and the Design of Multi-Programmed Computer Systems
JACM 12, 4 (Oct 1965) pp 589 - 602

(Den 66) Dennis, J B.
Programming Semantics for Multi-Programmed Computations
CACM 9, 3 (Mar 1966) pp 143 - 155

(Dij 68) Dijkstra, E W.
The Structure of the THE Multiprogramming System
CACM 11, 5 (May 1968) pp 341 - 346

(Dij 76) Dijkstra, E W.
A Discipline of Programming
Prentice - Hall 1976

(Doh 70) Doherty, W J.
Scheduling TSS/360 for Responsiveness
AFIPS FJCC (1970) pp 97 - 112

(Fag 76) Fagin, R.
A Counter-Intuitive Example of Computer Paging
CACM 19, 2 (Feb 1976) pp 96 - 97

(Fer 74) Ferrari, D.
Improving Locality by Critical Working Sets
CACM 17, 11 (Nov 1974) pp 614 - 620

(Fer 77) Ferrari, D, and Kobayashi, M.
Program Restructuring Algorithms for Global
LRU Environments
Proc. International Computing Symposium 1977
pp 277 - 283

(Fin 66) Fine, G H, Jackson, C W, and McIsaac, P V.
Dynamic Program Behaviour under Paging
Proc. 1966 ACM Nat. Conf. pp 223 - 228

(Fot 61) Fotheringham, J.
Dynamic Storage in the Atlas Computer Including
an Automatic Use of Backing Store
CACM 4, 10 (Oct 1961) pp 435 - 436

(Han 73) Hansen, P B.
Operating Systems Principles
Prentice - Hall 1973

(Hat 72) Hatfield, D J.
Experiments on Page Size, Program Access Patterns and Virtual Memory Performance
IBM Jour. of Res. and Dev. (Jan 1973) pp 58 - 66

(Hat 71) Hatfield D J, and Gerald, J.
Program Restructuring for Virtual Memory
IBM Systems Journal 10, 3 (1971) pp 168 - 192

(Hed 75) Hedges, R L, and Pooch, U W.
A Measure for Program Locality in Demand Paging
Proc. 1975 ACM Nat. Conf. pp 181 - 188

(Hoa 72) Hoare, C A R, and McKeag, R M.
A Survey of Store Management Techniques -
in Operating Systems Techniques
ed Hoare and Perrott
Academic Press 1972

(IBM a) IBM
OS / Virtual Storage 1 Features Supplement
No GC 20-1752-0

(IBM b) IBM
OS / Virtual Storage 2 Features Supplement
No GC 20-1753-0

(IBM 70) IBM
System / 360 Time-Sharing Operating System
Program Logic Manual
No S360-36 GY 28-2009-2

(IBM 72) IBM
IBM System / 360 Operating System Introduction
No GC 28-6534-3

(Ich 76) Ichbiah, J D, Rissen, J P, Heliard, J C, and Cousot, P.
The System Implementation Language LIS
Technical Report 4549 E1/EN
Compagnie Internationale pour l'Informatique

(Inn 77) Innes, D R.
Exploiting the Least Recently Used Page Replacement Algorithm
Software - Practice and Experience 7, 1977
pp 271 - 273

(Jen 74) Jensen, K, and Wirth, N.
PASCAL - User Manual and Report
Lecture Notes in Computer Science Vol. 18
Springer - Verlag 1974

(Kil 62) Kilburn, T, Edwards, D, Lanigan, M, and Sumner, F.
One - Level Storage System
IRE Trans EC - 11, 2 (April 1962) pp 223 - 235
(Reprinted in (Bel 71))

(Knu 68) Knuth, D E.
The Art of Computer Programming Vol. 1
Addison - Wesley 1968

(Lam 77) Lampson, B W et al.
EUCLID
SIGPLAN Notices February 1977

(Lan 69) Lanzano, B C.
Loader Standards for Overlay Programs
CACM 12, 10 (Oct 1969) pp 541 - 550

(Low 70) Lowe, T C.
Automatic Segmentation of Cyclic Program Structures Based on Connectivity and Processor Timing
CACM 13, 1 (Jan 1970) pp 3 - 6, 9

(McKe 67) McKeeman, W M.
Language Directed Computer Design
AFIPS FJCC (1967) pp 413 - 417

(McKe 71) McKeag R M.
Burroughs B5500 Master Control Program
Queen's University Belfast 1971

(Mad 73) Madnick, S E.
Storage Hierarchy Systems
MIT Project MAC Technical Report MAC-TR 107 (April 1973)

(Mat 70) Mattson, R L, Gecsei, J, Slutz, D R and Traiger, I L.
Evaluation Techniques for Storage Hierarchies
IBM Systems Journal 9, 2 (1970) pp 78 - 117

(Oli 74) Oliver, N A.
Experimental Data on Page Replacement Algorithm
AFIPS NCC 1974 pp 179 - 184

(Org 69) Organick, E I.
A Guide to Multics for Sub-System Writers
Project MAC 1969

(Par 77) Parent, M, and Potier, D.
A Note on the Influence of Program Loading on the Page Fault Rate
Acta Informatica 8, 4 1977 pp 359 - 370

(Pot 77) Potier, D.
Analysis of Demand Paging Policies with Swopped Working Sets
Proc 6th ACM SIGOPS Symposium (Nov 1977) pp 125 - 131

(Pri 73) Prieve, B G.
Using Page Residency to Select the Working Set Parameter
CACM 16, 10 (Oct 1973) pp 619 - 620

(Pri 76) Prieve, B G, and Fabry, R S.
VMIN - An optimal Variable Space Page Replacement Algorithm
CACM 19, 5 (May 1976) pp 295 - 297

(Pur 74) Purcell, C J.
The Control Data STAR-100 - Performance Measurements
AFIPS NCC 1974 pp 385 - 387

(Ran 69) Randell, B
A Note on Storage Fragmentation and Program Segmentation
CACM 12, 7 (July 1969) pp 365 - 369, 372

(Rit 78) Ritchie, D, and Thompson, K.
The UNIX Time-Sharing System
Bell System Technical Journal 57, 6(Jul - Aug 1978) Pt II pp 1905 - 1931

(Rod 71) Rodriguez-Rosell, J.
Experimental Data on How Program Behaviour Affects the Choice of Scheduler Parameters
Proc. 3rd ACM SIGOPS Symposium (Oct 1971) pp 156 - 163

(Rod 73) Rodriguez-Rosell, J.
Empirical Working Set Behaviour
CACM 16, 9 (Sept 1973) pp 556 - 560

(Sad 75) Sadeh, E
An Analysis of the Page Fault Frequency Replacement Algorithm
Proc. 5th ACM SIGOPS Symposium (Nov 1975) pp 6 - 13

(Say 69) Sayre, D.
Is Automatic Folding of Programs Efficient Enough to Replace Manual?
CACM 12, 12 (Dec 1969) pp 656 - 660

(Smi 76) Smith, A J.
Analysis of Optimal, Look-Ahead, Demand Paging Algorithms
SIAM Journal on Computing 5,4 (Dec 1976) pp 743 - 757

(Smi 78) Smith, A J.
Bibliography on Paging and Related Topics
ACM Operating Systems Review 12, 4 (Oct 1978)

(Spi 72) Spirn, J R, and Denning P J.
Experiments with Program Locality
AFIPS FJCC (1972) pp 611 - 621

(Spi 77) Spirn, J R.
Program Behaviour : Models and Measurements
Elsevier Computer Science Library 1977

(Var 67) Varian, L C, and Coffman, E G.
An Experimental Study of the Behaviour of Programs in a Paging Environment
ACM SOSP (Oct 1967) pp

(Wil 72) Wilner, W.
Design of the B1700
AFIPS FJCC (1972) pp 489 - 497

(Whi 73) Whitfield, H, and Wight, A S.
The Edinburgh Multi-Access System
Computer Journal 16, 4 pp 331 - 346

(Wir 76) Wirth, N.
Algorithms + Data Structures = Programs
Prentice-Hall 1976