A N A L Y S I S O F S O M E A L G O R I T H M S

F O R U S E O N

P A G E D V I R T U A L M E M O R Y C O M P U T E R S

J. C. Knight.

September, 1973.

Analysis Of Some Algorithms For Use On Paged Virtual Memory Computers

E R R A T A

Page 15 - Line 4b : Replace 'dx' by 'dy'.

Page 15 - Line 2b : Replace $\int_0^1 y^m(1 - y)^{n-m}\, dy$ by $\int_0^1 y^{m-1}(1 - y)^{n-1}\, dy$.

Page 16 - Line 3 : Replace $n(n - 1)/2(n + 1)$ by $n(n - 3)/2(n + 1)$.

Page 52 - Line 1b : Replace 'is' by 'is less than or equal to'.

Page 70 - Line 6 : Replace $\log (M - i)$ by $\log (M - 1)$

Page 120 - Line 9 : Replace H_{c-k-2} by $H_{c-k'-2}$.

Page 175 - Line 4b : Replace 'adjecant' by 'adjacent'.

Page 108 - Insert following line 5:-

The idea of not resolving clashes during address calculation sorting was suggested by Jones, 1970. The modification involving address calculation sorting in place was suggested by Kronmal and Tarter, 1965.

Page 174 - Insert following line 17:-

38. Jones, B.: "A Variation On Address Calculation Sorting", Comm. ACM, Vol. 13, No. 2, 1970.

39. Kronmal, R., M. Tarter: "Cumulative Polygon Address Calculation Sorting", Proc. 20th ACM National Conference, 1965.

This Thesis Is Dedicated To My Father.

ABSTRACT

Handling a single page fault involves execution of thousands of instructions, drum rotational delay and is usually so expensive that if it can be avoided, almost any other cost can be tolerated. Optimizing operating system performance is usually the main concern of computer scientists who deal with paged memories. However, redesigning the algorithm used by a problem program can often result in a very significant reduction in paging, and hence in program execution time. The redesigned algorithm frequently does not satisfy the more conventional efficiency criteria.

A sorting algorithm, Hash Coding and other search algorithms are considered. Analytic and simulation studies are presented, and some modifications are proposed to reduce the number of page faults produced by data set references. Analysis is in terms of three of the most commonly used page replacement algorithms i.e. least recently used, first in first out, and random selection.

The modifications are for the most part relatively minor and in some cases have appeared elsewhere in the context of searching on external storage media. The important aspects are the dramatic performance improvements which are possible, and the fact that classical internal algorithms are inappropriate for use in a paged virtual memory system.

ACKNOWLEDGEMENTS

I welcome this opportunity to thank the staff and my fellow students at the Computing Laboratory of the University of Newcastle Upon Tyne for providing an extremely stimulating environment for graduate study. In particular, I am grateful to Professor E. S. Page, the Director, for his extensive advice, encouragement and support, and to Dr. T. Anderson and Mr. T. Betteridge for many helpful discussions.

My wife must be praised for her endless patience and the meticulous care with which she typed this thesis.

Finally, none of this work would have been possible without the sponsorship of the Science Research Council, and the financial support of the University of Newcastle Upon Tyne and International Business Machines (United Kingdom) Limited.

TABLE OF CONTENTS

1. Introduction. 1

1.1 The Principles Of A Paged Memory. 1

1.2 Page Replacement Algorithms. 3

1.3 The Costs Of Paging. 6

1.4 The Problem Program Approach. 8

1.5 The Objectives Of This Dissertation. 10

1.6 Paging Analysis Model. 11

2. Element Selection 13

2.1 The FIND Algorithm. 13

2.1.1 Definition Of The Basic Algorithm. 13

2.1.2 A Modification To The Basic Algorithm. 14

2.1.3 The Expected Number Of Records Involved In The Second Step. 14

2.1.3.1 The Basic Algorithm. 15

2.1.3.2 The Modified Algorithm. 16

2.1.4 The Expected Number Of Records Involved In The Third And Subsequent Steps. 18

2.1.5 Paging Analysis. 20

2.1.5.1 L.R.U. Page Replacement. 21

2.1.5.2 F.I.F.O. Page Replacement. 23

2.1.5.3 Random Page Replacement. 25

2.2 The LOCATE Algorithm. 26

2.2.1 Definition. 26

2.2.2 Analysis Of The Second Scan. 27

2.2.3 A Simple Modification. 29

2.2.4 A Major Modification. 30

2.2.5 Paging Analysis. 32

2.2.5.1 The Major Modification Operating With L.R.U. Replacement. 32

2.2.5.2 The Major Modification Operating With F.I.F.O. Replacement. 33

2.2.5.3 The Major Modification Operating With Random Page Replacement. 34

2.3 The Advantages Of LOCATE. 34

3. Scatter Storage Tables. 36

3.1 Introduction. 36

3.2 Random Request Pattern. 37

3.2.1 The Basic Technique. 37

3.2.2 Minor Modifications. 40

3.2.3 A Major Modification. 46

3.3 Non-Random Request Pattern. 48

3.3.1 An Example Of A Non-Random Request Pattern. 48

3.3.2 A Model Of Non-Random Requests. 52

3.3.3 L.R.U. Page Replacement. 52

3.3.4 Random Page Replacement. 54

3.3.5 F.I.F.O. Page Replacement. 66

4. Searching. 67

4.1 Introduction. 67

4.2 Binary Searching. 67

4.2.1 Conventional Binary Searching. 67

4.2.2 Modified Binary Searching. 69

4.2.3 L.R.U. Page Replacement. 72

4.2.4 Random Page Replacement. 74

4.2.5 F.I.F.O. Page Replacement. 77

4.2.6 Simulation Results. 79

4.3 Binary Sequence Search Trees. 84

4.3.1 Storage Layout. 84

4.3.2 Low Frequency Of Searches. 85

4.3.3 High Frequency Of Searches. 86

4.4 An Optimum Search Technique. 87

5. Sorting And Merging. 92

5.1 Introduction. 92

5.2 Basic Algorithm. 93

5.3 Modifications. 98

5.4 Paging Analysis. 108

5.4.1 The Sort Phase. 109

5.4.1.1 L.R.U. Page Replacement. 109

5.4.1.2 Random Page Replacement. 119

5.4.1.3 F.I.F.O. Page Replacement. 122

5.4.1.4 The MIN Algorithm. 122

5.4.2 Simulation Of The Sort Phase. 123

5.4.3 The Merge Phase. 145

5.5 Parameter Setting. 161

5.5.1 The Ratio Of Source Area Size To Object Area Size. 163

5.5.2 The Source Area Size And Order Of Merging. 164

5.6 Conclusion. 165

6. Conclusion. 167

References. 171

Appendix. 175

Chapter 1.

INTRODUCTION

1.1 The Principles of a Paged Memory.

The cost per bit of computer memories almost always varies inversly with access time. In consequence the high speed storage, or main memory, from which the central processing unit (C.P.U.) executes programs is usually small. Slower, secondary storage devices often have great capacity but are far too slow to allow them to be directly referenced by the C.P.U.

Thus it is often found that programs require more main memory space than is available on a particular computer. In this situation it is usually left to the programmer to divide his program into smaller parts so that each separate part will fit into memory. He then has to arrange for one part at a time to be loaded into storage for execution, the remainder of the program residing on some form of backing storage. The programmer is supposed to understand his program and the operating system sufficiently well to be able to do this efficiently.

The main function of a paged virtual memory computer is to automate this overlaying process completely, using special hardware and software techniques. The adjective "virtual" is used because an address space is usually provided for the user which bears no obvious relation to real memory size or organisation. The virtual memory is usually much larger than the physical main memory available but the programmer is urged to ignore this and treat virtual memory as if it were real.

The terms real memory and real storage are synonymous with main memory.

At any particular time the contents of those areas of virtual storage actually required for execution to proceed are located in real memory. The remainder, and in some systems (e.g. I.B.M.'s OS/VS) copies of those areas located in main storage as they were prior to loading, reside on backing storage.

A mapping mechanism is employed to locate a given item from its virtual address. In this way several different pieces of information can be located in the same real memory cell at different times and the mapping is updated to indicate the real memory contents at any given time.

The mapping, or address translation as it is usually referred to, takes the form of a table of correspondences between virtual and real addresses. Ideally a table entry for each virtual memory word should exist so that at any instant, only the individual words involved in a program's execution need be in main memory but clearly this would make the table undesirably large. It would also make the operation of the secondary storage and its associated channel very inefficient because a complete I/O operation would be needed to place one word of information in main memory.

The compromise which occurs in practice is that virtual memory is divided into blocks of equal size and only the base address of each block is contained in the table. In this way the size of the table is drastically reduced and the transfers to and from backing storage involve blocks of information rather than single words. The blocks are referred to as pages and a region of main memory equal in size to

a page and with starting address equal to a multiple of the page size is known as a page frame. There is no clear cut optimum page size and a size which is suitable for one program may not be for another. Some systems (e.g. I.B.M.'s System 370) allow two different page sizes in an attempt to alleviate this problem. It is interesting to note that this use of the term page appears as far back as 1949 in the context of the machine built at Manchester University at that time (Kilburn et al, 1953).

Reference to an address not contained within a page in main memory is termed a page fault and clearly, the program which made the reference will be unable to continue execution until the required address is available in real storage. Thus servicing a page fault involves transferring a page into real storage and displacing a page from real storage if it is completely full, together with whatever housekeeping operations are necessary. The technique of waiting until a page is requested before moving it into main memory (i.e. no lookahead) is known as demand paging. For a very lucid and detailed description of virtual memory techniques see Denning, 1970.

1.2 Page Replacement Algorithms.

The choice of which page to displace from real storage when a page fault occurs is made by a page replacement algorithm. Several algorithms exist and four common ones are considered here.

They are:-

(a) Least Recently Used (L.R.U.). A linked list is maintained with one entry for each page in real storage.

At any given time the list entries are in the order of the last use of the real storage pages, the most recently used page is at the head and the least recently used is at the tail. A page is placed at the head of the list when it is in real storage and referenced, and when it enters real storage, but in the latter case the page at the bottom of the list is selected for displacement.

(b) First In First Out (F.I.F.O.). A list is maintained of the order in which pages entered real storage. When a new page comes in, it is placed at the head of the list and the page at the bottom of the list is displaced.

(c) Random selection (RAND). When a new page enters storage, a page already in real storage is selected at random and displaced.

It is possible that one of the pages in real storage is no longer needed or is not required for a 'considerable' time. Clearly such a page is a good candidate for displacement. However if a page is selected to be removed from real storage and is then immediately required again, a second page fault will result. With hindsight i.e. observation of execution after the page fault was serviced, it is possible to determine which page should have been displaced. This leads to the notion of the optimum replacement algorithm, defined by Belady, 1966:-

(d) Minimum Algorithm (MIN). This algorithm is for theoretical use only and is intended to operate on address traces produced by program execution. Its major use is to determine how close practical algorithms come to the optimal value. Its basic principle is to defer the decision about which page to displace when a page fault occurs, until there is sufficient information available about the future use of pages to make an optimum choice. For a detailed description see Belady, 1966 but an example should clarify the principle.

Suppose a virtual address space consists of five pages, labelled 1, 2, .., 5 for identification, and there is room for three pages in real storage. Consider the following page reference string:-

2, 1, 2, 3, 4, 2, 1, 4, 5, ...

The reference to page 4 will cause a page fault. L.R.U. will select page 1 to displace and F.I.F.O. will select page 2. Both of these are referenced immediately after page 4 and so the optimum choice for displacement is page 3, which is the one that MIN would select.

The working set model of program behaviour (Denning, 1968) provides a very elegant solution to the problem of storage allocation in a paged multiprogramming environment. It can also be used to make a decision about which page to replace when a page fault occurs.

Denning points out that it is extremely closely related to L.R.U. page replacement (Denning, 1972 (a); Denning & Schwartz 1972).

In fact if the working set model is used to set the number of real page frames that a program may use to say k (t) (k is a function of time), then working set replacement is just L.R.U. replacement within these k page frames. Working set replacement will not be considered separately here.

1.3 The Costs of Paging.

Unfortunately, servicing a page fault is extremely expensive in terms of system resources. Some of the costs are:-

(i) the execution of several thousand instructions (typically 5000 on a 360/67) by the program which controls the secondary storage device and in handling the interrupt which results from the page fault.

(ii) waiting for the secondary storage device to find the required page (e.g. rotation of a drum). This increases the program's elapsed time.

(iii) the program's pages which are already in real storage are locked there while the page fault is serviced, effectively reducing the amount of real storage available for other programs (see Randell & Kuehner, 1968 (a)).

(iv) the channel connecting the secondary storage device to main storage has to execute its program (see Denning, 1968).

(v) on I.B.M.'s System 360 Model 67 and System 370 there is usually a single Bus Control Unit handling main storage requests.

I/O channels have priority over the C.P.U. for its use and the very high data transfer rates which are possible with Selector and Block Multiplexor channels can cause serious interference to C.P.U. operation (see Gibson, 1966; Lauer, 1967; Nielson, 1967). This effect is still present, though reduced, when a high speed buffer memory (cache) is used (see for example I.B.M. publication GA22-7012-0).

The number of page faults being serviced can become excessive and then their high cost shows up clearly. Usually this happens when there is too high a level of multiprogramming and an over-committment of main memory. The result is a total breakdown in system performance and C.P.U. utilization drops considerably. This condition is known as thrashing and designers of operating systems have devoted considerable effort to its avoidance. However, even in a system which is not thrashing, the extreme cost of page faults as detailed above means that performance improvements will result if they can be avoided.

It is natural to ask why some kind of special hardware is not provided to reduce these costs. The Scientific Computer Corporation realize the need and include a microprogrammed miniprocessor within the main C.P.U. of the S.C.C. 6700 (Watson, 1970). This miniprocessor is dedicated to handling some of the housekeeping associated with a page fault in parallel with normal C.P.U. operation. In addition, the S.C.C. 6700 uses the "Berkeley Memory System" (Watson, 1970). This employs a main memory access mechanism in which the priority of pending storage requests can vary with time in order to enforce limited co-operation between devices using main storage. In this way interference is reduced.

Even if ideas such as these are used, paging still has a positive cost and page faults should be avoided if possible. Unfortunately, most manufacturers of presently available paged memory systems do not have the foresight of S.C.C. and provide no hardware assistance.

1.4 The Problem Program Approach.

The MIN page replacement algorithm is often used as a standard by which other replacement algorithms are judged. However it must be remembered that MIN only achieves the minimum number of page faults for a given page reference string. If a programmer is prepared to redesign his algorithm the performance achieved by MIN and the original algorithm can often be beaten (see Weinberg, 1972). There are only a few instances where this has been done, but the improvements are dramatic.

For example, a comprehensive study of matrix storage and matrix operations in a paged memory is presented by McKellar & Coffman, 1969. They show that the normal technique of row major (and equivalently column major) matrix storage is wholly inappropriate for a paged memory. Most matrix operations involve both row and column traversals and clearly a column traversal with row major storage will produce a page fault for every few elements referenced. It is shown that a much more efficient storage technique is to divide the matrix into suitable sized submatrices and store one submatrix per page. Algorithms are presented for the various matrix operations which, assuming this storage method, achieve reductions of three orders of magnitude in the paging produced by data references, compared with what would be produced by a naive approach using row or column major storage.

Several different approaches to the problem of sorting a large file located in a paged memory are discussed by Brawn, Gustavson and Mankin, 1970. Efficient Lisp implementations in a paged environment are considered by Bobrow & Murphy, 1967 and Cohen, 1967.

The redesigning process must take account of the fact that if extra computation is necessary to avoid a single page fault it may still be worthwhile. Clearly the amount of computation which it is worth doing will depend on the particular machine involved and will be less for the S.C.C. 6700 than say I.B.M.'s System 370. This implies the need to question conventional standards of efficiency. For example, sort algorithms are frequently compared in terms of the average number of comparisons required to sort a given number of records. This is a reasonable approach when the algorithm is to be executed from a conventional memory. In this case a comparision is a relatively lengthy operation and one would obviously want their number minimized. However a comparision is trivial when one considers the amount of work involved in a single page transfer. Indeed a thousand comparisions may be considered trivial.

The notion of a program's working set (Denning, 1968) is well established. The working set of a program involved in extensive data manipulation may be considered to consist of two parts, namely the working set of instructions and the working set of data.

The first of these is often relatively unimportant and comparatively small. For example, most sort algorithms can usually be programmed in just a few hundred machine level instructions which occupy less than a page. This will be the total size of the area used to contain instructions during execution of the entire sort operation.

Far more important and frequently much larger is the working set of data (W.S.D.). It can be loosely defined as that set of data pages which must be located in real storage for a program to execute for a "reasonable" length of time without causing a page fault. In fact a formal definition is not necessary because with most algorithms, the size and constituent pages of the working set of data are obvious. Several examples of this idea are included later in this thesis.

Given this concept, it is clear that the results presented by other authors, which are mentioned above, are in fact changes in the design of algorithms aimed at reducing the size of the W.S.D. so that efficient operation will be maintained with less real storage available. In addition, where possible it is important to use all of the data contained within the W.S.D. while it is located in real storage so that it need not re-enter real storage at a later time.

1.5 The Objectives Of This Dissertation

The primary aims of this thesis are to examine various commonly encountered algorithms which might be used on large data sets in a virtual memory, propose changes to improve performance and to take advantage, where possible, of the facilities provided by a paged memory. For the most part the suggested modifications are relatively minor but the improvement produced is usually substantial. An attempt is made to analyse these effects in terms of each of the replacement algorithms defined above.

The process of finding the i^{th} largest element in an unsorted data set i.e. element selection, is examined in Chapter 2.

This is really just a rather specialized search operation. One of the few algorithms available is FIND (Hoare, 1961) and it will be shown that in many ways that algorithm does not perform as well in a paged memory as an alternative which is defined in Chapter 2.

Another search algorithm, the extremely important and widely used Scatter Storage technique, is discussed in Chapter 3. Classical implementations and analyses are shown to be inappropriate.

More general searching methods are considered in Chapter 4 and a searching scheme is proposed which is optimum in many ways when used in a paged memory.

In Chapter 5 a sorting algorithm is proposed which makes use of techniques proposed by other authors (Brawn et al, 1970) for reducing the incidence of page faults. More importantly it takes advantage of the large virtual address space in a limited way to considerably increase sorting speed with only a very small increase in paging activity.

1.6 Paging Analysis Model

The model which will be assumed in the sections on paging analysis is that a program in execution has a certain number of real page frames available, and that the page replacement algorithm operates to keep this quantity fixed. This is somewhat reminiscent of the early Atlas scheme (Kilburn et al, 1962). The discrepency between this model and a real life situation varies considerably with different schedulers and operating systems. Some schedulers tend to fix the amount of real memory that a program can use depending on the amount of system activity. The problem of a high number of page demands at the beginning of a time slice is frequently circumvented by loading the set of pages that the

program was using when previously halted, before allowing it to begin execution once again (e.g. TSS, BCC1). If these techniques are used, the model provides a reasonable approximation and has been used by several authors (Belady 1966, Brawn & Gustavson 1968, Coffman & Varian 1968, Fine et al 1966, Joseph 1970, McKellar & Coffman 1969).

A recently announced operating system known as VM/370 has been produced by I.B.M. for running on their System 370 range of computers. It is a modified form of the CP67/CMS system developed at the I.B.M. Laboratories in Cambridge, Mass. for the 360/67, and provides terminal users with Virtual Machines (see I.B.M. GC20-1801-0,1972). It incorporates a VM/370 operator command "SET RESERVED xxx n" which allows a fixed number (n) of real storage page frames to be used exclusively by a particular virtual machine (xxx) and that virtual machine is then limited to n page frames even if more are available. Clearly the behaviour of programs being executed by such a virtual machine can be studied almost exactly by the use of the proposed model.

Extensive use is made of the ceiling and floor functions. For non-integral x the notations and definitions are as follows:-

Ceiling : $\lceil x \rceil$ = smallest integer greater than x.

Floor : $\lfloor x \rfloor$ = largest integer less than x.

and clearly:- $\lceil x \rceil = \lfloor x \rfloor + 1$.

For integral x:- $\lceil x \rceil = \lfloor x \rfloor = x$.

The Harmonic numbers are also used considerably. The n^{th} Harmonic number where $n > 0$, is denoted H_n and is defined:-

$$H_n = \sum_{i=1}^{n} \frac{1}{i}$$

Chapter 2.

ELEMENT SELECTION

Finding the i^{th} largest or smallest element of a data set of n items is a common problem. For example the cases i = 1 (extremes), i = n/2 (median) and certain percentiles. If the data set is ordered the problem is trivial, but if it is unordered a considerable amount of data inspection is necessary. In general the whole data set will have to be scanned at least once, and if it occupies M pages this gives a lower bound of M page faults for finding an element of specific rank.

2.1 The FIND Algorithm.

2.1.1 Definition Of The Basic Algorithm.

A frequently used algorithm on non-paged machines if FIND (Hoare, 1961). To find the i^{th} largest element two pointers are established, one at each end of the data set, and a record is chosen at random from the data set. One of the pointers is moved towards the centre until a record is found that is less than the selected record. The other pointer is then moved until a record is found which is greater than the selected record. The two records to which the pointers refer are then swapped. This process is repeated until the pointers meet. Thus the data set is partitioned into two pieces since all records on one side of the pointers are less than the selected record, and all those on the other side are greater. This partitioning process is then carried out again on that section of the data containing the desired i^{th} largest

element, and repeated until the desired element is found.

2.1.2 A Modification To The Basic Algorithm.

If the range of the data and its distribution are known, an alternative approach which does not appear in the original definition of FIND, is to use a constant for comparison during each partitioning process or step. Clearly if this is done, the constant for each step should be selected so as to minimize the expected number of records involved in the subsequent step.

2.1.3 The Expected Number Of Records Involved In The Second Step.

Suppose the i^{th} largest record is required from a data set which is a sample of size n from a distribution with density function f, distribution function F, and range $[a,b]$. Define E_j to be the expected number of records involved in the j^{th} step, then $E_1 = n$ = the total number of records in the file.

Consider the first partitioning process and suppose a quantity x is used to make comparisons. x may be a constant or set equal to a record in the file, depending on which version of the algorithm is in use. The expected number of records involved in the second partitioning process depends on the values of x, i, and the joint pointer location at the end of the first step. If at the end of the first step the pointers refer to the record in the m^{th} location ($1 \leq m \leq n$) then:-

$$\text{number of records which will be used in second step} = \begin{cases} m - 1 & \text{if } m > i \\ n - m & \text{if } m < i \end{cases}$$

$$= \quad (m-1)\, I_{m>i}(m) \quad + \quad (n-m)\, I_{m<i}(m)$$

where I is the indicator function i.e. $I_{m<i}(m) = 1$ if $m<i$ and 0 otherwise. Clearly if $m = i$ the algorithm can terminate.

2.1.3.1 The Basic Algorithm.

The number of records less than x is binomially distrbuted and so:-

$$E_2 = E(\text{number of records which will be used in second step.} \mid i, x)$$

$$= \sum_{m=1}^{n} \{(m-1)I_{m>i}(m) + (n-m)I_{m<i}(m)\}\{F(x)\}^m\{1-F(x)\}^{n-m}\, {}^nC_m$$

For the basic algorithm the distribution of x is known since x is selected at random from the file, and so the dependence of E_2 on x can be removed, giving:-

$$E_2 = E(\text{number of records which will be used in second step.} \mid i)$$

$$= \int_a^b \sum_{m=i}^{n} \{(m-1)I_{m>i}(m) + (n-m)I_{m<i}(m)\}\{F(x)\}^m\{1-F(x)\}^{n-m}\, {}^nC_m f(x)\, dx$$

Let $y = F(x)$ then $dy/dx = f(x)$ and the expression for E_2 can be simplified:-

$$E_2 = \int_0^1 \sum_{m=i+1}^{n} \{(m-1)y^m(1-y)^{n-m}\, {}^nC_m\} + \sum_{m=1}^{i-1} \{(n-m)y^m(1-y)^{n-m}\, {}^nC_m\}\, dx$$

$$= \sum_{m=i+1}^{n} \{(m-1)\, {}^nC_m \int_0^1 y^m(1-y)^{n-m} dy\} + \sum_{m=1}^{i-1} \{(n-m)\, {}^nC_m \int_0^1 y^m(1-y)^{n-m}\, dy\}$$

Now $\int_0^1 y^m(1-y)^{n-m}\, dy = B(m,n) = \dfrac{\Gamma(m)\,\Gamma(n)}{\Gamma(m+n)} = \dfrac{(m-1)!\,(n-1)!}{(m+n-1)!}$

where $\Gamma(x)$ is the Gamma function.

$$\therefore \quad E_2 = \sum_{m=i+1}^{n}(m-1)/(n+1) + \sum_{m=1}^{i-1}(n-m)/(n+1)$$

$$= \frac{1}{2(n+1)}\{n(n-3) + 2i(n+1-i)\}$$

The minimum value of E_2 is $n(n-1)/2(n+1) \simeq n/2$.

The maximum value of E_2 occurs at the median and is:-

$$\frac{n(n-3)}{2(n+1)} + \frac{(n+1)}{4}$$

$$\simeq 3n/4$$

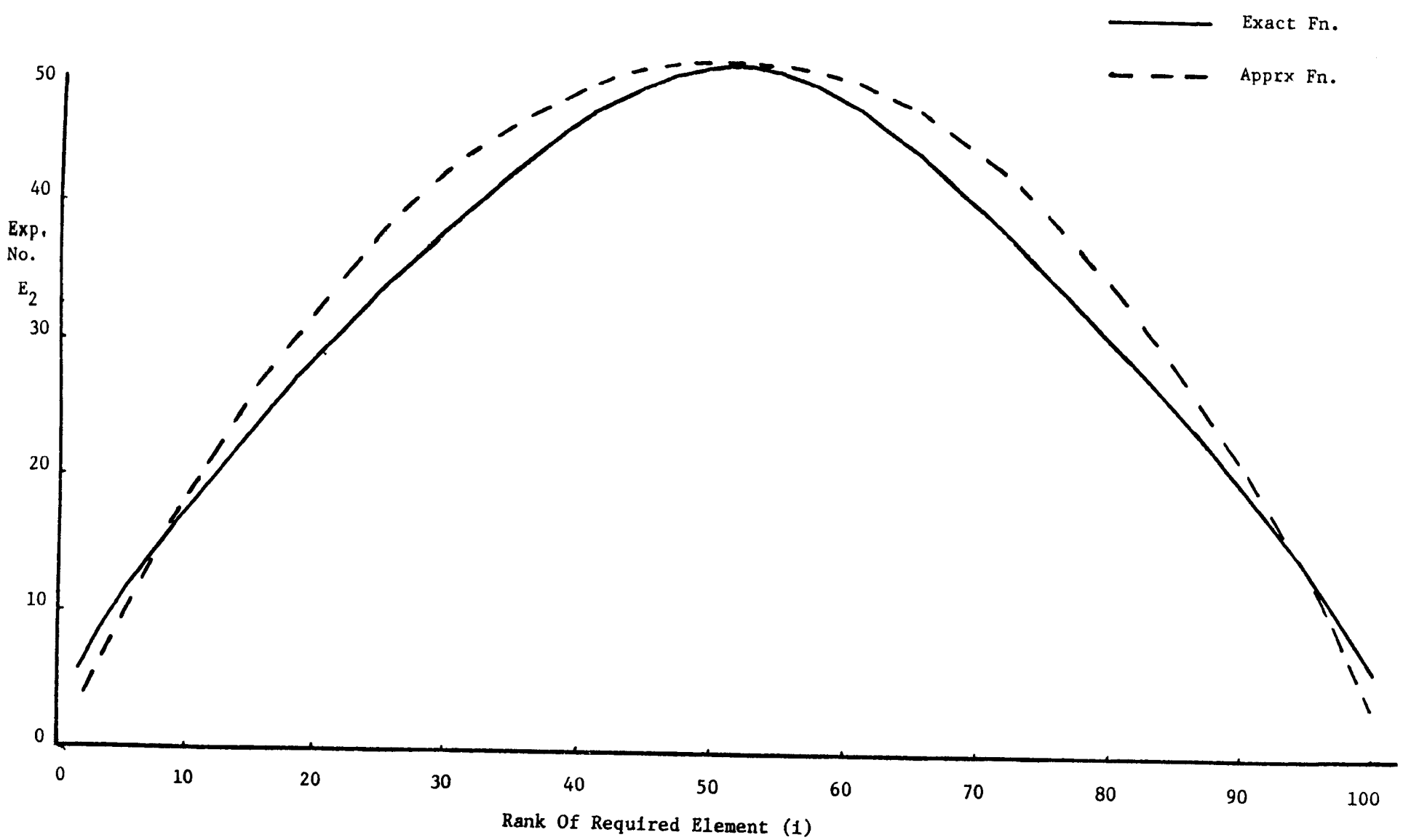
2.1.3.2 The Modified Algorithm.

For the modified form of FIND the expression for E_2 which is given at the beginning of section 2.1.3.1 still applies:-

$$E_2 = \sum_{m=1}^{n}\{(m-1)I_{m>i}(m) + (n-m)I_{m<i}(m)\}\{F(x)\}^m\{1-F(x)\}^{n-m}\,{}^nC_m$$

If a, b, f and F are known i.e. details of the distribution from which the sample comes, then x can be chosen so as to minimize E_2. As noted previously this is the way in which x should be selected for each step. No closed form expression for that value of x which minimizes E_2 in the general case, or the corresponding value of E_2 has been found. However in the special case where the data is uniformly distributed on $[0, 1]$, the minimum value of E_2 for each i has been evaluated numerically for a file size of one hundred records i.e. n = 100. These values are shown in graph 2.1.

G R A P H 2.1


Exact Fn.
Apprx Fn.
50
40
Exp.
No.
E_2
30
20
10
0
0
10
20
30
40
50
60
70
80
90
100
Rank Of Required Element (i)

2.1.4 The Expected Number Of Records Involved In The Third and Subsequent Steps.

During the second step if $i < m$ the algorithm is in fact required to find the i^{th} largest record out of m, and if $i > m$ the $i - m^{th}$ largest out of $n - m$. This means that several additional random variables are required in consideration of the number of records involved in the third and subsequent steps.

Define:-

i_j = rank of the desired record during j^{th} step.

n_j = number of records in the set known to contain the desired record during j^{th} step.

x_j = quantity used for comparison during j^{th} step.

m_j = joint pointer location at the end of j^{th} step. $1 \leq m_j \leq n_j$.

There are two relationships of interest:-

$$i_{j+1} = i_j I_{i_j<m_j}(m_j) + (i_j-m_j) I_{i_j>m_j}(m_j)$$

$$n_{j+1} = m_j I_{i_j<m_j}(m_j) + (n_j-m_j) I_{i_j>m_j}(m_j)$$

The quantities E_j are required because they are directly related to the expected value of the amount of paging which will occur.

Ideally, analytic solutions to these recurrence relations should be found, giving the E_j explicitely, but this is not practical. As noted in section 2.1.3.1, it was not possible even to obtain a general expression for E_2 when using the modified algorithm, and E_2 is conditional on fewer random variables than the other E_j.

In the paging analysis which follows some simplifying approximations will be made:-

(a) At the beginning of the j^{th} step, n_j records remain in the set being considered. n_j will be replaced by $E(n_j)$.

(b) For the basic algorithm, the relationship between E_2 and n (recall $n = E_1$) will be assumed to hold for all E_j giving:-

$$E_j \simeq \frac{1}{2(E_{j-1}+1)} \{E_{j-1}(E_{j-1}-3) + 2i_{j-1}(E_{j-1}-i_{j-1}+1)\}$$

(c) For the modified algorithm E_2 has been found numerically for one case (figure 2.1 shows E_2 as a function of i). Examination of the shape of the curve indicates that a parabola might be used as an approximation. Thus E_j will be approximated by:-

$$E_j \simeq \{E_{j-1} + 1 - i_{j-1}\}.2i_{j-1}/(E_{j-1} + 1)$$

For the case j = 2, this approximation to E_2 is also shown in figure 2.1.

Even these approximate recurrence relations for the E_j are non-linear making solution difficult. However they do allow approximate bounds to be placed on the values of E_j which in turn allow bounds to be placed on the amount of paging to be expected.

This is adequate for comparison purposes with a second algorithm which is defined in section 2.2.

2.1.5 Paging Analysis.

Since the number of records per page is a constant, the expected number of pages of the file which have to be examined during each step is directly related to the expected number of records which have to be examined in that step i.e. E_j. If it is assumed that the two pages containing the records to which the pointers refer remain in real storage as the pointers are updated, it is possible to estimate the number of page faults needed to locate one record.

Page faults occur during the partitioning process because one of the pointers, say pointer A, is being updated and crosses a page boundary. Clearly it is preferable to replace the page which pointer A has just left rather than the page being referenced by the other pointer, pointer B. This is an example of two important concepts. Firstly, the working set of data (W.S.D.) in this case consists of two pages since clearly execution can proceed relatively efficiently with just two data pages. Secondly, as a pointer crosses a page boundary a "page change" is required. A page change is the deletion from the W.S.D. of one page which is no longer required and the addition of another. Thus the size of the working set of data, denoted $|W.S.D.|$, remains fixed but one of the constituent pages changes. If the entire W.S.D. is located in real storage prior to a page change, several page faults may be necessary before the new W.S.D. is located wholly in real storage. The exact number depends on the page replacement algorithm but the MIN algorithm, by definition achieves the ideal situation of one page change causing one page fault.

2.1.5.1 L.R.U. Page Replacement.

Consider the first partitioning process and assume that c real page frames are available with c > |W.S.D.|. Initially setting up the pointers will cause two pages to be loaded into real storage and thereafter $\lceil E_1/b \rceil - 2$ page changes will be necessary before the pointers meet, where b is the number of records per page. c - 2 of these page changes will occur before real storage is filled, and for the remainder, it is reasonable to assume that pages from the W.S.D. will be the most recently referenced and will not be selected for displacement. Thus each page change will only induce one page fault and this gives a total of $\lceil E_1/b \rceil$ page faults for the first step.

Each of the second and subsequent steps will begin with one of their required pages already in real storage. Thus the expected number of page changes required by the j^{th} step (j > 1) will be $\lceil E_j/b \rceil - 1$ which will induce $\lceil E_j/b \rceil - 1$ page faults. If a total of t steps are needed before the size of the region in which the desired record is known to lie drops to less than one page, then the expected value of the total number of page faults induced will be:-

$$\sum_{j=1}^{t} \{\lceil E_j/b \rceil - 1\} + 1$$

If c = |W.S.D.| (recall |W.S.D.| = 2) then as a pointer is updated and crosses a page boundary, the page which it leaves is the page no longer required but is also most recently referenced. Thus L.R.U. will always select incorrectly and each page change will induce two page faults.

It follows that the total number of page faults which will occur in this case will be:-

$$
\begin{aligned}
& 2 && \text{Initially loading real storage.} \\
+ \quad & \{\lceil E_1/b \rceil - 2\}2 && \text{First step.} \\
+ \quad & \sum_{j=2}^{t} \{\lceil E_j/b \rceil - 1\}2 && \text{Other steps.} \\
= \quad & \sum_{j=1}^{t} \{\lceil E_j/b \rceil - 1\}2
\end{aligned}
$$

The values of the E_j $(1 \leqslant j \leqslant t)$ depend on which version of the algorithm is in use. In both cases, even with simplifying approximations, it was shown that E_j is the subject of a non-linear recurrence relation which also involves i_j. However approximate bounds on the values of the E_j can be found and these are adequate.

In the case of the basic algorithm:-

$$
\begin{aligned}
\text{Min}(E_j) &\simeq E_{j-1}/2 \\
\therefore \quad \text{Min}(E_j) &\simeq E_1/2^{j-1}
\end{aligned}
$$

$$
\begin{aligned}
\text{Max}(E_j) &\simeq 3E_{j-1}/4 \\
\therefore \quad \text{Max}(E_j) &\simeq (3/4)^{j-1}E_1
\end{aligned}
$$

and for the modified algorithm, assuming the approximating function for E_j is reasonable:-

$$\text{Min}(E_j) \simeq 0 \quad \text{for } j > 1. \quad E_1 = n = \text{const.}$$

$$\text{Max}(E_j) \simeq E_{j-1}/2$$

$$\therefore \quad \text{Max}(E_j) \simeq E_1/2^{j-1}$$

These can be used to give approximate bounds on the number of page faults involved in each case.

For the basic algorithm:-

$$\text{Min. page faults} \simeq \lceil E_1/b \rceil + \lceil E_1/2b \rceil + \ldots + \lceil E_1/2^{t-1}b \rceil$$
$$\simeq \lceil 2E_1/b \rceil$$

$$\text{Max. page faults} \simeq \lceil E_1/b \rceil + \lceil 3E_1/4b \rceil + \ldots + \lceil (3/4)^{t-1}E_1/b \rceil$$
$$\simeq \lceil 4E_1/b \rceil$$

For the modified algorithm:-

$$\text{Min. page faults} \simeq \lceil E_1/b \rceil = \lceil n/b \rceil$$

$$\text{Max. page faults} \simeq \lceil E_1/b \rceil + \lceil E_1/2b \rceil + \ldots + \lceil E_1/2^{t-1}b \rceil$$
$$\simeq \lceil 2E_1/b \rceil$$

2.1.5.2 F.I.F.O. Page Replacement.

In general F.I.F.O. replacement does not lend itself to analytic study. The primary reason is that the page which is displaced when a page fault occurs is not determined by the events immediately prior to that page fault but by the events occurring when the page in question entered storage.

If the changing of the contents of real memory as page faults occur is regarded as a Markov process then the new state following a page fault is determined by the set of page references which occurred a considerable period of time before, as well as the page which has just been referenced. This makes the transition matrix extremely complex in all but the simplest case.

The FIND algorithm is sufficiently simple that F.I.F.O. replacement can be analysed but fairly major assumptions have to be made (see below). Several algorithms are discussed in this dissertation for which even approximate analysis with F.I.F.O. replacement has not been possible.

The number of page changes which occur when using the FIND algorithm with F.I.F.O. replacement will be the same as with L.R.U. The assumptions which have to be made are:-

(i) In the case $c > |W.S.D.|$ pages of the W.S.D. are not the oldest in real storage.

(ii) In the case $c = |W.S.D.|$, as pointer A is being updated and crosses a page boundary, the page which pointer A leaves has been in real storage longer than the page referenced by pointer B. This is reasonable since there has been sufficient time for pointer A to traverse a complete page.

If these assumptions are made then for all values of c, F.I.F.O. always correctly selects the unwanted page and only one page fault will occur for each page change. This is the same performance which was found with L.R.U. replacement when $c > |W.S.D.|$ and the results obtained in that case apply here.

2.1.5.3 Random Page Replacement.

The number of page changes required with Random page replacement is the same as with the other replacement algorithms. When a page change becomes necessary only one of the pages in real storage is still in use. Thus the probability of selecting an unwanted page for displacement is (c-1)/c and the probability of selecting the wanted page is 1/c. If the wanted page is displaced a second page fault will occur when it is next referenced. Clearly these events can be repeated and a large number of page faults could be induced before both pages of the W.S.D. are located in real storage. The number of page faults has a geometric distribution with parameter (c-1)/c and expected value c/(c-1).

The expected value of the total number of page faults is therefore:-

$$
\begin{aligned}
& c && \text{Initially loading storage.} \\
& + && \\
& \{\lceil E_1/b \rceil - c\}.c/c\text{-}1 && \text{Completing first step.} \\
& + && \\
& \{\lceil E_j/b \rceil - 1\}.c/c\text{-}1 && \text{Second and subsequent steps.} \\
= \quad & \{\lceil E_j/b \rceil - 1\}.c/c\text{-}1 &&
\end{aligned}
$$

As with L.R.U. and F.I.F.O., approximate bounds can be found for this expression by using the approximate bounds for E_j.

2.2 The LOCATE Algorithm.

2.2.1 Definition.

Although FIND is reasonably efficient and has many excellent properties, it was not designed to operate within a paged memory. An alternative algorithm to be known as LOCATE is suggested here and it will be shown that it is preferable to FIND in many respects.

The basic version of LOCATE is defined as follows:-

(i) the range of the keys is divided into a set of intervals.

(ii) a set of counters is established, one for each interval, and initialized to zero.

(iii) the whole data set is scanned linearly and for each record the counter corresponding to the interval in which the key lies is incremented by one.

(iv) the interval which contains the desired record is determined from the values of the counters.

(v) a second scan of the file is made and each record lying in the same interval as the desired record is stored in a work area of the program.

(vi) the required record is selected from the records in the work area.

It is easily seen that at most 2M page faults are necessary, and the number produced is independent of the replacement policy since the size of the working set of data is only one.

Notice that LOCATE requires the range of the keys to be known. This is not a serious disadvantage since in most cases experimental data comes from sources where the bounds of the sample values are known.

2.2.2 Analysis Of The Second Scan.

In practice, the second scan through the data set can be terminated as soon as all the records in the interval of interest have been found. The number of records in the required interval and their distribution through the file will determine the number of page faults induced during the second scan.

Suppose there are n records occupying M pages in the file and q counters are to be used. Suppose the counter corresponding to the interval containing the desired record has a value of k at the beginning of the second scan. k is a random variable whose expected value depends on the distribution of the keys in question. If they are uniformly distributed then k has expected value n/q. In general, if a random variable X has distribution function F then the transformed variable F(X) has a uniform distribution. If F is known, this property can be used to transform the data and ensure equal expected values for the counters. In the case where the distribution function is not known, the uniform distribution can be assumed and the counters then provide an estimate of the density function of the distribution from which the data comes.

Assume that the data (or the transformed data) is uniformly distributed and define S as the length of a string of pages which do not contain any of the records which contribute to the counter value of interest, but which would be the last pages moved into main storage if the second scan were to examine the whole data set, then:-

$$E(\text{number of page faults during second scan} \mid k) = M - E(S)$$

Now $pr(S = 0) = pr(\text{last page examined contains a record})$

$$= 1 - (1 - 1/M)^k$$

$pr(S = 1) = pr(\text{penultimate page examined contains a record but last does not})$

$$= (1 - 1/M)^k(1 - (1 - 1/(M - 1))^k)$$

similarly $pr(S = j) = (1 - 1/M)^k(1 - 1/(M - 1))^k \ldots (1 - (1 - 1/(M - j))^k)$

$$= (1 - j/M)^k - (1 - (j + 1)/M)^k$$

$$\therefore \quad E(S|k) = \sum_{j=0}^{M-1} j\{(1 - j/M)^k - (1 - (j + 1)/M)^k\}$$

$$= (1/M)^k \sum_{j=0}^{M-1} j^k$$

$\therefore$ E(number of page faults during second scan | k)

$$= M - (1/M)^k \sum_{j=0}^{M-1} j^k$$

This function approaches M very rapidly as k increases but it is not impractical to keep k small by using a large number of counters. For example, a file of 5000 records occupying 20 pages yields only 5 as the expected value of k if 1000 counters are used. As only one byte need be used for each counter this requires a data area of only a quarter of one page on a machine like I.B.M.'s 360/67 which has a page size of 4096 bytes. The expected number of page faults for the second scan assuming k is 5 is then 17.

2.2.3 A Simple Modification.

A modification can be incorporated which makes the second scan unnecessary in certain cases. Suppose those records in the interval where the desired record is expected to lie are stored in a program work area during the first pass. If indeed it is found at the end of the first pass that the required record is located in the work area then the second scan is unnecessary. This modification is only worthwhile if the probability of success is reasonably high.

In general, the distribution of the i^{th} order statistic, x_i, for a sample of size n from a distribution F is given by:-

$$dG_i = \frac{\{F(x_i)\}^{i-1}\{1 - F(x_i)\}^{n-i}dF(x_i)}{B(i,n-i+1)}$$

where B(m,n) is the Beta function. In the case of a sample from the uniform distribution this becomes the Beta distribution.

The probability that the i^{th} largest record is located in the j^{th} interval is given by:-

$$P_{ij} = \int_{(j-1)/q}^{j/q} \frac{x^{i-1}(1 - x)^{n-i}}{B(i,n-i+1)}\, dx$$

where q is the number of intervals used, and assuming the data is uniformly distributed on $[0,1]$. The i^{th} largest record is expected to lie in interval number $\lceil iq/n \rceil$, and so the probability that the desired record is found is P_{ij} evaluated with $j = \lceil iq/n \rceil$.

This integral is the difference between two incomplete Beta functions. Extensive tables of the incomplete Beta function have been published but only for parameter values which are considerably less than those involved

here. Various methods of evaluation were considered but the use of a Normal approximation was selected as the most suitable. With the high values of the parameters involved the distribution of most of the order statistics can be assumed to be almost Normal, the exceptions being the x_i with extreme values of i, e.g. x_1. This approximation was made for various parameter values and the approximate value of the integral found to vary considerably. For example, searching for the 1200^{th} of 5000 records occupying 20 pages and using 1000 counters ($q = 1000$) the corresponding limits of the standard Normal integral are 0.00795 and -0.15766 giving a probability of 0.066. However, searching for the 400^{th} record in the same file and still using 1000 counters, the corresponding limits are 0.00417 and -0.25655 giving a probability of 0.103.

This modification seems worthwhile since it costs very little but the probability of success depends very much on the individual case.

2.2.4 A Major Modification.

If it is assumed that more than one real page frame is available for use by the LOCATE algorithm, the previously described modification can be extended. Instead of merely keeping copies of the small number of records lying in the interval which is expected to contain the desired record, copies of those records lying over a much wider range can be kept in the extra page. This could be done anyway since the virtual address space is not a limitation. The important aspect is that if two real page frames are available, this extra page plus at least one page of the file can reside in storage concurrently. In effect, the working set of data increases by one.

If the required record is found to lie in this extra page, the second scan over the data will not be necessary. In many cases, one page represents a large proportion of the file and hopefully the probability that the second scan can be omitted, or rather restricted to this one page, will be high.

Once again the Normal approximation provides a convenient way to determine the effect of the process. More than 99% of the probability of the Normal distribution lies within three standard deviations of the mean. Thus in the case of the order statistics being considered, it can be assumed that:-

$$\text{pr}(i^{th} \text{ largest record lies in the interval } \mu \pm 3\sigma) \simeq 0.99$$

where $\mu = i/(n+1)$ and $\sigma^2 = i(n+1-i)/(n+1)^2(n+2)$ are the mean and variance respectively of the i^{th} order statistic, the distribution of which was given above.

Using the previous example of a file of 5000 records and searching for the 1200^{th} record this becomes:-

$$\text{pr}(\text{required record lies in the interval } 0.24 \pm 0.018) \simeq 0.99$$

Thus if all the records from the sample which lie in this range can be copied into the extra data page the second scan can almost certainly be limited to this extra page. In the example given the expected number of records in this range is 180 which will fit into a single page since each page will hold 250 records.

It is important to realize that this modification need only be used during the first execution of LOCATE or when establishing the counters

cannot be achieved as a by-product of another process. Once the counters have been set, a single sequential scan through the data set is all that is required to locate any record.

2.2.5 Paging Analysis.

The basic algorithm will generate M page faults during the first scan, and a maximum of M - 1 during the second since one page of the file will be in real storage as the second scan begins. This figure is independent of the replacement policy.

The simple modification described in section 2.2.3 has the effect of sometimes obviating the need for the second pass and hence the maximum figure of 2M - 1 page faults will be necessary less often.

Paging analysis of the major modification is less straight forward because |W.S.D.| is two. Ideally, the extra data page should remain in real storage throughout processing, while the file pages enter in sequence with each one displacing one of its predecessors. Most replacement algorithms will not operate in this fashion, and most operating systems will not allow a user to "lock" pages in real storage selectively i.e. remove certain pages from the set being considered for paging out. An exception is I.B.M.'s VM/370 which permits the system operator to lock user pages in real storage. For the major modification each replacement policy will be considered separately.

2.2.5.1 The Major Modification Operating With L.R.U. Replacement.

In the case c > |W.S.D.|, both pages in the W.S.D. will have been referenced more recently than the other c - 2 pages in real storage, and so they will not be displaced when a page change is necessary.

The total number of page faults induced will be M to load the file pages plus one to load the extra data page.

When c = |W.S.D.| a page change will cause either the file page or the extra data page to be displaced, the former being the best choice since it is no longer needed. The final record examined will determine whether the file page is least recently used or not, but to guarantee that it always is, it is only necessary to make a 'dummy' reference to the extra data page. In this way a single page fault per page change can be achieved even in the case c = |W.S.D.| = 2.

2.2.5.2 The Major Modification Operating With F.I.F.O. Replacement.

When execution begins the first page to enter storage will be a page of the file and it is reasonable to assume that the second will be the extra data page. This will be followed by other file pages until all the available real storage is full. After this an additional M - (c-1) page changes will be necessary during the first scan. The second, the $c + 2^{nd}$, the $2c + 2^{nd}$, ... of these page changes will cause two page faults since the extra data page will be displaced. Every other page change will only induce one page fault since each will cause a file page which is no longer in use to be displaced. Thus the total will be:-

c	loading real storage.
+	
M - (c-1)	one page fault for every page change.
+	
$1 + \lfloor \frac{M - (c-1)}{c} \rfloor$	one additional page fault each time the extra data page is displaced.

$$= M + 2 + \lfloor \frac{M - c + 1}{c} \rfloor$$

2.2.5.3 The Major Modification Operating With Random Replacement.

Once the available real storage has been loaded a further M - (c-1) page changes are required as noted in the previous section. When a page change is necessary only one of the c pages in real storage is still required i.e. the extra data page. Thus the probability that one page fault will be sufficient is (c-1)/c and in fact the number of page faults necessary to complete a page change has a geometric distribution with parameter (c-1)/c and expected value c/(c-1). Thus the total number of page faults necessary has expected value:-

$$c + (M - (c-1))\frac{c}{c-1}$$

$$= Mc/(c-1)$$

2.3 The Advantages Of LOCATE.

Apart from the advantages already noted, LOCATE is to be preferred for several other reasons.

If the order of the data items is important, for example when run tests are to be applied, FIND is totally inappropriate because the partitioning process necessitates data movement. LOCATE however, operates efficiently with no changes to the data set.

The first scan of the LOCATE algorithm is only necessary for the first record which has to be found. Once the counters have been set, they can be used any number of times and for the second and subsequent searches at most M page faults are necessary.

The FIND algorithm becomes more efficient as execution is repeated because the file is effectively being sorted (i.e.Quicksort). However to avoid repeating the partitioning processes, pointers will have to be maintained indicating the limits of the partitions as they are established. This set of pointers will have to be updated with each search until the file is sorted. This is a great deal less convenient than the counters used by LOCATE. In addition, the evaluation of the counters can be combined very easily with some other process such as determination of the sample mean or even the initial input phase.

The working set of data for the FIND algorithm is two pages, which is reasonable, but the paging characteristics depend on the replacement policy. The basic form of LOCATE however, has a working set of data of just one page, and even when available main storage is severely limited, it is not affected by the page replacement policy.

Chapter 3.

SCATTER STORAGE TABLES

3.1 Introduction

Conventional methods of assessing the efficiency of hashing or scatter storage methods usually rely on comparisons of the average number of probes required. A probe may be defined as the examination of a single record in the file. More than one probe may be necessary in order to locate a specific record because of the possibility of clashes. The average number of probes is kept low by making the hash table for a given number of records, as large as practicable. The more sparsely it is occupied, the lower the average number of probes.

A paged machine which provides a large virtual address space might be thought very suitable because any previous limitation on table size is removed. A closer examination reveals that a large hash table will necessitate considerable paging activity if for each search, every record has the same probability of being accessed. The reason for this is that only part of the table will be located in main memory at any time and if a random pattern of requests is produced, there is a high probability that the data needed will be located on backing storage.

The high cost of paging as detailed in chapter 1 implies that the comparison of hashing techniques used on a paged machine can no longer rely on the number of probes required. Doubling the average number of probes and reducing the number of page faults by 5% may well decrease

a program's execution time.

Only scatter storage tables which are basically static once created are considered here. Tables which vary in size during program execution are as much of a problem in a paged memory as in a normal memory, and will require some form of periodic re-organisation.

3.2 Random Request Pattern

3.2.1 The Basic Technique.

Consider the basic technique of hash coding. In an attempt to keep the average number of probes down, tables are not usually filled to capacity, in fact rarely more than 75%. An average of 1.5 probes per retrieval is an achievable figure for this packing density (see Morris, 1968). The resolution of clashes can be done in several ways but most of them make no deliberate attempt to keep the record which clashes close to its hash location within the table.

Suppose a conventional table contains S records in p pages, is 100 f % full and c main memory pages are available. Let b be the maximum number of records which can be contained in one page. The probability that any record retrieved from this table is in main memory when requested is given by:-

$$\frac{\text{number of records in main memory}}{\text{number of records in file}}$$

$$= \frac{c\ b\ f}{S} = \frac{c\ b\ f}{p\ b\ f} = \frac{c}{p}$$

Thus the number of page faults generated by n requests has a binomial distribution with parameters n and $(1 - \frac{c}{p})$.

The expected number of page faults generated by n requests is therefore $n\ (1 - \frac{c}{p})$, assuming no clashes have to be resolved.

One technique for clash resolution is normally termed random probing. Table positions are examined in a sequence generated by a pseudo-random number generator starting at the hash address. It is usually arranged that the pseudo-random number generator will produce a sequence of displacements within the table such that each table location occurs exactly once. The record is placed in the first free location that is found. An example is shown in figure 3.1. This probably represents a worst case as far as paging is concerned because it is quite likely that references will be made to addresses located on the backing storage.

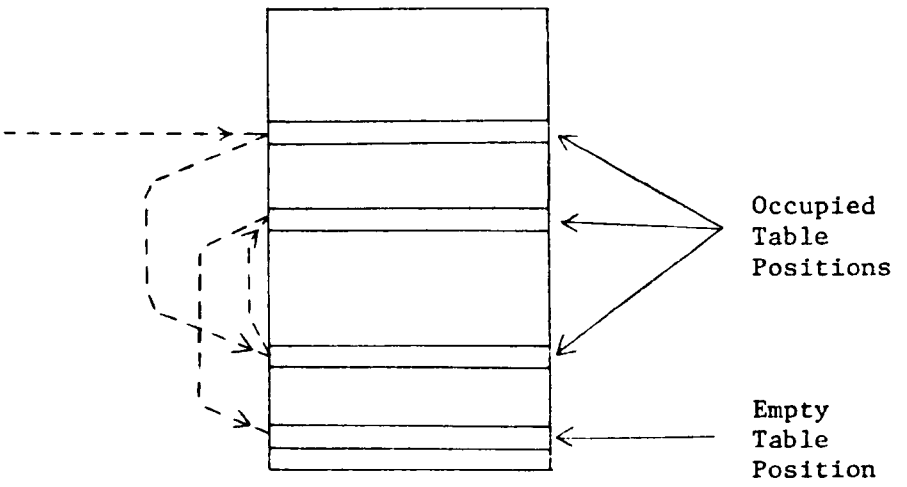


Fig 3.1

An alternative technique is linear scanning. Adjacent table positions are examined, starting at the hash address, until a vacant location is found. This method is illustrated in figure 3.2.

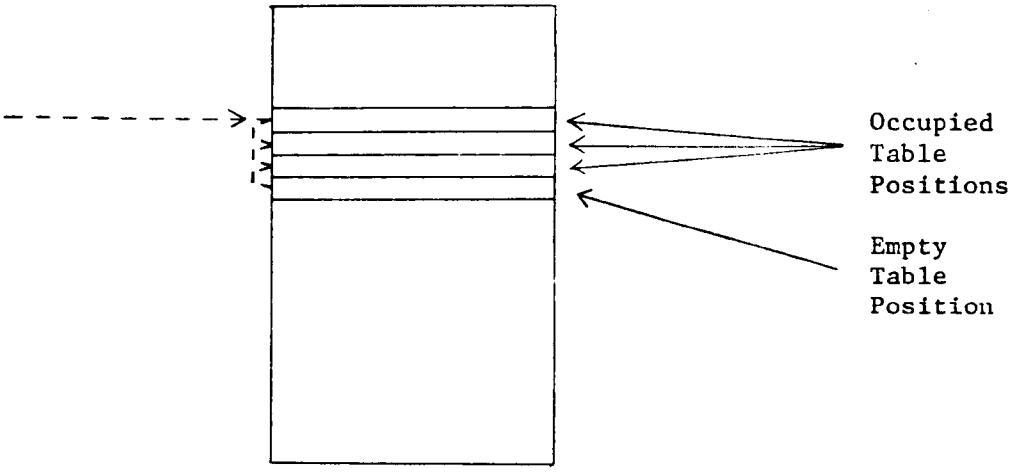


Fig 3.2

It is intuitively obvious that this is far better suited to a paging environment since a page boundary will be crossed fairly infrequently and the clashing record will often be located on the same page as the hash address. Morris, 1968 makes a similar point.

For the purpose of worst case analysis, suppose that random probing is being used. Assuming that for each key, every table location has the same probability of being selected by the hashing function, the expected number of probes required to insert the i^{th} element into a table of size N is:-

$$(1-\frac{i-1}{N}) + 2(\frac{i-1}{N})(1-\frac{i-2}{N-1}) + \ldots\ldots + i(\frac{i-1}{N})(\frac{i-2}{N-1})\ldots(\frac{i-(i-1)}{N-(i-1)+1}).1$$

$$= 1 + \frac{(i-1)}{N} + \frac{(i-1)}{N}\frac{(i-2)}{N-1} + \ldots\ldots + \frac{(i-1)}{N}\frac{(i-2)}{N-1}\ldots(\frac{1}{N-i+2})$$

$$= 1 + \frac{i-1}{N-i+2}$$

Thus the average number of probes per access will be:-

$$\sum_{i=1}^{S} \frac{1}{S} \left(1 + \frac{i-1}{N-i+2}\right)$$

$$= \frac{N+1}{S} \left(H_{N+1} - H_{N+1-S}\right)$$

where H_i is the i^{th} Harmonic number. Each probe has the same probability $(1 - c/p)$ of causing a page fault and so the average number of page faults for n accesses will be:-

$$n.(1 - c/p) \frac{N+1}{S} \left(H_{N+1} - H_{N+1-S}\right)$$

If N+1-S is large, the well known approximation for Harmonic numbers can be used:-

$$H_i \simeq \log_e i + \gamma$$

where γ is Euler's constant. In this case the expected number of page faults is approximately:-

$$n\ (1 - c/p) \frac{N+1}{S} \log_e \frac{1}{1-f}$$

This presupposes that table requests are sufficiently frequent to maintain c pages of main memory and that the initial loading of memory has negligible effect. This result is the same for each of the practical replacement policies being considered because the request sequence is assumed to be random.

3.2.2 Minor Modifications

To reduce the amount of paging the following simple modifications are proposed. Firstly the table size is reduced to the minimum possible i.e. fp pages, and secondly clashes are resolved within the page containing the calculated hash address.

Reducing the table size depends on knowing the file size or having an estimate of it. The problem is identical to that of conventional hash coding where one does not want the table to become more than 75% full. Resolving clashes may employ any of the standard techniques modified to act within one page.

The results of these changes are:-

(i) the reduced number of pages in the file increases the probability of a requested page being in main memory.

(ii) at most one page fault is produced per request.

(iii) a much higher average number of probes will be required.

Clearly the number of page faults still has a binomial distribution but now with parameters n and (1 - c/fp) and no extra page faults are induced in resolving clashes. Thus the expected number of page faults in satisfying n requests is just n.(1 - c/fp). This represents a saving of:-

$$\{n(1 - c/p)\frac{(pb + 1)}{S}\log_e\frac{1}{1-f}\} - n(1 - c/fp)$$

For the typical values b = 100, c = 10, p = 40 and f = 3/4, these modifications produce a saving of 0.725n page faults which is offset by the extra probes which are necessary.

As shown above, the average number of probes per access for random probing is:-

$$\frac{N + 1}{S}(H_{N+1} - H_{N-S+1})$$

Assuming the proposed modifications are used, this expression will apply with N = S = b = number of records per page and this gives:-

$$\frac{b+1}{b}\,(H_{b+1}-1)$$

In the typical case, b = 100 and so the average number of probes will be approximately 5. Thus the extra cost of the proposed modifications is approximately four probes per access. As an example, suppose each requires the execution of ten machine level instructions, then the cost for n requests is 40n instructions. The saving is 0.725n page faults in the typical case considered and since each of these could involve 5000 instructions it can readily be seen that the modifications should prove very worthwhile.

It will not be possible to fill completely all the fp pages of the table if the proposed clash resolution technique is used. As soon as an attempt is made to insert a record into an already filled page, the table itself must be regarded as full. Alternatively, performance could be sacrificed and clashes resolved within one of the remaining partially empty pages with the ensuing possibility of an extra page fault. Suppose this is not done, it is useful to know how much space will be wasted.

Recall that each page holds a maximum of b records and the table size is fp pages. When a page becomes full, the probability that the others contain $i_1,\ldots\ldots,i_{fp-1}$ with $i_j < b \forall j$ is:-

$$P = \frac{(fp)!\;(b-1+\sum_{j=1}^{fp-1} i_j)!}{i_1!\;\ldots\ldots\;i_{fp-1}!\;(b-1)!}\,(1/fp)^{b-1+\sum_{j=1}^{fp-1} i_j}(1/fp)$$

and so the expected value of i_1 say, is:-

$$E = \sum_{i_1=1}^{b-1} i_1 \sum_{i_2=0}^{b-1} \cdots\cdots \sum_{i_{fp-1}=0}^{b-1} P$$

Now

$$\sum_{i_1=0}^{b-1} \cdots\cdots \sum_{i_{fp-1}=0}^{b-1} P = 1$$

since it is a sum of probabilities and by a suitable change of variables, E can be reduced to a similar expression, allowing simplification. However after this transformation the limits involved in the expression for E do not correspond exactly with those in the sum of probabilities and the final general form for E is rather cumbersome in all but the simplest case i.e. fp=2.

Consider this case:-

$$E = \sum_{i_1=1}^{b-1} i_1 \frac{2(b-1+i_1)!}{i_1!\,(b-1)!}\left(\frac{1}{2}\right)^{b-1+i_1}\left(\frac{1}{2}\right)$$

$$= \sum_{i_1=1}^{b-1} \frac{(b-1+i_1)!}{(i_1-1)!\,(b-1)!}\left(\frac{1}{2}\right)^{b-1+i_1}$$

make the transformations $\ell = i_1 - 1$ and $d = b + 1$ then:-

$$E = \sum_{\ell=0}^{d-3} \frac{(d-1+\ell)!}{\ell!\,(d-2)!}\left(\frac{1}{2}\right)^{d-1+\ell}$$

$$= (d-1)\sum_{\ell=0}^{d-3} \frac{(d-1+\ell)!}{\ell!\,(d-1)!}\left(\frac{1}{2}\right)^{d-1+\ell}$$

Now

$$\sum_{i_1=0}^{b-1} \frac{(b-1+i_1)!}{i_1!\,(b-1)!} \left(\frac{1}{2}\right)^{b-1+i_1} = 1$$

(sum of probabilities.)

and so:-

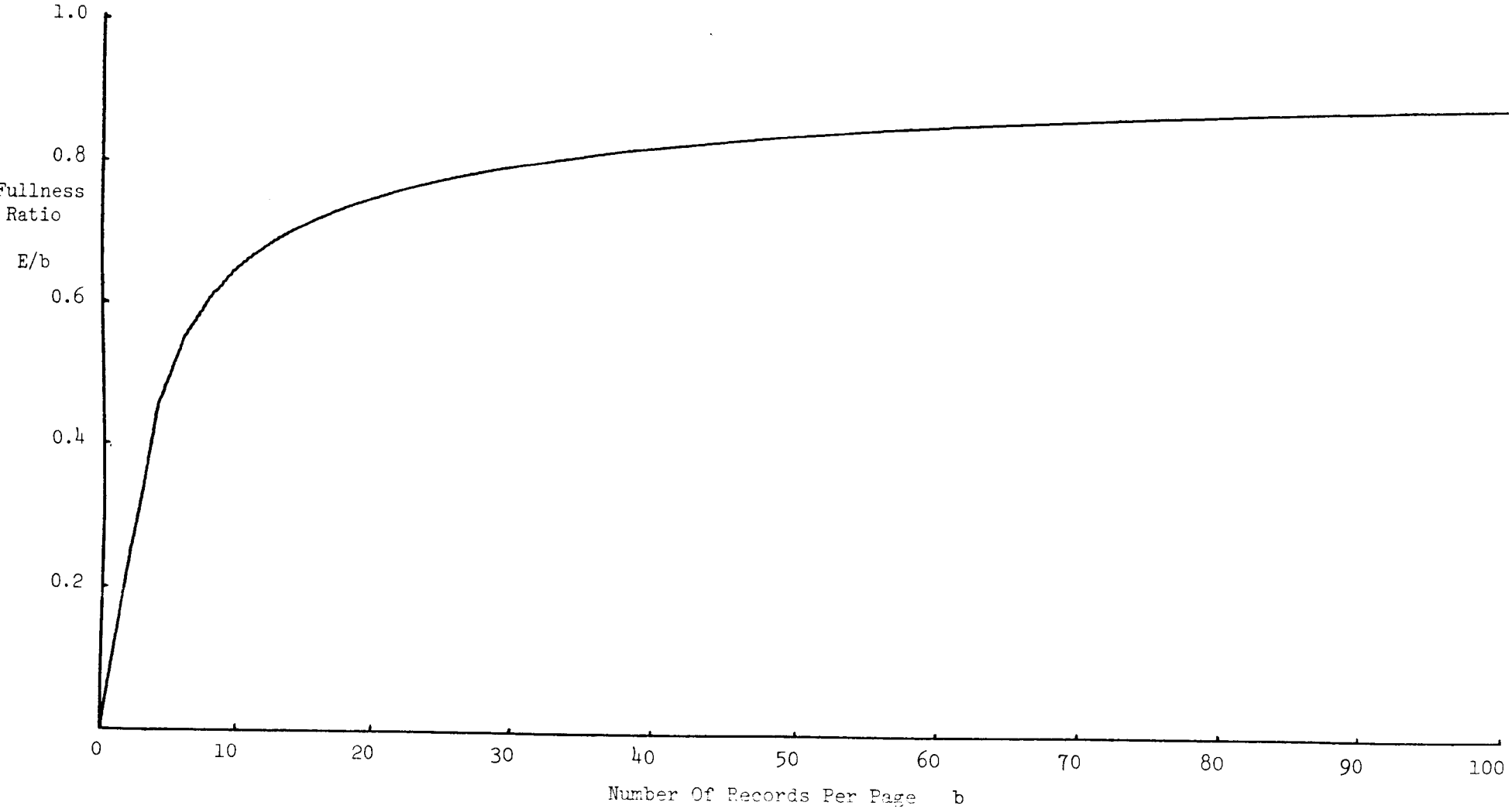
$$E = (d-1)\left(1 - {}^{2d-3}C_{d-1}\left(\frac{1}{2}\right)^{2d-3} - {}^{2d-2}C_{d-1}\left(\frac{1}{2}\right)^{2d-2}\right)$$

$$= b\left(1 - {}^{2b-1}C_{b}\,(1/2)^{2b-2}\right)$$

Corresponding, but more complex expressions can be derived similarly for other values of fp.

Graph 3.1 shows the variation of E/b with b for fp=2. For b> 100 the partially filled page can be expected to be more than 90% full. It is intuitively clear that for other values of fp and correspondingly large values of b, one can expect the partially filled page to be more than 90% full.

G R A P H 3 . 1


1.0
0.8
Fullness
Ratio
E/b
0.6
0.4
0.2
0
10
20
30
40
50
60
70
80
90
100
Number Of Records Per Page b

Morris, 1968 suggests that a scatter index table would be a suitable technique for use on a paged machine. A scatter index table contains entries consisting of a key and a pointer which shows where the rest of the record is located. The data fields of the records are stored sequentially. In this way the entries of the scatter storage table will be shorter than if the whole record were kept there. The advantage claimed is that the whole scatter index table could probably be located in main memory and so only one page fault per access would be necessary to pick up the data. This level of performance i.e. one page fault per access is automatically achieved if the proposed modifications are used and will frequently be surpassed because a proportion of the table will always be located in main memory.

3.2.3 A Major Modification

The discussion so far represents only a minor departure from conventional ideas and as far as programs using the table are concerned, there is little change. For instance a compiler or assembler which uses a hashing system for its symbol table could still acquire information about symbols as they appear in the source text, exactly as on non-paged machines.

If a more radical approach is taken to the use of hash tables a much greater saving in page transfers can be obtained. This is at the expense of altering the mode of operation of user programs. Suppose the previously described modifications are used and in addition, hash table requests are queued until sufficient have been accumulated to allow several accesses to be made for each page brought into real memory.

This is somewhat similar to the sector queuing system employed in paging drum processors. It might be thought that this is totally impractical since most programs which acquire data, do so because they are unable to continue without it. However, consider the operation of an assembler. If a partial scan of the source text is made prior to a phase which uses the symbol table, the information about a fixed number of symbols can be brought into the program's work area in an orderly fashion. The assembler could continue execution until it had used all the information in the work area and then the source text scan could resume.

Similarly, for operations on sparse matrices stored in a hash table, if data was brought into a work area in 'bulk', the page faults could be generated in an orderly fashion.

Suppose the number of requests is allowed to rise to q and they are held in a 'pending' buffer. They are sorted into individual queues, one for each page, and then all serviced at once. The number of page faults caused by servicing this set of requests is equal to the number of non-empty queues.

Let x_q = the number of non-empty queues after q requests are in the buffer.

then:-

$$x_{q+1} = \begin{cases} x_q & \text{if } q+1^{st} \text{ request is for a page which already has a non-empty queue.} \\ x_q + 1 & \text{otherwise.} \end{cases}$$

thus:-

$$E(x_{q+1} | x_q) = x_q \cdot x_q/fp + (x_q + 1)(1 - x_q/fp)$$

$$= x_q(1 - 1/fp) + 1$$

thus $E(x_{q+1}) = E(x_q)(1 - 1/fp) + 1$

which is a first order linear recurrence relation for $E(x_q)$.

Hence:-

$$E(x_q) = fp(1 - (1 - 1/fp)^q)$$

So the total for n requests is:-

$$(n/q).fp(1 - (1 - 1/fp)^q)$$

This assumes that the delay between bursts of paging activity is such that pages of the table do not remain in main memory between bursts. Note that this expression does not depend on c because the ordered hash table requests only require one main memory page to be serviced efficiently.

Suppose n = 5000, fp = 10 pages containing 1000 records, q = 25 which implies a pending buffer size of one quarter of a page and c = 4. Using the simple modifications this would produce 3000 page faults but only approximately 1850 if the requests are queued.

3.3 Non Random Request Pattern

3.3.1 An Example Of A Non Random Request Pattern.

Analysis of the use of hash tables usually assumes a random request pattern, as has been done overleaf. This greatly simplifies the mathematics but is rarely an accurate representation of real life. Using the example of the compiler symbol table, it has been found that identifiers consisting of a single letter are used far more frequently than their longer counterparts (up to 70% of the total identifier usage in some cases).

To investigate the way in which programmers use identifiers, a program was written to scan ALGOL W source text and produce counts of the gaps which occurred between uses of the same identifier. A gap of length n is defined as the occurrence of n instances of other identifiers between two successive uses of a particular identifier. Thus n is a variable taking non-negative integer values.

It is desirable to know what the expected values of these gap frequencies are, if it is assumed that identifiers are used randomly.

Suppose S different identifiers have been used and there are a total of d occurrences of identifiers in the entire source text.

Lemma 3.1

Let $U_{g,d}$ = the number of gaps of length g in a sequence of d symbols, then $E(U_{g,d}) = (d - (g+1))(1 - 1/S)^g/S$

Proof

Clearly:-

$$U_{g,i+1} = \begin{cases} U_{g,i} & \text{if } i+1^{st} \text{ symbol does not induce a gap of length g.} \\ U_{g,i} + 1 & \text{if } i+1^{st} \text{ symbol does induce a gap of length g.} \end{cases}$$

$$\therefore \quad E(U_{g,i+1}|U_{g,i}) = \begin{cases} 0 & \text{if } g > i-1 \\ U_{g,i} + (1 - 1/S)^g/S & \text{if } g \leq i-1 \end{cases}$$

$$\therefore \quad E(U_{g,i+1}) = \begin{cases} 0 & \text{if } g > i-1 \\ E(U_{g,i}) + (1 - 1/S)^g/S & \text{if } g \leq i-1 \end{cases}$$

which is a first order linear recurrence relation for $E(U_{g,d})$.

Hence:-

$$E(U_{g,d}) = (d - (g+1))(1 - 1/S)^g/S$$

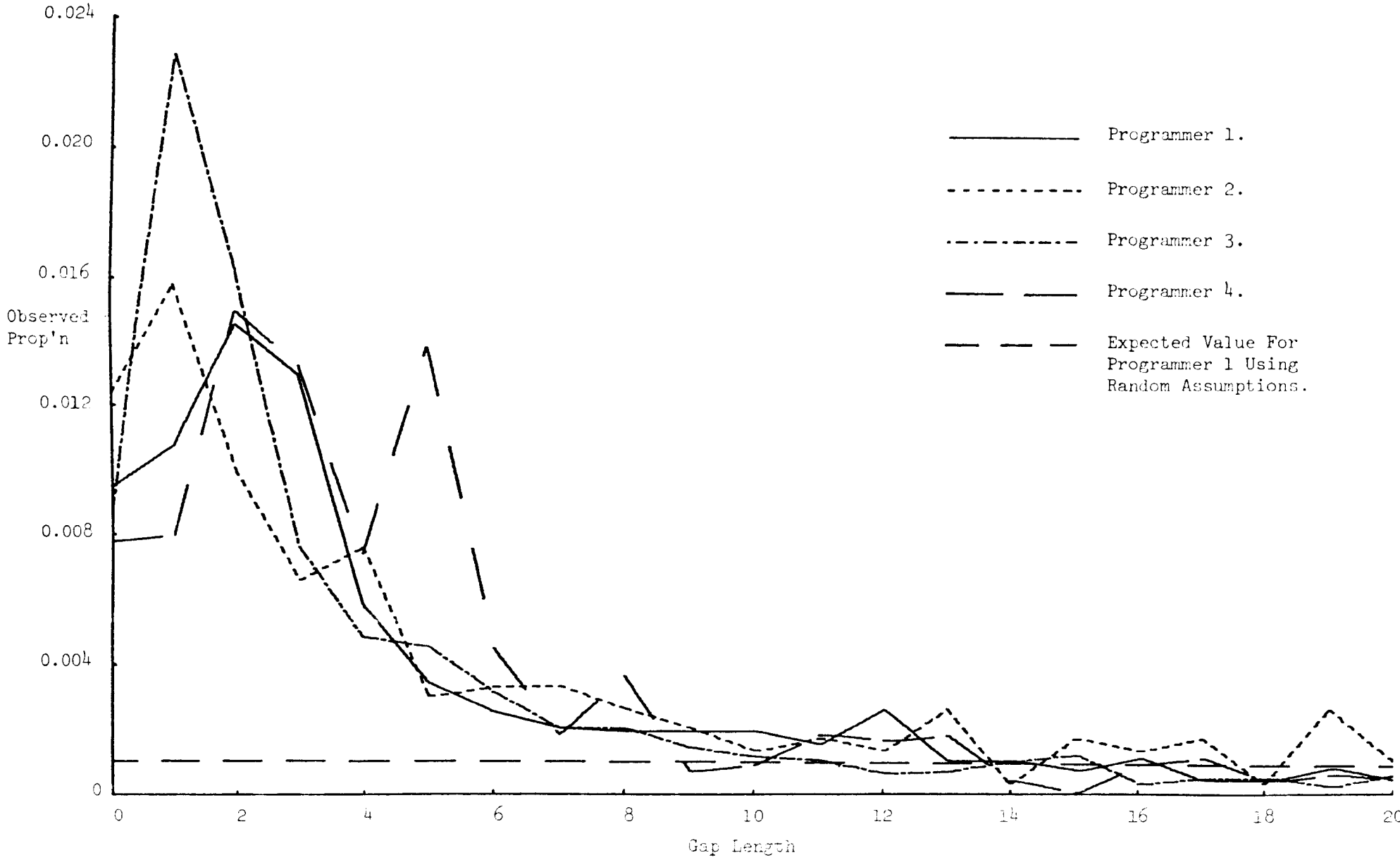
Table 3.1 shows the proportions of different gap lengths that were observed for four long programs (several hundred statements) written by different programmers, together with expected values computed using lemma 3.1 for programmer 1.

Graph 3.2 compares these observations.

Gap Length	Observed Prop'n Prgrmr 1	Expected Prop'n Prgrmr 1	Observed Prop'n Prgrmr 2	Observed Prop'n Prgrmr 3	Observed Prop'n Prgrmr 4
0	0.095	0.0104	0.124	0.087	0.078
1	0.108	0.0104	0.157	0.228	0.080
2	0.145	0.0104	0.100	0.162	0.149
3	0.129	0.0101	0.066	0.076	0.132
4	0.058	0.0101	0.076	0.048	0.074
5	0.034	0.0101	0.030	0.045	0.138
6	0.025	0.0098	0.033	0.031	0.045
7	0.020	0.0098	0.033	0.020	0.018
8	0.019	0.0098	0.026	0.020	0.036
9	0.019	0.0098	0.020	0.014	0.009
10	0.019	0.0095	0.013	0.011	0.009
11	0.015	0.0095	0.017	0.010	0.019
12	0.026	0.0092	0.013	0.006	0.016
13	0.010	0.0092	0.026	0.007	0.018
14	0.010	0.0092	0.003	0.010	0.004
15	0.007	0.0089	0.017	0.012	0.0
16	0.011	0.0089	0.013	0.008	0.009
17	0.004	0.0089	0.017	0.005	0.011
18	0.004	0.0086	0.003	0.005	0.004
19	0.008	0.0086	0.026	0.002	0.006
20	0.004	0.0083	0.010	0.006	0.004

Table 3.1

G R A P H 3 . 2


0.024
0.020
0.016
0.012
0.008
0.004
0
Observed Prop'n
0
2
4
6
8
10
12
14
16
18
20
Gap Length
Programmer 1.
Programmer 2.
Programmer 3.
Programmer 4.
Expected Value For Programmer 1 Using Random Assumptions.

3.3.2 A Model of Non Random Requests

Since the assumption of randomness appears to be inadequate as shown by this example, a more comprehensive analysis is required which can be used in the general case. The following model of request patterns is proposed as one possible approach. After each access, the next request to be serviced requires one of the set of r most recently referenced different records with a higher probability than the other (S-r) in the file. The most recently referenced is required again with probability K_1/S, the i^{th} (different) most recently referenced is requested again with probability K_i/S. Note that these probabilities only refer to different records, several occurrences of the same record do not affect its probability of being requested again. Clearly $K_i = 1$ for every i corresponds to the random case.

3.3.3 L.R.U. Page Replacement

Assume that previously generated addresses are required by the current request with probabilities $K_1/S,\ldots,K_\ell/S,K_{\ell+1}/S,\ldots,K_r/S$ and those with $i \leqslant \ell$ are still located in main memory. Other addresses will be generated with probability:-

$$\frac{S - \sum_{i=1}^{r} K_i}{S(S-r)}$$

Then the probability that the current request will generate a page fault is:-

$$1 - \{ \sum_{i=1}^{\ell} K_i/S + \frac{(S - \sum_{i=1}^{r} K_i)}{S(S-r)} .(cb - \ell) \}$$

$$= 1 - c/fp - \frac{\{ \sum_{i=1}^{\ell} K_i(S-r) + \sum_{i=1}^{r} K_i(\ell-cb) + rcb - S\ell \}}{S(S-r)}$$

where 1-c/fp is the probability found for the random case. Of course ℓ is a random variable and so this probability is conditional on the value of ℓ.

Similarly the expected value of the number of page faults is conditional and it is desirable to remove this dependence on ℓ by using the result:-

$$E(E(X|y)) = E(X)$$

However it has not proved possible to derive expressions for terms such as $E(\sum_{i=1}^{\ell} K_i)$ for general K_i and so an upper bound for the probability is derived by replacing ℓ by its lower bound c. Thus:-

probability of a page fault

$$\leqslant 1 - c/fp - \frac{\{ \sum_{i=1}^{c} K_i(S-r) - c \sum_{i=1}^{r} K_i(b-1) - c(S-rb) \}}{S(S-r)}$$

If $r \leqslant c$ further simplification is possible since in that case $\ell = r$. Thus the exact probability of a page fault can be derived:-

$$1 - c/fp - \frac{\{ \sum_{i=1}^{r} K_i(S-r) + \sum_{i=1}^{r} K_i(r-cb) + rcb - Sr \}}{S(S-r)}$$

$$= 1 - c/fp - \frac{\{(\sum_{i=1}^{r} K_i - r)(S - cb)\}}{S(S - r)}$$

$$= (1 - c/fp)\{1 - (\sum_{i=1}^{r} K_i - r)/(S - r)\}$$

and the expected number of page faults for n requests is just:-

$$n(1 - c/fp)\{1 - (\sum_{i=1}^{r} K_i - r)/(S - r)\}$$

3.3.4 Random Page Replacement.

With random page replacement the only record which is certain to be in main memory is the most recently used. Other previously referenced records are there with decreasing probability as new records are processed.

Let α = the probability that any request causes a page fault

K_i = the probability weights as before.

$P_{i,i_1,\ldots,i_m}$ = the conditional probability of a reference to the table generating a main memory address, given that the previously referenced records with probability weights K_1, K_{i_1}, ... , K_{i_m} are still in main memory.

then P_1 = probability of referencing the record with weight K_1 or one of the other cb-1 records in main memory.

$$= K_1/S + \frac{(S - \sum_{i=1}^{r} K_i)(cb - 1)}{S(S - r)}$$

and more generally:-

$$P_{1,i_1,\ldots,i_m} = K_1/S + \sum_{j=1}^{m} K_{i_j}/S + x(cb - (m+1))$$

where $x = \frac{(S - \sum_{i=1}^{r} K_i)}{S(S - r)}$ is defined for convenience.

Now let $Q_{1,i_1,\ldots,i_m}$ = probability that previously referenced records with probability weights $K_1, K_{i_1}, \ldots, K_{i_m}$ are still in main memory when a reference to the table occurs.

Suppose the movement of a particular record between the two storage levels as table requests are processed is regarded as a two state Markov process then:-

$$A = \begin{array}{l} \\ \text{Primary Memory} \\ \text{Secondary Memory} \end{array} \begin{array}{c} \begin{array}{cc} \text{Primary Memory} & \text{Secondary Memory} \end{array} \\ \begin{pmatrix} (1 - \alpha/c) & \alpha/c \\ \alpha/(p-c) & (1 - \alpha/(p-c)) \end{pmatrix} \end{array}$$

is the associated transition matrix. So the probability that a particular record is in main memory say after d requests have been serviced is just $A^d(1,1)$ or $A^d(2,1)$ depending on its initial location. For instance the probability that the record with weight K_r is in main memory is $A^{r-1}(1,1)$. $Q_{1,i_1,\ldots,i_m}$ is merely the product of such terms with A raised to powers i_1-1 , ... , i_m-1 and terms of the form $A^{k-1}(1,2)$ with k taking on values from the set $\{ 1, 2, \ldots , r \}\setminus\{ i_1, i_2, \ldots , i_m\}$.

The total probability rule:-

$$pr(\text{Event } \varepsilon) = \sum_{i=1}^{n} pr(\text{Event } \varepsilon | \text{Event } \beta_i) pr(\text{Event } \beta_i)$$

can now be invoked using $P_{1,i_1,\ldots,i_m}$ and $Q_{1,i_1,\ldots,i_m}$ to give the page fault probability in terms of an expression involving itself.

A closed form expression for α is clearly desirable but cannot be found in the general case, because a polynomial in α is obtained and there is no analytic solution for this polynomial when its order is greater than four.

However specific simple models are often adequate, in particular models where r is small (e.g. perhaps only K_1, or K_1 and K_2 are significant) and $K_i = K \forall i$ (K a constant). In such cases α can be found explicitly.

For example if r = 1, $K_1 = K$, then trivially:-

$$1 - \alpha = K/S + (1 - K/S)(cb - 1)/(S - 1)$$

$$\alpha = \{(S - cb)(S - K)\}/\{S(S - 1)\}$$

Alternatively if $r = 2$, $K_1 = K_2 = K$ then:-

$$P_1 = K/S + \{(S - 2K)(cb - 1)\}/\{S(S - 2)\} \qquad Q_1 = \alpha/c$$

$$P_{1,2} = 2K/S + \{(S - 2K)(cb - 2)\}/\{S(S - 2)\} \qquad Q_{1,2} = 1 - \alpha/c$$

$$1 - \alpha = P_1 Q_1 + P_{1,2} Q_{1,2}$$

$$\alpha = (c/S).\{(S - 2K)(S - cb)\}/(cS - 2c - K + 1)$$

The advantage of using L.R.U. page replacement when there is a non-random request sequence is intuitively clear. The records with the highest probability of being required again are the most recently referenced and L.R.U. will tend to keep these in real storage.

It is not possible to quantify the exact difference in page fault probability between random and least recently used replacement in the general caes because no general expression for the page fault probability has been found for random replacement. However for the two simple models discussed above the exact difference can be found.

In the first case ($r = 1$, $K_1 = K$), all replacement policies will have the same page fault probability since the most recently referenced record will always be in storage.

The difference in page fault probability between RAND and L.R.U. for the second simple model ($r = 2$, $K_1 = K_2 = K$) is:-

$$\frac{c(1 - cb/S)(S - 2k)}{(cS - 2c - K + 1)} - \frac{(1 - cb/S)(S - 2K)}{(S - 2)}$$

$$= (1 - cb/S)\frac{(S - 2K)}{(S - 2)}\cdot\frac{(K - 1)}{(cS - 2c - K + 1)}$$

This expression is positive since:-

$S > cb$	because it is assumed the file is larger than available real storage
$S > 2K$	the model of non-randomness requires $S > \sum_{i=1}^{m} K_i$
$K > 1$	$K = 1$ is the random case.

and

$$S > 2K \Rightarrow S > \frac{K}{c} + 2 \Rightarrow cS - 2c - K + 1 > 0$$

This suggests the following modification. When random page replacement is being used, having been selected on some other grounds, and non-random requests are being made to a scatter storage table, copies of the n most recently referenced records are maintained as a linked list in a working area within the program. This list is scanned before each access of the table and if the desired record is found, it is made the head of the list and no reference to the table is necessary.

If it is not found the table is searched with the ensuing possibility of a page fault, a copy of the record is placed at the head of the list and the last record of the list is deleted. This modification is similar in some senses to the cache memory of the 360/85 and the associative memory of the 360/67. The length of the list, n, is determined in an obvious way by r, the number of records contributing to the model of non-randomness, and the available space.

Suppose n = r, i.e. copies of all of the records in the model are kept in the list. By definition of the model, the probability that any search will find the required record in the list is $\sum_{i=1}^{r} K_i/S$. If c real storage pages are available and occupied by pages of the file, the probability that a page fault is necessary is approximately:-

$$\alpha \simeq 1 - \{ \sum_{i=1}^{r} K_i/S + (1 - \sum_{i=1}^{r} K_i/S)\ c/p \}$$

where p is the total size of the file. This probability is only approximate because those records of which copies exist in the list may be located in real storage, and this affects the second term of the expression within parentheses. The error introduced is extremely small if r is small compared with the number of records per page, and will be neglected.

The original purpose of introducing this modification was to gain some of the advantages of L.R.U. page replacement when random page replacement is in use. This is achieved since if c > r, the probability of a page fault with this modification in operation is very close to that obtained with L.R.U. replacement. However the modification has a second and more significant property. If real storage is limited such that c < r, then all of the records for which

non-uniformity exists will be available for reference even if the pages of the file on which they reside are not located in real storage. Thus the above expression for the page fault probability will apply for all practical page replacement algorithms and any amount of real storage. If the K_i are reasonably large there should be a significant reduction in paging for small amounts of real storage.

To provide some evidence that this is so, three non-random request sequences were generated and the resulting page reference strings processed by page replacement simulation programs. In each case the table size was taken to be 16 pages with 100 records per page and 2000 searches were performed in each sequence. Three models were used:-

(i) $r = 8$ $K_i = 100$ $(1 \leqslant i \leqslant 8)$

(ii) $r = 8$ $K_i = 150$ $(1 \leqslant i \leqslant 8)$

(iii) $r = 8$ $K_i = 175$ $(1 \leqslant i \leqslant 8)$

and the numbers of page faults for each replacement algorithm using various amounts of real storage are shown in tables 3.2 - 3.4 and graphs 3.3 - 3.5. The results labelled MOD are the values predicted to occur if the suggested modification is used. The data shown for the Random algorithm is the average of several runs since the initial value used in the random number generator of the paging simulator affects the number of page faults produced.

As expected the L.R.U. algorithm consistently outperformed the others which were considered (except MIN of course). This is particularly true in the third case ($K_i = 175$) where the probability is $\frac{8 \times 175}{1600} = 0.875$ that one of the eight most recently referenced records will be required again.

TABLE 3.2

Main Memory Pages	Page Faults				
	L.R.U.	F.I.F.O.	Random	MIN	MOD
2	1487	1492	1521	1183	884
3	1271	1276	1300	864	822
4	1044	1083	1154	657	760
5	859	907	980	519	698
6	682	743	842	416	636
7	558	641	730	339	574
8	437	558	600	274	512
9	423	489	537	219	450
10	362	404	432	174	388
11	302	354	370	134	326
12	237	262	274	100	264
13	167	203	222	71	202
14	104	136	155	48	140
15	55	89	80	30	78
16	16	16	16	16	16

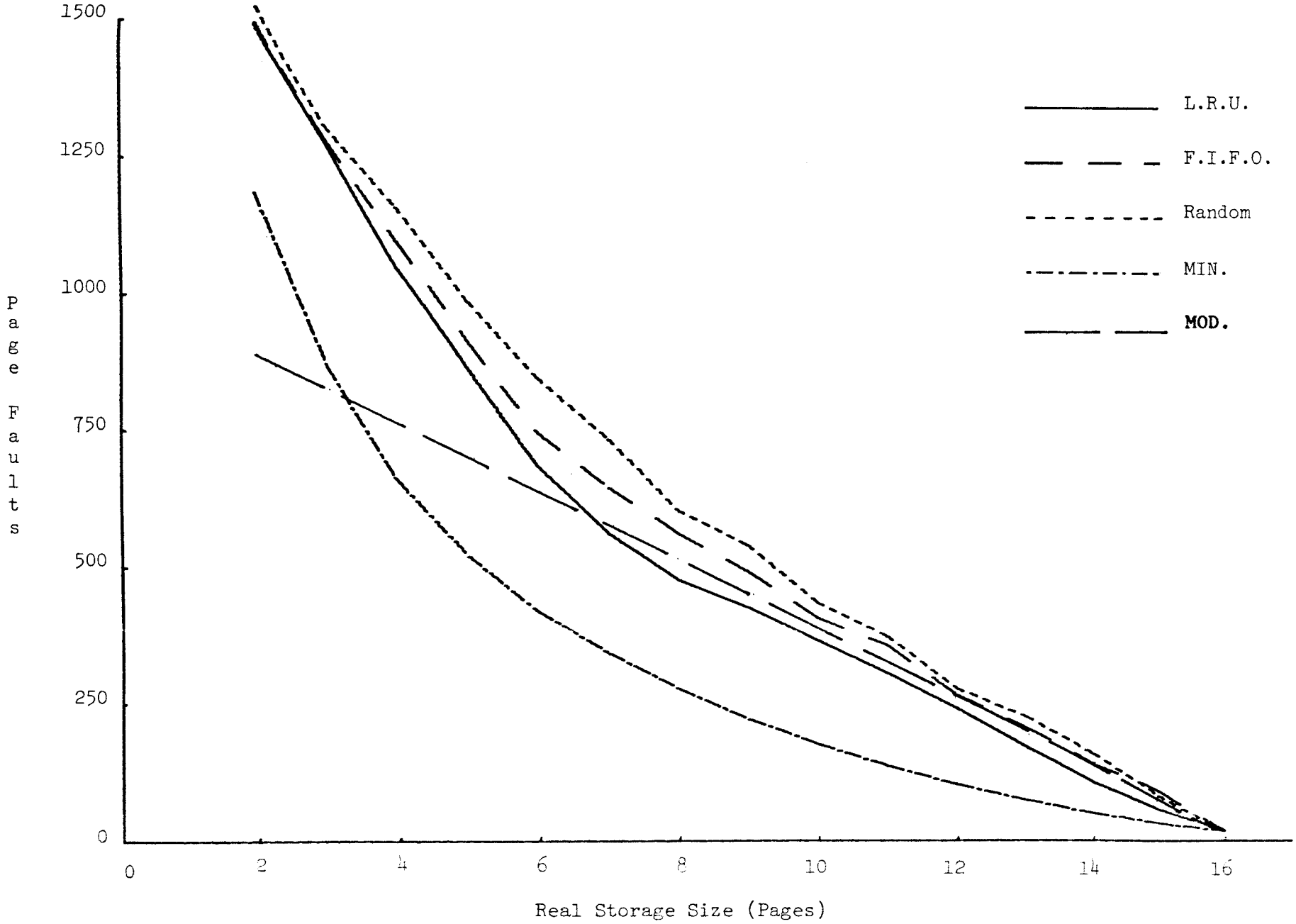
TABLE 3.3

Main Memory Pages	Page Faults				
	L.R.U.	F.I.F.O.	Random	MIN	MOD
2	1383	1396	1415	1081	450
3	1111	1123	1136	729	419
4	836	864	906	502	388
5	605	646	704	351	357
6	430	506	588	260	326
7	305	416	465	200	295
8	240	336	390	156	264
9	211	278	340	126	233
10	180	223	252	99	202
11	149	194	215	78	171
12	122	161	170	62	140
13	93	118	120	49	109
14	69	92	84	36	78
15	42	47	50	25	47
16	16	16	16	16	16

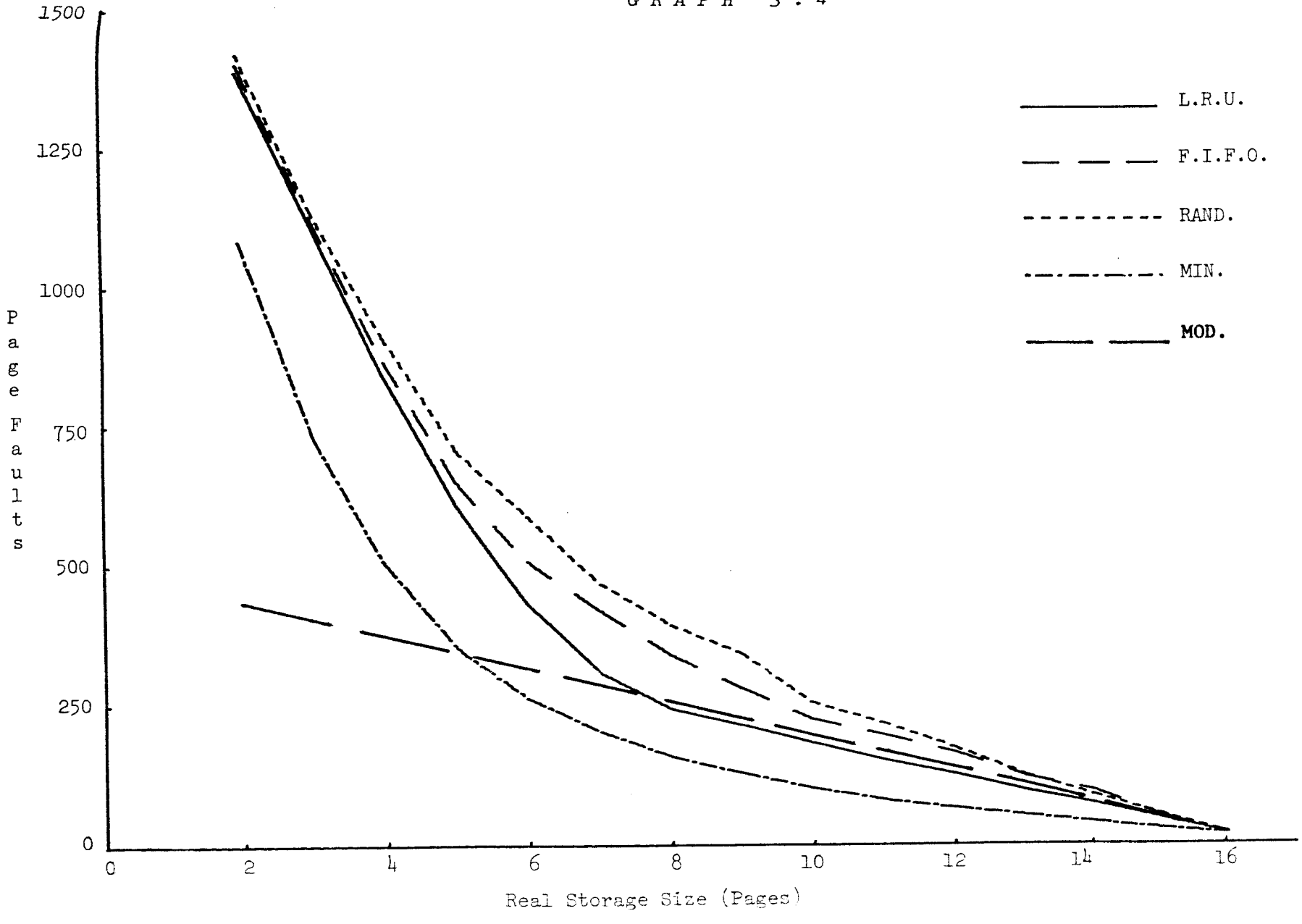
TABLE 3.4

Main Memory Pages	Page Faults				
	L.R.U.	F.I.F.O.	Random	MIN	MOD
2	1340	1351	1350	1047	233
3	1047	1067	1063	686	218
4	781	812	820	452	202
5	545	548	618	272	186
6	290	384	396	175	171
7	172	281	315	118	155
8	129	199	242	90	140
9	109	168	192	70	125
10	98	125	179	56	109
11	73	102	120	44	94
12	57	86	91	36	78
13	46	76	75	30	63
14	34	74	44	25	47
15	30	29	28	20	32
16	16	16	16	16	16

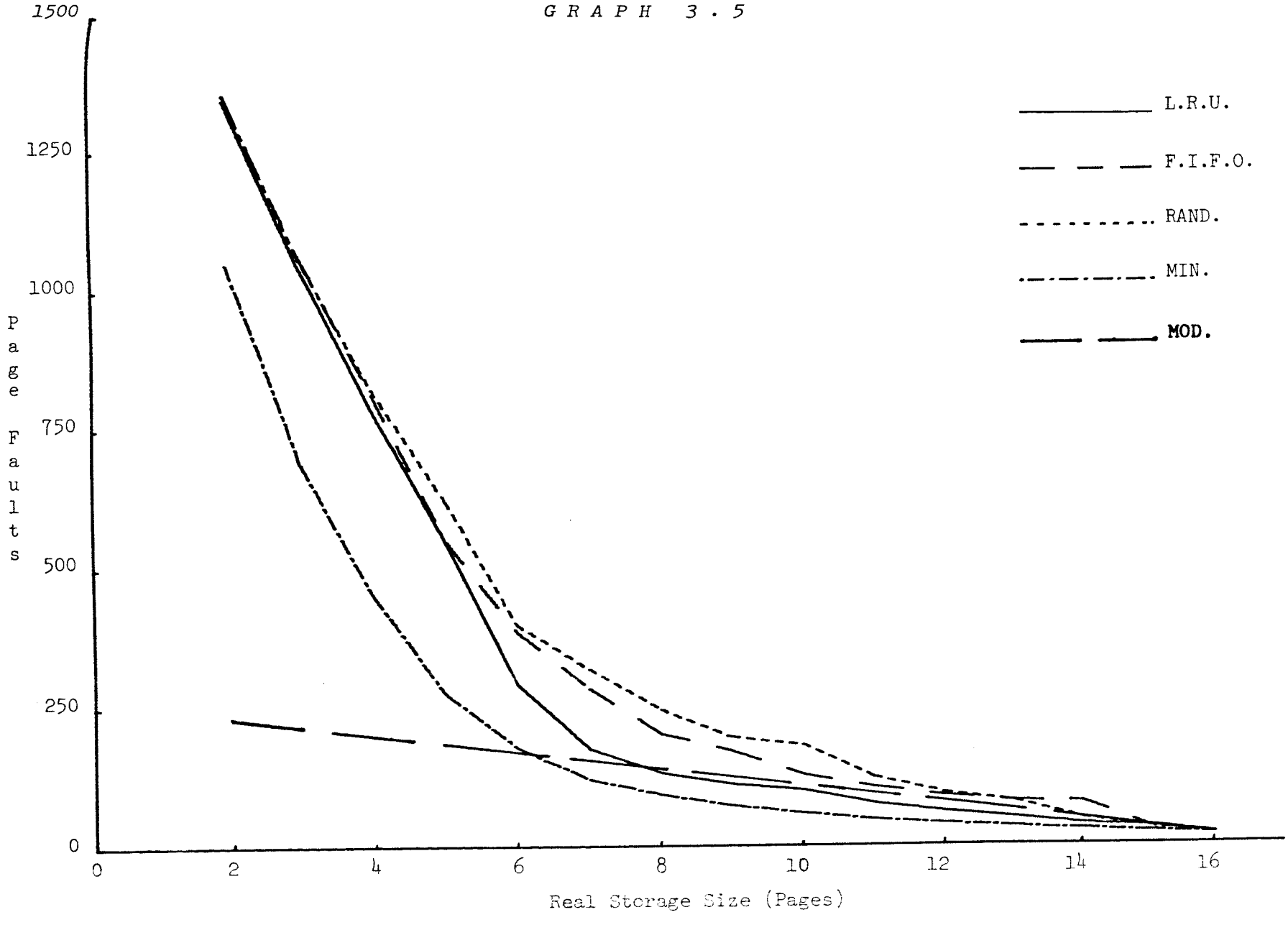
G R A P H 3 . 3


1500
1250
1000
750
500
250
0
Page Faults
0
2
4
6
8
10
12
14
16
Real Storage Size (Pages)
L.R.U.
F.I.F.O.
Random
MIN.
MOD.

GRAPH 3.4


1500
1250
1000
750
500
250
0
Page Faults
0
2
4
6
8
10
12
14
16
Real Storage Size (Pages)
L.R.U.
F.I.F.O.
RAND.
MIN.
MOD.

GRAPH 3.5


1500
1250
1000
750
500
250
0
Page Faults
0
2
4
6
8
10
12
14
16
Real Storage Size (Pages)
L.R.U.
F.I.F.O.
RAND.
MIN.
MOD.

It could also be anticipated that c = 8 (i.e. eight real page frames available) will be a critical value. The most recently used eight pages can be regarded as the W.S.D. and with L.R.U. at least, as c increases beyond eight the rate at which paging decreases is small.

The advantage of the proposed modification shows up clearly.

3.3.5 F.I.F.O. Page Replacement

It is reasonable to suppose that F.I.F.O. replacement lies somewhere between Random and L.R.U. in terms of efficiency. For example, if a record is referenced and is located on the secondary storage device, the page containing that record enters real storage and is certain to remain there until c additional page faults have occurred. If the record originally requested is required again in that time, which it may well be since a non-random request sequence is assumed, it will be available immediately. Random page replacement will be less efficient since all pages are candidates for displacement at all times, and L.R.U. will be more efficient since every reference to a page guarantees it a longer stay in real storage.

This conclusion is supported by the simulation studies, the results of which are shown in section 3.3.4.

Chapter 4.

SEARCHING

4.1 Introduction

Searching in this context means the location and retrieval from a file of a record containing some particular key field. Many techniques are available and one has already been covered in detail in chapter 3. Hash Coding is of sufficient importance and is used so commonly on large files that it warrants extensive separate treatment. Other methods will be examined here and the techniques established in the consideration of Hash Coding will be employed.

4.2 Binary Searching

4.2.1 Conventional Binary Searching

The standard binary search technique for locating records in a file which is ordered is well known. It is preferable to 'n'ary searches with n > 2 because it produces the least average number of comparisons. However when used with a large file on a paged machine it is surprisingly inefficient because the number of page transfers induced is unnecessarily large.

Consider a file of M pages and for simplicity suppose $M = 2^K$ for some integer K. During a binary search operation the desired record is known to be located in a region of the file which is halved in size by each step. Define R to be the set of pages constituting that region , and $|R|$ the number of pages in R.

Clearly $|R| = M$ initially and when $|R|$ drops to one the paging induced by the algorithm will cease.

$|R|$ will become one after $\log_2 M = K$ steps and as the record examined in each step will be located on a previously unreferenced page, K different pages will usually be referenced. The final step is concerned with deciding which of the two remaining pages in R contains the desired record. One of them will have been brought into main memory in order to do the comparison and so the probability is at least 1/2 that referencing K different pages is sufficient. If the desired record lies on the other of the two pages involved in the final step, K + 1 pages might be accessed and K + 1 page faults might be caused, depending on whether previously referenced pages of the file remain in main memory.

It is, of course, possible that the required record will be one of those involved in the comparisons performed before $|R|$ becomes one.

The probability of this event occurring is:-

$$(1/Mb) + (1 - 1/Mb)\frac{1}{Mb-1} + \ldots\ldots + (1 - 1/Mb)\ldots\frac{1}{Mb-(K-1)}$$

$$= \sum_{i=0}^{K-1} \frac{1}{Mb-i} \prod_{j=0}^{i-1} \left(1 - \frac{1}{Mb-j}\right)$$

$$= K/Mb$$

where b is the number of records per page.

Clearly, this will be a small quantity if b is large. For example if b = 100 and M = 16 then K = 4 and this probability is only 0.0025. In what follows, this event will be ignored.

Figure 4.1 illustrates the case M = 8 i.e. K = 3 and hopefully clarifies the discussion.

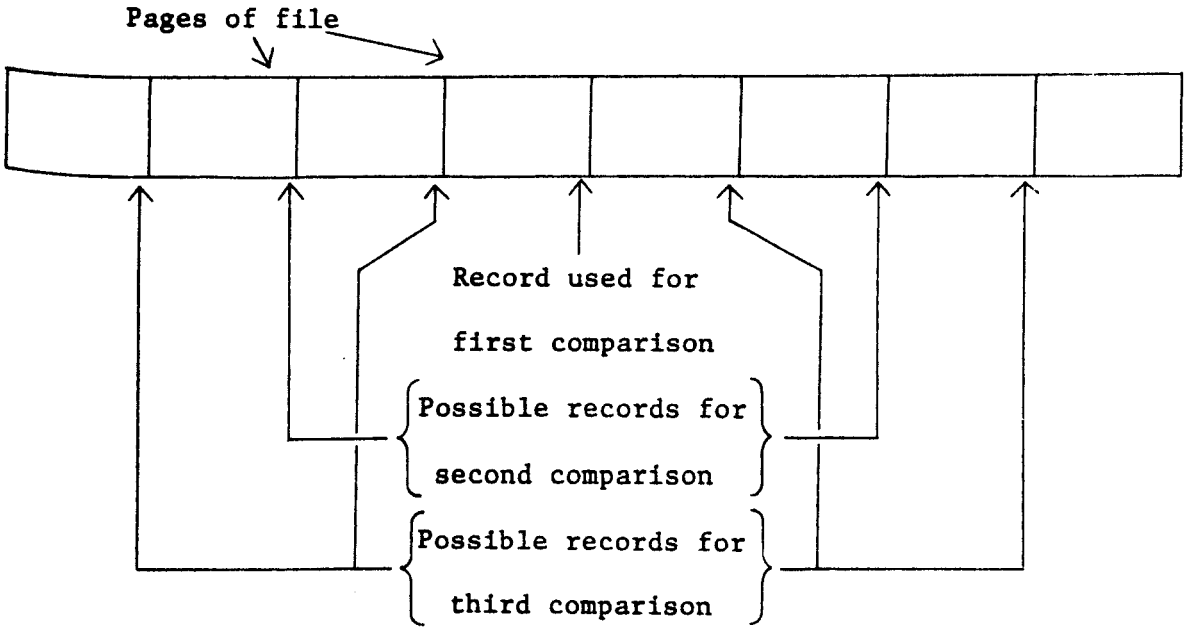


Fig 4.1

In the case of infrequent accesses of the file it has to be assumed that none of the file remains in main memory between requests and so all K (and sometimes K + 1) page faults will be induced for each record that has to be located.

4.2.2 Modified Binary Searching

Now consider the following modification. Suppose copies of the key fields of the first record located in the second and subsequent pages of the file are kept in a small working area of the program which

is referencing the file. Comparison of the desired key with this list will reveal which page contains the desired record because the file is ordered. In this way only one page fault need be caused by a reference to the file because R consists immediately of just one page. This process will involve $\frac{M}{2}$ comparisons on average if the list of keys is searched linearly or a maximum of $\lceil \log_2(M-i) \rceil$ if a binary search of the list is used. Clearly this is the first step in an M way search. The rest of the search on the single remaining page should be binary as this part of the algorithm will operate in main memory only.

The cost of this modification is the storage required for the M - 1 keys in the list and the associated searching of this list. The saving is K - 1 (and sometimes K) page faults per access. The modification is similar to the Estimated Entry technique described by Price, 1972 and other authors. It could also be regarded as the use of a dope vector.

The usual problems of addition and deletion of items are little different when the file is located in a paged memory. Moving large parts of the file to insert an item is costly and, if the proposed modification were used, the contents of the list of keys would have to be updated to reflect the changes.

The alternative of leaving unoccupied areas scattered through the file to make way for expansion is surprisingly attractive provided the proposed modification is used. They might take the form of unused spaces at the end of each page, the size of the spaces being dependent on the likely frequency of insertions. No extra page faults need be induced by the extra space requirements as only one is necessary per access.

In addition, this would take advantage of the virtual memory facilities since maintenance of the file is made easier by using more virtual memory than is strictly necessary but without the heavy cost of additional paging. Note that this is only true if the file is used infrequently. Where a working set can be maintained it is preferable to compress the file as was noted in chapter 3.

In terms of the number of page faults produced, the proposed modification is also almost independent of any non-random request pattern as any record can be accessed with at most one page fault. Again this is not true of the ordinary binary search technique where it would be helpful in reducing page faults to have the first comparison done on the most frequently needed item etc.

An entirely equivalent modification, which may be more convenient, is to keep copies in the program's work area of the key fields of the records involved in comparisons which occur before $|R|$ becomes 1.

The previous discussion is based on the assumption that requests to the file are so infrequent that as each occurs, the whole file is located on backing storage. If the access frequency is high enough, the program using the file may well reach the point where it maintains a working set of file pages in main memory. Suppose this latter situation exists and c real memory page frames are available for the program's data.

With a conventional binary search, the page containing the centre record will be accessed the most frequently, the two pages containing the records used for the second comparison will each be accessed half as frequently, on average, and so on.

The effect that this has on the paging activity depends on the replacement algorithm and so each will be considered separately.

4.2.3 L.R.U. Page Replacement

For those values of c which are less than K, there will be approximately K page faults per access of the file because the frequently used pages will be displaced from main memory between uses. This will be a lower bound since in some cases K + 1 page faults are required.

When c = K, the page containing the centre record will be available, although 'least recently used', at the beginning of each new search and immediately becomes 'most recently used', hence effectively resident. This will mean that only K - 1 page faults will usually be required to locate a record. However there is a probability of 1/2 that the initial comparison will have the same result as during the previous search and hence the record used for the second comparison will also be located in main memory.

Many different states can occur and the transition matrix is inherently complex but a reasonable approximation of the paging activity can be obtained if the following simplifying assumptions are made:-

(i) For each comparison, the probability of the same result occurring as on the previous search is 1/2.

(ii) All records required following a comparison with a different result to that of the previous search are located on secondary memory.

These assumptions imply:-

pr (K-1 p.f. necessary) = 1/2

pr (K-2 p.f. necessary) = 1/4

etc.

The approximate nature of the model is clear when it is realized that only in the infinite case do these probabilities add to one. However for n requests the approximate average number of page faults for c = K will be:-

$$\sum_{i=1}^{K} n\,(1/2)^{i}\,(K - i)$$

As c increases there will be little drop in the paging activity until c reaches a figure where both the pages containing the records used for the second comparisons will normally be in main memory. An example illustrates this. Figure 4.2 shows a 16 page file with the pages numbered for identification.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Fig 4.2

Suppose two records are to be located which lie in pages 3 and 15. The sequence of pages referenced would be:-

8 4 2 3 8 12 14 15 8 .....

If c = 7, then at the beginning of the search for the third record, the 'centre' page (8) and both pages which might be required for the second comparison (4 & 12) are located in main memory.

Hence the maximum number of page faults which will normally be required is K - 2. It follows easily from the example that in the general case this situation arises when c = 2K - 1.

Using the previously noted simplifying assumptions, and a similar set of probabilities, the approximate average number of page faults induced by n searches with c = 2K - 1 is:-

$$\sum_{i=2}^{K} n\,(1/2)^{i-1}\,(K - i)$$

At least two sources of error arise with this approximation. Firstly it is possible to select one of the second comparison pages several times in succession and probably displace the other because of the additional pages needed to complete the searches. Secondly, occasionally K - 1 page faults will be necessary - this corresponds to the use of page number 16 in the example.

The two drops in paging activity predicted above to occur at c = K and c = 2K - 1 are observed in simulation studies (see below). Similar drops which might be expected at higher values of c, corresponding to the presence in main memory of other pages involved in 'early' comparisons do not occur. This can be explained by the fact that the assumptions become increasingly invalid as c increases, and the frequency of use of the pages in question is correspondingly less than say the 'centre' page.

4.2.4 Random Page Replacement

In the previous section it was shown that the pages used for the first and second comparisons at least, require special

consideration because they are used relatively frequently. With this in mind and using the assumption that K pages have to be referenced to locate each record from a file of size 2^K, the following model of random page replacement is suggested.

Let

p_1 = probability that referencing the 'centre' record at the beginning of a search causes a page fault.

p_2 = probability that referencing the second record of a search causes a page fault.

p_3 = probability that i^{th} record of a search causes a page fault ($3 \leq i$). Thus comparisons after the second are treated equally.

then $1 - p_1$ = probability that second and subsequent comparisons do not displace the page containing the centre record

$= (1 - p_2/c)\ (1 - p_3/c)^{K-2}$

and $1 - p_2$ = probability that same second comparison record is required as for previous search and that it is still in main storage or if the other second comparison record is required, that it is in main storage.

$= (1/2)\ (1 - p_3/c)^{K-2}(1 - p_1/c) + (1/2)\ ((c-2)/(2^K-2))$

and $1 - p_3$ = probability that the record required for i^{th} comparison is in storage ($3 \leqslant i$). If the second comparison did not displace the page containing the 'centre' record there are c - 2 real storage pages that i^{th} comparison record could lie on. Otherwise c - 1. Clearly it will not lie on the pages containing the centre record or the two records which could be used for second comparison so it can only lie on one of $2^K - 3$ of the file pages. p_3 will vary slightly with i because real storage contents will change but this effect is neglected.

$$= (1 - p_2/c).((c-2)/(2^K-3)) + (p_2/c).((c-1)/(2^K-3))$$

This set of non-linear simultaneous equations in p_1 , p_2 and p_3 was solved approximately by iteration and the results are included in section 4.2.6. They compare favourably with the simulation studies except for extreme values of c.

The fact that random page replacement outperformed the other algorithms for small values of c should be noted. This effect is due to its ability to make the optimum choice (in the sense of the MIN algorithm) on some occasions purely by chance. In addition the drops in paging activity which occur for certain values of c with L.R.U. replacement are not observed with random replacement. This is to be expected since it is intuitively obvious that the

probability of frequently used pages being in main storage when required, increases smoothly and monotonically as c increases, rather than in jumps.

4.2.5 F.I.F.O. Page Replacement.

A complete analysis of binary searching with F.I.F.O. replacement involves the same problems as have been noted elsewhere with this paging algorithm. The large number of possible states make the calculations prohibitively complex. However a simpler model, similar to that employed with random page replacement, gives at least some insight into expected behaviour.

For $c < K$, the paging rates will be similar to that observed with L.R.U. Each search will usually induce K page faults and there will only be a small drop as c increases. So for n searches approximately nK page faults will be required.

When $c = K$ the centre page will be in main memory at the beginning of roughly one half of the searches. This is most easily understood with an example. Suppose the file pages are numbered 1 to 16 for identification and records are needed from pages 3 and 15. This is the example of section 4.2.3. The sequence of pages referenced will be:-

8 4 2 3 8 12 14 15 8 ....

Clearly if $c = K = 4$, the second reference to page 8 occurs while it is still in main memory. It will be displaced by the reference to page 12, and so the third reference to page 8 will cause a page fault, as did the first.

This process of causing a page fault for every other search continues for all n requests.

For c = K the proposed model is:-

(i) the probability that the 'centre' page is in main memory for any particular request is 1/2.

(ii) the probability that the same page is selected for the second comparison as on the previous search is 1/2 and that it is still in main memory is 1/2.

(iii) the probability that any one of the pages required to complete a search is in main memory when needed is:-

$$\frac{c-2}{2^K-3} = \frac{\left(\begin{array}{c}\text{no. of main mem. pages not used by} \\ \text{first or second compares}\end{array}\right)}{\left(\begin{array}{c}\text{no. of file pages which could be} \\ \text{requested}\end{array}\right)}$$

Thus an estimate of the number of page faults needed for n requests with c = K is:-

(Cont'd Overleaf)

$n.(1/2)$	using page for first comparison
$+ n.(1/2)$	using page for second comparison when different from previous search
$+ n.(1/2).(1/2)$	using page for second comparison when same as previous search
$+ n(1 - (c-2)/(2^K-3)).(K - 2)$	completing searches.

$$= 5n/4 + n(K - 2)(2^K - 1 - c)/(2^K - 3)$$

and as with L.R.U. replacement there is a considerable drop in the amount of paging as c becomes equal to K.

For c > K the number of page faults can be expected to drop as c increases but simulation is relied upon for actual numeric values.

4.2.6 Simulation Results

Conventional binary searching was simulated for two file sizes, 16 and 32 pages, and in each case two hundred searches were performed. A random request pattern was used. Page replacement using the different algorithms being considered here was simulated on the resulting page reference strings for various main memory sizes assuming a request frequency sufficiently high to use all the main memory made available. The results together with the figures predicted by the above analyses are shown in tables 4.1 and 4.2, and graphs 4.1 and 4.2 compare the various performances.

The final columns of tables 4.1 and 4.2 (labelled MOD) show the numbers of page faults expected to occur during two hundred searches using the proposed modification with a high request frequency. The very considerable improvement it provides in the case of restricted main memory should make it very worthwhile.

The drops in paging predicted for the conventional binary search algorithm are observed to occur when anticipated and to be of approximately the expected size.

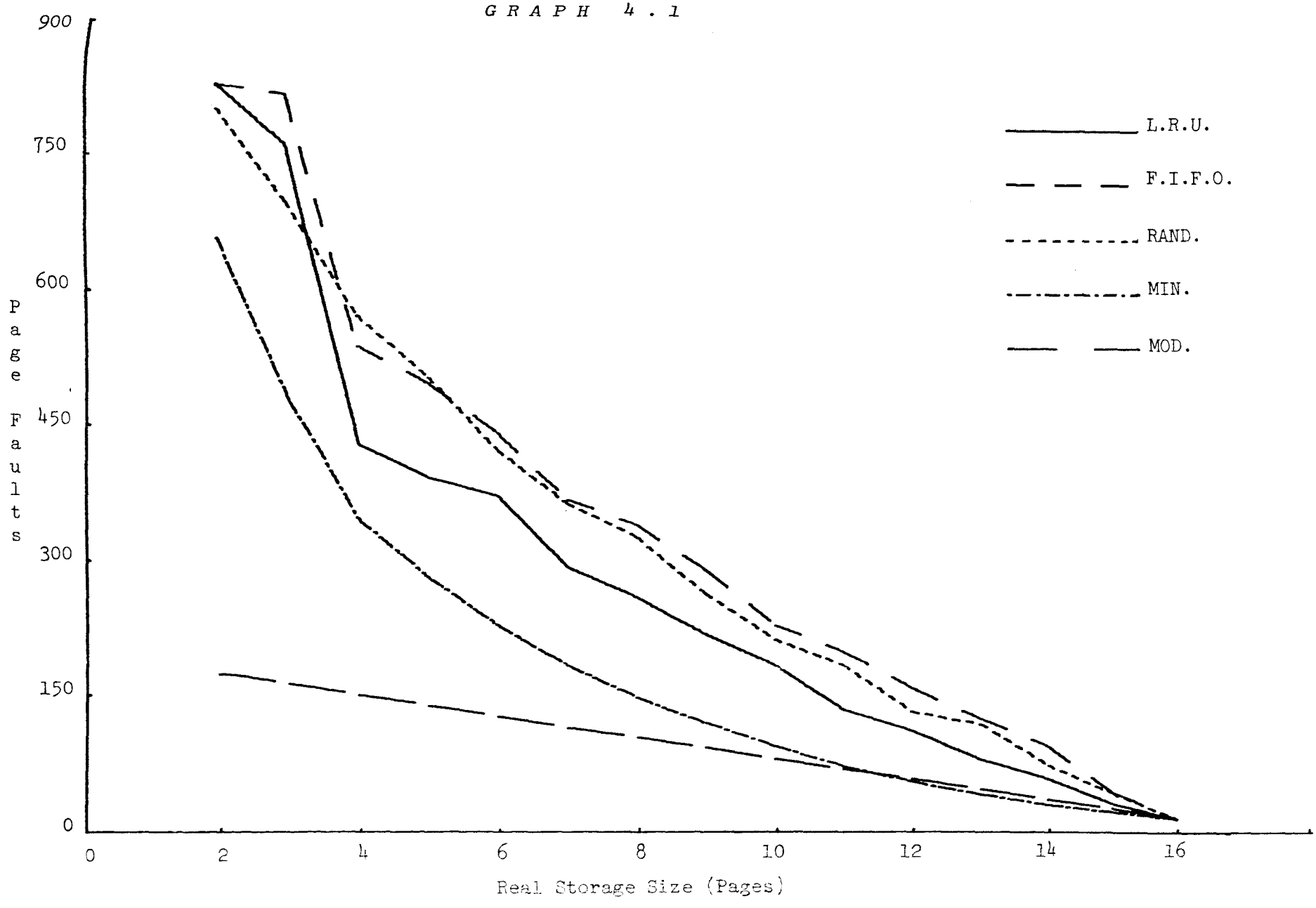
Real Pages	L.R.U.		F.I.F.O.		Random		Min.	Mod.
	Actual	Exp.	Actual	Exp.	Actual	Exp.		
2	824	800	824	800	798	745	657	175
3	758	-	813	-	695	643	478	163
4	427	425	536	586	570	559	345	151
5	390	-	494	-	501	489	280	139
6	370	-	440	-	421	427	227	127
7	292	250	367	-	362	371	183	115
8	258	-	338	-	325	319	148	104
9	217	-	289	-	262	271	120	93
10	184	-	228	-	212	224	95	81
11	135	-	199	-	184	179	73	70
12	112	-	159	-	133	135	56	59
13	80	-	125	-	119	92	42	48
14	59	-	95	-	74	50	31	37
15	32	-	45	-	44	-	23	27
16	16	16	16	16	16	16	16	16

Table 4.1

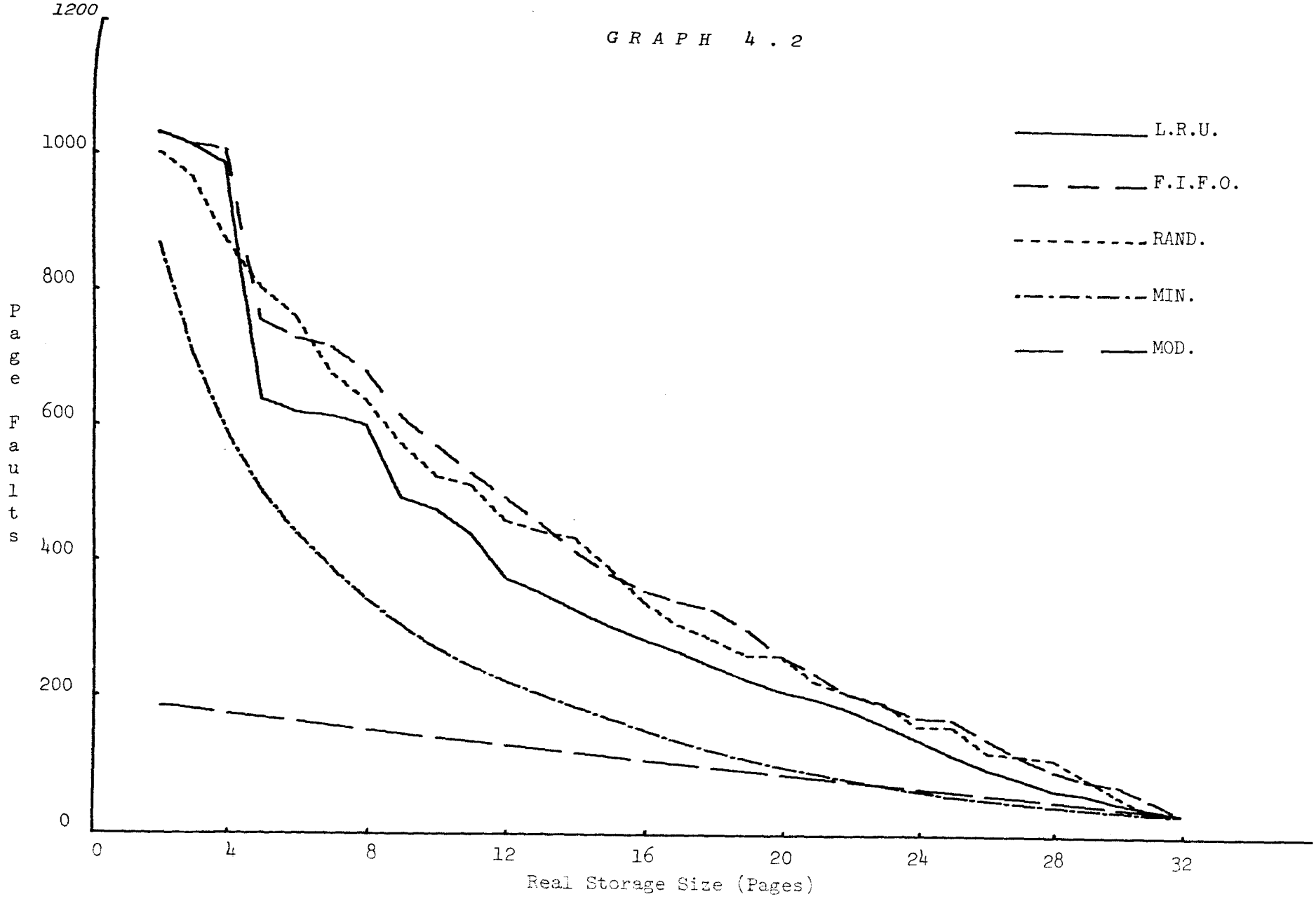
Real Pages	L.R.U.		F.I.F.O.		Random		Min.	Mod.
	Actual	Exp.	Actual	Exp.	Actual	Exp.		
2	1030	1000	1030	1000	1002	970	870	188
3	1011	-	1013	-	965	903	705	182
4	984	-	1005	-	874	838	591	175
5	639	612	754	786	803	781	504	170
6	619	-	729	-	760	732	442	164
7	614	-	716	-	677	689	390	158
8	600	-	681	-	637	650	345	152
9	493	425	614	-	573	611	305	146
10	476	-	571	-	526	576	272	140
11	440	-	531	-	512	544	245	135
12	376	-	495	-	461	512	224	130
13	355	-	458	-	445	482	205	124
14	329	-	416	-	436	453	186	119
15	305	-	382	-	391	424	169	113
16	285	-	358	-	341	396	152	108
17	268	-	342	-	308	367	136	103
18	247	-	331	-	287	340	122	98
19	227	-	302	-	264	313	110	93
20	210	-	263	-	263	287	99	88
21	199	-	237	-	225	260	90	83
22	183	-	207	-	207	234	81	78
23	162	-	191	-	194	207	72	73
24	140	-	173	-	160	181	64	68
25	117	-	170	-	160	156	57	63
26	97	-	142	-	122	130	52	59
27	82	-	115	-	117	104	47	54
28	66	-	95	-	112	79	43	50
29	60	-	81	-	85	-	39	45
30	48	-	74	-	59	-	36	41
31	40	-	54	-	36	-	33	36
32	32	32	32	32	32	32	32	32

Table 4.2

GRAPH 4.1


900
750
600
450
300
150
0
Page Faults
0
2
4
6
8
10
12
14
16
Real Storage Size (Pages)
L.R.U.
F.I.F.O.
RAND.
MIN.
MOD.

GRAPH 4.2


1200
1000
800
600
400
200
0
Page Faults
0
4
8
12
16
20
24
28
32
Real Storage Size (Pages)
L.R.U.
F.I.F.O.
RAND.
MIN.
MOD.

4.3 Binary Sequence Search Trees

4.3.1 Storage Layout

An obvious first attempt at storing and using binary trees in a paged memory is to keep the root (assumed at level 1), all nodes at level 2, ..., all nodes at level i on the same page for some i which will depend on the record length. The subtrees associated with each node at level i are then stored similarly. Figure 4.3 shows an example.

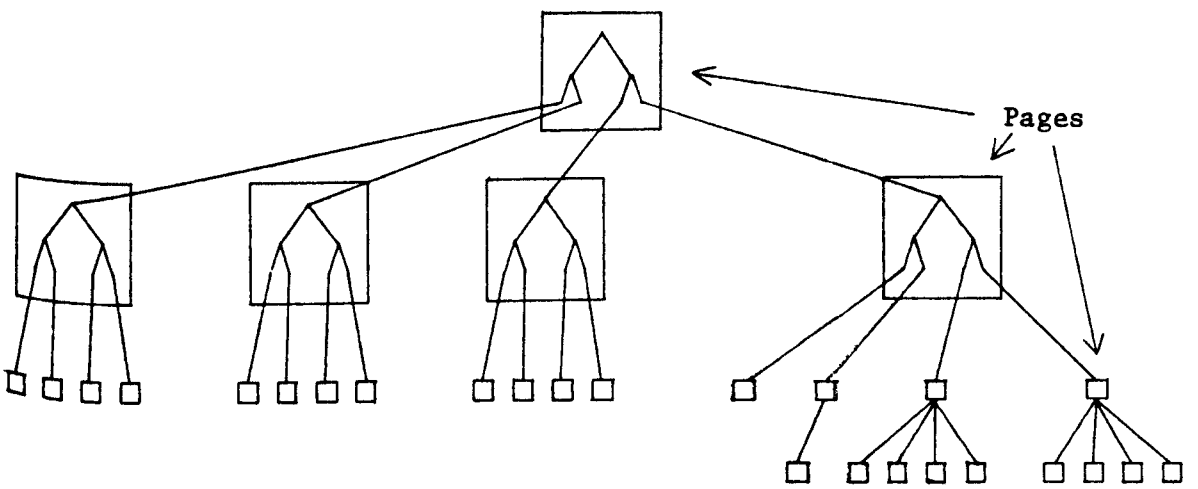


Fig 4.3

Let the binary tree itself be referred to as the data tree. If the suggested storage method is used, the pages containing the data tree have an n way tree structure imposed on them where each page constitutes a node. Let this tree be referred to as the page tree. Suppose there are b records per page and for simplicity $b = 2^{K+1}-1$ for some integer K, then $n = 2^K$. The data tree and page tree show up clearly in the example of figure 4.3 where K = 2 thus n = 4.

4.3.2 Low Frequency of Searches

Suppose firstly that the tree is accessed so rarely that all of its pages may be assumed to reside on backing storage when needed.

If the total file size is M pages, M may be written as $1 + n + n^2 + \ldots + n^j + \ell = (n^{j+1} - 1)/(n-1) + \ell$ for some integer j and $\ell < n^{j+1}$ since this is just the sum of the number of nodes at the various levels of the page tree. Nodes at level i require i page faults when accessed. Thus assuming a random request pattern over all pages of the page tree, the expected number of page faults for N searches of the data tree is:-

$$E = N\,(1 + n.2 + n^2.3 + \ldots + n^j.(j+1) + \ell.(j+2))/M$$

$$= N\,(\sum_{t=1}^{j+1} t.n^{t-1} + \ell.(j+2))/M$$

$$= N\,(\frac{(n-1)(j+2)n^{j+1} - (n^{j+2}-1)}{(n-1)} + \ell.(j+2))/M$$

In the example of figure 4.3, n = 4, M = 30, j = 2 and ℓ = 9. Thus $E = \frac{93N}{30}$ or roughly three page faults for each record located. The only consolation is that the minimum number of comparisons will be required.

Where there are a large number of records per page n may be large and there may exist only one level other than the root in the page tree. For example with $63 = 2^6 - 1$ records per page, n = 32 and a thirty page file, containing over 1800 records, can be stored with only one level other than the root. In this case n = 32, M = 30, j = 0 and ℓ = 29.

Thus $E = \frac{59N}{30}$ or approximately two page faults for each record located.

For large records i.e. where n is small, an alternative approach is to construct a data tree using detached keys with pointers to the associated data. Hopefully this tree structure would occupy just a few pages and have a high value of n. This would allow any record to be located with just two or three page faults.

Suppose the key field, together with the necessary pointers, is a fraction α of the length of the actual record. The tree constructed of these keys will have $\frac{b}{\alpha}$ records per page and occupy $\lceil M\alpha \rceil$ pages. If $\alpha = 0.1$ in the example of figure 4.3 then the keys can be contained in $\lceil M\alpha \rceil = 3$ pages. Thus using the expression E, the expected number of page faults required to locate a key is 1.67. Each search will require one more page fault to locate the data giving an expected number for N searches of 2.67N. Where α is small, allowing the tree of detached keys to occupy just a few pages, a very considerable saving may be possible using this technique. In the example of figure 4.3, for $0<\alpha\leq.034$ clearly $\lceil M\alpha \rceil = 1$ and only two pages will be referenced in retrieving any record.

4.3.3 High Frequency of Searches

Now consider the case where searches are made sufficiently frequently that file pages remain in main storage between searches and an ordinary non-detached key system is employed. A similar effect to that noted with binary searching will occur.

The root page of the page tree will be used most frequently and if 'sufficient' real storage is available this page will tend to become resident, exactly as was found with the centre page of a file

using binary searching and giving a similar drop in paging activity. Sufficient here means j + 1 i.e. the number levels because that is the number of pages usually required to complete a search. Most of the conclusions arrived at in considering binary searching will apply.

The advantages of constructing detached key trees are far greater when there is a high search frequency. The page tree will usually be considerably smaller than if the data were included (occupying only $\lceil M\alpha \rceil$ pages) and as a result only a small amount of real storage is necessary for it to become permanently resident. Suppose c, the amount of real storage available, satisfies $c > \lceil M\alpha \rceil$ then the pages containing the keys will tend to become resident (particularly with L.R.U.) and so at most one page fault per search will be necessary to locate the data. The probability that even this page fault is not necessary is clearly $\frac{c - \lceil M\alpha \rceil}{M}$.

4.4 An Optimum Search Technique

The proposed modification to the binary search technique and the detached key storage of trees are examples of mapping mechanisms which allow records to be located with fewer page faults. A better approach, for tree structures at least, is to use a fast simple mapping mechanism specifically designed to indicate merely which page of the file contains the desired record. The search procedure within the selected page can use any of the sophisticated techniques developed for conventional non-paged memories.

Coffman and Eve, 1970 suggest a tree structure constructed from transformed or 'hashed' keys rather than the keys themselves.

The advantages of this technique that they illustrate are not relevant here but the idea of hashing keys into a set of integers (representing pages), which would correspond to the first step in their algorithm, seems to fulfil the conditions for the mapping suggested above. If this mapping is used to set up the table then, during a search, the page on which a particular record lies can be determined directly from its key. Once again these ideas are related to the Estimated Entry technique.

The records mapped into a given page need bear no obvious relation to one another or can be closely related depending on the choice of transformation function. It might be desirable to map all records with some specific property into a given page for separate analysis later.

The proposed search technique is thus:-

(i) Acquire M pages of free virtual memory based on an estimate of the size the file will be.

(ii) Establish a suitable function to map the key range into the set of integers {1,.....,M}

(iii) Insert records into the file or locate them by computing the page number from the key and performing a normal binary search or any other conventional technique which suits the individual case, within the selected page.

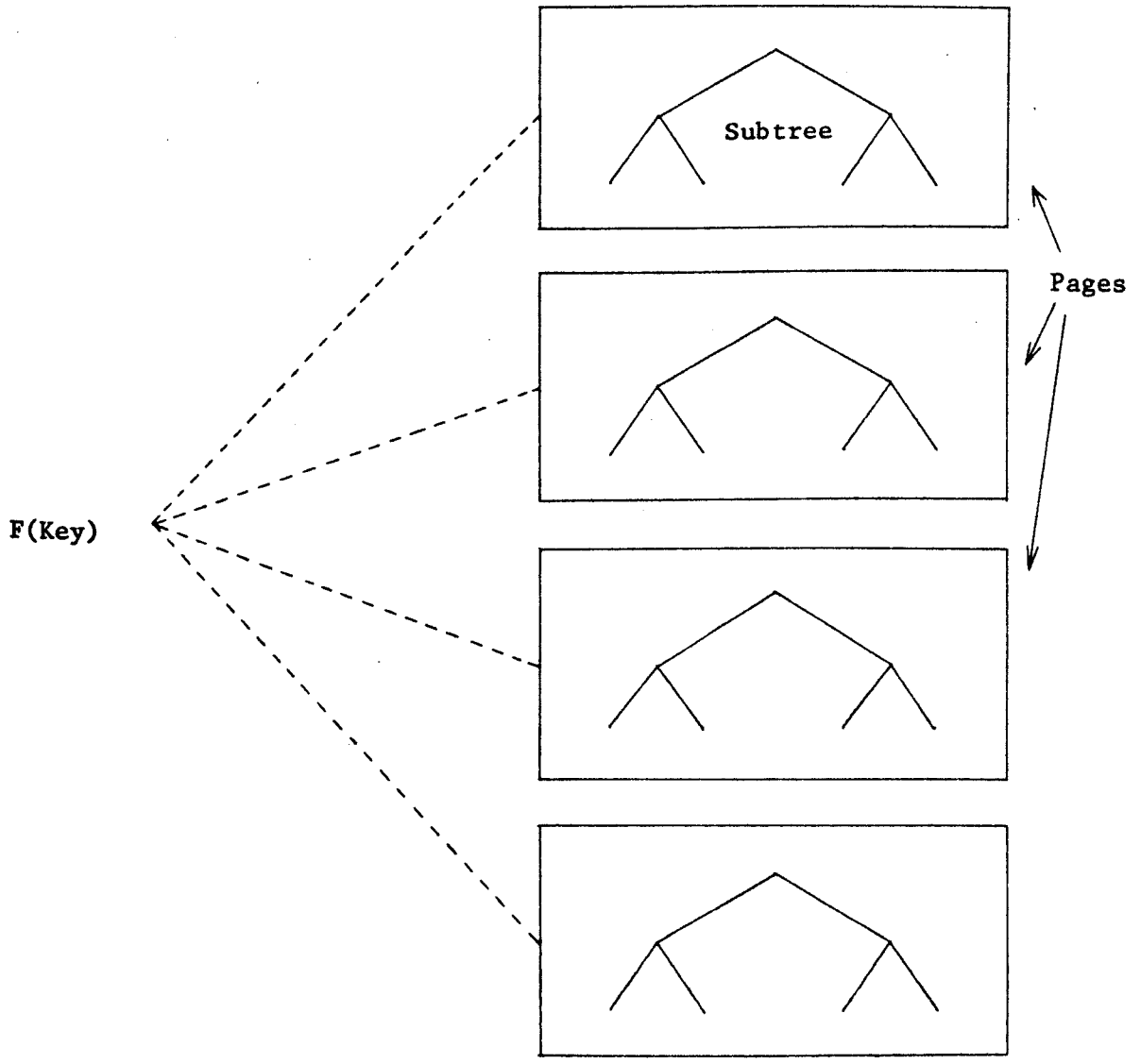


Fig 4.4

This may be thought of as a cross between hash tables and trees. The usual problem of dealing with hash table clashes is no longer a problem - it is intentional. The previously noted problem of having to reference several pages in searching a binary tree is no longer present - each tree is smaller than a page.

If a bad choice of M, the file size, is made it is not too serious. If a particular page becomes full another free page can be obtained and the tree or whatever structure is in use extended onto this page. Only on occasions when records contained in this page are required will it be referenced and this will only be via one other page. If several of these overflow pages become necessary then a serious error in estimating M was made. The file may have to be reorganised and a new mapping function defined.

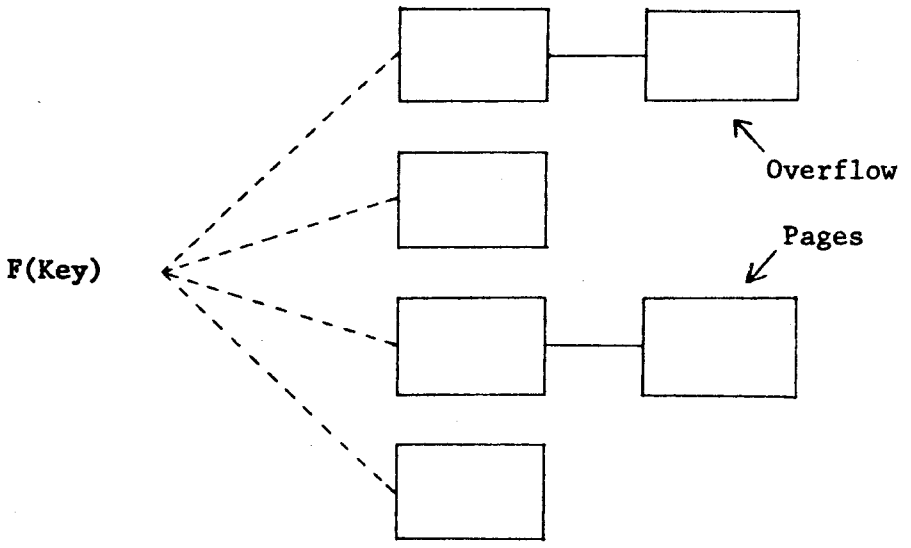


Fig 4.5

The adjective optimum is used with this technique in the context of a paged memory for several reasons. If searches are made infrequently only one page fault is caused each time. Where the access frequency is high, the technique works efficiently with any available quantity of real storage. In this case at most one page fault per search is caused and clearly there is a probability $\frac{c}{M}$ that no page fault will be necessary. No restrictions such as file ordering are made and insertions and deletions are fairly easy.

Minor errors in estimating M can be catered for and have very little effect on searching efficiency. Finally, no extra storage is needed for this mapping mechanism and none of the file is used unnecessarily frequently, as the root page is with a binary tree.

The basis of this proposed method is very similar to Hash Coding and many of the conclusions of chapter 3 apply. For instance if requests can be queued and ordered into page queues a significant reduction in paging can be achieved. In addition, if the request pattern is non-random the results of chapter 3 apply directly.

It must be observed that this search method is very similar to the "buckets" technique which is often used with external searching. The important aspect is the apparent lack of realization in general, that an external method is appropriate for a paged environment because a paging drum is an external storage device.

Chapter 5.

SORTING & MERGING

5.1 Introduction

The implementation of a conventional internal sorting algorithm for use on large files wholly located in virtual memory is not practical on paged machines. The necessarily non-sequential access pattern leads to excessive paging activity which is undesirable as shown in chapter 1.

This problem was considered by Brawn, Gustavson and Mankin, 1970 who proposed and analysed strategies for sorting in a paging environment. Basically, their approach is to divide the data set into convenient length sublists (integral number of pages or a fraction of a page) and to sort each of these by a conventional internal technique. As groups of sorted sublists become available they can be merged into one longer sublist. The algorithm terminates when only one sublist remains.

The work by Brawn et al shows that the use of sublists can produce a very significant reduction in paging activity but they limit their analysis to F.I.F.O. page replacement. In this chapter the use of sublists is assumed and a sorting algorithm is proposed to take advantage of the special features of a paged memory. Analysis is in terms of all of the page replacement policies described in chapter 1.

In practical sorting situations different types of files occur.

Records can vary in length between a few tens of bits and many hundreds of bytes although not all of the record need be used in the key. Three different techniques are usually employed to cater for the different cases:-

(a) Record sort. The sort algorithm is applied directly to the keys of the records in the stored file, treating each record as a unit whenever a transfer is dictated by the algorithm.

(b) Detached key sort. A table of keys and addresses is set up and this table is sorted, the records in the file are only re-arranged during the output phase.

(c) Nondetached key sort. A table of address pointers is set up and records examined via this table. The table is sorted into the order dictated by the record keys.

These alternatives have been considered in some detail by Brawn et al, 1970, and the various implementation methods which they discuss will not be pursued here.

In what follows, 'record' will mean the complete record in a record sort and key + pointer in a detached key sort. Nondetached key sorting is omitted because of its fundamentally different behaviour.

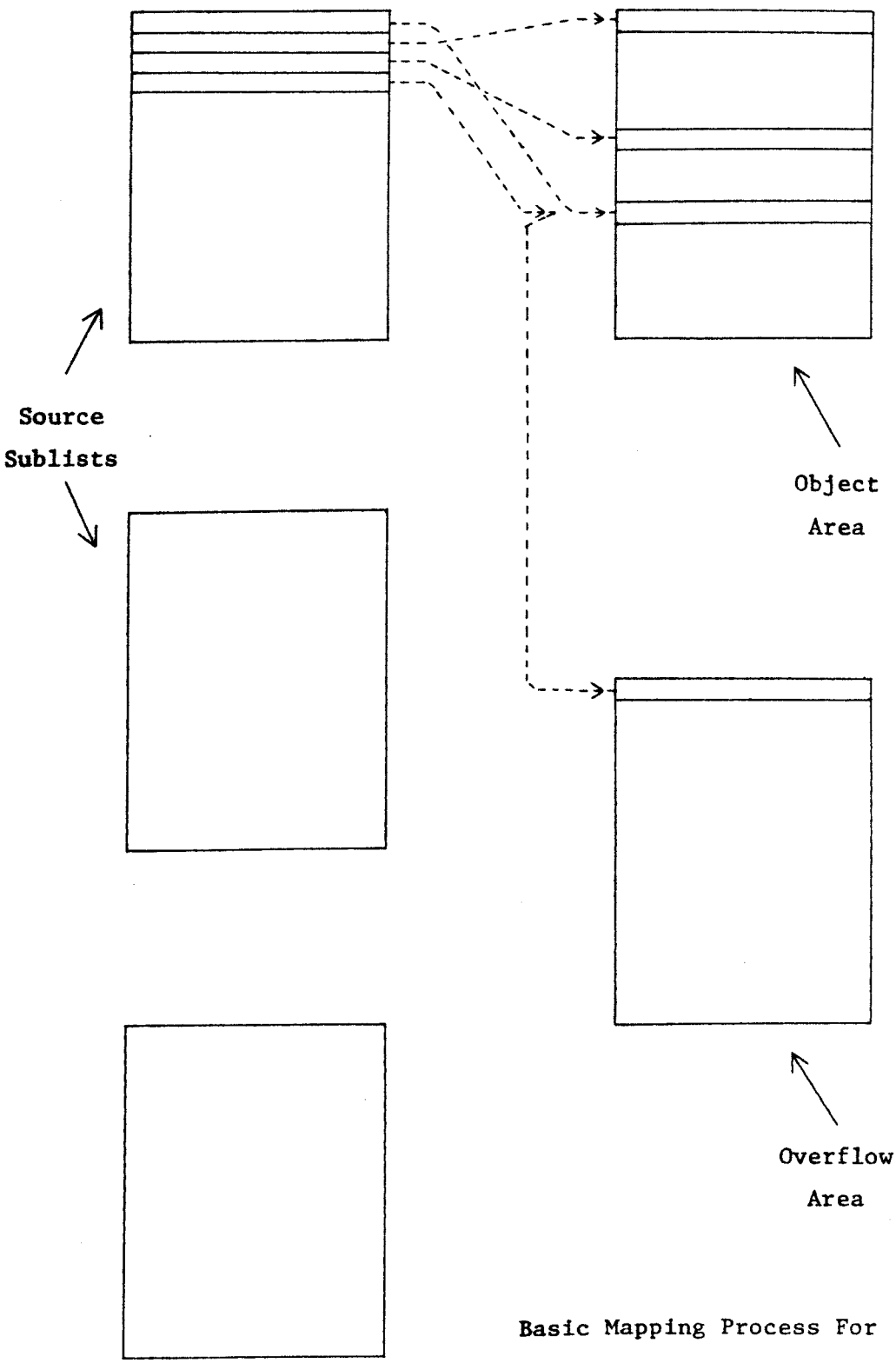
5.2 Basic Algorithm

In terms of almost all criteria, the most efficient internal sorting method for a set of data is that in which each key itself (or a simple function of it) indexes the location of the corresponding region in the store and the record can be directly "pigeonholed".

For most data no such exact pigeonholing is possible but some knowledge of the distribution of the keys in the data allows an approximate calculation of the address of each record in the sorted list. For this method, usually called address calculation, to be reasonably efficient excessive movement of the records once placed must be avoided and this leads to space requirements rather greater than the size of the data. This has rendered address calculation less attractive for non-paged systems than certain other sorting methods.

In a paging environment, however, reducing the amount of memory space used is not of primary importance (provided excessive paging is avoided) and so address calculation may be considered. If no attempt is made to resolve clashes, the most time consuming aspect of the algorithm is removed and since it is practical to keep the whole data set in virtual memory, the clashes can be easily concatenated onto the end of the data set for treatment later. The object area so produced will be fairly sparsely populated but if the modified algorithm executes quickly, this can be tolerated. Figure 5.1 illustrates the basic movement of the data.

For the analysis of the address calculation sorting method described, suppose that t records from a data set i.e. one sublist, are sorted into a working or object area k records long. The mapping may consist of any convenient transformation from the range of the keys to the set of integers 1 ....k. For example if $k = 2^r$ then the r most significant bits of the key would be relevant where the distribution of keys does not differ significantly from the rectangular.



Basic Mapping Process For
Modified Address Calculation.

Fig 5.1

Alphabetic keys can be treated identically provided the natural ordering of the letters is reflected in their internal coding e.g. as in EBCDIC.

Firstly it is necessary to determine just how sparsely filled the object area will be.

Lemma 5.1

Assuming each record can be mapped into any of the object area locations with equal probability, if t records are mapped into a region k records long the expected number of successful mappings is:-

$$k(1 - (1 - 1/k)^t)$$

Proof

Let $U_{y,k}$ = the number of records successfully mapped out of y. Consider mapping the $y+1^{st}$ record.

$$U_{y+1,k} = \begin{cases} U_{y,k} & \text{if } y+1^{st} \text{ record clashes.} \\ U_{y,k} + 1 & \text{otherwise.} \end{cases}$$

Clearly the $y+1^{st}$ record clashes with probability $U_{y,k}/k$. Thus:-

$$E(U_{y+1,k} | U_{y,k}) = U_{y,k}.U_{y,k}/k + (1 + U_{y,k})(1 - U_{y,k}/k)$$

$$= 1 + U_{y,k}(1 - 1/k)$$

Using the result $E(E(X|z)) = E(X)$ this becomes:-

$$E(U_{y+1,k}) = 1 + E(U_{y,k})(1 - 1/k)$$

Thus:-

$$E(U_{t,k}) = k(1 - (1 - 1/k)^t)$$

An alternative derivation can be obtained from the probabilities themselves. Define $q_{m,n}$= pr(n records have been successfully mapped after m records have been processed).

then $q_{m,n}$ satisfies:-

$$q_{m,n} = q_{m-1,n}.n/t + q_{m-1,n-1}(1 - (n-1)/t)$$

which may be solved as before and leads to the previous result for $E(U_{t,k})$.

The expression derived in Lemma 5.1 appears in several other contexts, in particular Morris, 1968.

The variance of the mapping efficiency can be similarly obtained.

Lemma 5.2

The variance of the number of records successfully mapped is given by:-

$$(1 - 1/k)^t(k(k - 1)(1 - 1/(k-1))^t + k - k^2(1 - 1/k)^t)$$

Proof

As in Lemma 5.1 it can easily be shown that:-

$$E(U^2_{y+1,k}|U^2_{y,k}) = (1 + U_{y,k})^2(1 - U_{y,k}/k) + U^2_{y,k}.U_{y,k}/k$$

and so:-

$$E(U^2_{y+1,k}) = 1 + E(U_{y,k})(2 - 1/k) + E(U^2_{y,k})(1 - 2/k)$$

and the result follows from the solution of this recurrence relation.

If t and k are sufficiently large a very convenient approximation can be made:-

$$(1 - 1/k)^t \simeq e^{-t/k}$$

Thus:-

$$E(U_{t,k}) \simeq k(1 - e^{-t/k})$$

$$var(U_{t,k}) \simeq e^{-t/k}(k(k-1)e^{-t/(k-1)} + k - k^2 e^{-t/k})$$

5.3 Modifications

Several modifications have been considered to try and improve the efficiency of the basic mapping without seriously affecting execution speed. The first two deal with the treatment of clashes and the third with a reduction in the space requirements. The fourth aims at reducing page references.

The basic algorithm makes no attempt to resolve clashes whereas there may well be a suitable empty slot close to the calculated address. The first modification is to store the clashes in an overflow area and when the whole source area has been mapped, attempt to insert each clash into the object area using a modified addressing function. This function is:-

$$\text{NEW ADDRESS} = \text{CALCULATED ADDRESS} \begin{cases} +1 \text{ IF CLASH KEY} > \text{OCCUPANT KEY} \\ -1 \text{ IF CLASH KEY} < \text{OCCUPANT KEY} \end{cases}$$

It is useful to know how many extra records will be mapped using this technique.

Suppose that after the initial mapping a total of n records have been successfully mapped into the object area which is of length k. The expected number inserted during the second pass is given by:-

$$\sum_{i=1}^{k-n} i \sum_{r=1}^{\min(n,k-n)} \sum_{\ell=\max(0,2r-(k-n))}^{r-1} \frac{{}^{r}C_{\ell}\ {}^{k-n-r-1}C_{k-n+\ell-2r}\ {}^{n+1}C_{r}}{{}^{k}C_{n}}$$

$$\sum_{j=\max(0,i-m)}^{\min(i,\ell)} {}^{\ell}C_{j}\ {}^{m}C_{i-j} \sum_{u=0}^{i-j} (-1)^{i-j-u}\ {}^{i-j}C_{u} \sum_{v=0}^{j} (-1)^{j-v}\ {}^{j}C_{v} \left(1 - \frac{\ell-v}{n} - \frac{m-u}{2n}\right)^{n'}$$

where $m = 2(r - \ell)$ and $n' = k - n$.

This is derived in the Appendix.

There does not appear to be any obvious way of simplifying this expression as there was with the results of Lemmas 5.1 and 5.2 It is not even possible to use integrals to approximate the summations because of the great variability of the summation limits.

The basic mapping process was simulated and the results together with the predicted values from Lemmas 5.1 and 5.2 are shown in Table 5.1. Two hundred simulations were performed for each value of t and the random assumption stated in Lemma 5.1 was made.

The modification to the basic process was also simulated for various values of t and the results are shown in Table 5.2. Again, two hundred simulations were made for each value of t and for each simulation the object area was pre-loaded with the observed average number of records mapped by the basic process for that value of t.

TABLE 5.1

Number in Source Area	Size of Object Area	Mean Number Mapped	Calculated Mean	Observed Variance	Calculated Variance
128	1024	120	120.38	6.72	6.46
256	1024	226	226.60	22.14	21.09
384	1024	320	320.34	40.00	38.66
512	1024	402	403.06	62.18	56.01
640	1024	476	476.06	71.92	71.36
768	1024	540	540.47	85.38	83.86
896	1024	597	597.32	101.92	93.24
1024	1024	648	647.48	111.90	99.56

TABLE 5.2

Number in Source Area	Size of Object Area	Number Mapped in First Pass	Number Mapped in Second Pass	Total Number Mapped
128	1024	120	7.955	128
256	1024	226	22.67	250
384	1024	320	43.00	363
512	1024	402	62.38	464
640	1024	476	76.61	553
768	1024	540	90.37	630
896	1024	597	99.72	697
1024	1024	648	107.28	755

Evaluation of the expression relating to the second pass, for significant values of t and k has proved difficult because of its inherent complexity. The amount of processor time involved is excessive because of the number of nested summations, and accuracy is doubtful because of the occurrenee of ${}^{i}C_{j}$ with very large values of i and j. Even if great care is taken in the way that the expression is evaluated rounding errors are considerable.

The second modification is similar to the first but attempts to resolve clashes as they occur rather than storing them. If the calculated address is found to be occupied, the modified address proposed above is examined immediately. If this location is also occupied the record is transferred to the overflow area.

An analytic study of the properties of this modification was attempted using the assumption that successfully mapped records are randomly distributed over the object area, but this is incorrect. In fact there will tend to be clumping of records because the empty cells adjacent to an occupied cell have double the probability of being filled by each record that is mapped. A similar point in a different context appears in Morris, 1968. Flores, 1960 appears to base his analysis on this assumption but its incorrectness may well only have a second order effect because his theoretical results are in approximate agreement with his simulation studies.

Theoretical analysis seems difficult if this simplifying assumption cannot be made and so this modification has only been studied by simulation. Table 5.3 shows the simulation results and graph 5.1 compares all three suggested mappings. Notice that as the ratio of source area size to object area size increases rather

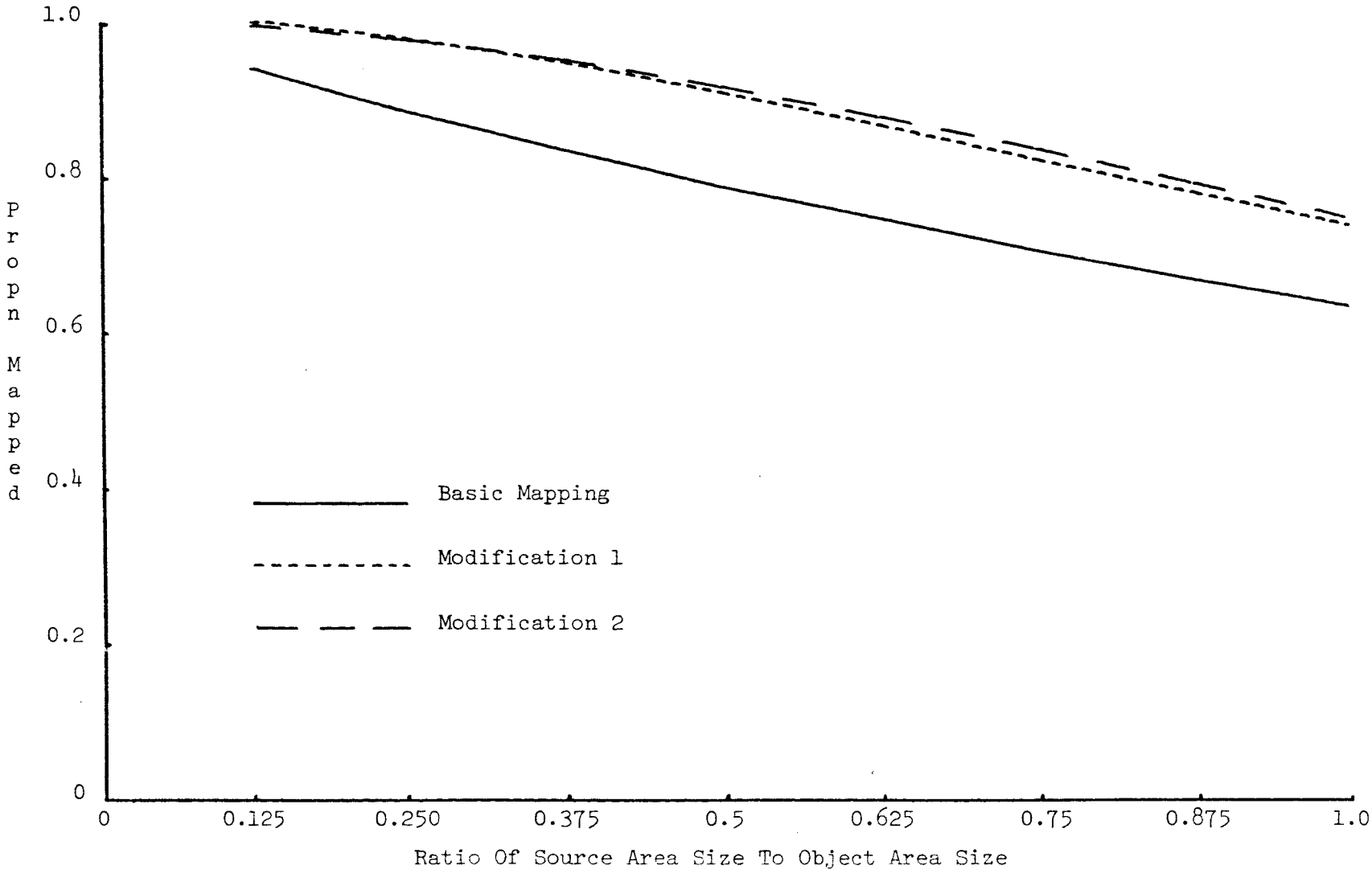
less records are mapped directly than are mapped by the basic process. This is because for each record mapped, the object area will be more fully occupied using this modification than when using the basic process.

Number in Source Area	Size of Object Area	Mean Number Mapped Directly	Mean Number into Adjacent Cells	Mean of Total Number Mapped.
128	1024	120.125	7.3	127.4
256	1024	224.2	25.4	249.6
384	1024	314.0	50.4	364.4
512	1024	389.31	78.62	467.9
640	1024	454.29	105.4	559.7
768	1024	506.9	133.7	640.6
896	1024	550.7	157.0	707.7
1024	1024	584.85	179.66	764.5

TABLE 5.3

A natural extension of these modifications is to examine several locations, not just one, on the side of the calculated address dictated by the rank of the clashing record. If movement of data is necessary to fit a record in then it is placed in the overflow area, otherwise in the object area. This rather more complex process was simulated and rather surprisingly found to be only a very small amount more efficient in terms of the mean number successfully mapped than the previously described mappings.

G R A P H 5 . 1


1.0
0.8
0.6
0.4
0.2
0
Propn Mapped
Basic Mapping
Modification 1
Modification 2
0
0.125
0.250
0.375
0.5
0.625
0.75
0.875
1.0
Ratio Of Source Area Size To Object Area Size

Thus it was discarded as being of very little benefit for a fairly heavy cost.

During the mapping processes described previously records are transferred from the source area to the object and overflow areas. If the initial source sublist contains t records then at most t records will be located in source and object areas during the mapping, and the rest of the space will in effect be wasted. The third modification is designed to include the source area within the object area. They will in fact be exactly the same space if t = k.

The algorithm consists basically of placing a record at the calculated displacement within the source area and using the displaced record as the new source record. Two flags for each record are necessary to distinguish between mapped and unmapped records. Figure 5.2 illustrates the basic movement of data.

A slight problem arises if there is no source record at the calculated address. The algorithm maintains a pointer, initially at the beginning of the source area, which scans forward when necessary until it locates an unmapped source record. When the pointer passes the top of the source area the sublist is completely processed.

An alternative solution to this problem is suggested by a clash resolution technique used in hash coding (Morris, 1968). If the calculated hash address is occupied, the program scans from that address through the table looking for an empty slot. In the sorting case, if the calculated address no longer contains a source record but is within the source area, the program scans forwards from the calculated address until it locates a source record, wrapping around from the end to the beginning of the source area if necessary.

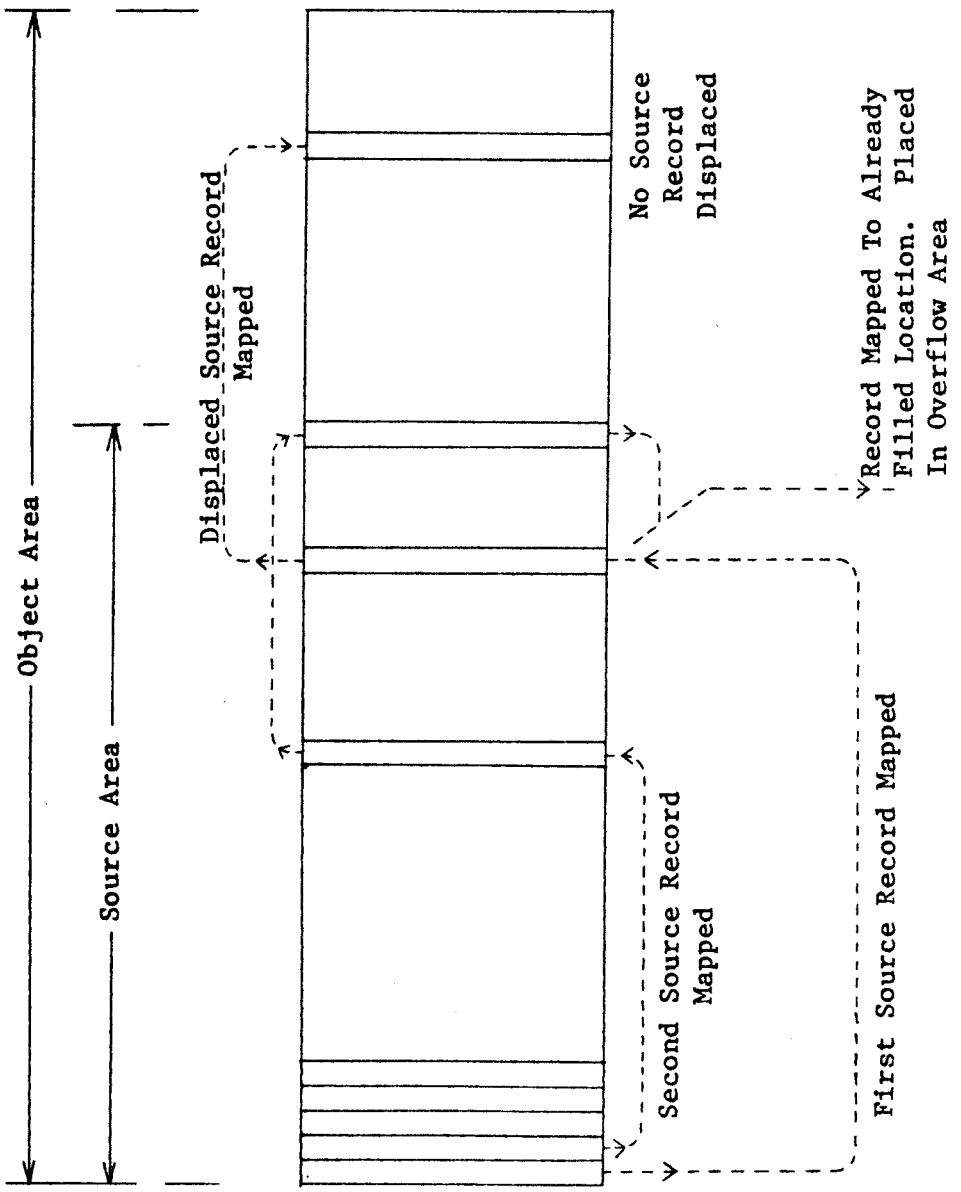

Object Area
Source Area
Displaced Source Record Mapped
No Source Record Displaced
Record Mapped To Already Filled Location. Placed In Overflow Area
Second Source Record Mapped
First Source Record Mapped

Fig 5.2

Clearly this method becomes less and less efficient as the source sublist empties and in fact the same location may be examined more than once. It is fairly easy to show that the use of a single pointer as described initially is the more efficient method in terms of the average scan length for $t > 7$.

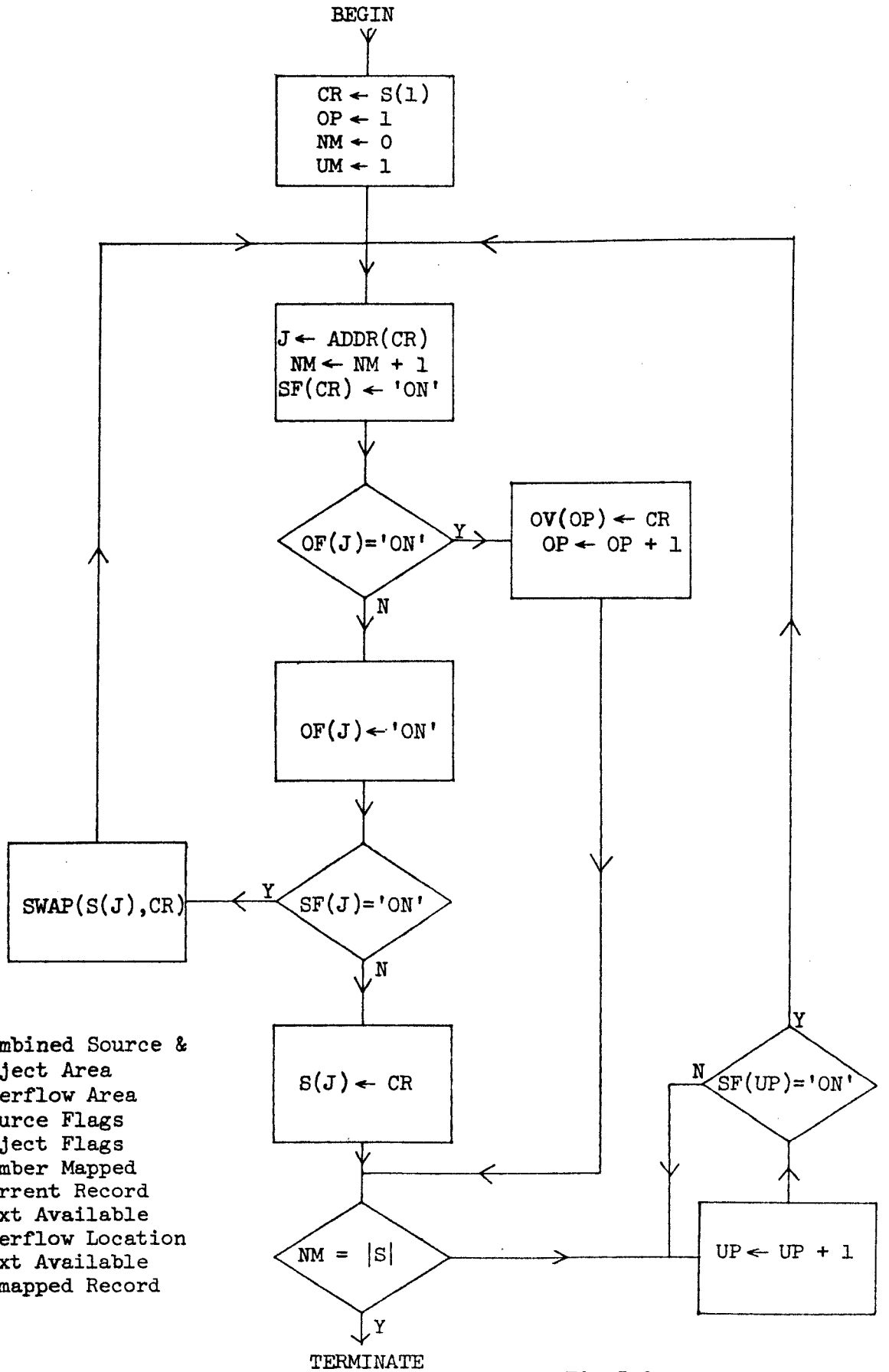
This modification is shown in detail in figure 5.3

Although initially it appears attractive this modification is probably not worth further consideration. Its only advantage is that it reduces by one page the main memory requirements for efficient execution (see defn. of data working set) and this is at a cost of two flags per record and the delays entailed in their examination.

The fourth modification which is proposed is the use of bit patterns to represent the state of the object area.

A bit string with one bit for each possible record location in the object area is established and each bit is set to zero. As a record is mapped into the object area the corresponding bit is switched to one. To check whether a particular address within the object area is already occupied it is only necessary to look at the corresponding bit. In this way, if a clash occurs and the object page concerned is not in real storage, it need never be referenced.

Once again this seems to be a worthwhile idea. There is no reduction in virtual space requirements but processing a record now always involves referencing two data pages rather than sometimes three. However, unless a bad choice of the algorithm's parameters (see below) has been made, it will be operating so that the saving in page references has negligible effect. In addition, the cost of



S Combined Source & Object Area
OV Overflow Area
SF Source Flags
OF Object Flags
NM Number Mapped
CR Current Record
OP Next Available Overflow Location
UP Next Available Unmapped Record

Fig 5.3

manipulating the bit patterns is considerable, especially on System 360 and System 370.

This modification does make paging analysis somewhat simpler and if required the performance to be expected can easily be derived from the results presented in section 5.4.

5.4 Paging Analysis

The paging characteristics of both parts of this algorithm are intuitively clear in that as the amount of real memory increases the paging decreases, rapidly at first and very slowly after a certain point. This is confirmed by simulation which also gives estimates of the actual number of page faults to be expected. However, analytic studies are worthwhile since they help to reveal the exact causes of excessive paging and the detailed differences between the different replacement algorithms, even if exact predictions of the number of page faults cannot be made. Although the sort technique described is really quite simple, it has proved sufficiently complex to defeat analytic study in certain cases.

Define M to be the total file size in pages, and β to be the expected value of the proportion of a source sublist successfully mapped into an object area. β depends on which mapping is being used and the parameters involved. The value of β for the basic mapping process was derived in lemma 5.1.

The two separate parts of this algorithm, the sort phase and the merge phase have entirely different properties and will be analysed separately.

5.4.1 The Sort Phase

Define the working set of data (W.S.D.) to be the total object area, one source page and one overflow page. Clearly if the program can maintain its W.S.D. in main memory it can execute for relatively long periods without causing a page fault. Suppose that there are c real memory pages available for data should the program require them, then there are three cases to consider:-

(i) $|W.S.D| < c$

(ii) $|W.S.D| = c$

(iii) $|W.S.D| > c$

where $|W.S.D|$ is the size of the W.S.D. in pages. Each case must be considered separately for each replacement algorithm.

5.4.1.1 L.R.U. Page Replacement.

Suppose the source sublist length is t' pages, the object area size is k' pages and there are b records per page (then $k = k'b$, $t = t'b$ and $|W.S.D.| = k' + 2$).

Consider the first case, $|W.S.D.| < c$.

The first page of the first source sublist will be loaded as it is referenced followed by the various pages of the object area and the overflow page as they are required. This will generate $2 + k'$ page faults. As $2 + k' < c$ some further pages may be referenced and moved into main memory each generating only one page fault until the total reaches c. These additional pages will be the second and subsequent pages of the source sublist (if $t' > 1$) or new overflow pages.

As soon as the available main memory is full, references to addresses located on the backing store will cause one of the main memory pages to be paged out.

Clearly the program's W.S.D. will have been referenced most recently. If full L.R.U. is being used, loading a new source or overflow page will displace a page which the program is no longer using since $|W.S.D.| < c$. Similarly when changing sublists, although almost a complete change of the W.S.D. is necessary, each new page required will only generate one page fault.

The file is of length M pages but the overflow pages which are produced effectively constitute an extension of the source file. On average, $M(1 - \beta)$ overflow pages are produced during processing of the original file and of course this quantity is not generally an integer.

Processing the overflow pages produces more overflow pages, and so on, and thus the effective length of the source file upon which the algorithm operates is:-

$$M_s = M + M(1 - \beta) + M(1 - \beta)^2 + \ldots\ldots + M(1 - \beta)^n$$

$$= \frac{M(1 - (1 - \beta)^{n+1})}{\beta}$$

where n depends on the criterion used to terminate the sort phase. The simplest and most obvious of these is termination when the length of the unprocessed part of the source file is less than one page. This short remaining sublist can be sorted using a more conventional algorithm. Clearly the final sublist which is processed may not

involve t' pages but for simplicity it will be assumed that all k' object pages are used with this sublist.

The sort phase simulation and the paging analysis which follows use this criterion, and clearly n is determined by:-

$$\begin{array}{l}\text{Fraction of a page left}\\ \text{unprocessed by final sublist}\end{array} + \begin{array}{l}\text{Overflows produced}\\ \text{by final sublist}\end{array} < 1$$

i.e.

$$\frac{M(1-(1-\beta)^{n+1})}{\beta} - \left\lfloor\frac{M(1-(1-\beta)^{n+1})}{\beta}\right\rfloor + M(1-\beta)^{n+1} < 1$$

and n is usually small, particularly if the 'best' mapping is used in which case $\beta \geq 0.75$.

Each source page will be part of a source sublist but only has to be brought into real storage once, thus $\lfloor M_s \rfloor$ page faults will be produced by loading the source sublists.

A fairly obvious modification to make is to use the source area just released as a part of the new object area when changing source sublists. This implies that one of the pages of the new object area might be in main memory as processing of a new sublist begins. This page will be either the second or third most recently referenced page in main memory depending on whether the last record was successfully mapped or not. Since $k' + 2 < c$ by definition and it is necessary to load only one new source page plus $k' - 1$ new object pages, a total of k', the number of undisturbed pages in main memory is at least three. Thus the previously used source page will remain in main storage.

Loading the constituent pages of the first object area will produce k' page faults and using the suggested modification, each

subsequent object area will be loaded with $k' - 1$ page faults. Hence the total for the object areas is:-

$$\lceil \lfloor M_s \rfloor / t' \rceil (k' - 1) + 1$$

The total number of overflow pages used will be $\lceil M_s - M \rceil$, one of which will be the partially empty overflow page remaining after the algorithm terminates. Each of these will only cause one page fault when loaded giving a total of $\lceil M_s - M \rceil$.

Thus the entire sort phase will produce:-

$$\lfloor M_s \rfloor + \lceil \lfloor M_s \rfloor / t' \rceil (k' - 1) + 1 + \lceil M_s - M \rceil$$

page faults in the case $|W.S.D.| < c$.

Consider as examples, a twelve page file ($M = 12$) sorted using sublist lengths of 1, 2, 3, 4 and 6 with source area and object area of equal size, and using the least efficient mapping ($\beta \simeq 2/3$). M_s is then 17.8 and the numbers of page faults predicted by this expression for the various sublist lengths are:-

t'	Page Faults.
1	24
2	33
3	36
4	39
6	39

If $|W.S.D.| << c$ it is possible that more than one completely processed source page will still be resident in real storage at a

change of sublist. This would slightly reduce the total number of page faults, the reduction increasing with c, and this fairly small effect is observed in simulation studies (see section 5.4.2).

Now consider the second case $|W.S.D.| = c$ i.e. $k' + 2 = c$. Surprisingly, for this value of c, there is a small but significant rise in the paging activity compared with the case $k' + 2 < c$. The reason for this effect is discussed below.

Suppose the letters S, O and V are used to denote a source area page, an object area page and overflow page respectively. Let strings of these letters represent main memory contents, the order from left to right being the order of most recent use. Thus the letter at the extreme right represents the most recently used page, and with L.R.U. replacement, a page fault is represented by the letter at the extreme left of the string being displaced and a new letter appearing at the extreme right.

Possible main memory arrangements prior to a change of overflow page are:-

O ..... O V S O

O ..... V O S O

.

.

.

.

V ..... O O S O

In all but the last case, a reference to the new overflow page, V', will displace an object area page. Before V can leave main storage it is necessary for it to become the least recently used page in memory. Suppose at any time before being paged out, it is j^{th}

most recently referenced and that immediately following the first reference to V' it is ℓ^{th} most recently referenced. For example in the first memory state shown above $\ell = 4$ since S, O and V' will have been referenced more recently than V. V will have been displaced by V' in the last memory contents shown and clearly in that case ℓ has no value.

Each record sorted will have one of three possible effects:-

(i) increase j by 1 by referencing an object page in main memory which is to the left of V in the string i.e. less recently used than V, and hence not cause a page fault.

(ii) increase j by 1 by referencing the object page which is not in main storage and hence cause a page fault.

(iii) reference an object page to the right of V i.e. more recently used than V and not cause a page fault or an increase in j.

If it is assumed that the data is randomly distributed, each object page is equally likely to be selected for each record mapped and so:-

$$\text{probability((i))} = (c - j)/k'$$

$$\text{probability((ii))} = 1/k'$$

$$\text{probability((iii))} = (k' - c + j - 1)/k'$$

Let Y_j denote the number of page faults occurring as j increases by one.

Clearly, Y_j can only be 0 or 1.

$$pr(Y_j = 1) = pr(\text{sequence of type (iii) events of any length occurs followed by a type (ii) event.})$$

$$= \sum_{n=0}^{\infty} (1 - (c - j + 1)/k')^n \, 1/k'$$

$$= 1/(c - j + 1)$$

and $$pr(Y_j = 0) = (c - j)/(c - j + 1)$$

thus $$E(Y_j) = 1/(c - j + 1)$$

The expected number of page faults between V becoming full and its displacement from storage is:-

1	initial reference to V'
$+ E(\sum_{j=\ell}^{c-1} Y_j)$	making V the least recently used page
+ 1	finally displacing V

$$= 2 + \sum_{j=\ell}^{c-1} E(Y_j)$$

$$= 2 + \sum_{j=\ell}^{c-1} 1/(c - j + 1)$$

$$= H_{k'-\ell+3} + 1 \qquad \text{since } c = k' + 2$$

Possible main memory contents prior to a change in S are:-

```
O ..... O S O V
O ..... O V S O
        .
        .
        .
        .
V ..... O O S O
```

The expression derived for a change of V may be used for a change in S except that the only values for ℓ are 3 and 4.

It was shown previously that for the case $|W.S.D.| < c$ only one page fault is induced by a change in V or S but in the case being considered i.e. $|W.S.D.| = c$, more than one page fault will usually occur. This is part of the reason for the increase in paging activity as c drops to the size of the working set of data.

Exactly which state main memory is in prior to a change in S or V will depend on the actual data. An upper bound on the total number of page faults could be derived by assuming the worst possible case i.e. k' and $k' + 1$ page faults in changing V and S respectively, which will occur very rarely. A more realistic figure can be obtained by assuming that the highest expected value occurs each time. For simplicity this is taken as $H_{k'-1} + 1$ for V i.e. $\ell = 4$ and $H_{k'} + 1$ for S i.e. $\ell = 3$.

Changing sublists is a rather more complex process than it at first appears. At least k' page faults will occur because a new object area is being used and only one of its pages (the discarded source page) will be located in real storage. In addition, a new source page will be referenced.

These k' page faults will be sufficient only if each one causes the displacement of one of the pages of the previous object area. However, at the beginning of the processing of a new sublist, clashes are extremely unlikely to occur, and it is very possible that all of the k' pages of the new object area will be referenced before the overflow page. As a result the overflow page may be displaced and $k' + 1$ page faults will have to occur before the new W.S.D. is located in real storage. Several memory states can occur which lead to more than $k' + 1$ page faults, though these are less likely. To cater for these various situations, it is assumed that on average $k' + 1$ rather than k' page faults occur when a sublist is changed.

Assuming that discarded source pages are used as part of the new object area as before, an estimate of the number of page faults induced is:-

c	Initially loading real storage with the first W.S.D. Recall
$+$	$c = k' + 2$ in the case being considered.
$\{\lfloor M_s \rfloor - \lceil \lfloor M_s \rfloor / t' \rceil\}(H_{k'} + 1)$	Loading those source pages which are not loaded at the time that sublists are changed. Recall
$+$	that there are $\lfloor M_s \rfloor$ source pages and $\lceil \lfloor M_s \rfloor / t' \rceil$ sublists.
$\{\lceil \lfloor M_s \rfloor / t' \rceil - 1\}\{(k' - 1) + 1 + 1\}$	Changing sublists. When a sublist change is necessary,

$k' - 1$ new object pages which are not in real storage, and one source sublist page, are referenced.

+

$\{\lceil M_s - M \rceil - 1\}(H_{k'-1} + 1)$ Changing overflow pages. There are $\lceil M_s - M \rceil$ overflow pages involved of which one was brought into real storage when the c real page frames were originally loaded.

$$= \lceil \lfloor M_s \rfloor / t' \rceil (k' - H_{k'}) - H_{k'-1} + \lfloor M_s \rfloor (H_{k'} + 1) + \lceil M_s - M \rceil (H_{k'-1} + 1)$$

Consider the previously defined examples (M = 12 etc). The numbers of page faults predicted by this expression for the various sublist lengths are:-

t'	Page Faults.
1	40
2	58
3	68
4	77
6	86

For the case $|W.S.D.| > c$, the large number of possible states before a page change and the complexity of the associated transition matrix make it necessary to rely on simulation studies.

5.4.1.2 Random Replacement

For simplicity in consideration of random replacement, it is assumed that the sublist changes and changes of overflow pages do not overlap or interfere with each other.

Firstly consider the case $|W.S.D.| < c$ i.e. $k' + 2 < c$. When only one new page is required e.g. changing an overflow or source page, the probability of displacing one of the remaining members of the working set of data is $(k' + 1)/c$. This will produce a further page fault when the displaced page is required again. Clearly several page faults may be induced in changing a single page of the W.S.D. before all of the modified W.S.D. is in main memory. This number of page faults has a geometric distribution with parameter $1 - (k'+1)/c$ and the expected value is:-

$$\frac{1}{1 - (k'+1)/c}$$

$$= \quad c/(c - k' - 1)$$

The worst situation in the case being considered is $c = k' + 3$ when this expected value is $(k' + 3)/2$.

When a source sublist and object area change is required the problems are considerably worse because most of the W.S.D. has to be replaced, but the random algorithm will not always choose members of the old W.S.D. to displace. Initially two pages of the new W.S.D. will be in main memory. They are the current source page, which will become part of the new object area, and the current overflow page. Let Z_i be the number of page faults necessary to increase the number of new W.S.D. pages in main memory from i to i + 1.

Z_i has a geometric distribution with parameter $(c - i)/c$. Note that once i pages of the new W.S.D. have been loaded into main memory it is not possible for this number to decrease although the constituent pages may change. So the total number of page faults induced by changing W.S.D.'s is $\sum_{i=2}^{k+1} Z_i$. This has expected value:-

$$E\left(\sum_{i=2}^{k+1} Z_i\right) = \sum_{i=2}^{k+1} E(Z_i)$$

$$= \sum_{i=2}^{k+1} c/(c - i)$$

$$= c \sum_{j=c-k-1}^{c-2} 1/j \qquad \text{where } j = c - i$$

$$= c(H_{c-2} - H_{c-k-2})$$

where H_i denotes the i^{th} Harmonic number.

An expression for the expected value of the total number of page faults can now be derived. Again it is necessary to work in terms of average values and hence only to be able to obtain an approximate result.

Since $c - |\text{W.S.D.}|$ is an unspecified quantity in the case being considered, the exact state of the algorithm when the available real storage becomes full is unknown. This can be derived of course but the expressions involved are extremely lengthy and the algebra is tedious. In the paging analysis which follows it is assumed that real storage becomes full once the first W.S.D. has been loaded. There is in fact no logical difference between the two cases $|\text{W.S.D.}| < c$ and $|\text{W.S.D.}| = c$ when random page replacement is used and the 'jump' in

the number of page faults which was observed with L.R.U. replacement will not occur. Thus the results presented below apply to both cases but because of the above mentioned assumption the expressions become increasingly inaccurate as c increases.

The total number of page faults induced will be approximately:-

c	Initially loading real storage
$+$	with the first W.S.D.
$(\lfloor M_s \rfloor - \lceil \lfloor M_s \rfloor / t' \rceil).c/(c - k' - 1)$	Loading those source pages which are not loaded at the time
$+$	that sublists are changed.
$(\lceil \lfloor M_s \rfloor / t' \rceil - 1)\{c(H_{c-2} - H_{c-k'-2})\}$	Changing sublists.
$+$	
$(\lceil M_s - M \rceil - 1).c/(c - k' - 1)$	Changing overflow pages.

Consider once again the previously defined examples (M = 12 etc). The numbers of page faults predicted by this expression are:-

$t' \backslash c$	3	4	5	6	7	8	9
1	69	48	42	-	-	-	-
2		104	71	60	-	-	-
3			126	83	69	-	-
4				158	102	84	-
6						167	105

Recall that $t' = k'$ in this example and that for the sort phase $|W.S.D.| = k' + 2$.

These results appear in more detail in tables 5.9 - 5.13.

For the case |W.S.D.| > c simulation studies are relied upon but it is intuitively obvious that the increase in paging activity will be considerable.

5.4.1.3 F.I.F.O. Page Replacement.

Again the F.I.F.O. replacement algorithm is considerably more difficult to analyse than either L.R.U. or Random because the order of loading of pages into main memory varies to a great extent with the actual data being used. Simulation studies are therefore relied on to provide information on the paging characteristics of the algorithm using F.I.F.O. replacement for all three cases.

5.4.1.4 The MIN Algorithm

Firstly consider the case |W.S.D.| < c. The L.R.U. algorithm does not make many wrong choices of which page has to be displaced because the working set of data is the most recently used set of pages.

However MIN is able to make the best possible use of the c - |W.S.D.| extra pages available and thus one would expect the number of page faults to be less than the number obtained with L.R.U., and to decrease monotonically as c increases.

For the case |W.S.D.| = c, MIN is always able to displace the page which is no longer needed when a page fault occurs but there are no extra pages of which it can take advantage. L.R.U. was expected to behave like this in the case |W.S.D.| < c, especially as

the c - |W.S.D.| extra pages were for the most part ignored in that analysis. Thus the number of page faults which MIN will produce in this case may be expected to be the number predicted for L.R.U. in the case |W.S.D.| < c.

As for all the other replacement policies, the amount of paging increases very rapidly for values of c such that |W.S.D.| > c.

5.4.2 Simulation Of The Sort Phase

In order to check on the expressions derived in section 5.4.1 and to provide data where theoretical analysis was not possible, the sort phase was simulated.

Clearly the number of variable quantities prohibits simulation of all possible or even desirable cases. Further problems are that execution of the simulation program requires considerable C.P.U. time and several simulations with a given set of parameters are required in order to obtain a reliable estimate of an unknown quantity (e.g. amount of paging). To overcome these difficulties, specific values of parameters were selected and hopefully they are typical. In addition, only a limited number of simulations were carried out for each parameter set and the results serve only to indicate the order of magnitude of the unknowns.

The simulation was intended to reflect fairly accurately the actual processes involved in the sort phase. A typical main memory management system was modelled and the sorting using only the basic mapping was actually carried out on randomly generated data sequences. The modification involving reclamation of recently released source pages was incorporated.

Page reference strings were output for latter use by programs simulating different page replacement algorithms with various amounts of real memory.

As observed elsewhere, the problems of simulation with random page replacement are even greater, since the same page reference string can exhibit different amounts of paging with different initial values for the random number generator used in the page replacement simulator. Each page reference string produced by the sort simulation program was subjected to random page replacement using three different initial values for the random number generator.

The random number generator used throughout is that described by Lewis et al, 1969 which is claimed to be designed especially for I.B.M.'s System 360. Some difficulty was experienced with some of the initial values for this generator and so all of the simulation results included here were obtained using starting values from the "Million Random Digits" produced by the Rand Corporation.

The file size used was M = 12 pages with b = 128 records per page. The five values 1, 2, 3, 4 and 6 were used for t', the length of the source sublist in pages, and the ratio of t' to k' was set at unity. Each set of parameters was simulated with three different initial values for the random number generator and amounts of main memory between 2 and 12 pages.

Tables 5.4 - 5.8 show the simulation results for L.R.U. page replacement together with the values predicted by the expressions derived in section 5.4.1.1. Graph 5.2 shows the average page fault values for each value of t'.

The results shown in tables 5.9 - 5.13 and graph 5.3 are for Random page replacement.

For each set of sort algorithm parameters, only the average of the three random paging simulations is shown.

Tables 5.14 - 5.18 and graph 5.4 show the results obtained using F.I.F.O. replacement, and the performance of the MIN algorithm is shown in tables 5.19 - 5.23 and graph 5.5.

Graphs 5.6 - 5.8 compare the performances of the different replacement policies for three of the source sublist lengths considered.

The most important thing to observe in these results is the very rapid increase in page faults as the amount of main memory (i.e. c) drops below |W.S.D.|. This is observed with all programs but rarely is the effect so pronounced as in this case, where with c = |W.S.D.| - 1, there is an increase of more than an order of magnitude over c = |W.S.D.|. This emphasises the need for accurate prediction of system loading so that c can be estimated, or dynamic variation of program parameters during execution. The advantage of saving some data movement during the merge phase by setting t' (and hence |W.S.D.|) at a high value is vastly offset by the paging activity which results if |W.S.D.| > c.

With L.R.U. page replacement, the expected slight decrease in paging as c becomes very much greater than |W.S.D.| is observed for all values of t' but is not included in the analytic results. Also, the slight increase in the number of page faults as the available real storage drops to |W.S.D.| is observed to occur.

An unexpected result which is also as yet unexplained is the amount of paging in the cases where the source sublist length i.e. t', is 3 and 6 pages with |W.S.D.| > c.

Observing the results for the other values of t', the amount of paging for a given value of c increases as t' increases, except in these cases. This shows up quite clearly in graph 5.2.

In examining the results of random page replacement, it is a little strange initially to come across situations where the amount of paging increases as main memory size increases by one page. This can happen of course where, by chance, random page replacement makes very good choices of which pages to replace at one memory size, but very poor choices at the next.

Again the most striking and important feature of these results is the dramatic paging increase as the amount of main memory available drops below the size of the W.S.D. However as expected, random page replacement is able, in a sense, to make use of large amounts of main memory in that it then has less probability of making incorrect choices of which pages to displace. Thus the amount of paging drops at a significant rate as main memory size increases and $|W.S.D.| = c$ is observed, as predicted, not to be a special case as it is with L.R.U.

Although random page replacement does not perform as well as either F.I.F.O. or L.R.U. for $|W.S.D.| < c$, in cases of limited main memory it does perform consistently very much better than F.I.F.O. and often better than L.R.U. As noted previously, the reason is that in most cases, neither L.R.U. nor F.I.F.O. make the optimum choice when main memory is limited (optimum in the sense of the definition of the MIN algorithm). Random page replacement will make an optimum choice on some occasions purely by chance. Consider for example the case $t' = k' = 1$, $c = 2$. During the sort phase, a record will be obtained from the source page and then a location examined in the object page.

If this location is occupied the overflow page will be needed. Since c = 2 any uses of the overflow page will cause a page fault, and both L.R.U. and F.I.F.O. would displace the source page in this example. The optimum choice would be to displace the object page as the source page will be needed immediately to obtain the next record. Random page replacement is capable of making this optimum choice purely by chance.

Except in the case c = 2, the L.R.U. algorithm produced less paging for all the parameter values used and this fact deserves comment. Although full L.R.U. as modelled here is considered somewhat impractical, the results indicate that the extra cost associated with implementing it in some form may well be worthwhile. In addition, the remarkable closeness of the L.R.U. results to the MIN results in many cases, illustrates how very efficient L.R.U. is for this particular sort algorithm. This fact was indicated in the analytic paging studies.

Although no analytic results have been obtained for F.I.F.O. page replacement the simulation results show its behaviour to be very similar to that obtained with other algorithms.

TABLE 5.4

Main Memory Pages	SEED 1	SEED 2	SEED 3	Average Page Faults	Predicted Page Faults
2	2718	2621	3079	2806	-
3	41	42	44	42	40
4	24	25	24	24.3	24
5	24	24	24	24	24
6	23	24	23	23.3	24
7	23	23	23	23	24
8	23	23	23	23	24
9	22	22	22	22	24
10	22	22	22	22	24
11	22	21	21	21.3	24
12	21	21	21	21	24

L.R.U. Page Replacement With Sublist Length Of One Page.

TABLE 5.5

Main Memory Pages	SEED 1	SEED 2	SEED 3	Average Page Faults	Predicted Page Faults
2	3263	3306	4576	3715	-
3	1147	1074	1365	1195	-
4	53	51	65	56.3	58
5	30	29	29	29.3	33
6	25	25	23	24.3	33
7	25	25	23	24.3	33
8	24	24	23	23.7	33
9	24	24	23	23.7	33
10	23	23	22	22.7	33
11	22	22	21	21.7	33
12	22	22	21	21.7	33

L.R.U. Page Replacement With Sublist Length Of Two Pages.

TABLE 5.6

Main Memory Pages	SEED 1	SEED 2	SEED 3	Average Page Faults	Predicted Page Faults
2	3267	3259	3223	3249.7	-
3	1545	1549	1542	1543.3	-
4	726	738	739	732.3	-
5	65	60	61	62	68
6	31	30	32	31	36
7	30	30	31	30.3	36
8	26	28	27	27	36
9	24	24	24	24	36
10	24	24	24	24	36
11	23	23	23	23	36
12	23	23	23	23	36

L.R.U. Page Replacement With Sublist Length Of Three Pages.

TABLE 5.7

Main Memory Pages	SEED 1	SEED 2	SEED 3	Average Page Faults	Predicted Page Faults
2	4150	4119	4492	4253.7	-
3	2089	2039	2153	2093.7	-
4	1332	1262	1364	1352.7	-
5	670	622	675	655.7	-
6	84	79	84	82.3	77
7	32	30	31	31	39
8	28	28	30	28.7	39
9	27	27	28	27.3	39
10	26	24	27	25.7	39
11	24	24	24	24	39
12	24	24	24	24	39

L.R.U. Page Replacement With Sublist Length Of Four Pages.

TABLE 5.8

Main Memory Pages	SEED 1	SEED 2	SEED 3	Average Page Faults	Predicted Page Faults
2	3444	3490	3457	3463.7	-
3	2060	2076	2076	2070.1	-
4	1574	1552	1591	1572.3	-
5	1130	1151	1115	1132	-
6	763	762	729	751.3	-
7	424	417	386	409	-
8	80	80	89	83	86
9	32	32	33	32.3	39
10	32	32	32	32	39
11	32	32	32	32	39
12	32	32	32	32	39

L.R.U. Page Replacement With Sublist Length Of Six Pages.

TABLE 5.9

Main Memory Pages	Average For SEED 1	Average For SEED 2	Average For SEED 3	Average Page Faults	Predicted Page Faults
2	2196	2122	2473	2263.7	-
3	76	767	64.3	72.3	69
4	41.7	53	44.3	46.3	48
5	37.3	40	42.0	39.8	41
6	34.3	37.3	32.3	34.7	39
7	33.7	34	31.3	33	39
8	29	30.3	28.7	29.3	-
9	28.3	26.3	27.3	27.3	-
10	27.3	25.7	23.3	25.4	-
11	24	25.7	25	24.9	-
12	22.3	25	23.7	23.7	-

RANDOM. Page Replacement With Sublist Length Of One Page.

TABLE 5.10

Main Memory Pages	Average For SEED 1	Average For SEED 2	Average For SEED 3	Average Page Faults	Predicted Page Faults
2	3202.3	3185.7	4165	3517.7	-
3	1433.3	1415.0	1867.3	1571.9	-
4	123	134.7	122.7	126.8	104
5	59.3	62	63.7	61.7	70
6	48	51.3	44.3	47.9	59
7	35.7	37.3	40	37.7	54
8	32.3	37.3	32	33.7	-
9	32.3	32.7	34.7	33.2	-
10	32.3	30.3	24.0	28.7	-
11	29	27.7	26	27.6	-
12	28.3	29.3	29.3	29.0	-

RANDOM Page Replacement With Sublist Length Of Two Pages.

TABLE 5.11

Main Memory Pages	Average For SEED 1	Average For SEED 2	Average For SEED 3	Average Page Faults	Predicted Page Faults
2	3368.3	3364	3330.3	3354.2	-
3	2035.3	2024.7	2018.3	2026.1	-
4	962.3	959	991.7	971	-
5	116	135.3	134.7	128.7	125
6	69.7	73	78	73.6	83
7	49	55	46.7	50.2	69
8	47.3	48.3	41.7	45.8	62
9	36.7	37	35	36.2	-
10	34.3	35.3	35	34.9	-
11	32.7	28	30.7	30.5	-
12	29.7	29	28.3	29	-

RANDOM Page Replacement With Sublist Length Of Three Pages.

TABLE 5.12

Main Memory Pages	Average For SEED 1	Average For SEED 2	Average For SEED 3	Average Page Faults	Predicted Page Faults
2	4176	4141.7	4439	4252.2	-
3	2834.7	2740.3	2966	2847	-
4	1764.3	1703	1809	1758.7	-
5	865	845.7	924	878.2	-
6	166.7	165.3	196.7	176.2	158
7	92.3	79	77	82.8	102
8	64.7	54	62.3	60.3	84
9	47	45.7	48	46.9	78
10	36	45	39.7	40.2	-
11	33	33.3	38.7	35	-
12	31.3	32.7	32.7	32.2	-

RANDOM Page Replacement With Sublist Length Of Four Pages.

TABLE 5.13

Main Memory Pages	Average For SEED 1	Average For SEED 2	Average For SEED 3	Average Page Faults	Predicted Page Faults
2	3728.7	3729.3	3736.3	3731.4	-
3	2751	2737.3	2760	2749.4	-
4	2051	2009.7	2067.3	2042.7	-
5	1455.7	1480.7	1421.3	1452.6	-
6	998.7	949	956.7	968.1	-
7	518.7	511.7	515.7	515.4	-
8	160.3	156.3	180.7	165.8	167
9	117	83	97	99	105
10	71.3	65.3	72	69.5	84
11	55.3	58	63	58.8	74
12	61	49.3	57	55.8	-

RANDOM Page Replacement With Sublist Length Of Six Pages.

TABLE 5.14

Main Memory Pages	SEED 1	SEED 2	SEED 3	Average Page Faults
2	2718	2621	2785	2708
3	50	59	68	59
4	26	26	26	26
5	24	24	24	24
6	24	24	24	24
7	24	24	24	24
8	23	23	23	23
9	23	23	23	23
10	23	22	22	22.3
11	22	22	22	22
12	21	22	21	21.3

F.I.F.O. Page Replacement With Sublist Length Of One Page.

TABLE 5.15

Main Memory Pages	SEED 1	SEED 2	SEED 3	Average Page Faults
2	3453	3500	4800	3917.7
3	1458	1505	1881	1614.7
4	56	56	78	63.3
5	36	36	35	35.7
6	33	33	32	32.3
7	26	26	24	25.3
8	25	25	23	24.3
9	24	24	22	23.3
10	23	23	22	22.7
11	23	23	22	22.7
12	23	23	22	22.7

F.I.F.O. Page Replacement With Sublist Length Of Two Pages.

TABLE 5.16

Main Memory Pages	SEED 1	SEED 2	SEED 3	Average Page Faults
2	3601	3613	3573	3595.7
3	2067	2082	2080	2076.3
4	926	948	951	941.7
5	61	61	61	61
6	41	41	41	41
7	32	32	32	32
8	29	29	29	29
9	25	25	25	25
10	24	24	24	24
11	23	23	23	23
12	23	23	23	23

F.I.F.O. Page Replacement With Sublist Length Of Three Pages.

TABLE 5.17

Main Memory Pages	SEED 1	SEED 2	SEED 3	Average Page Faults
2	4526	4467	4845	4612.7
3	2874	2782	3001	2885.7
4	1768	1725	1833	1775.3
5	881	807	885	857.7
6	94	101	107	100.7
7	53	54	56	54.3
8	39	39	39	39
9	28	28	28	28
10	26	26	26	26
11	24	24	25	24.3
12	24	24	24	24

F.I.F.O. Page Replacement With Sublist Length Of Four Pages.

TABLE 5.18

Main Memory Pages	SEED 1	SEED 2	SEED 3	Average Page Faults
2	3889	3940	3912	3913.7
3	2770	2771	2802	2781
4	2047	2026	2022	2031.7
5	1468	1450	1439	1452.3
6	949	943	923	938.3
7	489	497	482	489.3
8	102	106	93	100.3
9	56	58	58	57.3
10	48	49	49	48.7
11	39	35	35	36.3
12	33	33	33	33

F.I.F.O. Page Replacement With Sublist Length Of Six Pages.

TABLE 5.19

Main Memory Pages	SEED 1	SEED 2	SEED 3	Average Page Faults	Predicted Page Faults
2	1538	1475	1725	1579.3	-
3	26	26	28	26.7	24
4	23	24	23	23.3	
5	22	22	22	22	
6	21	21	21	21	
7	20	20	20	20	
8	20	20	20	20	
9	20	20	20	20	
10	20	20	20	20	
11	20	20	20	20	
12	20	20	20	20	

MIN Page Replacement With Sublist Length Of One Page.

TABLE 5.20

Main Memory Pages	SEED 1	SEED 2	SEED 3	Average Page Faults	Predicted Page Faults
2	2277	2295	2993	2521.7	-
3	786	761	995	847.3	-
4	35	35	40	36.7	33
5	26	26	24	25.3	
6	23	23	22	22.7	
7	22	22	21	21.7	
8	21	21	20	20.7	
9	21	21	20	20.7	
10	21	21	20	20.7	
11	21	21	20	20.7	
12	21	21	20	20.7	

MIN Page Replacement With Sublist Length Of Two Pages.

TABLE 5.21

Main Memory Pages	SEED 1	SEED 2	SEED 3	Average Page Faults	Predicted Page Faults
2	2406	2400	2370	2392	-
3	1116	1122	1111	1116.3	-
4	436	427	427	430	-
5	36	36	36	36	36
6	30	30	30	30	
7	25	25	25	25	
8	23	23	23	23	
9	22	22	22	22	
10	21	21	21	21	
11	21	21	21	21	
12	21	21	21	21	

MIN Page Replacement With Sublist Length Of Three Pages.

TABLE 5.22

Main Memory Pages	SEED 1	SEED 2	SEED 3	Average Page Faults	Predicted Page Faults
2	3025	2978	3210	3071	-
3	1588	1552	1657	1599	-
4	828	814	864	835.3	-
5	354	339	358	350.3	-
6	40	40	41	40.3	39
7	29	29	30	29.3	
8	25	25	26	25.3	
9	23	23	24	23.3	
10	22	22	23	22.3	
11	21	21	22	21.3	
12	21	21	21	21	

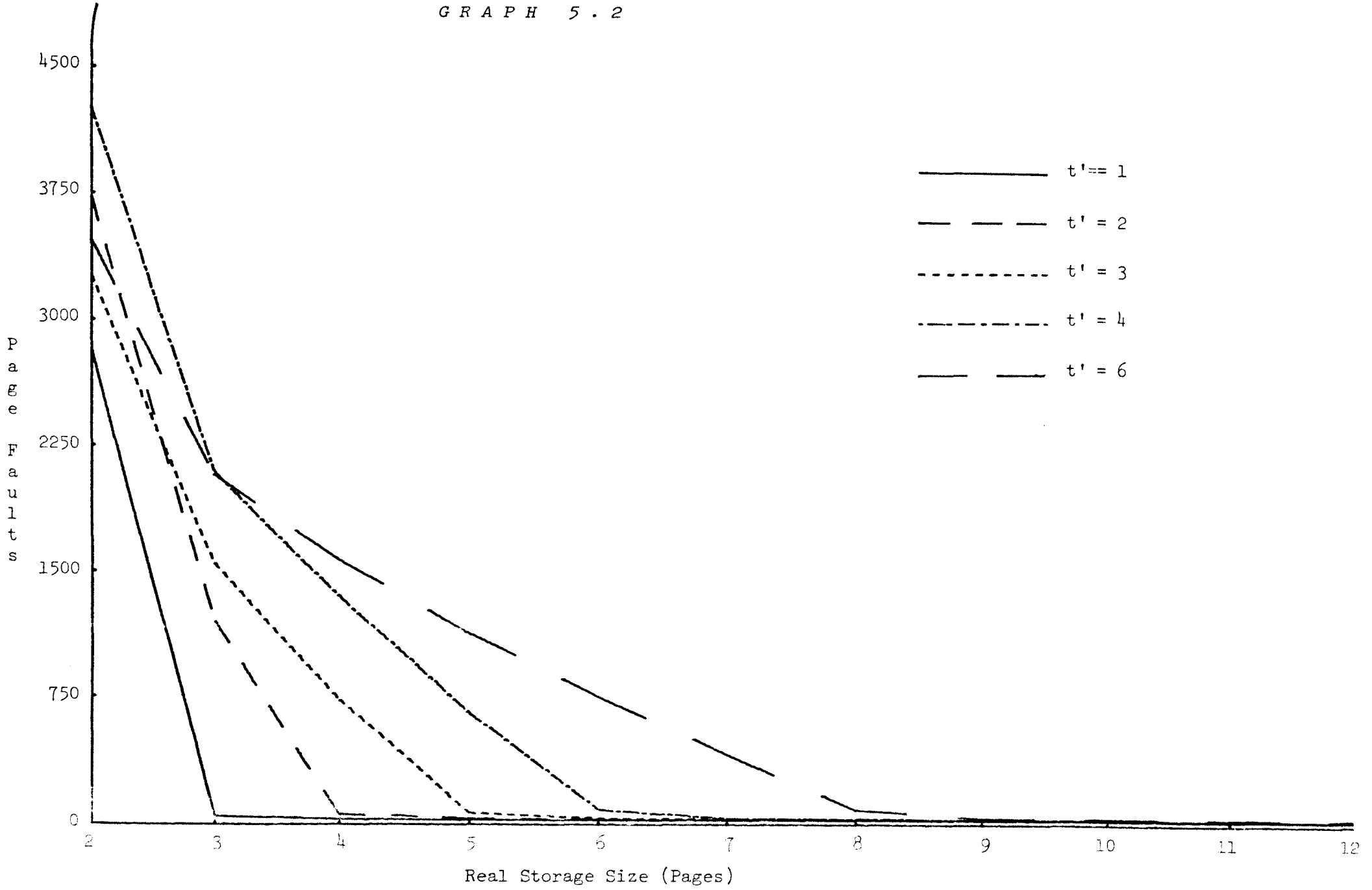
MIN Page Replacement With Sublist Length Of Four Pages.

TABLE 5.23

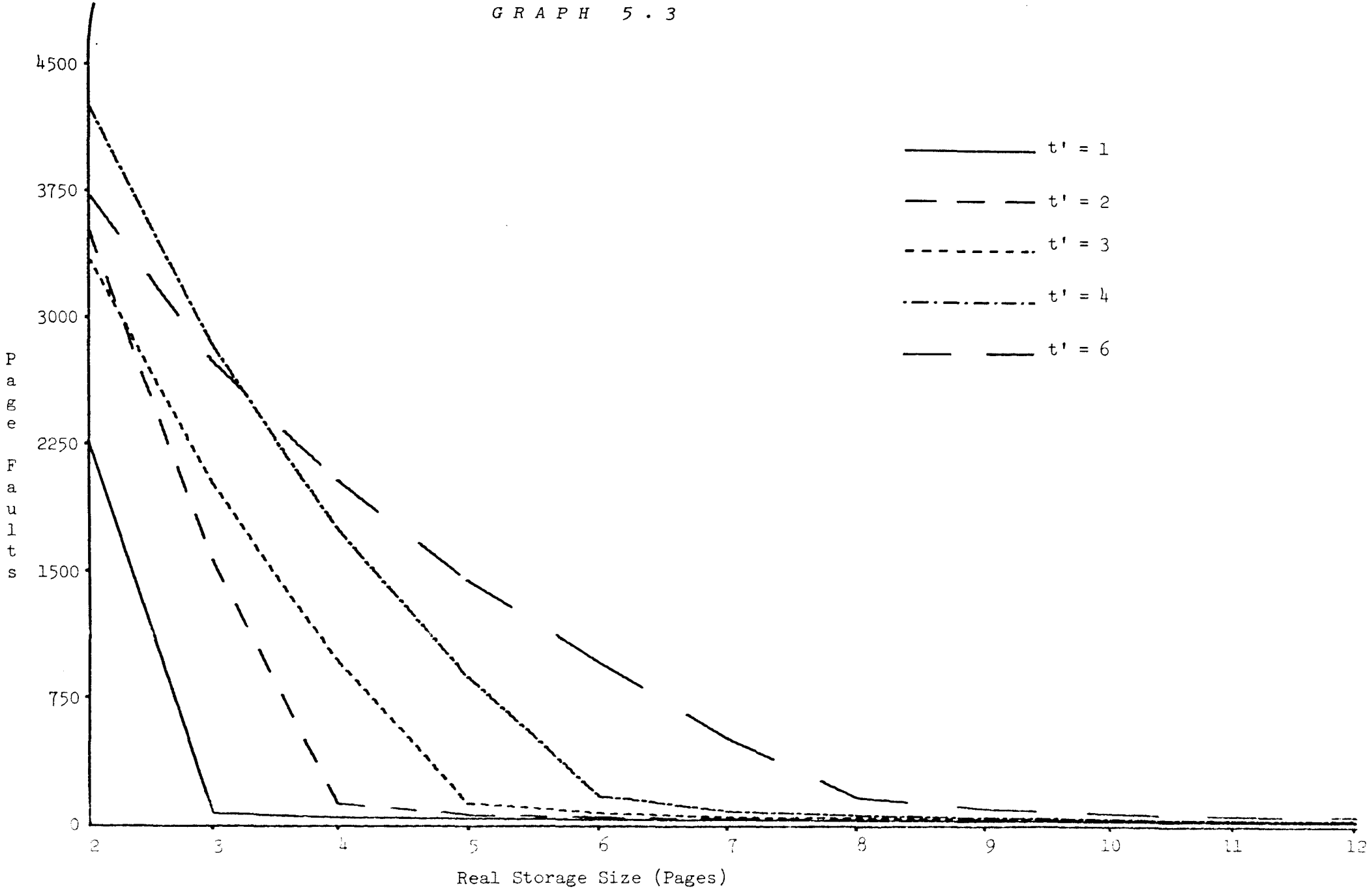
Main Memory Pages	SEED 1	SEED 2	SEED 3	Average Page Faults	Predicted Page Faults
2	2639	2667	2652	2657.7	-
3	1597	1583	1600	1593.3	-
4	1030	1028	1028	1028.3	-
5	653	659	647	653	-
6	394	381	378	384.3	-
7	186	181	179	182	-
8	39	39	39	39	39
9	32	32	32	32	
10	30	30	30	30	
11	28	28	28	28	
12	26	26	26	26	

MIN Page Replacement With Sublist Length Of Six Pages.

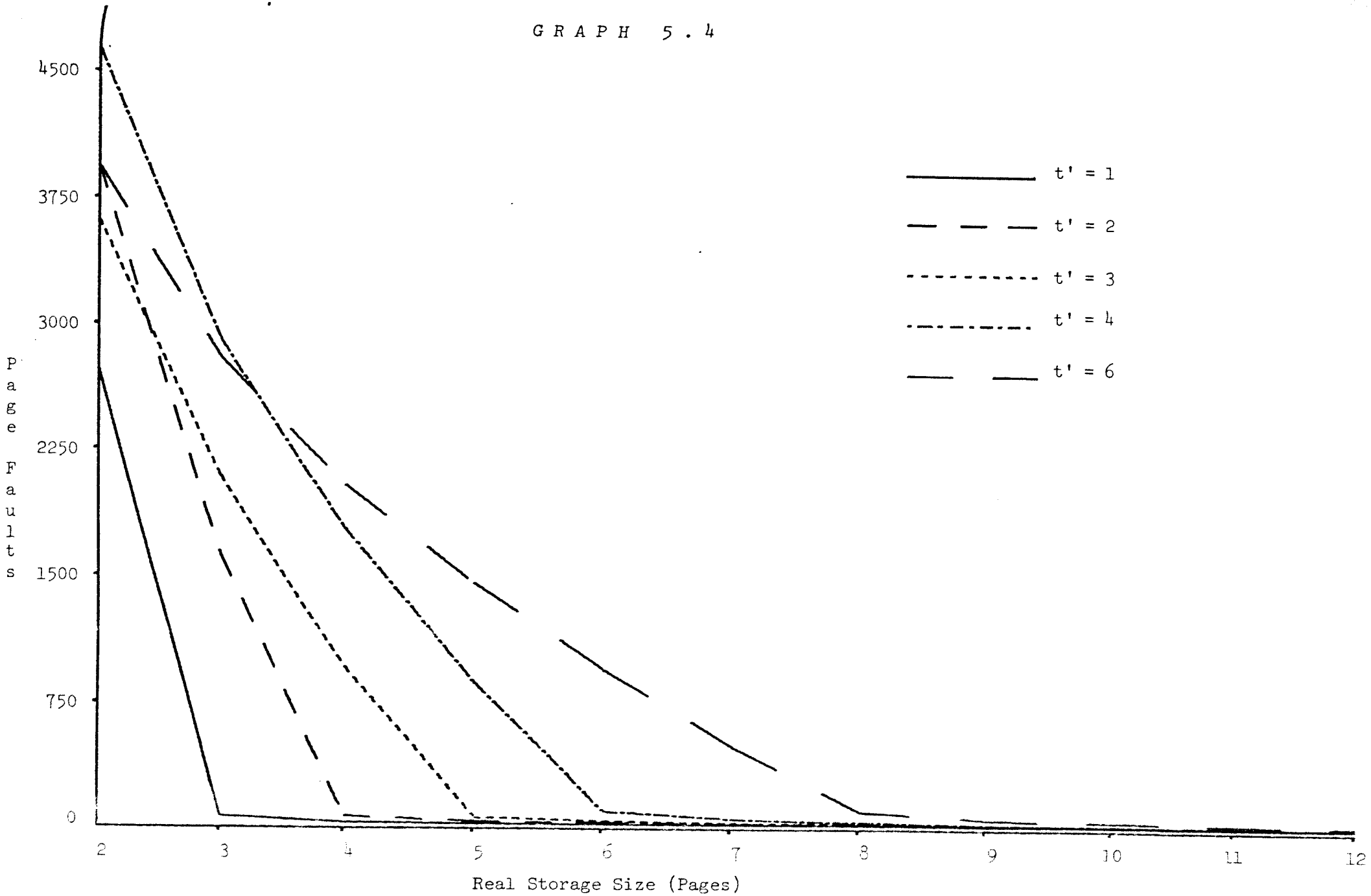
GRAPH 5.2


4500
3750
3000
2250
1500
750
0
Page Faults
2
3
4
5
6
7
8
9
10
11
12
Real Storage Size (Pages)
t' = 1
t' = 2
t' = 3
t' = 4
t' = 6

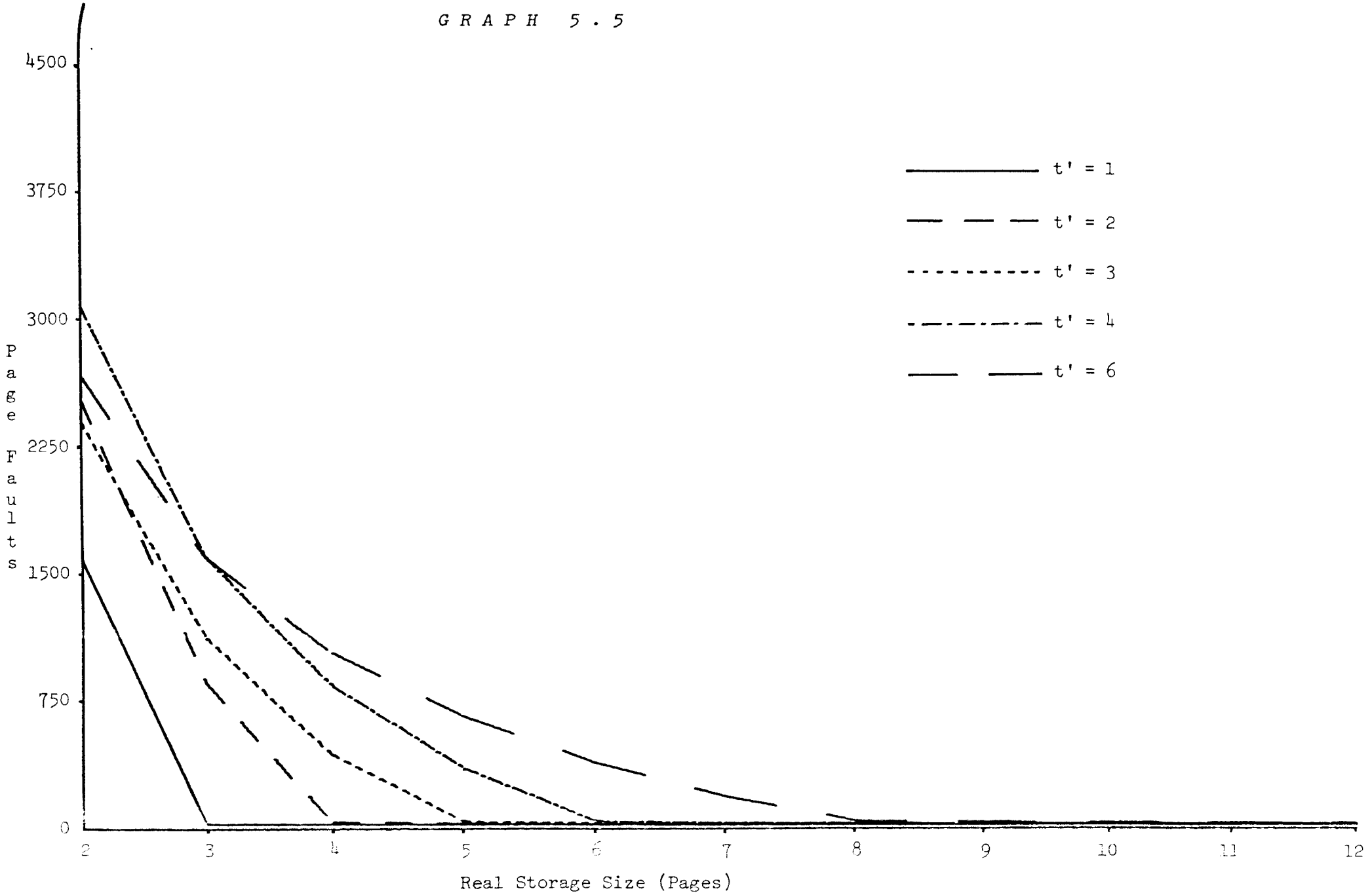
GRAPH 5.3


4500
3750
3000
2250
1500
750
0
Page Faults
2
3
4
5
6
7
8
9
10
11
12
Real Storage Size (Pages)
t' = 1
t' = 2
t' = 3
t' = 4
t' = 6

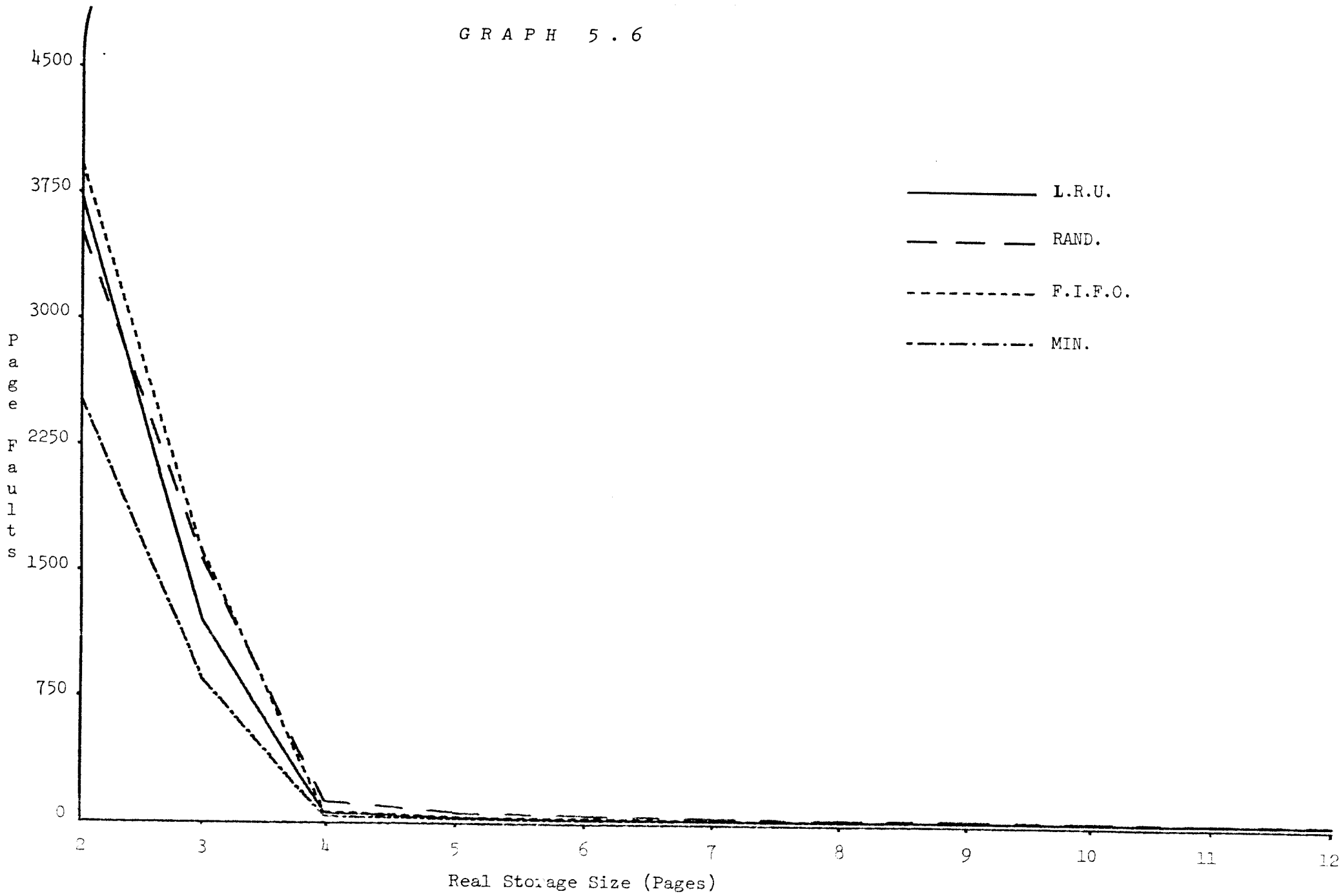
GRAPH 5.4


4500
3750
3000
2250
1500
750
0
Page Faults
2
3
4
5
6
7
8
9
10
11
12
Real Storage Size (Pages)
t' = 1
t' = 2
t' = 3
t' = 4
t' = 6

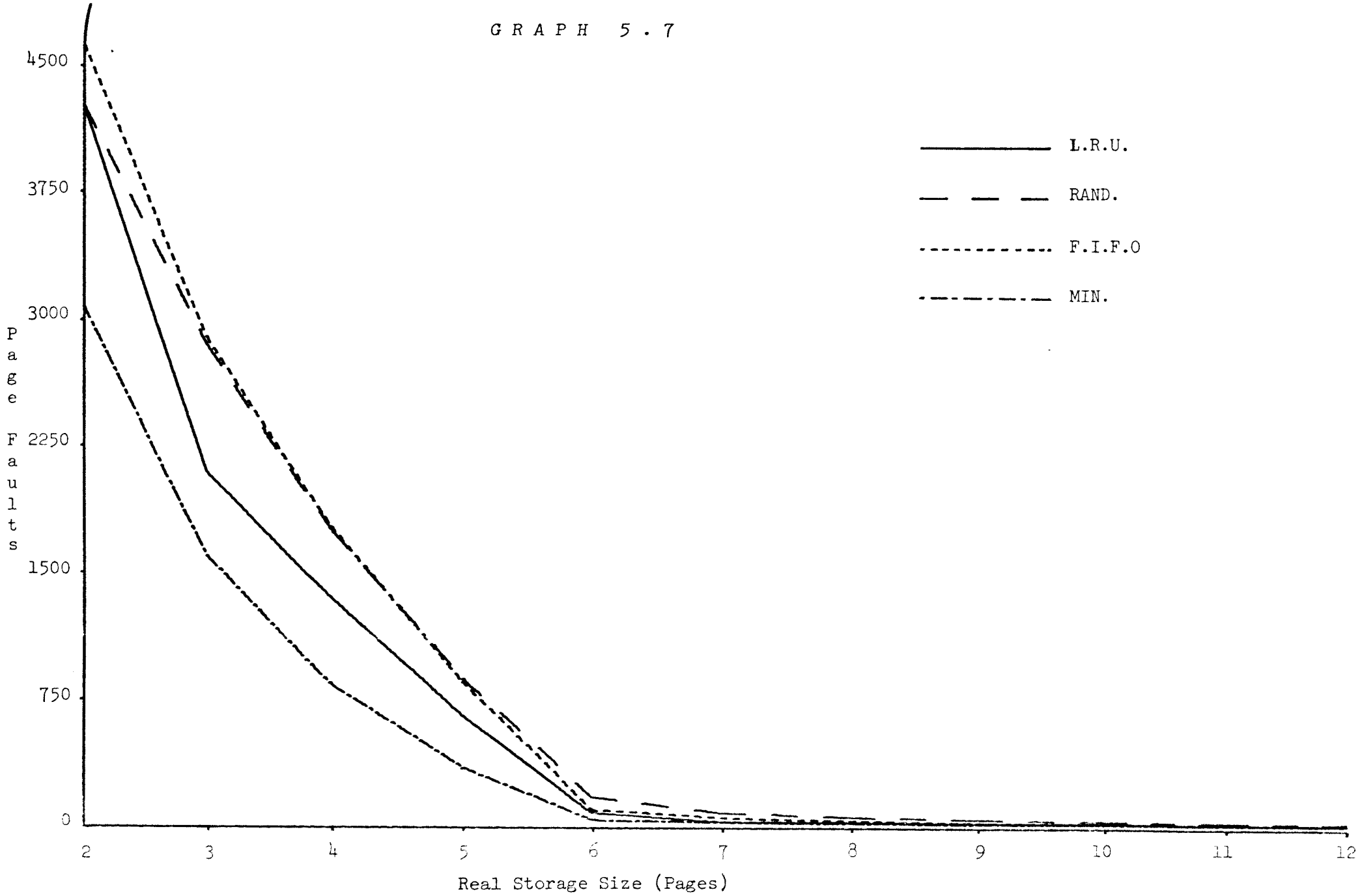
GRAPH 5.5


4500
3750
3000
2250
1500
750
0
Page Faults
2
3
4
5
6
7
8
9
10
11
12
Real Storage Size (Pages)
t' = 1
t' = 2
t' = 3
t' = 4
t' = 6

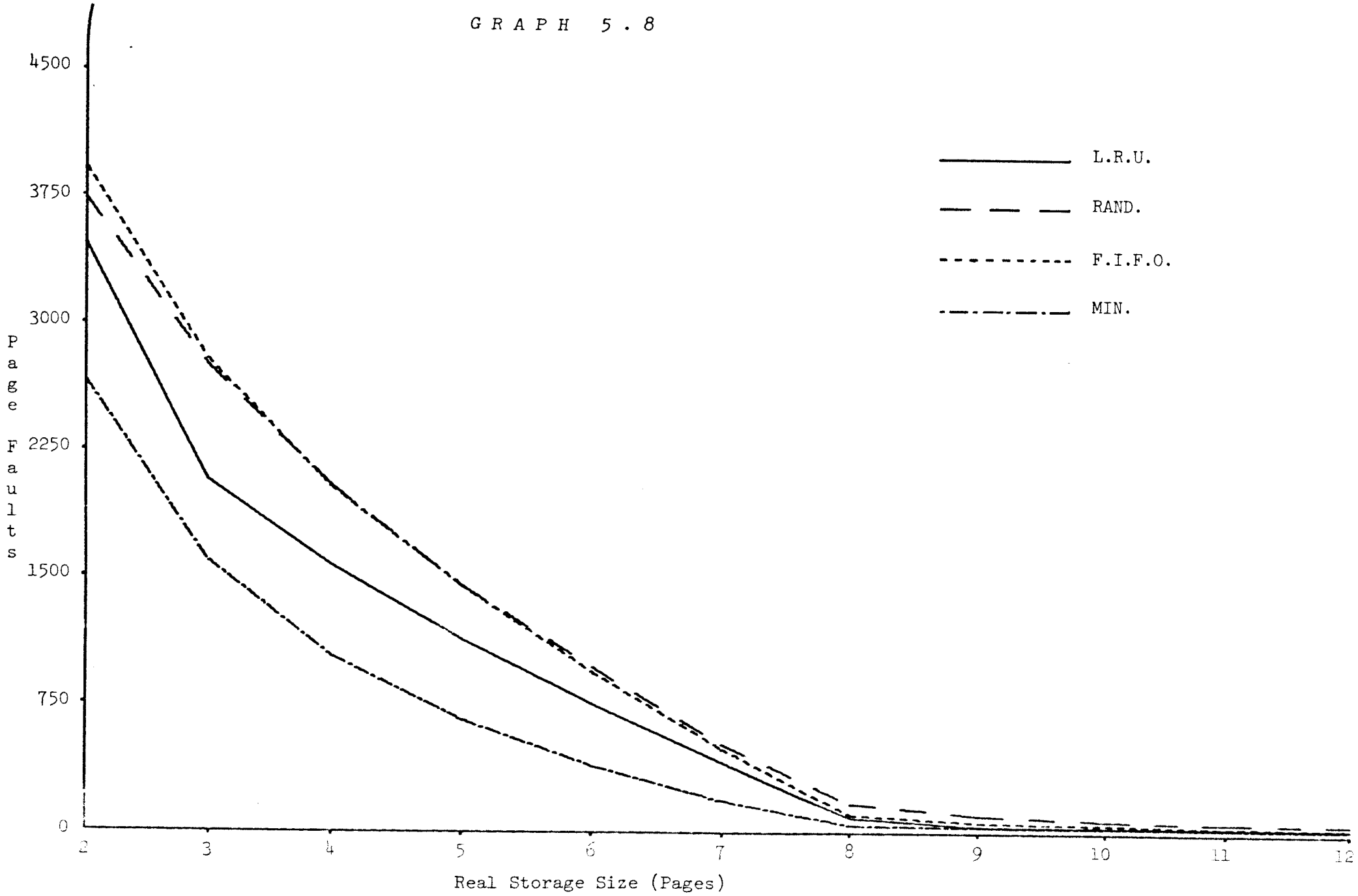
GRAPH 5.6


4500
3750
3000
2250
1500
750
0
Page Faults
2
3
4
5
6
7
8
9
10
11
12
Real Storage Size (Pages)
L.R.U.
RAND.
F.I.F.O.
MIN.

GRAPH 5.7


4500
3750
3000
2250
1500
750
0
Page Faults
2
3
4
5
6
7
8
9
10
11
12
Real Storage Size (Pages)
L.R.U.
RAND.
F.I.F.O
MIN.

GRAPH 5.8


4500
3750
3000
2250
1500
750
0
Page Faults
2
3
4
5
6
7
8
9
10
11
12
Real Storage Size (Pages)
L.R.U.
RAND.
F.I.F.O.
MIN.

5.4.3 The Merge Phase

In order to produce a completely sorted file it is necessary to merge together the sublists produced by the sort phase. Clearly if an S way merge is to be used it would be preferable for at least one page from each of the S sublists to be present in main memory together with one page from the new sublist being produced. This will allow merging to continue without being unduly interrupted by page faults, and leads to the definition of the working set of data for the merge phase as these S + 1 pages. Many of the problems which occurred in the sort phase analysis arise in the merge phase, and there are the same three cases to consider:-

(i) $|W.S.D.| < c$

(ii) $|W.S.D.| = c$

(iii) $|W.S.D.| > c$

Fortunately it is not necessary to formulate a completely separate analysis for each replacement algorithm because merging is a reasonably simple process and behaves similarly under the various replacement policies. In order to be able to refer easily to the various aspects of the process, it is useful to make the following definitions:-

Define a 'phase' to be the entire merging process.

Define a 'step' to be the merging necessary to take a whole set of sublists (input sublists) and produce a whole new set of longer sublists (output sublists).

Define a 'unit' to be the merging of a set of input sublists to produce one new output sublist.

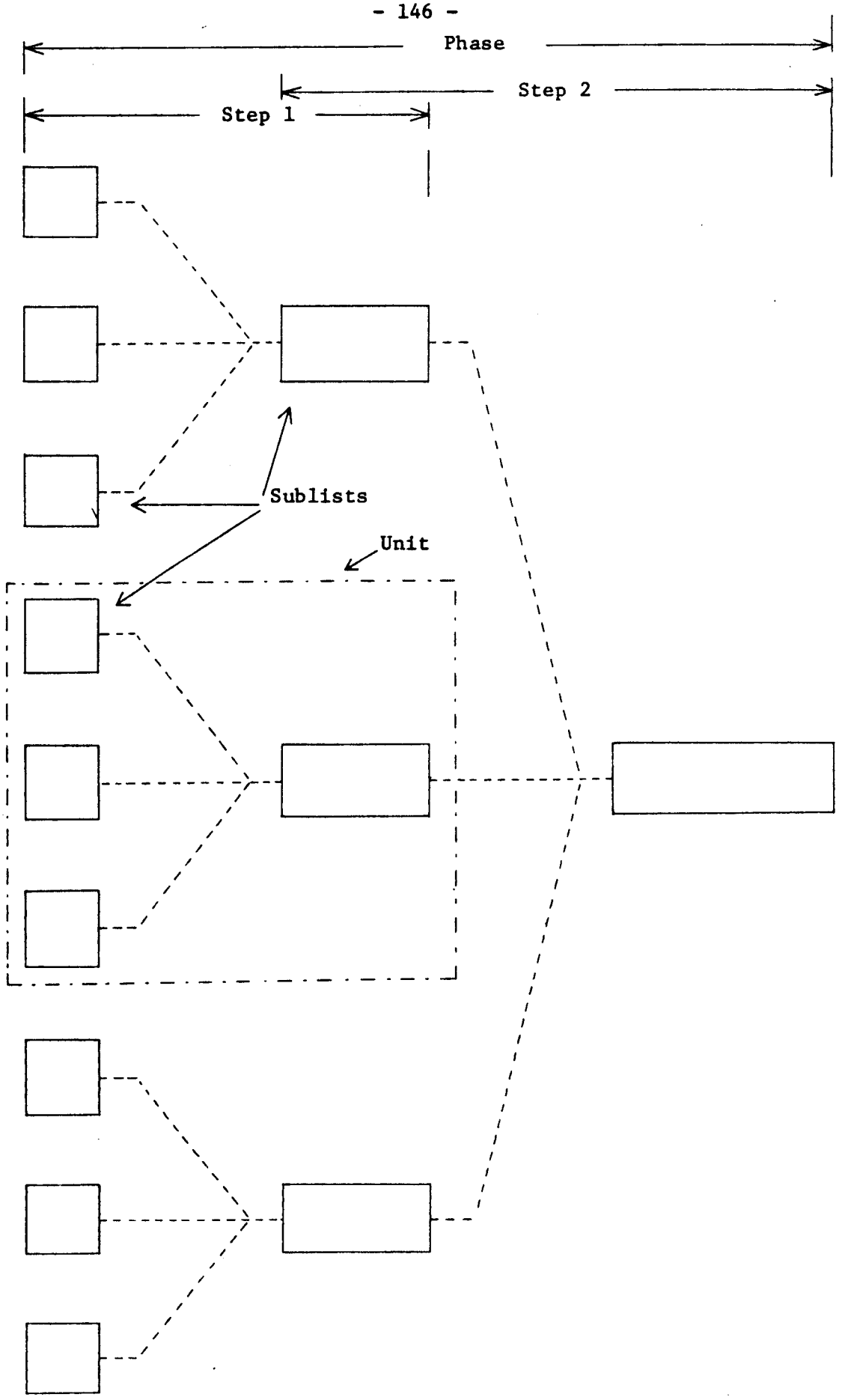


Fig. 5.4.

Thus a phase consists of several steps and a step consists of several units. Hopefully figure 5.4 clarifies these definitions.

Recall the definition of a 'page change' as the replacement of one of the members of the W.S.D. Thus |W.S.D.| remains the same but one of the constituent pages is changed. If the entire W.S.D. is located in real storage, a page change could induce several page faults before the modified W.S.D. is entirely located in real storage.

As has been done previously, dynamic acquisition of a free virtual page is treated as a page fault. This can be regarded as a worst case and the analysis presented can easily be modified to predict the performance of a specific operating system.

Consider the first case i.e. $|W.S.D.| < c$. During each unit the only page faults which occur will be when the page of the output sublist being produced becomes full, or one of the pages of the input sublist is exhausted.

With L.R.U. each of these page changes will only cause one page fault because no member of the W.S.D. will be least recently used.

Similarly with F.I.F.O., usually the oldest page in real storage (i.e. first in) will not be a member of the W.S.D., so only one page fault will be caused for each page change.

Random page replacement could induce any number of page faults before successfully effecting one page change. The probability of displacing an unwanted page is $(c - S)/c$ and so the expected number of page faults required to change a page of the W.S.D. is $c/(c - S)$ (expected value of a geometric distribution). The average number of page faults produced in the change of W.S.D. which occurs when a new unit begins is $S + 1$ for the first unit and $c(H_c - H_{c-S-2})$ for subsequent units (see section 5.4.1.2.).

Let F be defined as this average number of page faults which occur when a unit begins then:-

$$F = \begin{cases} S + 1 & \text{for the first unit} \\ c(H_c - H_{c-S-2}) & \text{for subsequent units.} \end{cases}$$

A unit which merges S sublists each of length ℓ pages will require S + 1 page changes to load the W.S.D. initially, and:-

$S(\ell - 1)$	page changes to load the remainder of each sublist.
+	
$S\ell - 1$	page changes to load the remainder of the new sublist.
$= 2S\ell - S - 1$	

This will produce:-

$S + 1 + 2S\ell - S - 1$	page faults with L.R.U. and F.I.F.O.
$= 2S\ell$	
$F + (2S - S - 1)\frac{c}{c - S}$	page faults on average with random page replacement.

The output of the sort phase described previously will consist of sparsely occupied sublists. The first merge step will compact them, as a side effect. In this case the expressions derived above require the following modifications.

Suppose the input sublists for a unit are only 100f% full, $0 < f \leq 1$.

If there are S sublists each of length ℓ pages, the output sublist length will be $\lceil S\ell f \rceil$ pages and the above expressions modify to:-

$$S\ell + \lceil S\ell f \rceil \qquad \text{L.R.U. and F.I.F.O}$$

$$F + (S\ell + \lceil S\ell f \rceil - S - 1)\frac{c}{c - S} \qquad \text{Random}$$

All the input sublists to a unit may not be the same length, in particular, just one may be shorter than the rest. This special case occurs frequently and expressions for the paging to be expected are required. Clearly a unit merging S input sublists, of which one is shorter than the rest, will not necessarily exhaust the shorter sublist before the other S - 1. The distribution of the keys in the shorter sublist is over the same range as in the longer sublists, there are just fewer records.

Suppose (S - 1) input sublists are of length ℓ and one is of length ℓ' where $\ell' \leqslant \ell$. Using the expressions derived above the total number of page faults will be:-

$$\begin{aligned} & S + 1 && \text{loading the initial W.S.D.} \\ & + && \\ & (S - 1)(\ell - 1) + (\ell' - 1) && \text{loading the rest of the} \\ & + && \text{input sublists.} \\ & \lceil ((S - 1)\ell + \ell')f \rceil - 1 && \text{loading the rest of the} \\ & && \text{output sublist.} \end{aligned}$$

$$= S\ell - (\ell - \ell') + \lceil ((S - 1)\ell + \ell')f \rceil \qquad \text{for L.R.U. and F.I.F.O.}$$

and correspondingly:-

$$F + (S\ell - (\ell - \ell') + \lceil ((S - 1)\ell + \ell')f \rceil - S - 1)\frac{c}{c - S} \qquad \text{for Random.}$$

In the case |W.S.D.| = c the expression derived for random page replacement for the case |W.S.D.| < c is still valid.

An understanding of the problems associated with F.I.F.O. replacement when |W.S.D.| = c is best obtained using a modified version of the notation employed previously to describe main storage contents. Suppose the letters I_1, ..., I_S denotes the set of pages from the S input sublists and O_i denotes the i^{th} page of the output sublist. A sequence of these letters then represents main storage contents and the order from left to right indicates the order in which the pages entered storage i.e. first in is the extreme leftmost letter of the string. As the algorithm begins the storage contents will be:-

$$I_1, I_2, I_3, \ldots\ldots, I_S\ O_1$$

Clearly, the output page will be filled before any of the input pages are exhausted and a reference to O_2 will result. This will displace I_1 :-

$$I_2, I_3, \ldots\ldots, I_S\ O_1, O_2$$

Referencing I_1 will displace I_2 and so on and a total of S + 1 page faults will occur before O_1 is finally displaced. This is an example of one page change requiring several page faults. The final main storage contents will be:-

$$O_2, I_1, I_2, \ldots\ldots, I_S$$

If O_2 is filled and a reference to O_3 occurs before one of the input pages becomes exhausted, only one page fault will occur and the

new memory contents will be:-

$$I_1, I_2, \ldots\ldots, I_s, O_3$$

Each of the input sublists has the same probability of being the first to become exhausted. Suppose input page I_j becomes exhausted and a reference to its successor I_j' occurs. Main storage contents immediately following this will be either:-

$$I_1, I_2, \ldots\ldots, I_s, I_j'$$

or

$$I_2, I_3, \ldots\ldots, I_s, O_i, I_j'$$

depending on how many output pages have been referenced. Either j or j - 1 additional page faults will occur before I_j is displaced giving totals of j + 1 and j, and main storage contents following the displacement of I_j of:-

$$I_{j+1}, \ldots\ldots, I_s, I_j', O_i, I_1, \ldots\ldots, I_{j-1}$$

or

$$I_{j+1}, \ldots\ldots, I_s, O_i, I_j', I_1, \ldots\ldots, I_{j-1}$$

Changing other input pages will have a similar 'mixing' effect on storage contents and several page changes (the order of which will depend on the actual data) will leave the pages in real storage in what is virtually an unpredictable order.

For simplicity it is assumed that when a page change is necessary, the page which has to be displaced occupies any of the possible positions in the string with equal probability giving an average number of page faults per page change of (S + 2)/2. Regrettably, there is no way to improve on this figure since the program is not able to rearrange the order of the pages in storage.

The fact that a page is no longer required in main storage is not evident until its successor is referenced. As far as the output sublist is concerned this will mean the filled page is second most recently referenced page in storage and so S - 1 page faults must occur with L.R.U. before it is displaced. An exhausted page of the input sublists could be anything, except most recently used, giving an average of (S + 1)/2 page faults to displace it.

If the program which is using the data is able to detect that an input page is exhausted or an output page is full before referencing its successor, for example by counting records or checking addresses, the page which is no longer needed could be made least recently used by deliberatly referencing the other pages in storage.

Thus in the case $|W.S.D.| = c$, a unit which merges S sublists each of length ℓ pages will require $S(\ell - 1) + S\ell - 1$ page changes plus S + 1 page faults to initially load the W.S.D. giving totals of:-

$$S + 1 + S(\ell - 1)\frac{(S + 1)}{2} + (S\ell - 1)(S - 1)$$

$$= S(S + 1)\frac{(3\ell - 1)}{2} - 2(S\ell - 1)$$

page faults on average with L.R.U.

$$S + 1 + S(\ell - 1)\frac{(S + 1)}{2} + (S\ell - 1)\frac{(S + 1)}{2}$$

$$= (S + 1)\left(S\ell - \frac{(S - 1)}{2}\right)$$

page faults on average with F.I.F.O.

$$F + \frac{c}{(c - S)}(2S\ell - S - 1)$$

$$= F + (S + 1)(2S - S - 1)$$

page faults on average with Random. (recall c = S + 1)

The corresponding expressions for sparsely occupied input sublists ($f < 1$) are:-

$$(S + 1)(S(\frac{\ell - 1}{2}) + \lceil S\ell f \rceil) - 2(\lceil S\ell f \rceil - 1) \qquad \text{L.R.U.}$$

$$S(\ell - 1)\frac{(S + 1)}{2} + (\lceil S\ell f \rceil + 1)\frac{(S + 1)}{2} \qquad \text{F.I.F.O.}$$

$$F + (S\ell + \lceil S\ell f \rceil - S - 1)\frac{c}{(c - S)} \qquad \text{Random.}$$

$$= F + (S + 1)(S\ell + \lceil S\ell f \rceil - S - 1)$$

When one sublist is shorter than the rest, these expressions modify very easily. As before, suppose there are $S - 1$ sublists of length ℓ and one of length ℓ'. The expressions for the number of page faults induced are:-

$$S + 1 + (S - 1)(\ell - 1)\frac{(S + 1)}{2} + (\ell' - 1)\frac{(S + 1)}{2} \qquad \text{L.R.U.}$$

$$+ ((S - 1)\ell + \ell' - 1)(S - 1)$$

$$= \frac{(S + 1)}{2}((S - 1)(\ell - 1) + (\ell' + 1)) + ((S - 1)\ell + \ell')(S - 1)$$

$$S + 1 + ((S - 1)(\ell - 1) + (\ell' - 1))\frac{(S + 1)}{2} \qquad \text{F.I.F.O.}$$

$$+ ((S - 1)\ell + \ell' - 1)\frac{(S + 1)}{2}$$

$$= \frac{(S + 1)}{2}(2 + (S - 1)(2\ell - 1) + 2(\ell' - 1))$$

$$F + \frac{c}{(c - S)}(2(S - 1)\ell + 2\ell' - (S + 1)) \qquad \text{Random.}$$

$$= F + (S + 1)(2(S - 1)\ell + 2\ell' - (S + 1))$$

When real storage is very limited i.e. |W.S.D.| > c page faults will occur during merging as well as when a page change is necessary. As a result, a very considerable increase in paging is to be expected.

Suppose there are b records per page. After a record has been written to the output sublist a new record must be obtained from an input sublist. Thus processing a single record can cause either 0, 1 or 2 page faults:-

0 page faults if the page of the input sublist required is in real storage.

1 page fault if the page of the input sublist required is not in real storage and loading it does not displace the page of the output sublist.

2 page faults if the page of the input sublist required is not in real storage and loading it does displace the page of the output sublist.

Assuming randomly distributed data the probabilities of each of these cases for each replacement algorithm are:-

L.R.U.: $pr(0\ p.f.) = (c - 1)/S$

$pr(1\ p.f.) = 1 - (c - 1)/S$

$pr(2\ p.f.) = 0$ must be zero since the page of the output sublist is most recently used.

Random: $pr(0\ p.f.) = (c - 1)/S$

$pr(1\ p.f.) = (1 - (c - 1)/S)(1 - 1/c)$

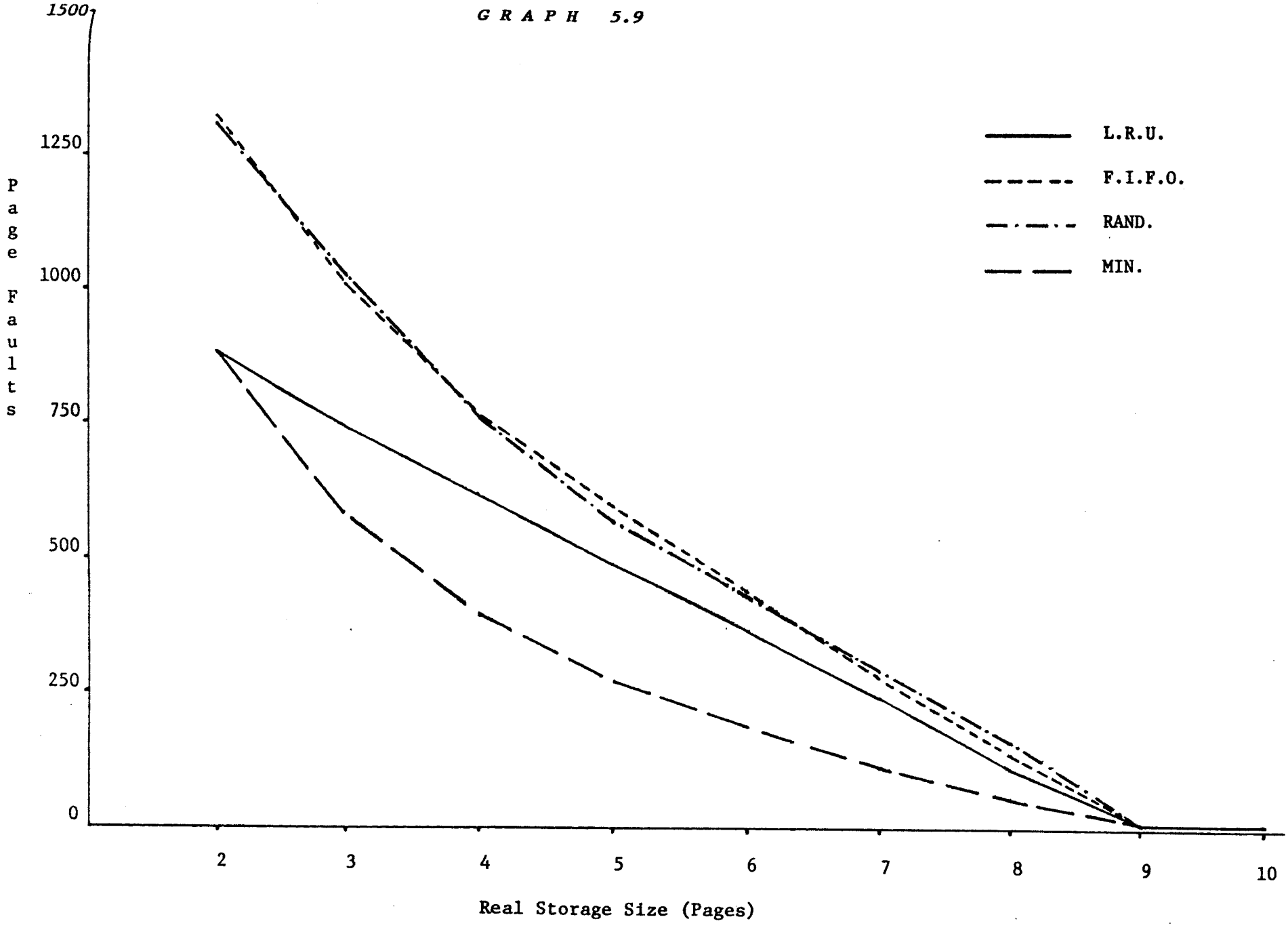
$pr(2\ p.f.) = (1 - (c - 1)/S).1/c$

F.I.F.O.: A reasonable approximation is obtained if the probabilities obtained for random page replacement are used. This is intuitively reasonable since pages are entering real storage in a 'random' order. Further evidence is provided by simulation. Table 5.24 and graph 5.9 show the result of simulating the merging of eight, one page sublists each containing 1000 records, using various amounts of real storage.

Main Memory Pages	Page faults			
	L.R.U.	F.I.F.O.	RAND.	MIN.
2	882	1322	1319	882
3	747	1010	1027	582
4	626	769	765	406
5	492	599	570	278
6	372	444	434	186
7	245	279	299	112
8	116	146	163	54
9	9	9	9	9
10	9	9	9	9

Table 5.24

GRAPH 5.9


1500
1250
1000
750
500
250
0
Page Faults
L.R.U.
F.I.F.O.
RAND.
MIN.
2
3
4
5
6
7
8
9
10
Real Storage Size (Pages)

Thus for b records the expected number of page faults is $b(1 - (c - 1)/S)$ for L.R.U. and $b(1 - (c - 1)/S)(1 + 1/c)$ for F.I.F.O. and Random.

Changing a page of the W.S.D. will cause slight changes in these expressions until the page leaving the W.S.D. has been displaced from real storage. This effect is negligible when compared with the paging which is occurring anyway and will be ignored. Thus a unit merging S sublists of length ℓ pages with b records per page will induce an average of $S\ell b(1 - (c - 1)/S)$with L.R.U. and $S\ell b(1 - (c - 1)/S)(1 + 1/c)$ for F.I.F.O. and Random.

Consider as an example the case $S = 4$, $\ell = 3$, $f = 1$, and $b = 200$. Then $|W.S.D.| = 5$ and the expressions derived above give:-

	L.R.U.	F.I.F.O.	RANDOM
$c > \|W.S.D.\|$	24	24	$5 + 19c/(c - 4)$
$c = \|W.S.D.\|$	58	53	100
$c < \|W.S.D.\|$	$600(5 - c)$	$600(5 - c)(1 + 1/c)$	$600(5 - c)(1 + 1/c)$

The fact that paging increases by an order of magnitude if the available real storage drops to one page less than the $|W.S.D.|$ should be noted.

A step usually consists of several units. The expressions derived above for the paging to be expected for a single unit are 'self contained' in the sense that all the paging occurring during processing of a single unit is accounted for. As a result the paging which will occur during a complete step consisting of say U units, can be expected to be just U multiplied by the expression for a single unit. However the total number of sublists involved in a single step may not be a multiple of S, the order of merging.

In this case, one unit will require special attention. Even if the number of sublists is a multiple of S, one may be shorter than the rest because one of the previous steps may have involved a unit merging less than S sublists.

Suppose step i has E_i input sublists, all but one of them of length L_i, then it will consist of $\lceil E_i/S \rceil$ units and hence the input sublists supplied to the next step will satisfy $E_{i+1} = \lceil E_i/S \rceil$ and $L_{i+1} = \lceil SL_i f \rceil$.

The final unit of each step will have Q_i input sublists where $Q_i = E_i - \lfloor E_i/S \rfloor S$ if E_i is not a multiple of S, and $Q_i = S$ otherwise. One of these sublists may be shorter than the rest. Its length will be V_i and it is easily shown that $V_i = ((Q_{i-1} - 1)L_{i-1} + V_{i-1})f$. Clearly $V_1 = L_1$ on most occasions since it is reasonable to assume that all the sorted sublists produced by the sort phase are of the same length. As an example figure 5.5 shows the case $E_1 = 20$, $S = 3$, $L_1 = 1$.

The importance of the special case unit rises with each step until the final step which consists of a single unit.

The number of steps involved in the entire phase will be $\lceil \log_S E_1 \rceil$. Suppose, as input to the merge phase, there are E_1 sublists each of length L_1 pages and an S way merge is desired. Using the notation and expressions derived above, it is relatively easy to compute the total number of page faults which will occur for each replacement policy with various amounts of real storage available. However the general expressions which result are extremely lengthy and so only one case, $|W.S.D.| < c$, is given here as an example.

$E_1 = 20$	$E_2 = 7$	$E_3 = 3$
$L_1 = 1$	$L_2 = 3$	$L_3 = 9$
$V_1 = 1$	$V_2 = 2$	$V_3 = 2$
$Q_1 = 2$	$Q_2 = 1$	$Q_3 = 3$

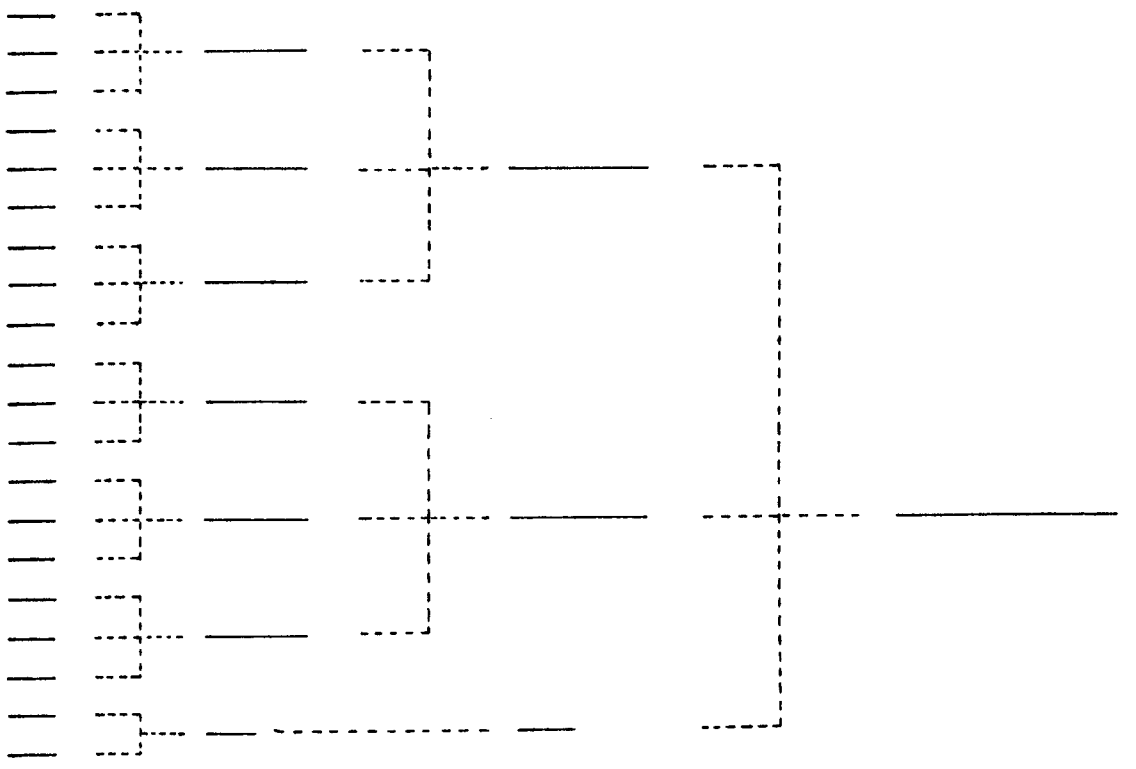


Fig 5.5

It is a great deal easier to deal with specific instances of merges of particular interest where the actual parameters are available, using the expressions derived for individual units, than to formulate all encompassing generalized expressions.

Thus for the case $|W.S.D.| < c$, it is assumed that $f < 1$ for units of the first step and $f = 1$ thereafter as the second and subsequent steps will normally operate with compacted sublists. In addition it is assumed that all the input sublists to the first step are of equal length (i.e. equal numbers of sparsely occupied

pages but not necessarily equal numbers of records). The total number of page faults expected is:-

L.R.U. and F.I.F.O.:-

$$(SL_i + \lceil SL_i f \rceil)(\lceil E_i/S \rceil - 1)$$ All but the last unit of the first step.

$$+$$

$$(Q_i L_i + \lceil Q_i L_i f \rceil)$$ Last unit of the first step.

$$+$$

$$\sum_{i=2}^{\lceil \log_2 E_i \rceil} \{2SL_i(\lceil E_i/S \rceil - 1)$$ All but the last unit of second and subsequent steps.

$$+ 2Q_i L_i - 2(L_i - V_i)\}$$ Last unit of second and subsequent steps.

(Cont'd Overleaf.)

Random:-

$$(S + 1) + (SL_1 + \lceil SL_1 f \rceil - S - 1)\frac{c}{c - S}$$ First unit of first step.

$$+$$

$$(\lceil E_1/S \rceil - 2)(c(H_c - H_{c-S-2}) + (SL_1 + \lceil SL_1 f \rceil - S - 1)\frac{c}{c - S})$$ All of first step except first and last units.

$$+$$

$$c(H_c - H_{c-Q_1-2}) + (Q_1 L_1 + |Q_1 L_1| - Q_1 - 1)\frac{c}{c - S}$$ Last unit of first step.

$$+$$

$$\sum_{i=2}^{\lceil \log_2 E_1 \rceil} \{c(H_c - H_{c-S-2}) + (2SL_i - S - 1)\frac{c}{c - S}$$ All but the final units of the second and subsequent steps.

$$+ c(H_c - H_{c-Q_i-2}) + (Q_i L_i - (L_i - V_i + \lceil ((Q_i - 1)L_i + V_i)f \rceil - Q_i - 1\}$$ Final units of second and subsequent steps.

5.5 Parameter Setting

There are several variable quantities involved in this sorting technique which play important roles in determining its efficiency. They are the ratio of the source area size to the object area size i.e. t to k or t' to k' , the actual size of the source area i.e. the sublist length, and the order of merging, S. This raises the question of a suitable definition of efficiency.

Conventional internal comparative sorting algorithms are usually assessed in terms of the average number of comparisons required to sort n records. This relies on the assumption that execution time increases as the average number of comparisons increases. Other criteria which might be considered are the number of storage references, amount of working space required, total execution time, or the number of page faults induced.

For modified address calculation, the average number of comparisons is not appropriate since two record keys are never compared in the basic algorithm, and only a relatively small number of such comparisons take place if the first or second modification is used. Checking to see whether an object area location is occupied need not involve a comparison (on I.B.M.'s System 360 and System 370 at least). For example, if all object area locations are initialized to zero then during execution, the logical OR can be used to set a condition code if a particular object area location is other than zero.

Normal address calculation sorting is known to be a high speed algorithm and with the changes suggested here it should operate even more quickly. Excessive paging will cause considerable delays and Brawn et al, 1970 have shown how the use of sublists dramatically reduces the number of page faults produced. In order that modified address calculation does not lose the benefits gained from the use of sublists, the criterion used in the setting of parameters is that the amount of paging should be low.

5.5.1 The Ratio Of Source Area Size To Object Area Size.

Suppose $\gamma = t/k$. Lemma 5.1 shows that as γ is reduced, the mapping efficiency, β, increases but the number of object pages produced also increases. This is offset by less records clashing and producing fewer overflow pages which have to be sorted. However with $\gamma = 1$, the largest value considered, β is almost .75 for the most efficient mapping and so very few overflow pages will be produced anyway. For example, if the source file occupies 12 pages, table 5.25 shows the effect of varying γ with the most efficient mapping.

γ	EFFICIENCY OF MAPPING	ACTUAL NUMBER OF SOURCE PAGES MAPPED	ACTUAL NUMBER OF OBJECT PAGES PRODUCED
0.125	.995	13	104
0.25	.975	13	52
0.375	.95	13	35
0.5	.91	14	28
0.625	.875	14	24
0.75	.835	14	19
0.825	.79	15	17
1.0	.745	16	16

TABLE 5.25

Clearly for any value of γ other than unity, the efficiency of the merge phase is going to be drastically reduced.

5.5.2 The Source Area Size And Order Of Merging

Once the ratio of source area size (t records or t' pages) to object area size (k records or k' pages) is determined, the value of t will set the size of the W.S.D. for the sort phase. The value of S determines the size of the W.S.D. for the merge phase. It was shown in section 5.4 that provided $|W.S.D.| < c$ reasonable performance can be achieved and this is the fundamental basis for the choice of these parameters.

If they are chosen such that $|W.S.D.| + 1 < c$ then main memory is available which the program cannot immediately use, although extra data will be available in main memory for use at a later time. By definition $|W.S.D.| = k' + 2$ for the sort phase and the optimum choice of t is clearly that which makes $|W.S.D.| + 1 = c$ i.e. $k' = c - 3$. Similarly, the 'best' choice of S is that which makes $|W.S.D.| + 1 = c$ i.e. $S = c - 2$.

Under most operating systems this is not possible because c usually varies with time and there is no easy way of measuring it. A guess at the value of c could be made by counting the number of tasks in execution and perhaps use a function of this as an upper bound. Alternatively the program could be written to be self modifying so as to adjust its working set size as the paging traffic varies. Another approach is to "play safe" and always set the parameters so that the working set of data is as small as possible. In this way reasonable performance can be achieved while using only the minimum system resources.

These problems do not arise when VM/370 is being used. The virtual machine performing the sort can be assigned a certain number

of real page frames using the SET RESERVED command and the algorithm's paraméters set accordingly.

5.6 Conclusion

A sorting technique has been proposed which differs from the more conventional algorithms in that it requires rather more memory space than is necessary to hold the data. However it does exhibit extremely good locality of reference, and this is an ideal way to take advantage of a paging system. In addition, the result of removing the most time consuming aspect (resolution of clashes) from address calculation sorting, which is an attractive algorithm anyway, is to produce an extremely fast and simple sort technique. Comparing the execution time of high level language implementations of the sort phase indicates that modified address calculation will be approximately an order of magnitude faster than Quicksort (Hoare, 1962). There are so many variables involved in the precise comparison of algorithms that it is not possible to quantify the exact performance of the suggested algorithm. For example different coding, different data, different hardware and many other factors affect practical performance.

The way that the paging performance can be expected to vary with changes in the amount of real storage available was shown in section 5.4. With correctly set parameters, the number of page faults produced is of the same order as the number which would occur with a conventional algorithm using sublists (Brawn et al, 1970).

The most important point to note is that modified address calculation will execute a great deal faster than conventional

address calculation, and most other algorithms, without losing the benefits of the file organisation suggested by Brawn et al, 1970. They show that the naive approach of treating virtual storage as if it were real can produce an increase in paging of three orders of magnitude or more.

Modified address calculation as suggested here is capable of various extensions. For example, in a system where real storage is extremely limited, the source area being vacated during execution could be used to hold the overflow records, or the suggested use of a combined source and object area could be investigated in more detail.

The analysis of the merge phase is in terms of the previously defined sort algorithm. However the conclusions apply to any merge operation, no matter how the sublists are obtained. If the order of merging is not less than the amount of real storage available, the high costs computed in section 5.4 will be incurred.

Finally, the simplicity of the algorithm makes it suitable for a microprogrammed implementation. Data fetching and address calculation within registers can easily be overlapped at the microprogram level and it ought to be possible to implement the entire algorithm in a few specially constructed machine instructions. This approach is suggested by Husson, 1970.

Chapter 6.

CONCLUSION

It seems unlikely that an inexpensive, extremely high capacity, random access storage device with a cycle time of just a few nanoseconds will become available in commercial quantities in the near future. Magnetic bubble technology and cryogenics look promising but the cycle times so far achieved are generally longer than presently available magnetic core and electronic storage. Hardware designers will always wish to use the fastest available storage and such storage will never be economic in massive quantities soon after its introduction. Thus it seems reasonable to assume that memory heirarchies will continue to be used. Since paged memories have been made to work fairly well, and at least one manufacturer has made available a whole range of machines with paged memories, it also seems reasonable to assume that memory heirarchies utilizing paging will continue to be used. If these assumptions are correct, two broad areas for further research are suggested.

Firstly, all algorithms which are to be implemented in a paged memory and which use large amounts of data should be examined to determine how well they use virtual memory facilities. Hopefully this thesis shows areas where improvement is possible but there are many algorithms used in numerical analysis and statistics for example which have not been considered.

Secondly, the architecture of computers with paged memories must be reviewed with the aim of reducing the costs of paging. This is achieved to a great extent in the S.C.C. 6700 (Watson, 1970), but other commercially available machines contain few innovations in the

architectural sense and are basically similar to the original Atlas (Kilburn et al., 1962). However, even if hardware improvements are made, paging will continue to be an expensive operation simply because the difference in access times between the two storage levels will mean delays in acquiring information located on the secondary storage device.

A fact which has appeared on several occasions through this thesis is that L.R.U. page replacement is generally superior to both RAND and F.I.F.O., at least as far as data references are concerned. Frequently it can approach the performance level of the MIN algorithm.

An objection which might be raised to the analyses presented is that the full L.R.U. algorithm is modelled. In practice a much simplified version of L.R.U. is usually used because of hardware deficiencies.

For example, I.B.M.'s System 360 Model 67 and all System 370 models have a 'use' bit and a 'change' bit associated with each half page of real storage. Thus it is possible to distinguish between those pages which have been referenced in a given time interval and those which have not. No hardware facilities exist for classification beyond this simple partitioning. Random selection among non-referenced pages is the simple version of L.R.U. which is often used, but very frequent inspection of these bits by the operating system can reveal the least recently used page.

Rather surprisingly, a hardware implementation of L.R.U. replacement is relatively simple on these machines and requires only a minor addition to the hardware. One register for each real storage page frame is all that is required.

These registers should be continuously incremented by clock pulses and will be referred to here as use timers. Since another register (the storage protection key) is interrogated for every storage reference anyway, it is relatively trivial to reset the corresponding use timer to zero. Full L.R.U. replacement within the set of pages being used by a program then consists of selecting the page from that set whose timer has the highest reading. This selection is also a comparatively simple hardware operation and could be a continuously operating process so that no delay is incurred when a page has to be selected for displacement.

If this type of hardware is used, it is reasonable to implement in addition, hardware measurement of the frequency of use of each page. The decision of which page to displace could then be based on a combination of frequency of use and time since last use (L.R.F.U. page replacement). Further investigation is needed to determine how worthwhile this would be.

Working Set allocation is becoming increasingly popular. If modified hardware were available, it might be beneficial to redefine a program's working set in terms of the frequency of use of its pages rather than merely those pages referenced during the 'window' interval.

Although they are not paging situations in the normal sense, similar principles are used in the cache memories of I.B.M.'s System 360 Model 85 and System 370, and a hardware implementation of L.R.U. replacement is used when a block has to be displaced. It should be noted that many of the results obtained here may be used to advantage on these machines.

The idea of the Working Set of Data has been used extensively. It has the useful property of frequently showing exactly how much real storage is required for efficient operation and this value is often clearly marked by a very sharp rise in paging when it is not available.

In contrast, algorithms have been considered for which no W.S.D. can be defined (e.g. the optimum search technique). In this case, efficient operation is possible with any quantity of real storage, the amount of paging decreasing smoothly as real storage increases.

REFERENCES.

1. Belady, L.A.: "A Study of Replacement Algorithms For A Virtual Storage Computer", I.B.M. Systems Journal Vol. 5, No. 2, 1966.

2. Bobrow, D.G., D.L. Murphy: "Structure Of A Lisp System Using Two Level Storage", Comm. ACM, Vol. 10, No. 3, 1967.

3. Brawn, B.S., F.G. Gustavson: "Program Behaviour In A Paging Environment", A.F.I.P.S., F.J.C.C., Vol. 33, 1968.

4. Brawn, B.S., F.G. Gustavson, E.S. Mankin: "Sorting In A Paging Environment", Comm. ACM, Vol. 13, No. 8, 1970.

5. Brooker, R.A.: "Some Techniques For Dealing With Two Level Storage", Computer Journal, Vol. 2, 1960.

6. Coffman, E.G., J. Eve: "File Structures Using Hashing Functions", Comm. ACM, Vol. 13, No. 7, 1970.

7. Coffman, E.G., L.C. Varian: "Further Experimental Data On The Behaviour of Programs In A Paging Environment", Comm. ACM, Vol. 11, No. 7, 1968.

8. Cohen J.: "A Use of Fast And Slow Memories In List Processing Languages", Comm. ACM, Vol. 10, No. 2, 1967.

9. Denning, P.J.: "The Working Set Model For Program Behaviour", Comm. ACM, Vol. 11, No. 5. 1968.

10. Denning P.J.: "Virtual Memory", Computing Surveys, Vol. 2, No. 3, 1970.

11. Denning, P.J.: "Third Generation Computer Systems", Computing Surveys, Vol. 3, No. 4, 1971.

12. Denning, P.J.: "On Modelling Program Behaviour", A.F.I.P.S., F.J.C.C., Vol. 40, 1972.

13. Denning, P.J., S.C. Schwartz: "Properties Of The Working Set Model", Comm. ACM, Vol. 15, No. 3, 1972.

14. Fine G.H., C.W. Jackson, P.V. M^{C}Isaac: "Dynamic Program Behaviour Under Paging", Proc. 21st ACM National Conference, 1966.

15. Flores, I.: "Computer Time For Address Calculation Sorting", Jour. ACM, Vol. 7, No. 4, 1960.

16. Gibson, C.T.: "Time Sharing On The 360/67", A.F.I.P.S., F.J.C.C., Vol. 28, 1966.

17. Hatfield, D.J., J. Gerald: "Program Restructuring For Virtual Memory", I.B.M. Systems Journal, Vol. 10, No. 3, 1971.

18. Hoare C.A.R.: "Quicksort", Computer Journal, Vol. 5, No. 1, 1962.

19. Hoare C.A.R.: Algorithm 64 (Quicksort), Comm. ACM, Vol. 4, No. 7, 1961.

20. Husson S.S.: "Microprogramming: Principles And Practices", Prentice Hall, 1970.

21. Isaac E.J., R.C. Singleton: "Sorting By Address Calculation", Jour. ACM, Vol. 3, 1956.

22. Joseph M.: "An Analysis Of Paging And Program Behaviour", Computer Journal, Vol. 13, No. 1, 1970.

23. Kendall, M.G., A. Stuart: "The Advanced Theory Of Statistics", Griffen & Co., 1968.

24. Kilburn T., D.B.G. Edwards, M.J. Lanigan, F.H. Summer: "One Level Storage System", I.R.E. Transactions On Electronic Computers, Vol. 11, No. 2, 1962.

25. Kilburn T., G.C. Tootill, D.B.G. Edwards, B.W. Pollard: "Digital Computers At Manchester University", Proceedings I.E.E., Vol. 100, 1953.

26. Lauer, H.C.: "Bulk Core In A 360/67 Time Sharing System", A.F.I.P.S., F.J.C.C., Vol. 31, 1967.

27. Lewis, P.A.W., A.S. Goodman, J.M. Miller: "A Pseudo Random Number Generator For System 360", I.B.M. Systems' Journal, Vol. 8, No. 2, 1969.

28. McKellar, A.C., E.G. Coffman: "The Organisation of Matrices And Matrix Operations In A Paged Multiprogramming Environment", Comm. ACM, Vol. 12, No. 3, 1969.

29. Moler C.B.: "Matrix Computation With Fortran And Paging", Comm. ACM, Vol. 15, No. 4, 1972.

30. Morris, R.: "Scatter Storage Techniques", Comm. ACM, Vol. 11, No. 1, 1968.

31. Nielson, N.R.: "The Simulation Of Time Sharing Systems", Comm. ACM, Vol. 10, No. 7, 1967.

32. Parmlee, R.P., T.I. Peterson, C.C. Tillman, D.J. Hatfield: " Virtual Storage And Virtual Machine Concepts ", I.B.M. Systems Journal, Vol. 11, No. 2, 1972.

33. Price, C.E.: "Table Look Up Techniques", Computing Surveys, Vol. 3, No. 2, 1971.

34. Randell, B.R. C.J. Kuehner: "Demand Paging In Perspective", A.F.I.P.S., F.J.C.C., Vol. 33, 1968.

35. Randell, B.R., C.J. Kuehner: "Dynamic Storage Allocation Systems", Comm. ACM, Vol. 11, No. 2, 1968.

36. Watson, R.W.: "Timesharing Design Concepts", M^{c}Graw Hill, 1970.

37. Weinberg, G.M.: "Programming And Compiling Strategies For Paging Systems", Software Practice And Experience, Vol. 2, No. 2, 1972.

APPENDIX.

Using the basic mapping process, suppose n records out of a possible t have been mapped into the object area which is of length k. In the object area the total number of possible arrangements of the n 'full' cells and the k - n 'empty' cells is ${}^{k}C_{n}$ and the n records inserted on the first pass will be uniformly distributed over the object area.

Where a group of adjacent cells are full/empty they constitute a full/empty block. The total number, k - n of empty cells in the object area will be made up of a collection of empty blocks of lengths $\geqslant 1$ and $\leqslant k - n$, each separated by one or more full cells. The lengths of the empty blocks thus constitute a composition of k - n.

By virtue of the second pass address function only those two empty cells immediately adjacent to a full block stand any chance of being filled.

Consider one of the two end cells of an empty block of length > 1. The probability that it gets filled by any particular record processed in the second pass is the probability that the record has a first pass address value of the adjacent occupied cell and that it has the required rank with respect to the occupant. This probability is (1/n).(1/2).

An empty block of length 1 has double the probability of being filled i.e. 1/n because clashes on either of the adjecant cells could be mapped into it.

The above is not strictly correct at the end points of the object area.

If the extreme end cells are full or part of an empty block of length >1 then all is well. If either end cell constitutes an empty block of length 1 it has probability 1/2n rather than 1/n of being filled. k and n will be quite large in practice and neglecting the end effect is unlikely to invalidate the results to any great extent.

If there are r blocks constituting the composition of k - n and ℓ of these are of length 1 ($0 \leqslant \ell \leqslant r-1$ in general) then clearly there are $2(r - \ell)$ cells with probability 1/2n and ℓ cells with probability 1/n, which can be filled. If k - n source records are processed during the second pass then the numbers of these mapped into each of the $2r - \ell$ cells have a multinomial distribution provided a suitable variable is included to represent those records not mapped into any empty cell. This leads to the following:-

Theorem.

During the second pass the expected number of extra successful mappings is given by:-

$$\sum_{i=1}^{k-n} i \sum_{r=1}^{\mathrm{Min}(n,k-n)} \sum_{\ell=\mathrm{Max}(0,2r-(k-n))}^{r-1} \frac{{}^{r}C_{\ell}\ {}^{k-n-r-1}C_{k-n+\ell-2r}\ {}^{n+1}C_{r}}{{}^{k}C_{n}} \sum_{j=\mathrm{Max}(0,i-m)}^{\mathrm{Min}(1,\ell)} {}^{\ell}C_{j}\ {}^{m}C_{i-j}$$

$$\sum_{u=0}^{i-j} (-1)^{i-j-u}\ {}^{i-j}C_{u} \sum_{v=0}^{j} (-1)^{j-v}\ {}^{j}C_{v} \left(1 - \frac{\ell-v}{n} - \frac{m-u}{2n}\right)^{k-n}$$

The proof requires two lemmas.

Lemma A.1

$$\sum_{X_{\ell+i-j-z}=1}^{n-S_j-T_{i-j-1-z}} \frac{(n-S_j-T_{i-j-1-z})!}{X_{\ell+i-j-z}!(n-S_j-T_{i-j-z})!}\left(\frac{1}{2n}\right)^{X_{\ell+i-j-z}} \sum_{r=0}^{z}(-1)^{z-r}\left(1-\frac{\ell}{n}-\frac{(m-r)}{2n}\right)^{n-S_j-T_{i-j-z}} {}^{z}C_r$$

$$= \sum_{r=0}^{z+1}(-1)^{z-r+1}\ {}^{z+1}C_r\left(1-\frac{\ell}{n}-\frac{(m-r)}{2n}\right)^{n-S_j-T_{i-j-z-1}}$$

where $S_j = \sum_{q=1}^{j} X_q$, $T_{i-j} = \sum_{q=\ell+1}^{\ell+i-j} X_q$

Proof

$$\text{L.H.S.} = \sum_{r=0}^{z}(-1)^{z-r}\ {}^{z}C_r \sum_{X_{\ell+i-j-z}=1}^{n-S_i-T_{i-j-1-z}} {}^{n-S_j-T_{i-j-1-z}}C_{X_{\ell+i-j-z}}\left(\frac{1}{2n}\right)^{X_{\ell+i-j-z}}\left(1-\frac{\ell}{n}-\frac{(m-r)}{2n}\right)^{n-S_j-T_{i-j}}$$

reversing the order of summation.

$$= \sum_{r=0}^{z}(-1)^{z-r}\ {}^{z}C_r\left[\left(1-\frac{\ell}{n}-\frac{m-(r+1)}{2n}\right)^{n-S_j-T_{i-j-z-1}} - \left(1-\frac{\ell}{n}-\frac{m-r}{2n}\right)^{n-S_j-T_{i-j-z-1}}\right]$$

using the binomial theorem.

$$= \sum_{r=1}^{z+1}(-1)^{z-r+1}\ {}^{z}C_{r-1}\left(1-\frac{\ell}{n}-\frac{m-r}{2n}\right)^{n-S_j-T_{i-j-z-1}}$$

$$+ \sum_{r=0}^{z}(-1)^{z-r+1}\ {}^{z}C_r\left(1-\frac{\ell}{n}-\frac{m-r}{2n}\right)^{n-S_j-T_{i-j-z-1}}$$

$$= \sum_{r=1}^{z} (-1)^{z-r+1} \, {}^{z+1}C_r \left(1 - \frac{\ell}{n} - \frac{m-r}{2n}\right)^{n-S_j-T_{i-j-z-1}}$$

$$+ \; {}^{z}C_z \left(1 - \frac{\ell}{n} - \frac{m-(z+1)}{2n}\right)^{n-S_j-T_{i-j-z-1}} - {}^{z}C_0 \left(1 - \frac{\ell}{n} - \frac{m}{2n}\right)^{n-S_j-T_{i-j-z-1}}$$

Now ${}^{z}C_z = {}^{z+1}C_{z+1}$ and ${}^{z}C_0 = {}^{z+1}C_0$ and so:-

$$\text{L.H.S.} = \sum_{r=0}^{z+1} (-1)^{z-r+1} \, {}^{z+1}C_r \left(1 - \frac{\ell}{n} - \frac{m-r}{2n}\right)^{n-S_j-T_{i-j-z-1}}$$

= R.H.S.

Lemma A.2

Probability that a composition of k - n occurs with r parts, ℓ of which are 1. $= \dfrac{{}^{r}C_{\ell} \; {}^{k-n-r-1}C_{k-n-1-2r} \; {}^{n+1}C_r}{{}^{k}C_n}$

Proof

The generating function for the frequencies of compositions with r parts, ℓ of which are 1 is:-

$$C_r(t) = t^{2r-\ell} \, (1-t)^{-(r-\ell)} \, {}^{r}C_{\ell}$$

the coefficient of t^{k-n} is required.

$$C_r(t) = {}^{r}C_{\ell} \, t^{2r-\ell} \sum_{a=0}^{\infty} {}^{r-\ell+a-1}C_a \, t^a$$

$$= {}^{r}C_{\ell} \sum_{a=0}^{\infty} {}^{r-\ell+a-1}C_{a}\ t^{2r-\ell+a}$$

Let $k - n = 2r - \ell + a$ then $a = k - n - 2r + \ell$

Thus the coefficient of t^{k-n} in $C_r(t)$ is ${}^{r}C_{\ell}\ {}^{k-n-r-1}C_{k-n-2r+\ell}$

Each of these may occur in several ways because the occupied cells may be distributed in any way provided each empty block is separated, i.e. there is at least one full cell between two empty blocks. Thus in effect there are r + 1 different locations for the remaining n - (r - 1) like (full) cells and these can be arranged in ${}^{n+1}C_{r}$ different ways. The total number of arrangements of the n full cells and k - n empty cells is ${}^{k}C_{n}$ and so:-

$$\text{Required probability} = \frac{{}^{r}C_{\ell}\ {}^{k-n-r-1}C_{k-n+\ell-2r}\ {}^{n+1}C_{r}}{{}^{k}C_{n}}$$

Proof Of Theorem.

Consider a particular composition of k - n.

Let $X_1, \ldots, X_\ell$ represent the numbers of records mapped into the ℓ cells with probability 1/n of being filled. Let $X_{\ell+1}, \ldots, X_{\ell+m}$ represent the numbers of records mapped into the m cells with probability 1/2n of being filled, where $m = 2(r - \ell)$. Let X represent the number of records not mapped into any of the above $\ell + m$ cells with probability $(1 - \ell/n - m/2n)$.

The probability that i of the $X_1, \ldots, X_{\ell+m}$ are greater than zero is the probability that i are successfully mapped during the second pass with this particular composition of k - n. If j of the i are of the type $X_1, \ldots, X_\ell$ i.e. mapped with probability 1/n, and the rest, i - j, are of the type $X_{\ell+1}, \ldots, X_{\ell+m}$ i.e. mapped with probability 1/2n, then clearly if the probability of successfully mapping i records

is known it is only necessary to sum over j.

Since $X_1, \ldots, X_\ell$ are effectively the same one need only consider $X_1, \ldots, X_j > 0$ and multiply by ${}^{\ell}C_j$, similarly $X_{\ell+1}, \ldots, X_{\ell+i-j} > 0$ and multiply by ${}^{m}C_{i-j}$.

Let E_i = probability(i are successfully mapped during second pass with this particular composition of k - n)

then
$$E_i = \sum_{j=\mathrm{Max}(0,i-m)}^{\mathrm{Min}(i,\ell)} \{ {}^{\ell}C_j \, {}^{m}C_{i-j} \, \mathrm{pr}(X_1>0, \ldots, X_j>0, X_{j+1}=0, \ldots, X_\ell=0,$$
$$X_{\ell+1}>0, \ldots, X_{\ell+i-j}>0, X_{\ell+i-j+1}=0, \ldots, X_{\ell+m}=0, X=k-n-\sum_{q=1}^{\ell+m} X_q) \}$$

Define $S_j = \sum_{q=1}^{j} X_q$, $T_{i-j} = \sum_{q=\ell+1}^{\ell+i-j} X_q$ and $n' = k - n$ as in Lemma A.1.

then:-

$$E_i = \sum_j \{ {}^{\ell}C_j \, {}^{m}C_{i-j} \sum_{X_1=1}^{n'} \sum_{X_2=1}^{n'-S_1} \cdots \sum_{X_j=1}^{n'-S_{j-1}} \sum_{X_{\ell+1}=1}^{n'-S_j} \cdots \sum_{X_{\ell+i-j}=1}^{n'-S_j-T_{i-j-1}} \frac{n!}{X_1! \ldots X_{\ell+i-j}! \, (n'-S_j-T_{i-j})!}$$
$$\cdot (1/n)^{S_j} \cdot (1/2n)^{T_{i-j}} \cdot (1 - \ell/n - m/2n)^{n'-S_j-T_{i-j}} \}$$

$$= \sum_j {}^{\ell}C_j \, {}^{m}C_{i-j} \sum_{X_1=1}^{n'} \frac{n'!}{(n'-S_1)!\,X_1!} (1/n)^{X_1} \sum_{X_2=1}^{n'-S_1} \frac{(n'-S_1)!}{(n'-S_2)!\,X_2!} (1/n)^{X_2} \cdots$$
$$\cdots \sum_{X_j=1}^{n'-S_{j-1}} \frac{(n'-S_{j-1})!}{(n'-S_j)!\,X_j!} (1/n)^{X_j} \sum_{X_{\ell+1}=1}^{n'-S_j} \frac{(n'-S_j)!}{(n'-S_j-T_1)!\,X_{\ell+1}!} (1/2n)^{X_{\ell+1}} \cdots$$
$$\cdots \sum_{X_{\ell+i-j}=1}^{n'-S_j-T_{i-j-1}} \frac{(n'-S_j-T_{i-j-1})!}{(n'-S_j-T_{i-j})!\,X_{\ell+i-j}!} (1/2n)^{X_{\ell+i-j}} (1 - \ell/n - m/2n)^{n'-S_j-T_{i-j}}$$

$$= \sum_{j} \{ {}^{\ell}C_{j} \; {}^{m}C_{i-j} \sum_{u=0}^{i-j} (-1)^{i-j-u} \; {}^{i-j}C_{u} \sum_{v=0}^{j} (-1)^{j-v} \; {}^{j}C_{v} \left(1 - \frac{\ell-v}{n} - \frac{m-u}{2n}\right)^{n'} \}$$

by repeated use of lemma A.1.

Let $q_{r,\ell}$ represent the probability obtained in lemma A.2.

Then:-

$$\text{pr(i successfully mapped)} = \sum_{r,\ell} q_{r,\ell} E_i$$

and so:-

$$\text{Exp. no. of successful mappings} = \sum_{i=1}^{k-n} i \sum_{r,\ell} q_{r,\ell} E_i$$

$$= \sum_{i=1}^{k-n} i \sum_{r=1}^{Min(n,k-n)} \sum_{\ell=Max(0,2r-(k-n))}^{r-1} \frac{{}^{r}C_{\ell} \; {}^{k-n-r-1}C_{k-n+\ell-2r} \; {}^{n+1}C_{r}}{{}^{k}C_{n}} \cdot$$

$$\sum_{j=Max(0,i-n)}^{Min(i,\ell)} {}^{\ell}C_{j} \; {}^{m}C_{i-j} \sum_{u=0}^{i-j} (-1)^{i-j-u} \; {}^{i-j}C_{u} \sum_{v=0}^{j} (-1)^{j-v} \; {}^{j}C_{v} \cdot$$

$$\left(1 - \frac{\ell-v}{n} - \frac{m-u}{2n}\right)^{n'}$$