20.12.78

VIRTUAL MEMORY TECHNIQUES FOR SMALL COMPUTERS

John Nguyen Thanh THUY, B.E. (Hons.)

Being a Thesis submitted

for the

DEGREE OF MASTER OF ENGINEERING SCIENCE

in the

ELECTRICAL ENGINEERING DEPARTMENT

UNIVERSITY OF ADELAIDE

November, 1976

Awarded April 1978

To my wife

Catherine-Nuong

DECLARATION

This thesis contains no material which has been accepted for the award of any other degree or diploma in any University, and to the best of my knowledge and belief it contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

J.N.T. THUY

ACKNOWLEDGEMENTS

I wish to thank my supervisor, Dr. D.A. Pucknell for his wholehearted help, guidance and encouragement throughout the work of this thesis. I must also acknowledge the valuable assistance from C. Beare, C. Nettle, J. Rogers and especially B. Ackland for making use of his program.

Finally, I would like to thank my wife for her encouragement which has made everything possible.

TABLE OF CONTENTS

		Page
–	SUMMARY	i
–	KEYWORDS AND SYMBOLS	ii
CHAPTER I	INTRODUCTION TO THE VIRTUAL MEMORY CONCEPT	
1.1	Why Virtual Memory	1
1.2	History	3
1.3	Basic Concepts	3
1.4	Characteristics of a Virtual Memory System	7
CHAPTER II	AN ANALYSIS OF A VIRTUAL MEMORY SYSTEM	
2.1	Introduction	8
2.2	Replacement Policy	10
2.3	Prediction of Page Requests	17
CHAPTER III	SIMULATION OF A VIRTUAL MEMORY SYSTEM	24

CHAPTER IV REPLACEMENT ALGORITHMS

4.1 Direct Mapping 28
4.2 LRU Stack with Page-Associative Mapping 29
4.3 Simulation Results 30
4.4 Conclusion 35

CHAPTER V PREDICTION METHODS

5.1 Introduction 37
5.2 First Prediction Method 38
5.3 Second Prediction Method 40
5.4 Third Prediction Method 43
5.5 Conclusion 46

CHAPTER VI THE PREVENTION OF PUSHES

6.1 Introduction 48
6.2 Modified-Page Checking Scheme 49
6.3 Bottom-Page Avoiding Scheme 52

CHAPTER VII MODELS OF FAULT RATE AND AVERAGE ACCESS TIME FUNCTIONS

7.1 Introduction 59
7.2 Fault Rate Function 59
7.3 Average Access Time Function 62

CHAPTER VIII AN EVALUATION OF VIRTUAL MEMORY TECHNIQUES USING SUGGESTED MODELS

8.1 Introduction 65
8.2 Cost 65
8.3 Time 69
8.4 Limitations 73
8.5 Conclusion 74

CHAPTER IX — SUGGESTIONS FOR FUTURE WORK

9.1 Introduction 75

9.2 Restructuring of Programs 75

9.3 Expansion of the Virtual Address Space 78

CHAPTER X — CONCLUSION 81

REFERENCES 84

APPENDIX A — FLOW CHARTS OF THE SIMULATION PROGRAM A-1

APPENDIX B — SIMULATION PROGRAM LISTING B-1

APPENDIX C — NUMERICAL BACK-UP RESULTS C-1

APPENDIX D — ADMENDMENT D-1

SUMMARY

This thesis describes an investigation into a cost - effective method of expanding main memory of a computer. The method used is known as Virtual Memory which has the significant advantage of being transparent to programmers. Special attention is paid to small computers in a single - user environment. This is the focus of the study.

Basically this thesis consists of four main parts. Part 1 (Chapter 1 and 2) is a literature survey of the virtual memory concept for the last 15 years since it came to existence. Part 2 (Chapter 3, 4, 5 and 6) presents the results of the investigation in 3 main areas: Replacement algorithms, Prediction methods and the Prevention of Pushes. In Part 3 (Chapter 7,8), mathematical models of the virtual memory system are derived. An evaluation is made on all the techniques investigated using the suggested models. Part 4 (Chapter 9 and 10), discusses possible extensions of this project for future study which includes: Program restructuring and the expansion of the Virtual Address Space. Proper conclusions are then drawn.

Appendices contain the Listing and Flow Charts of the Simulation Program. Tables of Back-up numerical results are also included.

KEYWORDS AND SYMBOLS

- virtual memory	:	virtual storage.
- auxiliary memory	:	external storage, backing store.
- a page	:	a block: a group of words.
- page fault	:	page exception, translation exception.
- pre-paging	:	lookahead, predictive loading.
- a push	:	a write back, a paging-out, a transfer of data from main memory to backing store.
- a pull	:	a paging-in; a transfer of data from backing store to main memory.
- page frame	:	space in main memory for one page.
- fault rate	:	ratio of page faults and total number of references expressed in percentage.
- reference pattern	:	a sequence of page references in a program.
- address space	:	maximum range in which the CPU is capable of addressing. In V.M. System, it is called Virtual Address Space.

P	:	page size
M	:	main memory size
C	:	No. of page frames in main memory
F	:	total number of page faults of a program
f	:	fault rate
T	:	average access time
n	:	number of references executed in a program
t_1	:	access time of main memory
t_s	:	settling time of disk head or tape head
t_t	:	data transfering time per work of disk or tape
α	:	co-efficient of page transfers
R	:	ratio of Pushes over Pulls
HR	:	Hit Rate
HR	=	1-f
HR%	=	100%-f%

Note: The term "memory" in this thesis is meant to be "main memory". For example -
memory size : main memory size

CHAPTER I

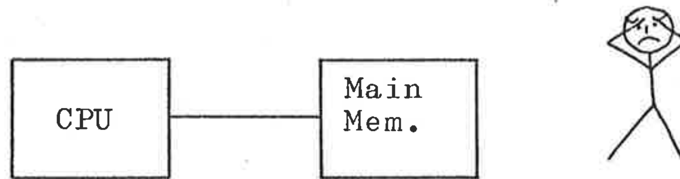
INTRODUCTION TO THE VIRTUAL MEMORY CONCEPT

1.1 Why Virtual Memory?

Since the earliest days of electronic computing it has been recognized that, one of the basic limitations of a digital computer is the size of its available memory. In most cases, it is neither feasible nor economical for a user to insist that every problem program fit into memory. The number of words of information in a program often exceeds the number of cells in memory. Since a cell can not hold more than one word at a time, extra words must be held in external storage. Therefore computer memories of very large overall capacity must be organized hierarchically, comprising at least two levels : Main Memory and Auxiliary Memory. Conventionally, overlay techniques are employed to exchange memory words and external-memory words whenever needed. This, of course, places an additional planning and coding burden on the programmer. For several reasons, it would be advantageous to rid the programmer of this function by providing him with a "Virtual Memory" larger than his program. A virtual memory system provides automatic executions of the memory overlay functions and thereby becomes transparent to the programmer. The philosophy of the Virtual Memory Concept is illustrated in Fig. 1.1.

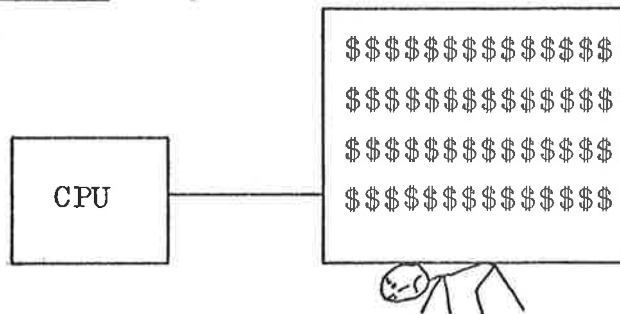
Why Virtual Memory?

* Problem

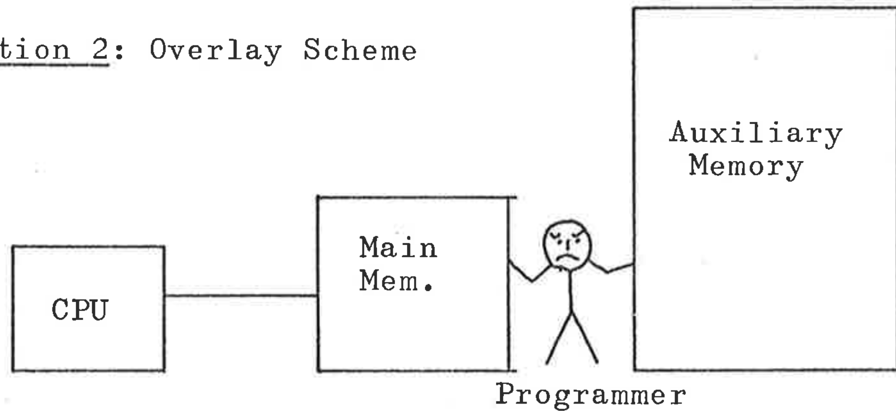


- High-level languages : Core eaters
- A need for more memory

* Solution 1: Buy more



* Solution 2: Overlay Scheme



* Solution 3: Virtual Memory

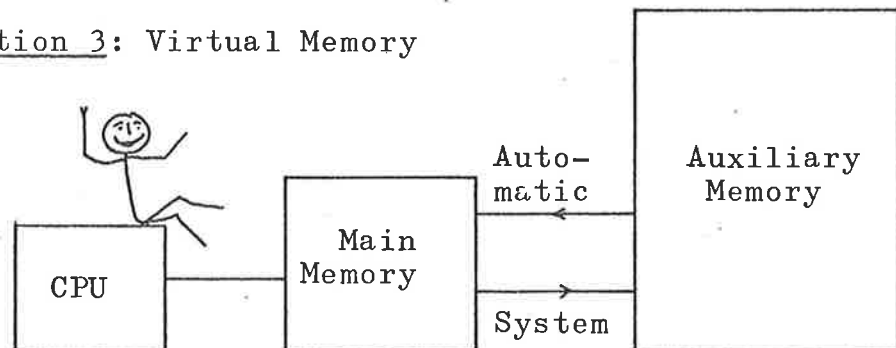


Fig. 1.1 The Philosophy of the Virtual Memory Concept

1.2 History

The first virtual memory computer, the Atlas (4), (5) appeared in 1961. A group at Manchester, England, published a proposal for a one-level store on the Atlas Computer which has had profound influence on computer system architecture. Their idea, known as "virtual memory" now gives the programmer the illusion that he has a very large main memory at his disposal, even though the computer actually has a relatively small main memory.

By the mid-1960s, the ideas of virtual memory had gained widespread acceptance, and had been applied to the internal design of many large processors IBM 360/85, CDC 7600 etc. (1). Since then the virtual memory system has been improved to meet the demand of speed and memory required by the advance of computer software (high level languages).

1.3 Basic Concepts

The basic concept of virtual memory system is the distinction between "logical address" in a program and "physical location" in main memory of a computer. In early computers, they were regarded as identical, therefore it seems impossible to have more addresses than actual locations. In a virtual memory system, an address used by the programmer is called a "name" or "virtual address", while an address used by the memory is called a "memory location". There are more virtual addresses than memory locations and that is

the reason why it is called "virtual".

Usually only a part of the program resides in main memory while the rest of the program is in auxiliary memory. In order to convert a given virtual address into a main memory location, an address translation mechanism is required. The program is divided into "blocks" or "pages" of equal sizes or segmented into segments of unequal sizes. Whenever a reference is made by the CPU, the virtual address is loaded into a register in the address translator. The first part of the virtual address is the page number and the rest is the location of a particular word within that page (Fig. 1.2). The address translator will check the page table (which shows all present pages in memory) to see if the required page is present in main memory. If it is, the address translator will point to a memory location which contains the required information. If it is not in main memory, a "page fault" occurs which will trigger an interrupt routine to bring the requested page in from auxiliary memory. This is called "paging" (Fig. 1.3). Usually before bringing in a new page to main memory (a pull), a page in main memory has to be written back to the auxiliary memory (a push). The policy which decides which page has to go to make room for the new page is called "replacement policy". There are a few replacement algorithms which will be discussed in Chapter II. The basic operation is shown in the Flow Chart of Fig. 1.4.

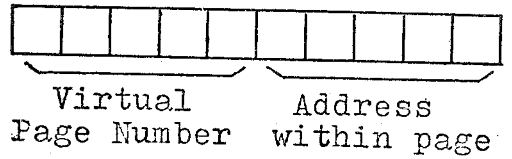


Fig. 1.2 Virtual Address

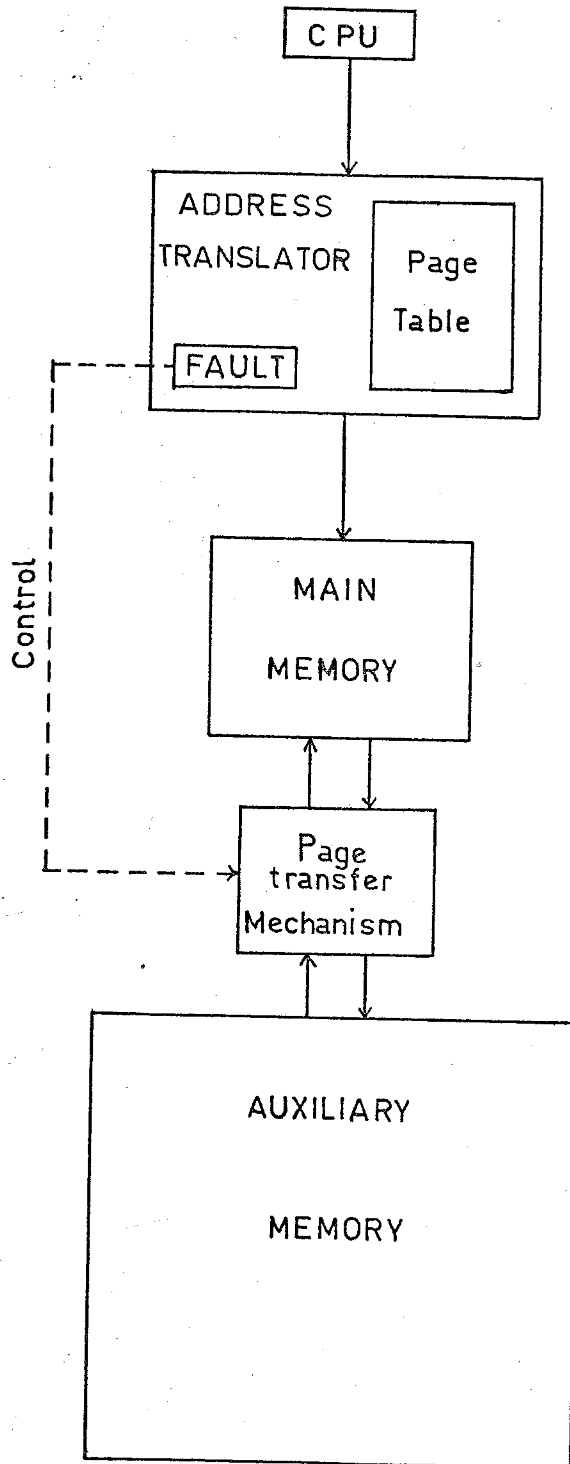


Fig. 1.3
A simple illustration of
a Virtual Memory System

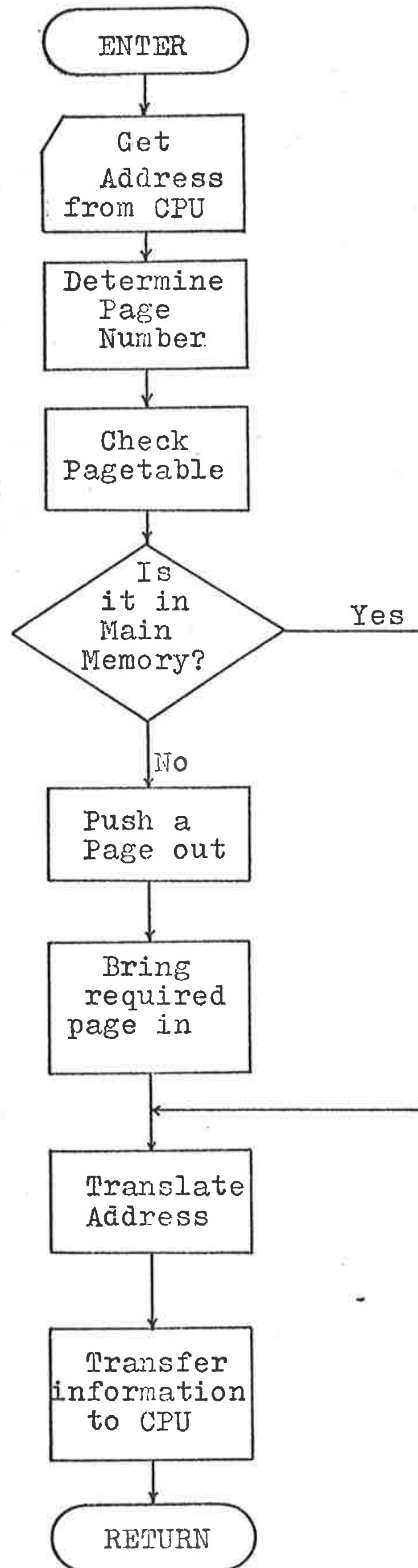


Fig. 1.4 Basic operation of a Virtual Memory System.

1.4 Characteristics of a Virtual Memory System

1.4.1 Fault rate f

The failure to locate a page in main memory is called a page fault or page exception. When this happens, a hardware mechanism is triggered to bring the requested page in from auxiliary memory. Let F be the total number of page faults for a particular program and n be the total number of references made during execution, fault rate is defined as the ratio $\frac{F}{n}$ which can also be expressed in percentage. It is an important function since it directly measures the system performance.

1.4.2 Parachore Curve

A parachore curve is the basic characteristics of a Virtual Memory System. It is the graph of the fault rate f or F versus the amount of main memory available M or C. This is usually done at a fixed page size. The shape of the curves are easily explained. If a program has less main memory space available to it, then usually more page faults will result. Typical parachore curves are shown in Fig. 1.5.

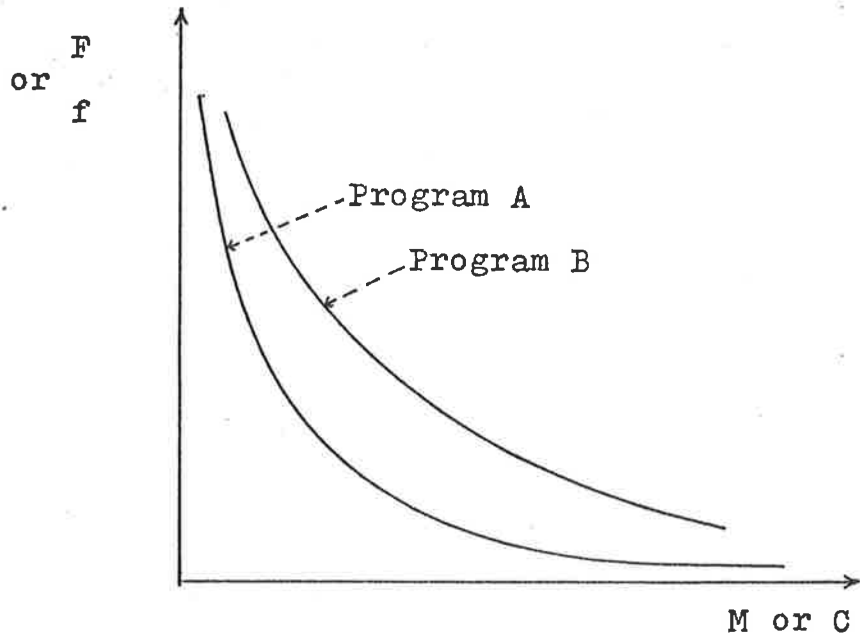


Fig. 1.5 Parachore curves of a Virtual Memory System.

Note

M: Main Memory Size
C: No. of Page Frames in Main Memory

CHAPTER II

AN ANALYSIS OF A VIRTUAL MEMORY SYSTEM

2.1 Introduction

A simple model of a virtual memory system is illustrated in Fig. 1.2. However, in general, a virtual memory system is not always like that for a number of reasons. For example, a page can be brought into main memory before a reference to it is made (pre-paging). Generally speaking, there are three policies which are used to supervise the activities of the system.

- Replacement Policy. A policy which is used to decide which page in main memory has to go to make room for the new page.
- Fetch Policy. A policy which is used to determine when a page shall enter main memory.
- Placement Policy. A policy which is used to decide where the new page is to be placed in the main memory. This policy only becomes distinct at special circumstances because usually it is included in the Replacement Policy.

In this analysis we will assume that the mapping between main memory and auxiliary memory is "page

associative mapping" i.e. a page in auxiliary can reside in any page frame in main memory. This mapping will be discussed carefully in Chapter III.

2.2 Replacement Policy

Knowing the basic operation of virtual memory system, one immediately recognizes that a policy is needed to determine which page in main memory ought to go to make room for the in-coming new page when it is requested. Theoretically, the best replacement pushes a "dead" page, i.e. a page no longer needed by the program run. The worst replacement occurs when a page is referenced immediately after being pushed out. Several algorithms have been suggested which have different assumptions about the behaviour of the running program.

2.2.1 Random Algorithm (RAND)

This algorithm assumes that all pages in a program are equally likely to be referenced at any time. Therefore a page is selected randomly to be pushed out of main memory. This algorithm is not based on "historical" information about memory usage. It is said to be "static".

2.2.2 First In First Out Algorithm (FIFO)

Another algorithm in the "static group" is the FIFO algorithm. The strongest argument

for FIFO probably is the fact that it is easier to step a cyclic counter than to generate a random number. The first to come is the first to leave.

These "static" types of algorithm are usually easy to implement. However, the assumption that all pages have equal chances to be referenced at a time does not seem to hold in most cases. It can become disastrous in dealing with program loops which appear in almost all programs. A study made by L.A. Belady in the mid 1960s has shown unimpressive results given by these "static" algorithms. However these algorithms, especially FIFO have theoretical value. They can be used as a yardstick to measure the efficiency of other "dynamic" algorithms.

2.2.3 Atlas Algorithm (ATLAS)

This is the algorithm used by the first virtual memory computer: the Atlas in 1961 (4). It is based on the history of the absence and presence of a page in memory. Information is recorded about all blocks (pages).

Let t be the length of time since a page has been used, T be the length of the last period of inactivity of this page.

The page to be pushed is selected by 3 simple tests:

(i) Any page for which $t > T+1$

(ii) That page with $t \neq 0$ and $(T-t)$ max

(iii) That page with Tmax (all $t=0$)

The first rule selects any page which has been concurrently out of use for longer than its last period of inactivity. Such a page has probably ceased to be used by the program and is therefore an ideal one to be transferred to the backing store. The second rule ignores all pages with $t=0$ as they are in current use, and then selects the one which, if the pattern of use is maintained, will not be required by the program for the longest time. If the first two rules fail to select a page, the third would ensure that if the page finally selected is wrong, in that it is immediately required again, then, as in this case, T will become zero and the same mistake will not be repeated.

This algorithm attempts to detect loop behaviour in page reference patterns - which is not hoped to be done by RAND or FIFO - and then to minimize page traffic by maximizing the time between page transfers. However, L.A. Belady in his study (2) has shown that it only performs slightly better than RAND or FIFO. It is only

successful for looping programs. Performance is unimpressive for programs exhibiting random reference patterns. Implementation is costly (9). P.J. Denning in his paper (9) saw this algorithm as an extreme one which used too much "historical" data and therefore can have adverse effects.

2.2.4 Least Recently Used Algorithm (LRU)

This algorithm uses the history of the pages' most recent use in memory. Whenever a fresh page is demanded, the page unreferenced for the longest time is removed.

To store the "historical" data, each page has a flag, set to ON each time the page is referenced. The reset occurs immediately and automatically whenever the last flag is set to ON.

In another system, it is organized by a stack process illustrated in Fig. 2.1. Whenever a page is referenced, it is brought to the top of the stack. Other pages are pushed down one place. At any instant, the page at the bottom has the first preference to "retire", i.e. to be removed.

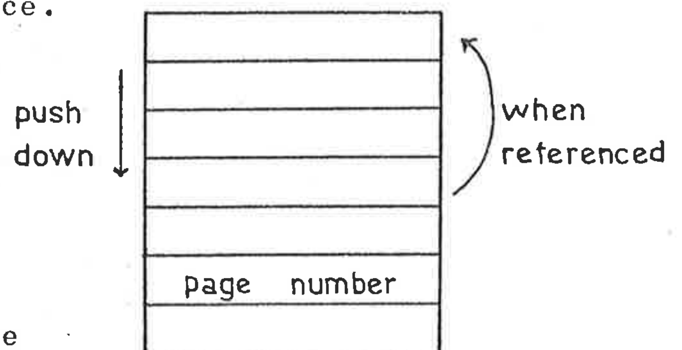


Fig. 2.1
Stack for LRU Algorithm

The study made by L.A. Belady (2) has indicated that this algorithm is the most effective algorithm among the four discussed so far (i.e. RAND, FIFO, ATLAS and LRU). It is still used widely in today's computers because of its relatively simple implementation. There is one disadvantage: In multi-programming situation, a page in one process can compete with another page in another process causing inefficiency.

2.2.5 Working Set Algorithm (WS)

This algorithm is explained by P.J. Denning in his paper (9) in 1968.

A working set is defined as a set $W(t,\tau)$ of a process at time t which contains information referenced by the process during the past τ seconds of the process time, i.e. the time interval $(t-\tau,t)$.

The basic replacement policy is to keep in the main memory those pages which have been referenced during the last τ msec. τ is an important parameter which affects the performance of the WS algorithm.

In general, the WS Algorithm can be considered as an LRU with variable size memory allocation. Using an LRU algorithm, pages are

always replaced when a page fault occurs.
This does not apply to WS algorithm. Here, page frames are freed whenever they have not been accessed during the last τ msec. Thus when a fresh page is needed, some page of a non-active process or some non-working-set page of an active process is removed. The WS Algorithm is applied individually to each program in a multiprogrammed memory, therefore avoiding the competing situation of the LRU.

The WS was shown to perform better than LRU in multiprogramming situation. However this algorithm appears to be expensive to implement (8).

2.2.6 Optimal or MIN Algorithm

This is an optimal algorithm which is the ideal algorithm wherein the storage page that is replaced is the one that will, in fact, be referenced farthest in the future. The MIN algorithm, though not realizable in practice, was used by Belady in his experimental work (2) as a basis for evaluating other algorithms. Actual virtual memory systems generally employ various approximate forms of the Working Set and LRU algorithms that typically generate 10 to 50 percent more translation exceptions (page faults) than the MIN algorithm.

2.2.7 Page Fault Frequency Replacement Algorithm (PFF)

This algorithm was introduced in 1972 by W.W. Chu and H. Opderbeck (8). It uses the measured page fault frequency P' as a feedback to control the system. A high page fault frequency indicates that a process is running inefficiently because of a shortage of page frames in main memory. A low page fault frequency might result in waste of memory space. Therefore to improve the system performance (e.g. space-time product), one or more page frames could be free.

The basic policy of the PFF algorithm is: whenever P' rises above a given critical P'_c, all referenced pages which cause page faults since they are not in memory, are brought into memory without replacing any pages. This results in an increase in the number of allocated page frames which usually reduces P'. On the other hand, if P' falls below P'_c, some page frames may be freed. These page frames are only freed at the time of a page fault and only those which have not been referenced since the last page fault occurred.

The advantage of this algorithm is program-independent since the page fault frequency P' is kept at a pre-determined level most of the time

irrespective to what kind of program. The time-space product is higher than that of the WS or LRU Algorithms.

The PFF can be considered as a WS algorithm with variable τ, where τ is determined by the page fault frequence P' and the lower limit of τ is $\tau_1 = \frac{1}{P'_c}$. It may also be viewed as a LRU algorithm with variable size of memory allocation where the size is determined by τ_1 and the inter-page-fault times.

This algorithm was claimed by the authors: W.W. Chu and H. Opderbeck to be easy to implement. It is very effective in multi-programming situation. However, it would have little use in single-user programming since the memory size cannot be varied.

2.3 Prediction of Page Requests

Fig. 1.3 has shown a simple illustration for a demand-paging policy which brings a page into memory only when requested (page fault occurs). This is called demand paging. The fetch-policy which allows a page to enter main memory before it is actually needed is called "pre-paging" or Prediction algorithm. There are a few algorithms: described by M. Joseph in his paper (7).

2.3.1 First Preciction Algorithm

Statistically, it seems that when a page

is being referenced, the next adjacent page is likely to be requested very soon. This occurs especially to small-sized pages. The first prediction algorithm is very simple: whenever a page is referenced, the next adjacent page is immediately brought into main memory. This only happens, of course, when it is not in main memory.

This algorithm is advantageous in some situations but the disadvantage is the penalty of faulty predictions. As illustrated in Fig. 2.2 if at certain time, page S is referenced, page S+1 will be brought into main memory. If the prediction fails and Page T is needed, T+1 will also be brought into main memory. Similarly if prediction fails again when V is referenced next instead of T+1, V+1 will also enter memory.

If prediction fails N times, in the worst case, we would have N page-frames occupied by N un-requested pages. This would certainly decrease the active main memory size and more page faults will follow.

S			
S+1	T		V
	T+1		V+1

Fig. 2.2 Disadvantage of the first prediction algorithm is the decrease in active main memory size by faulty predictions.

2.3.2 One Block Lookahead Algorithm (OBL)

One method of reducing this disadvantage would be to keep one page frame as a buffer to hold the predicted page. Whenever a new page R is demanded, the next page R+1, is loaded into a buffer. This is kept locked out and if the program next demands R+1, the lock is removed for immediate access; further, a new prediction is made and R+2 is loaded into another page buffer. There is only one buffer at a time and any page frame can be used. If prediction fails and the next requested page is T, then T+1 is brought into the buffer and S+1 is overwritten. This is shown in Fig. 2.3. In this way, a program would never use more than one page above its demand space and its total store usage with prediction would be well below that used with the demand algorithm and a page twice as large (7). The reason for locking out the predicted page is that by this it is possible to have a program drive the prediction algorithm by informing the supervisor exactly when prediction had succeeded.

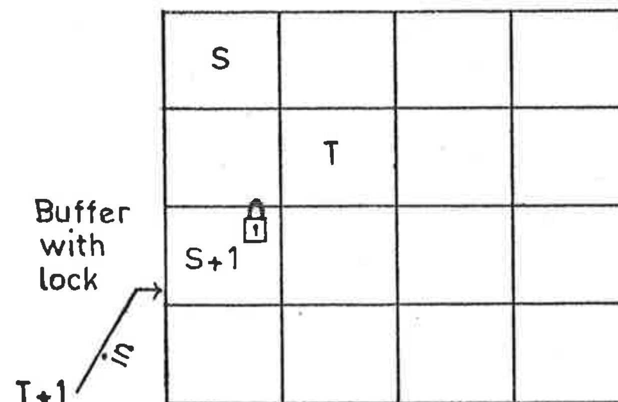


Fig. 2.3 Illustration of One Block Lookahead Algorithm.

2.3.3 Simple Prediction Algorithm (SP)

In many cases where the program's address space is growing in more than one direction. For example in a matrix multiplication program, references are made alternately to each matrix in different pages, say S and T. OBL Algorithm will result in the buffer being repeatedly overwritten and prediction failing. It could result in more paging when S+1, say is needed just after being overwritten by T+1.

To overcome this situation, the Simple Prediction keeps two buffers at any time. This is illustrated in Fig. 2.4.

This can be seen as a compromise between the first two algorithms. Of course we can introduce 3 or more buffers in the memory at the expense of main memory active space in case predictions fail. However it would result in more complication.

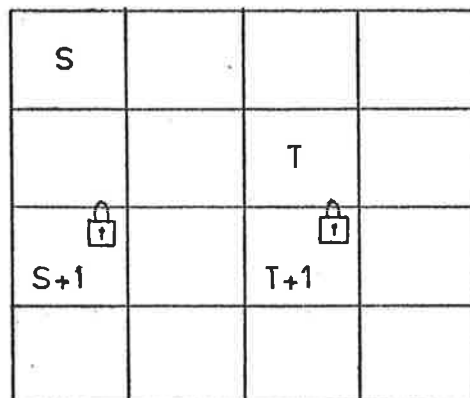


Fig. 2.4 Illustration of Simple Prediction Algorithm

2.3.4 Other Prediction Methods

J.P. Baer (17) has recently described two new prediction methods which may improve the One Block Lookahead method (OBL). They are named Temporal Lookahead and Spatial Lookahead.

The departure of these techniques from OBL is that the predicted page is no longer the next adjacent page. A prediction function PRED is defined and updated with the running of the program while PRED (i) = i+1 is always assumed in OBL. Note that in all cases, predictive loading only occurs at page faults.

a) Spatial Lookahead

Initially, the prediction function PRED (i) is set to i+1, i.e. the next adjacent page. At each page fault, if it is found that predictive loading of last fault did not occur or occurred but resulted in failure, then PRED is updated. The method is summarized below :

1. LAST = Ø
2. PRED (i) = i+1
3. NEXT FAULT, LOAD q
4. IF PREDICTIVE LOADING DIDNOT OCCUR LAST TIME, GOTO 6
5. IF PREDICTION LAST TIME HAD SUCCEEDED, GOTO 3

6. PRED (LAST) = q
7. IF PRED (q) IS IN MAIN MEM., GOTO 3
8. LOAD PRED (q)
9. LAST = q
10. GOTO 3

    end

b) Temporal Lookahead

Similarly to SL, this technique updates the Prediction function when the program is being executed. The main difference is that updating occurs at every fault. Suppose the LRU Stack is used and page q has highest priority, i.e. LRU (1st) = p and LRU (2nd) = q. The prediction function is then updated PRED (q) = p, hoping that the pattern will happen sometime later. The method is summarized below :

1. PRED (i) = i+1
2. NEXT FAULT, LOAD q
3. UPDATE PRED (LRU(2nd)) = q
4. IF PRED (q) IS IN MAIN MEM., GOTO 2
5. LOAD PRED (q)
6. GOTO 2

    end

TL is simpler than SL but Baer claimed more successful results for SL than for TL. In either case, the worst condition occurs when all predictions result in failure. Effective size of main

memory will become one page shorter.

The main criticism about these methods is the storage in main memory needed to store the PRED function. It may become critical in many situations.

For more details about these techniques readers are referred to Reference 17.

CHAPTER III

SIMULATION OF A VIRTUAL MEMORY SYSTEM

A great deal has been said in literature about the Virtual Memory techniques for large computers, but little attention has been paid on small computers with a single user. This is the reason for this study. Investigation of the virtual memory techniques is carried out here on mini or micro-computers in a single-user environment.

Most of the theory of the Virtual Memory Concept has been presented in Chapter I and II. It is important to verify this theory by experiments. Since a real model (hardware) is not flexible enough to vary at will, the system under study must be simulated in some way. In order to measure the performance, each machine instruction must be followed closely. Therefore high-level languages are not suitable for this purpose. A simulation program is written in ASSEMBLY Language of the NOVA computer which is shown in Appendix B. It is run on the NOVA 2 computer of Dept. of Electrical Engineering. The 32K core memory available is divided in 3 parts. Part 1 contains the simulation program, Part 2 is the Simulated Main Memory and the rest is Part 3 : the Simulated Backing Store. This is shown in Fig. 3.1.

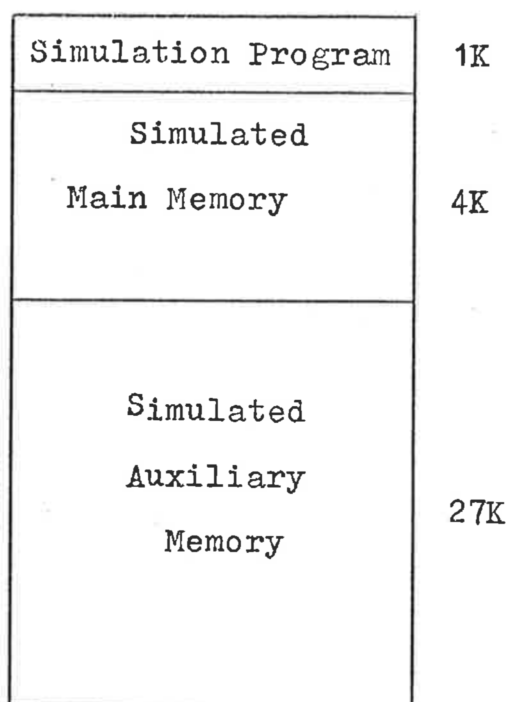


Fig. 3.1 Software construction of the Virtual Memory System on 32K Main Memory of the Nova 2 Computer.

The simulation program contains all simulated parts of the Virtual Memory System except memory. It resides on top of the core memory of the Nova Computer. Initially, a test program is stored in the Simulated Auxiliary Memory. The simulated main memory is then filled up with adjacent pages, beginning from the starting page. When running, instead of executing the test program directly, the Nova CPU passes control over to the simulation program which will in effect "run" the test program under virtual memory mode. In this way, all activities of the simulated virtual

memory system can be monitored and measured by a set of software counters.

Three test programs are used in this study :

* ASSEMBLY : FFT Program, 5K, $1.5\text{x}10^6$ references
* ALGOL : Inversion of matrix
  8K, $2.1\text{x}10^6$ references
* FORTRAN : Octal - Decimal Number Conversion
  5K, $4.\text{x}10^5$ references

Simulation are carried out in three main areas : which are presented in Chapter IV, V and VI.

1. Replacement Algorithm

In a single-user environment, techniques for multi-programming are eliminated. Therefore LRU stack is studied and compared with the simplest mapping of Auxiliary Memory onto Main Memory : the Direct Mapping.

2. Prediction Methods

This is also known as Predictive Loading. The first three prediction methods mentioned in Chapter II are compared. Spatial Lookahead and Temporal Lookahead methods are not investigated in this study because much of the main memory must be used to store the PRED Function for each page. This becomes unsuitable for small computers in most cases.

3. The Prevention of Pushes

This study is concerning with unnecessary write-back or push. Two schemes are suggested to cut down these unnecessary pushes.

CHAPTER IV

REPLACEMENT ALGORITHMS

A number of replacement algorithms have been discussed in Chapter II. The choice of a good algorithm is important since it reduces the fault rate and hence increases the system performance. Two algorithms which have different mappings are compared here: Direct Mapping and Least Recently Used Stack with Page Associative Mapping.

4.1 Direct Mapping

This is the simplest method, probably, to map Auxiliary Memory onto Main Memory. It can be summarized as follows:

Suppose Main Memory has C page frames, then in Auxiliary Memory, pages 1, C+1, 2C+1, 3C+1 etc. are mapped to frame 1 of Main Memory; pages n, C+n, 2C+n, 3C+n etc. are mapped to frame n of Main Memory. This mapping is illustrated in Fig. 4.1

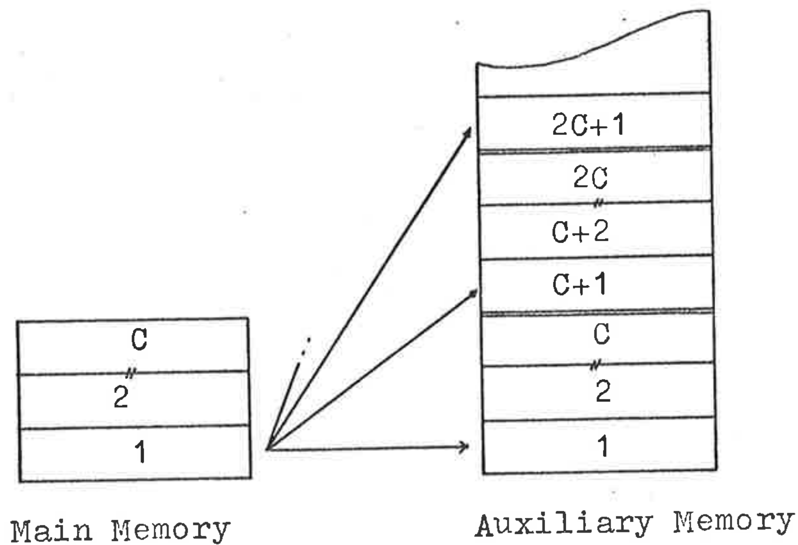


Fig.4.1 Direct Mapping

Arranging the mapping in this way, given a page in a particular program, there is only one frame in Main Memory for it to reside. When it is requested, any page which currently occupies this frame has to go.

4.2 LRU Stack with Page-associative Mapping

In contrast with Direct Mapping, the Page-associative Mapping provides more flexibility. Any page in Auxiliary Memory can reside in any frame in Main Memory. To decide where it should reside at a particular time, the Least Recently Used (LRU) Stack is employed. When a page is referenced, it is brought to the top of the stack and other pages are then pushed down to fill the 'hole' created. This algorithm is shown in Fig. 3.2 which is a reproduction of Fig. 2.1 in Chapter II.

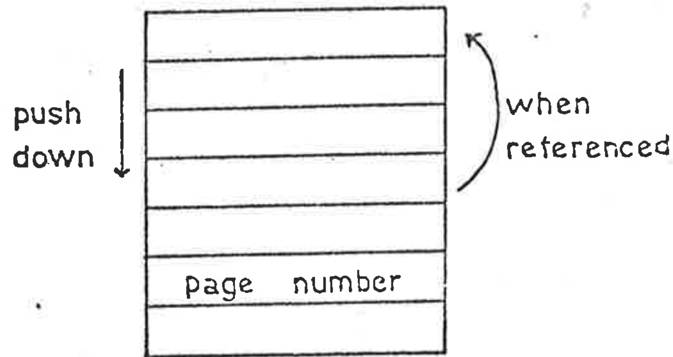


Fig. 4.2 Stack for LRU Algorithm

During actual program execution the pages cannot be shuffled physically, therefore a set of priority numbers are introduced. Each page frame in Main Memory is given a Priority number. When a page is referenced, its priority becomes highest, other numbers are decremented by 1 until the 'hole' is filled. At any time, the page which has lowest priority number (i.e. zero) is the first to be replaced.

4.3 Simulation Results

In a virtual memory system, two important variables are Memory size M and Page size P. A dependent variable $C = \frac{M}{P}$ is added which is the number of page frames in Main Memory. Fault Rate is a function of these variables.

$$f = f(C,M,P)$$

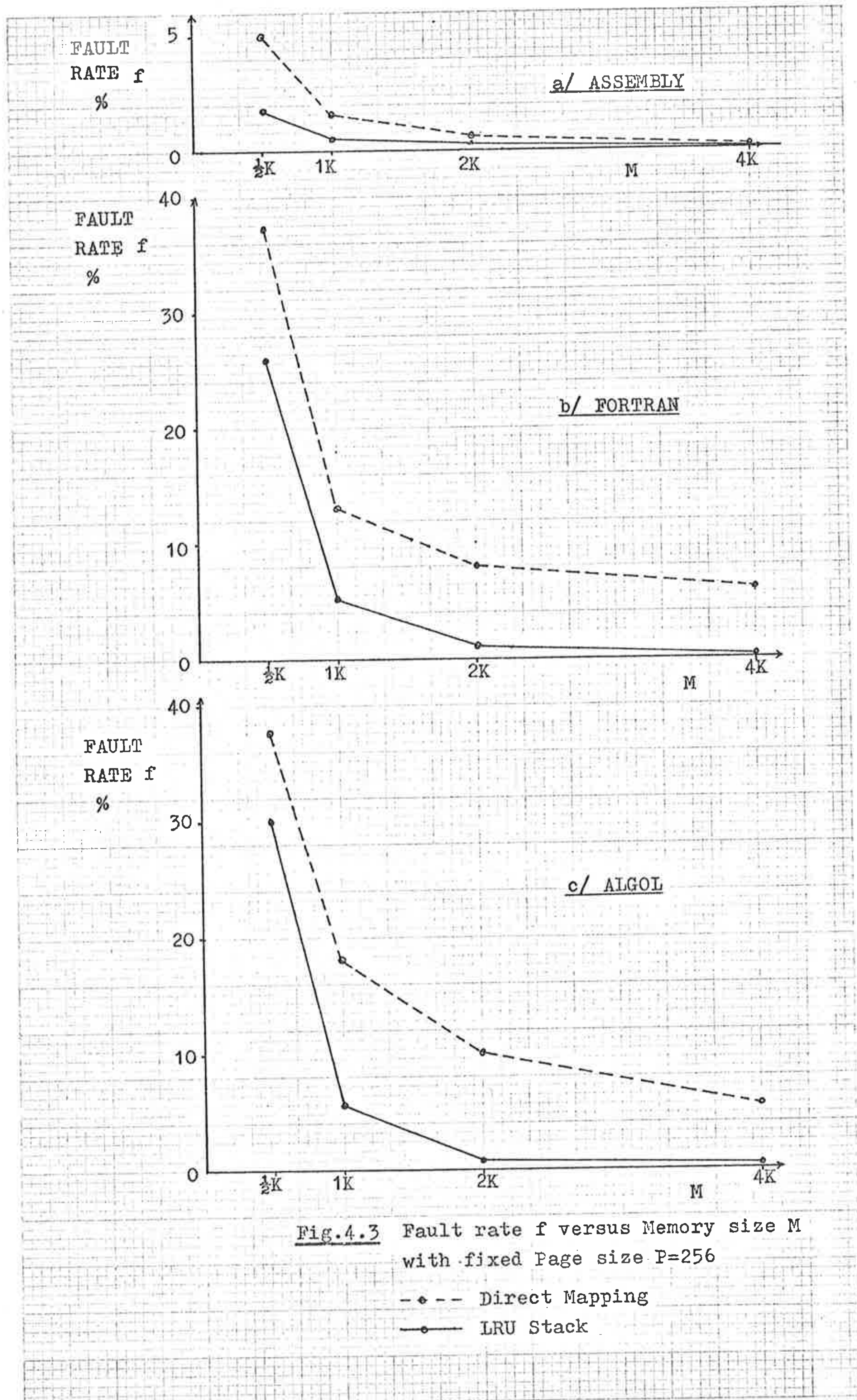


Fig.4.3 Fault rate f versus Memory size M with fixed Page size P=256

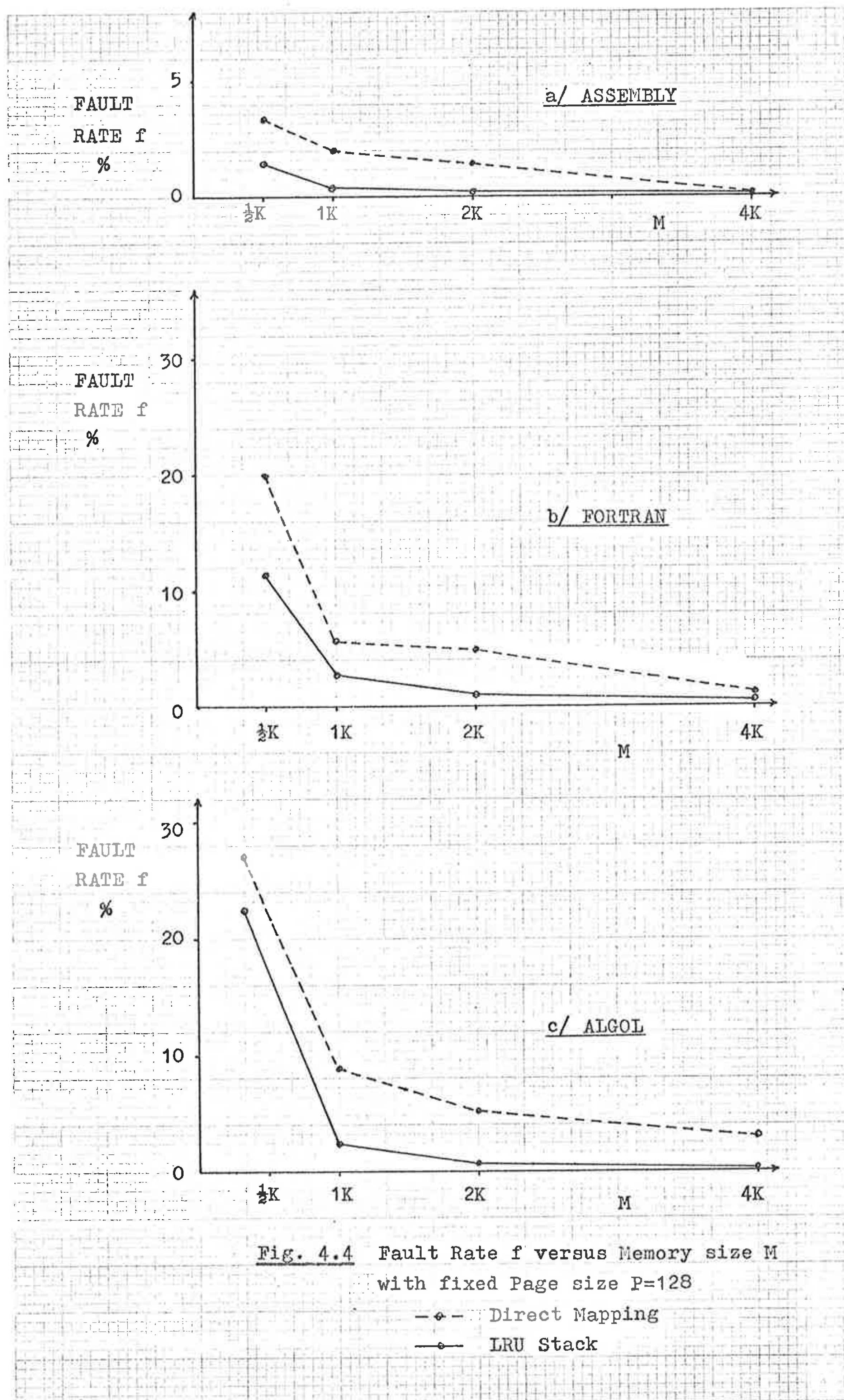


Fig. 4.4 Fault Rate f versus Memory size M with fixed Page size P=128

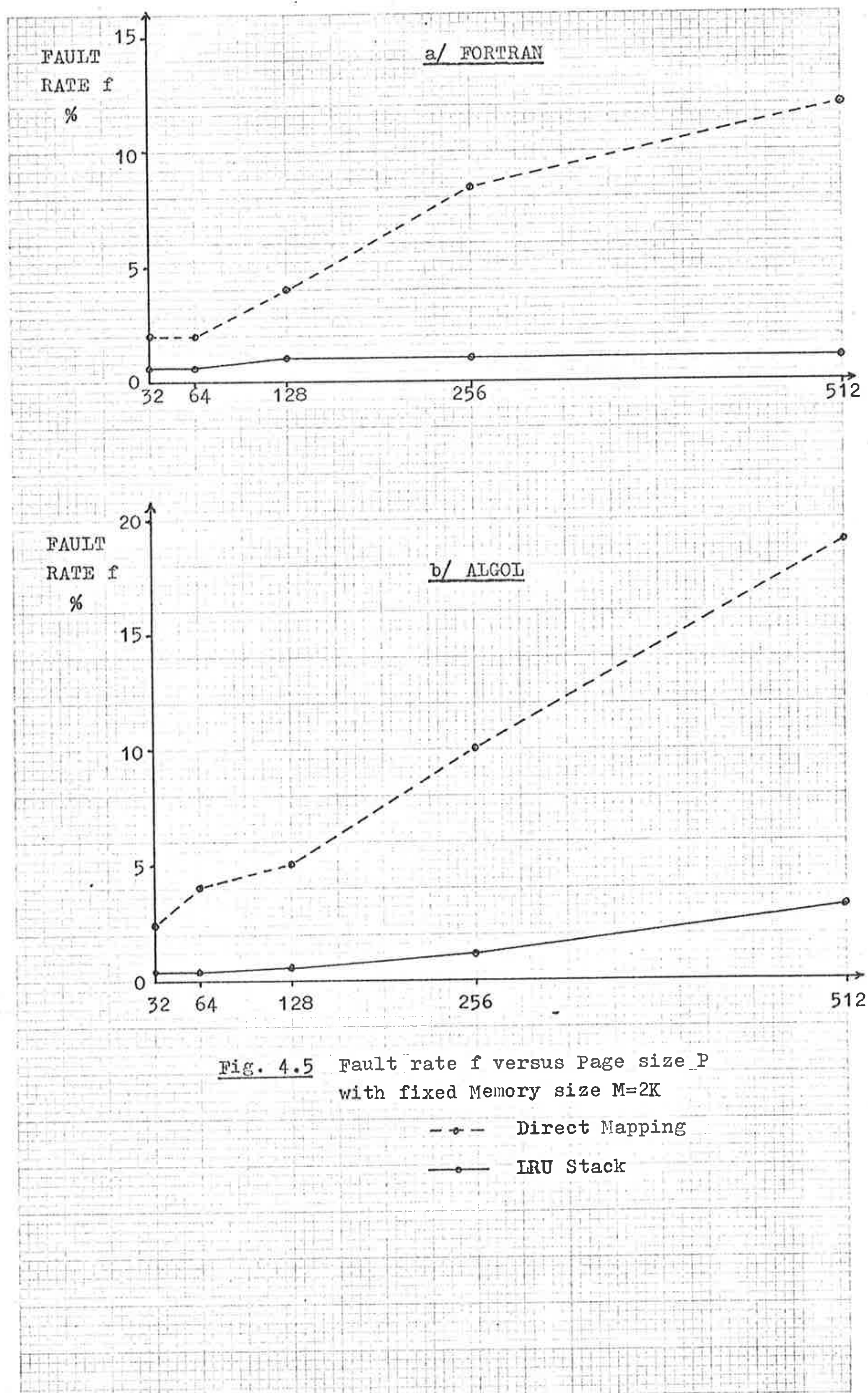


Fig. 4.5 Fault rate f versus Page size P with fixed Memory size M=2K

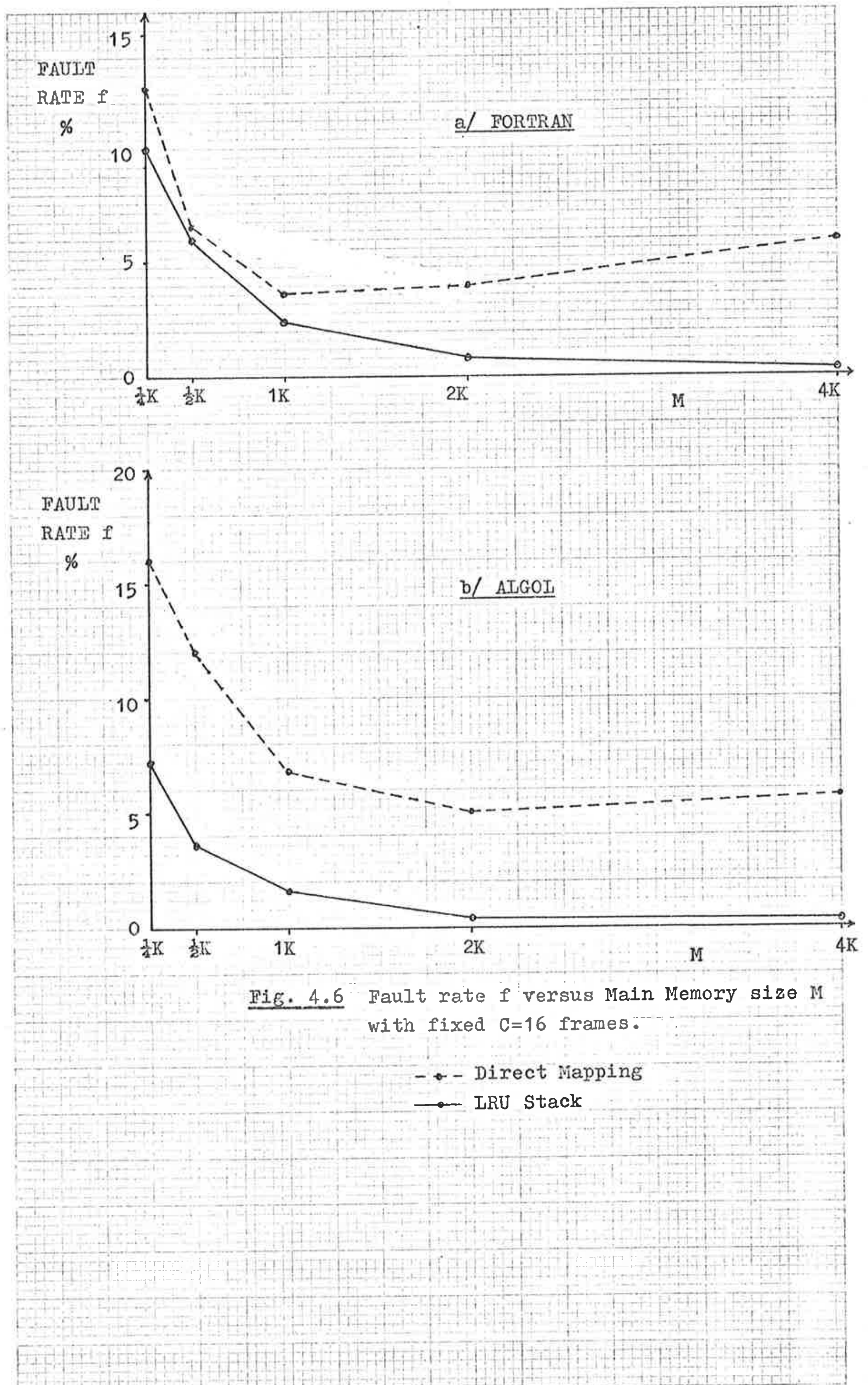


Fig. 4.6 Fault rate f versus Main Memory size M with fixed C=16 frames.

Fault Rate of Direct Mapping and LRU Stack are measured by simulations. With each variable in turn, kept fixed, Fault Rate is plotted versus the others. Fixed page sizes of 256 and 128 are shown in Figs. 4.3 and 4.4, fixed memory size 2K in Fig. 4.5 and a fixed number of page frame C=16 in Fig. 4.6.

4.4 Conclusion

The Direct Mapping and the LRU Stack have been compared along four directions (Program, M,C,P) in the constituted space of these dimensions (Program, M,P). The effectiveness of the latter has been verified. In other words, the LRU Stack does indeed perform better than the Direct Mapping.

Other points can also be made from these results :

- Fault Rate is lower with Assembly language than with High-level languages for a given Replacement Algorithm.
- In Direct Mapping algorithm, the Fault Rate increases with page size P. (Fig. 4.5). An increase in memory size would not always mean a decrease in Fault Rate (Fig. 4.6). This is probably due to the fact that large page size usually results in more conflicts between pages.

- The optimum page size would be 128 or 64 (Fig. 4.5). Page size 64 is not preferred because it would increase the number of page frames to a comparable

number, i.e. with 4K main memory, 64 page frames are needed which will result in unnecessary hardware complexity. As shown in Fig 4.4 page size of 128 gives better performance than page size of 256. (Fig. 4.3). With a memory size as small as 1K, P = 128 gives less than 3% Fault Rate which is really excellent.

For these reasons, a main memory of 2K or 4K with page size 128 and LRU stack is probably in the optimum area for mini or micro computers in a single - user environment.

CHAPTER V

PREDICTION METHODS

5.1 Introduction

Usually, a page is brought into main memory only when it is requested. If a page is brought in before it is demanded, the method is called pre-paging or Prediction. Simulations have been carried out to test these prediction methods which have been discussed in Chapter II. Since the page size of 128 words has been chosen in Chapter IV as the optimum page size, investigation is made at three points on the parachore curve of P = 128 words (Fig. 5.1) Point X : M = 1K or C = 8; Point Y : M = 2K or C = 16 and Point Z: M = 4K or C = 32. The same three test programs will be used. For the reason of convenience C will be taken as variable instead of M, for example F(32) means the Number of Page Faults at C = 32 page frames.

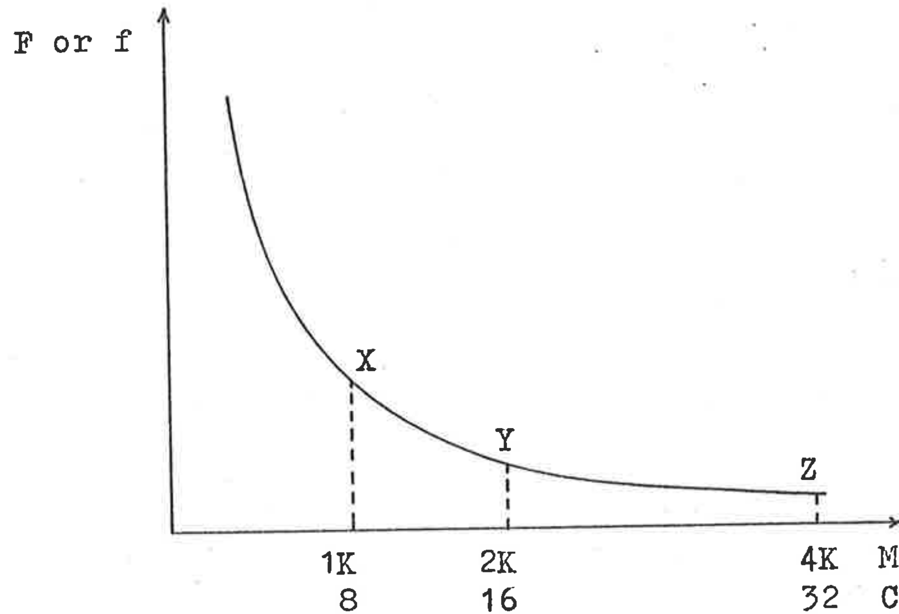


Fig. 5.1 Points of interest on the parachore curve of the Algol program with fixed P=128

5.2 First Prediction Method

In this method, whenever page N is requested to be transferred from auxiliary memory, the next adjacent page N+1 is also brought into Main Memory. It is based on the assumption that when a page is needed, the next adjacent page will be needed soon. Loading of the predicted page happens only at page faults and at every fault. After being brought into Main Memory, page N will be given the highest priority number and page N+1 the second highest number. Table 5.1 a,b,c show simulation results using this method compared with the "normal" LRU case (i.e. demand-paging). Numbers shown are numbers of page faults for a particular program.

a) C = 8 (M = 1K)

	Normal F(8)	First Method $F_1(8)$	Improvement %
ASSEMBLY	1583	5493	-247%
FORTRAN	10497	10771	-2.6%
ALGOL	47625	56115	-17.8%

b) C = 16 (M = 2K)

	Normal F(16)	First Method $F_1(16)$	Improvement %
ASSEMBLY	1008	872	13.5%
FORTRAN	2713	2655	2%
ALGOL	10879	11569	-6.3%

c) C = 32 (M = 4K)

	Normal F(32)	First Method $F_1(32)$	Improvement %
ASSEMBLY	248	151	39%
FORTRAN	235	180	23%
ALGOL	1726	1477	14%

Table 5.1 Page Faults of First Prediction Method

Results show that the improvement is totally negative at Point X, i.e. C=8, a small improvement at Point Y and a large improvement at Point Z.

5.3 Second Prediction Method (OBL)

The disadvantage of the First Prediction Method as discussed in Chapter II is the penalty incurred by unsuccessful predictions. We have no knowledge of how severe this penalty would be in any particular situation and therefore the method is more or less a "gamble". The performance of the system is totally dependent on "luck". We can win a great deal if we are "lucky" and lose very badly if we are not.

To avoid this disadvantage, the second method is employed which is similar to the first method. The departure from the first method is that after pages N and N+1 are brought into Main Memory, page N+1 will be given the lowest priority number instead of the second highest one. If a prediction fails, no reference is made to N+1 and it will be replaced immediately at the next page fault. Therefore the worst situation is the case of Main Memory with only one page less. (Hence the name One Block Lookahead)

Let F(C) and F(C-1) be the number of faults at 'memory size' C and (C-1) page frames respectively. The probability of successful prediction is called h. In Reference (15), J.L. Baer has derived an approximation of the performance of this technique.

The number of faults generated at C given h is :

$$F_2(C) = \frac{F(C)\ F(C-1)}{F(C) + hF(C-1)}$$

Worst case: h = 0 $\qquad F_2max(C) = F(C-1)$

Best case : h = 1 $\qquad F_2min(C) = \frac{F(C)F(C-1)}{F(C)+F(C-1)}$

Results are shown in Table 5.2 a, b, c along with the two estimated bounds $F_2max(C)$ and $F_2min(C)$.

a) C=8 (M=1K)

	Normal F(8)	Second Method $F_2(8)$	Upper Bound F(7)	Lower Bound $F_2min(8)$	Improvement %
ASSEMBLY	1583	2627	5022	1203	-66%
FORTRAN	10497	10329	12587	5723	1.6%
ALGOL	47625	50415	81945	30119	-5.8%

b) C+16 (M=2K)

	Normal F(16)	Second Method $F_2(16)$	Upper Bound F(15)	Lower Bound $F_2min(16)$	Improvement %
ASSEMBLY	1008	960	1087	523	4.7%
FORTRAN	2713	2595	3584	1326	4.3%
ALGOL	10879	9590	12109	5730	11.8%

Table 5.2 Page Faults of Second Prediction Method.

(Continued over page)

c) C=32 (M=4K)

	Normal F(32)	Second Method $F_2(32)$	Upper Bound F(31)	Lower Bound $F_2min(32)$	Improvement %
ASSEMBLY	248	191	258	126	23%
FORTRAN	235	187	284	128	20%
ALGOL	1726	1292	1788	878	25%

Results show that the improvement is negative at point X (C=8), a small improvement at point Y (C=16) and a large improvement at point Z (C=32).

Comparing with the First Prediction Method the Performance of this method is more predictable. The number of page faults lie exactly between the predicted upper and lower bounds. Note that with the First Method, the ASSEMBLY language program generated 5493 faults (Table 5.1a) which is more than the upperbound of the second method at this point, i.e. 5022 (Table 5.2a).

The reason why large improvement is achieved at C=32 over performance at C=16 and C=8 is due to the fact that at C=32 the upper bounds are very close to the normal number of faults. In other words $F(C-1) \simeq F(C)$

Therefore:

$$\text{Worst Case: } F_2max(C) = F(C-1) \simeq F(C)$$

$$\text{Best Case: } F_2min(C) \simeq \frac{F(C)}{2}$$

5.4 Third Prediction Method

The argument against the Second Method is that page N+1 might not have stayed in Main Memory long enough to gain benefit of the prediction. The Third Prediction Method is a compromise between the first two methods. In this arrangement, page N+1 is given priority number "3" after being brought in (this number is not affected by C). This will make it to stay at least until the page fault after next occurs. Doing it this way, the worst case would be the case in which Main Memory appears to be 2 pages shorter. The best case is rather difficult to derive analytically. However it is certain that it should be better than the best case of the Second Method.

Table 5.3 a, b, c show the results obtained for this prediction method.

a) C=8 (M=1K)

	Normal F(8)	Third Method $F_3(8)$	Upper Bound F(6)	Improvement %
ASSEMBLY	1583	5311	7515	-235%
FORTRAN	10497	10389	14497	1%
ALGOL	47625	50906	73697	-6.8%

Table 5.3 Page Faults of Third Prediction Method

(Continued over page)

b) C=16 (M=2K)

	Normal F(16)	Third Method $F_3(16)$	Upper Bound F(14)	Improvement %
ASSEMBLY	1008	940	1120	12%
FORTRAN	2713	2526	4318	6.8%
ALGOL	10879	9508	13893	6.7%

c) C=32 (M=4K)

	Normal F(32)	Third Method $F_3(32)$	Upper Bound F(30)	Improvement %
ASSEMBLY	248	164	273	33%
FORTRAN	235	185	340	21%
ALGOL	1726	1346	2002	23%

Table 5.3 Page Faults of Third Prediction Method

Results show that the Third Prediction Method is a compromise between the other two methods. It performs better than the Second Method in some cases, but not all. The effect of this method is widening the gap between the upper and lower bounds of F. There is no guarantee that it would perform better or worse than the second method.

To summarize the performance of the three prediction methods, percentage of improvement is plotted against C in Fig. 5.2

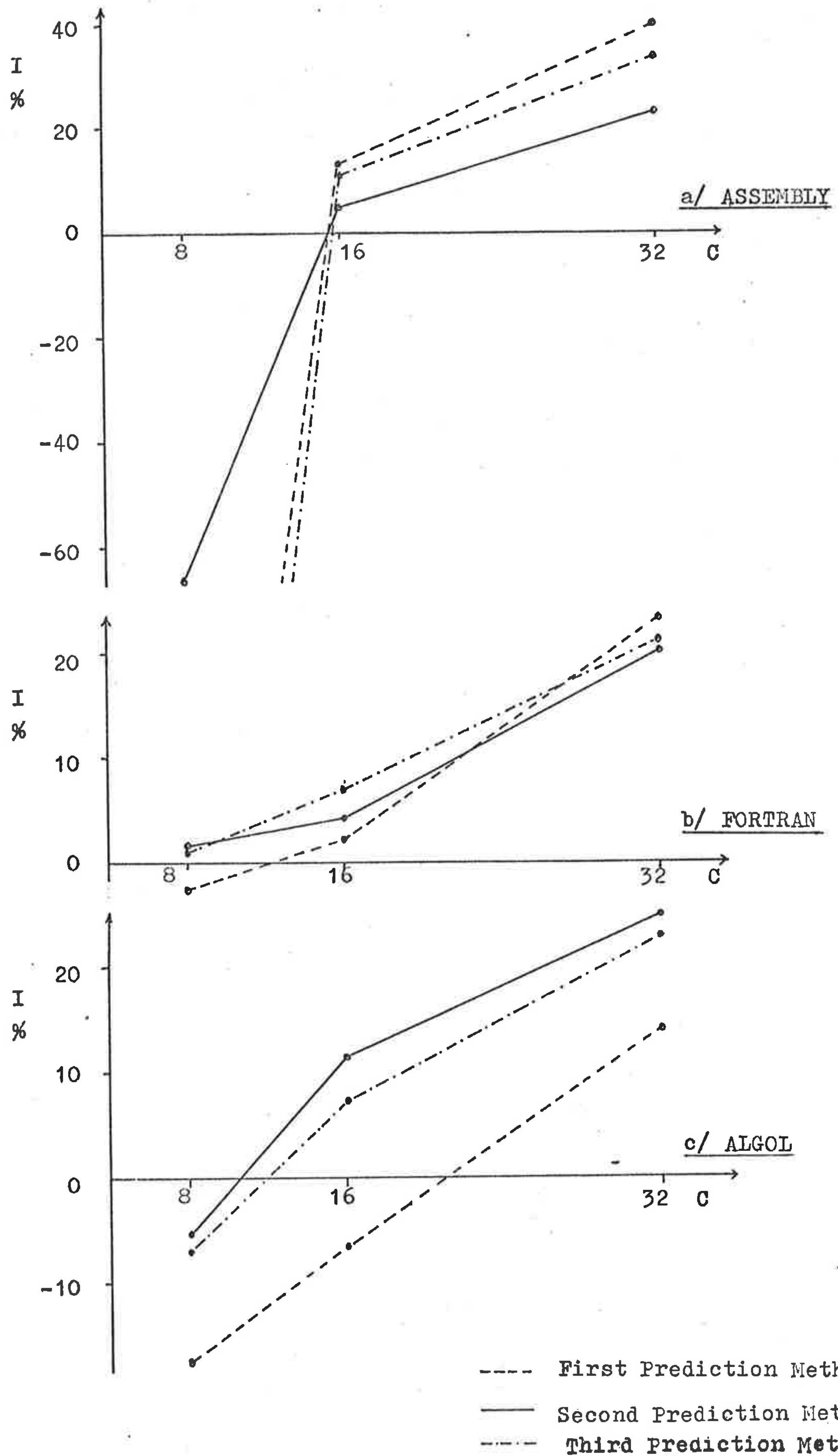


Fig. 5.2 Percentage of improvement I of the three Prediction methods

5.5 Conclusion

Based on the results obtained, it is concluded that the First Prediction Method is risky, totally dependent on luck. The Second and Third methods are somewhat comparable. However, as far as the amount of hardware and software involved in the implementation of these methods, the Second method is simplest of all. To demonstrate this point, recall that in a LRU Stack, when a page fault occurs, the requested page will replace the page of lowest priority. A procedure is then called to update the priority order and top priority number is given to the requested page. In case of the Second Method, when a predicted page is loaded, that procedure is simply avoided and the priority of the predicted page is automatically lowest (i.e. zero). Comparing with the First or Third Method in which the number "3" or the second largest priority number (C-2) is required, the implementation of the Second Method is much simpler. No additional hardware or software is needed. Furthermore, the Second Method is less risky than others because its upper limit F(C-1) on page faults is lowest. In other words, the Second Method has been found to be the best because it is the safest and simplest method.

Another important conclusion is that this method is successful only at the far right hand part of the parachore curve where F(C-1) is not far from

F(C) (Fig. 5.3). In this region, it is particularly useful because a large improvement cannot be obtained by increasing the main memory size. The main memory size has little effect on fault rate in this region.

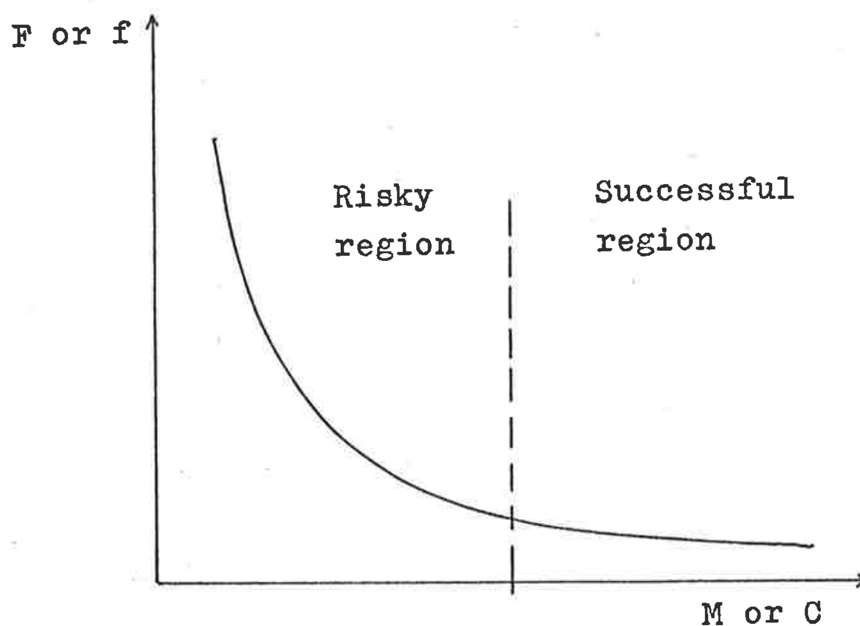


Fig. 5.3 Successful and Risky regions of the Second Prediction Method.

CHAPTER VI

THE PREVENTION OF PUSHES

6.1 Introduction

Under the simple system described in Flow Chart of Fig. 1.3, whenever a page is brought into main memory (a pull), the replaced page has to be written back to auxiliary memory beforehand to update its contents in that memory (a push). Therefore a push always precedes a pull. Any number of pulls will create the same number of pushes. In many cases, not all the pushes made are necessary. Since page transfers between the two memories are the most time-consuming operation, it would be advantageous to reduce these pushes down to a minimum number.

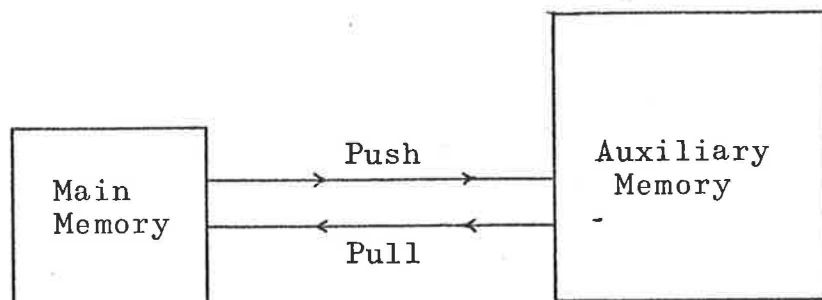


Fig. 6.1 Page transfers between Main and Auxiliary Memories.

6.2 Modified-Page Checking Scheme

The argument against the simple system discussed above is that a push is necessary only when the outgoing page has been modified while residing in main memory. One way of preventing the unnecessary pushes is to provide a flag for each page in Main Memory. A flag is set to ON whenever one or more words in the corresponding page is modified. When a page fault occurs, the supervisor will check if the outgoing page has been modified. If it has, then a push is needed. If it has not, the new page can be brought in to overwrite it immediately.

6.2.1 Co-efficient of Page Transfers

If this scheme is employed, the number of pushes will not be the same as the number of pulls. For the purpose of investigation, a co-efficient α , called co-efficient of page transfers is introduced.

$$\alpha = \frac{\text{Pushes + Pulls}}{\text{Pulls}}$$

If the number of pushes is the same as the number of pulls (i.e. simple system) then α becomes maximum; $\alpha_{max}=2$. If there is no pushes occuring, α becomes minimum; $\alpha_{min}=1$.

Hence $1 \leq \alpha \leq 2$ and in general, α is a function of page size P and Memory size M.

Since the number of pulls is exactly the same as the number of page faults F, α can also be defined as

$$\alpha = \frac{\text{pushes} + \text{pulls}}{F}$$

or $\quad \alpha F = \text{pushes} + \text{pulls}$

Thus αF denotes the total number of page transfers. This is the significance of the co-efficient. Also if R is the ratio of pushes over pulls, α can be expressed as

$$\alpha = \frac{\text{Pushes}}{\text{Pulls}} + 1 = R + 1$$

6.2.2 Simulation Results

A simulation of the Modified-Page checking scheme has been carried out to measure the number of pushes and pulls with different parameters. In Fig. 6.2, the ratio of pushes over pulls R is plotted versus Memory Size M and Page Size P.

Results show that there is no clear general shape of these plots although for high-level languages, R seems to decrease with an increase in Main Memory size.

The most important point of these results is that the average R is 0.75 for ASSEMBLY language

ADELAIDE UNIVERSITY LIBRARY

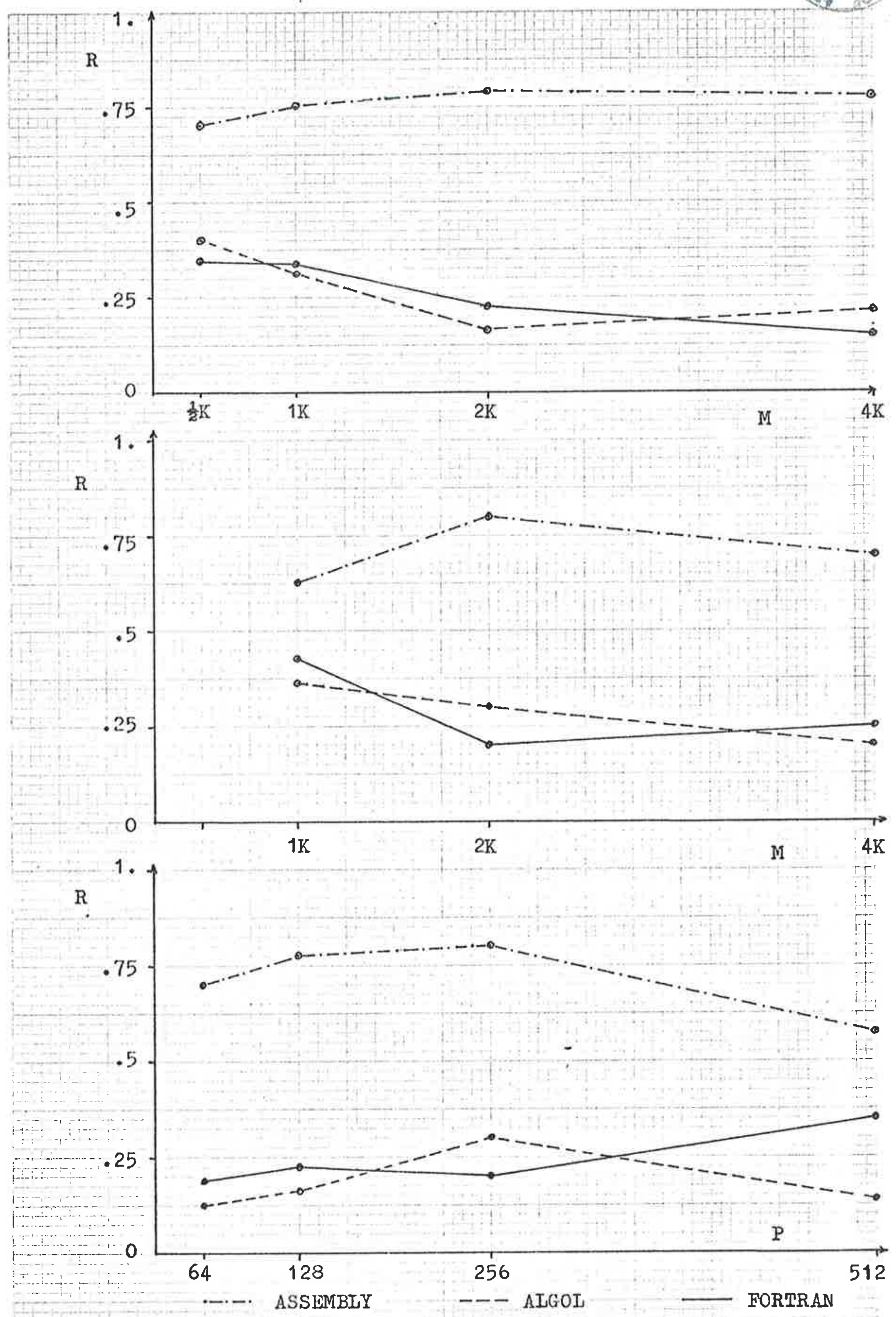


Fig. 6.2 Ratio $R=\frac{push}{pull}$ versus a/ M with fixed P=128
b/ M with fixed P=256
c/ P with fixed M=2K

program and is 0.2 for high-level language programs. Probably, this is due to the fact that high-level languages always need a subroutine library whose contents are never changed. For high-level languages, the use of this scheme seems justified because 80% of the pushes can be avoided. As a result, a good increase in speed can be achieved which will be discussed in Chapter VIII.

6.3 Bottom-Page Avoiding Scheme

6.3.1 Description

The Modified-Page Checking Scheme is a scheme which is completely independent of Replacement Algorithms. It can be used with any replacement algorithm. According to results obtained in Chapter IV, it is seen that LRU Stack should be employed alongside the Modified-Page checking system. The two techniques together result in a co-operative scheme which will be called Bottom-Page Avoiding Scheme.

Under the Modified-Page Checking Scheme, whenever a page fault occurs, the supervisor will check the bottom page of the LRU Stack (i.e. zero-priority page) to see if it has been modified. That page will be replaced afterwards. Under the Bottom-Page Avoiding Scheme, when the bottom page has been modified, the supervisor will check the

second-bottom page. If this page has not been modified then it will be replaced instead. Doing it this way, the supervisor can "climb" the LRU stack ordering and check up to, say, N number of pages. Obviously after k successive checking, if all N pages have been modified, then the bottom-page has to be replaced along the normal route. This scheme is summarized in the flow chart of Fig. 6.3.

6.3.2 Simulation Results

Results of this scheme are shown on Fig. 6.4, 6.5 and 6.6 at memory size 2K and 4K. The page size P is kept fixed at 128 words. The number of pushes, pulls and sum are plotted against k. k = 2 means that the supervisor can check not more than 2 pages up the LRU stack. Table 6.1 shows the improvement percentage of this scheme on the co-efficient α over the normal case of Modified - Page scheme and LRU stack. (k=0)

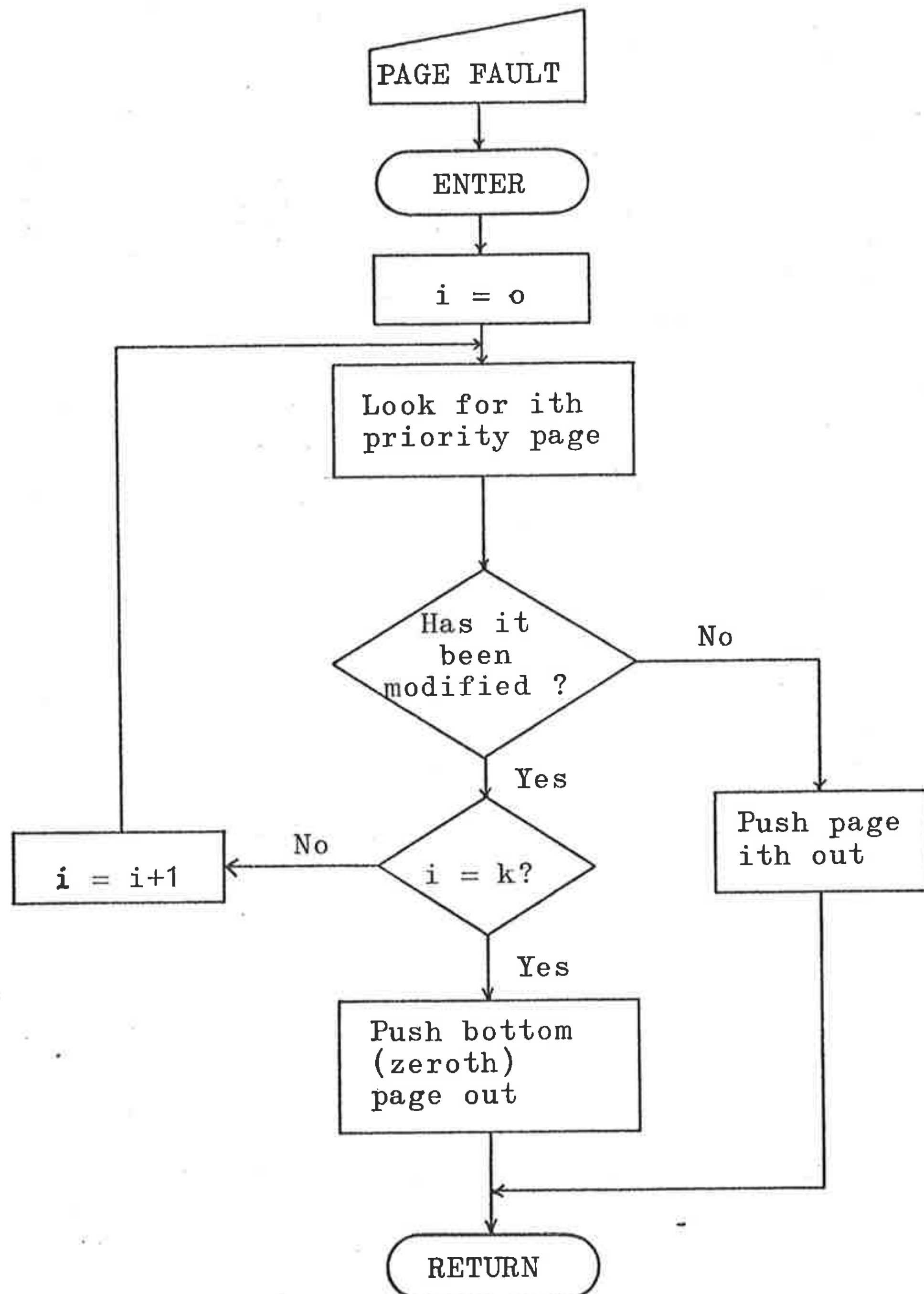


Fig. 6.3 Flow Chart of Bottom - Page Avoiding Scheme

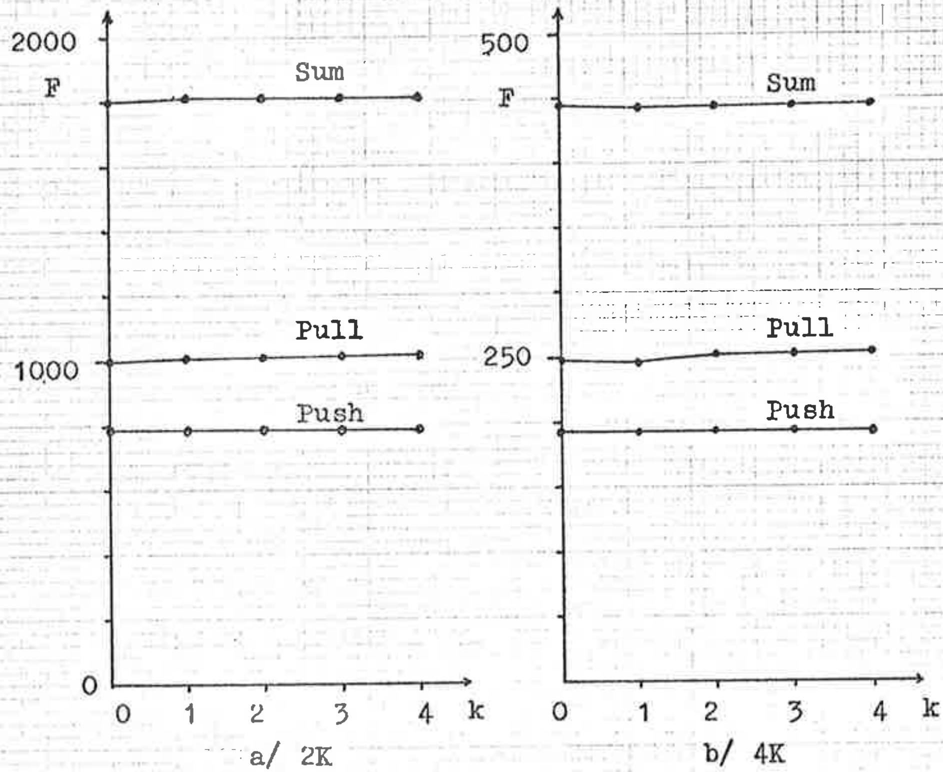


Fig. 6.4 Bottom-page avoiding Scheme of the Assembly Program at fixed P= 128 : Faults versus k

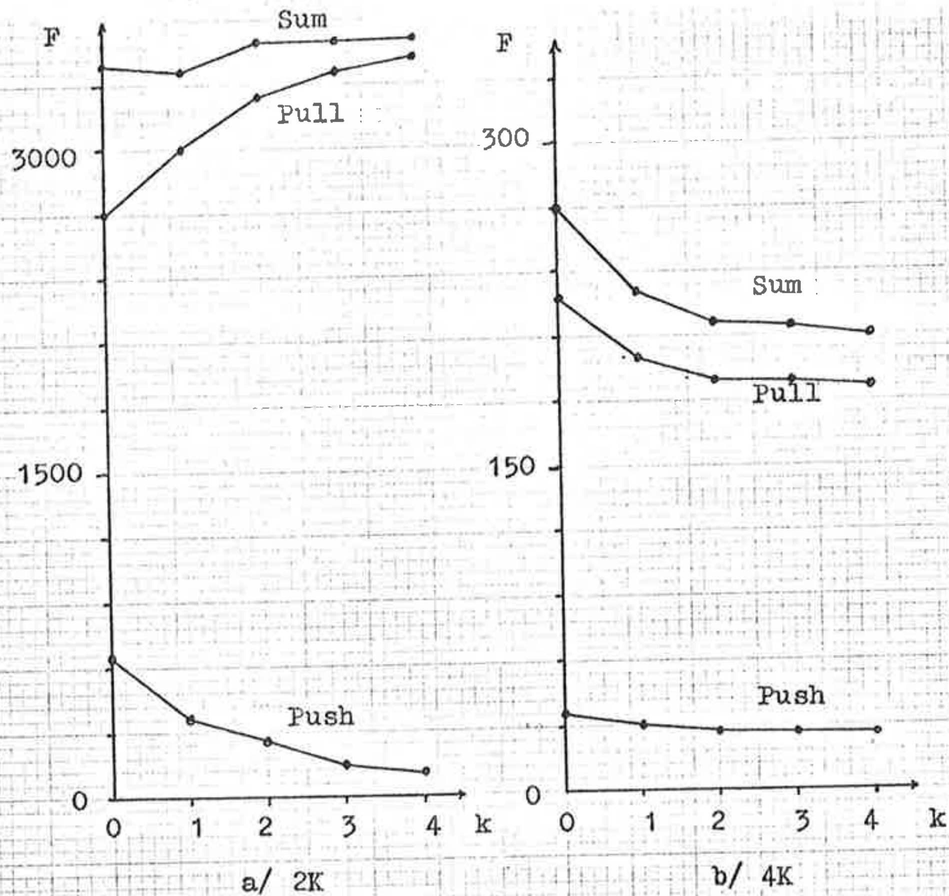


Fig. 6.5 Bottom-page avoiding Scheme of the Fortran program at fixed P=128 : Faults versus k

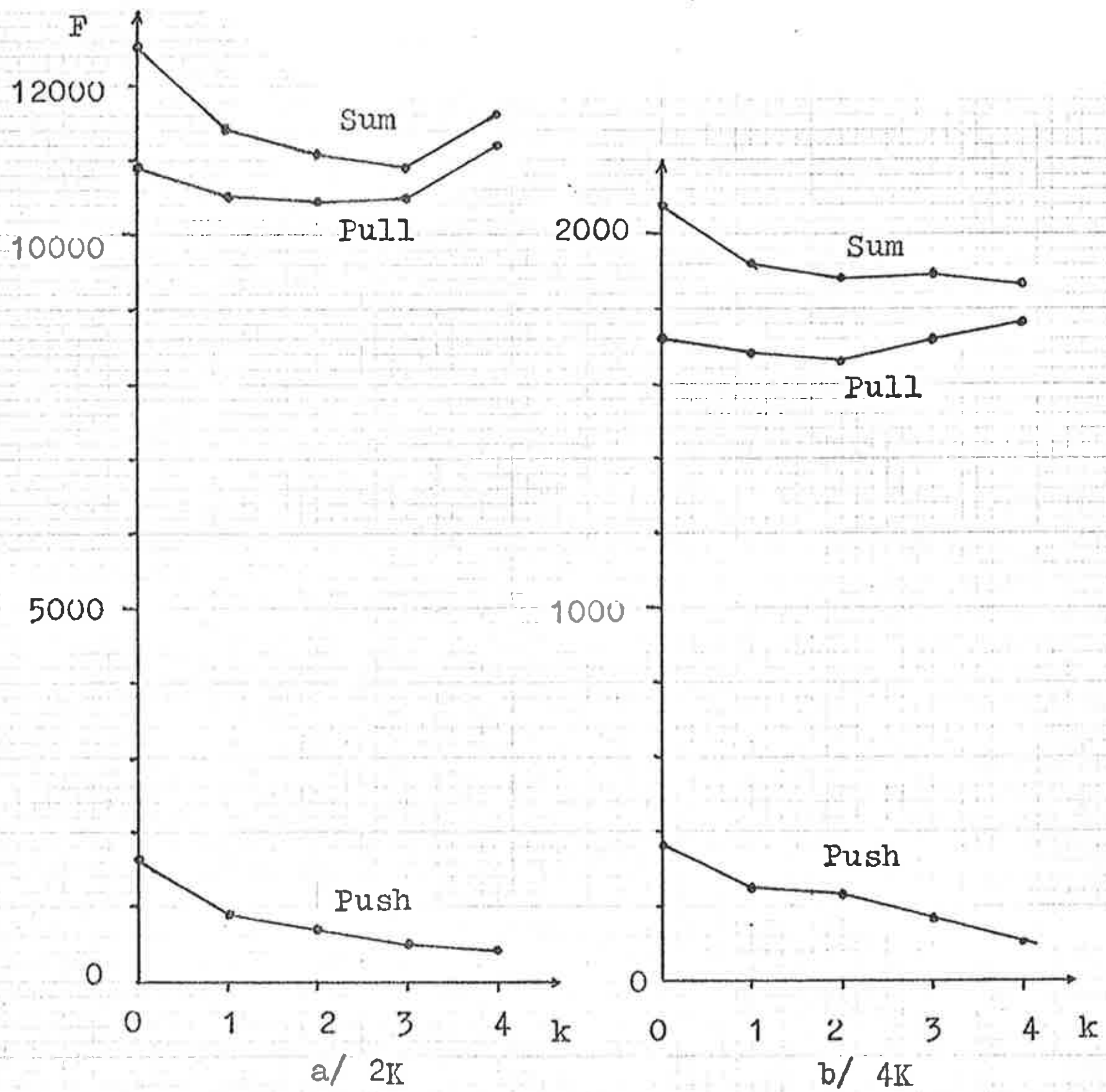


Fig. 6.6 Bottom-page Avoiding Scheme of the Algol Program at fixed P=128: Faults versus k.

M=2K, C=16

	k=1	k=2	k=3
ASSEMBLY	-0.4%	-.3%	-0.4%
FORTRAN	0.6%	3.8%	-3.17%
ALGOL	8.7%	11.4%	12%

M=4K, C=32

	k=1	k=2	k=3
ASSEMBLY	0.9%	0%	.4%
FORTRAN	15%	19%	18%
ALGOL	6.4%	8.7%	7.5%

Table 6.1 Improvement percentage on co-efficient α

Results show that the number of pushes are further reduced by this scheme. The number of pulls can either increase or decrease. The improvement percentages are all positive with M=4K or C=32. In case M=2K, there are some negative results. This can be explained by the worst situation which happens when the pages at the bottom become "stagnant". They always stay in Main Memory and as a result, effective Main Memory becomes k pages less.

The situation now becomes exactly the same as that of the Prediction Methods in Chapter V. This scheme is

only effective at the right part of the parachore curve where the worst case is not far from the normal case. (Fig. 6.7) Also, similar to Prediction Methods, for the reason of safety and simplicity, k should not be larger than 1. In other words, the supervisor has two choices ; either the bottom page or the next one up is replaced when a page fault occurs.

The reason for small improvement in case of the ASSEMBLY program is that the ratio R of that program is high (i.e. 0.75). Almost every page has been modified and the scheme supervisor would have little chance of avoiding the bottom page.

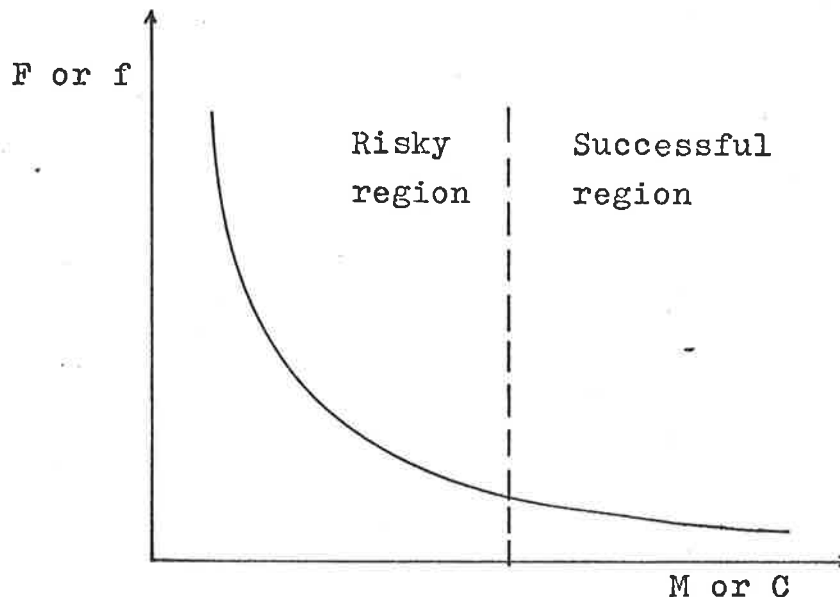


Fig. 6.7 Successful and Risky regions of Bottom-page avoiding scheme.

CHAPTER VII

MODELS OF FAULT RATE AND AVERAGE ACCESS TIME FUNCTIONS

7.1 Introduction

In a virtual memory system, the absolute measure of performance is the average access time to memories. This time function depends on many system parameters and techniques. A direct link to this function is the fault rate which has been discussed in previous chapters. In this chapter, mathematical models for these functions are suggested to enable an evaluation of all techniques investigated using the absolute yardstick: time.

7.2 Fault Rate Function

Strictly speaking, the fault rate of a particular program is a function of both Memory Size M and Page Size P for a given Replacement Algorithm. With LRU stack, f is much more sensitive to M than to P as is seen in Figures 4.3, 4.4 and 4.5 in Chapter IV. For this reason, P is usually kept fixed at a pre-chosen value and f becomes a sole function of M or C (C: number of page frames). This is a basic characteristic of the Virtual Memory Concept.

The parachore curve of a virtual memory system is a graph of F or f against M or C. Many models have been suggested to approximate this curve. J.H. Saltzer in his paper (13) proposed that fault rate is inversely proportional to main memory size or $f = f_o M^{-1}$ where f_o is a constant. Although this model serves some purposes, it is far too simple and hardly fits in many situations. On the other hand, in (18) C.K. Chow proposed that $f=f_o M^{-\gamma}$ where γ is a positive real number depending on system parameters. This is probably an excellent model, but it is rather too complicated.

The model of the fault rate function suggested here is a compromise between these two extremes. It can be stated as f or F is inversely proportional to the square of M or C. In simple mathematical terms:

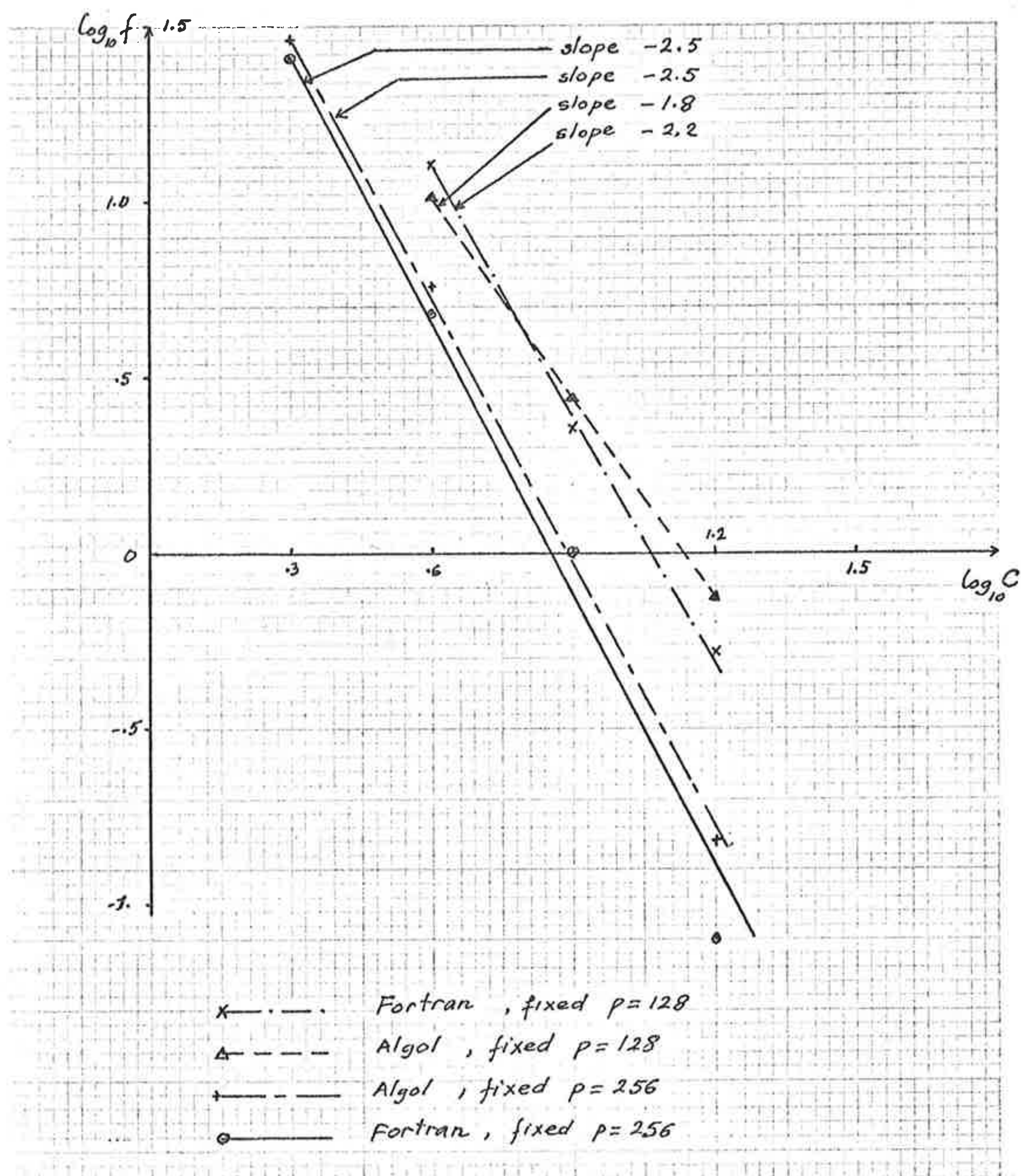
$$f=f_o M^{-2} = f'_o C^{-2}$$

or

$$F=F_o M^{-2} = F'_o C^{-2}$$

where f_o, f'_o, F, F'_o are constants.

Experimental results shown in Fig. 7.1 agree quite closely with this model. Four plot lines of log f versus log C show slopes which are close to -2.



<u>Fig. 7.1</u> Log. Fault Rate f versus log. C with fixed P.

7.3 Average Access Time Function

As previously discussed, the average access time is the most important function which directly represents the system performance. Since it depends on the fault rate f and memory access time, it can be approximate as $T=(1-f)t_1+ft_2$ where t_1 and t_2 are access time of main memory and auxiliary memory respectively. However there is an increasing need to express this function more accurately. In this section, an attempt is made to derive a better mathematical model for this function. In this analysis, the auxiliary memory is assumed to be either a disk or a magnetic tape.

For any program executed under a virtual memory system, let:

- n : number of references executed
- t_1 : access time of Main Memory
- t_s : settling time of disk or tape head
- t_t : data transfering time per word or disk or tape.
- F : Total number of page faults
- α : co-efficient of page transfers.
- α : $\dfrac{\text{Push + Pull}}{\text{Pull}} = R+1 \;;\; 1 \leq \alpha \leq 2$

Total execution time for the program will be:

$$\sum T = nt_1 + \alpha F(t_s + Pt_t)$$

where α and F are functions of both M and P in general and αF denotes the total number of page

transfers between the two memories.

Average access time T for a reference is:

$$T = t_1 + \alpha \frac{F}{n}(t_s + Pt_t)$$

$$T = t_1 + \alpha f(t_s + Pt_t)$$

where f is the fault rate

Typically the access time ratio $t_s/t_1 \simeq 10,000$ for a disk-core system and is much larger for a tape-core system. If $f \geq .001$ or .1%, t_1 in the expression becomes insignificant.

$$\boxed{T \simeq \alpha f(t_s + Pt_t)}$$

What has been done in previous chapters can now be seen as trying to minimize T by minimizing other quantities in the expression. Fault rate f can be reduced by LRU Stack and Prediction Methods and α can be reduced by the Prevention of Pushes. These improvements will be discussed in details in Chapter VIII.

In case of a typical Floppy disk and for P=128, t_s and Pt_t are comparable (t_s=10ms, $Pt_t \simeq$ 8ms). It must be noted here that P in the expression is not intended to be a variable. A reduction in P would seem to boost the system performance, but it cannot be done without limit. For a reasonable main memory size, smaller page size would mean an increase in number of frames C which results in extra hardware

complexity. For these reasons, P is kept at some pre-chosen value, i.e. in this case P=128.

Recall from Section 7.2, for a fixed P

$$f=f_oM^{-2}=f_o'C^{-2}$$

then $$T \simeq \alpha f_o M^{-2}(t_s+Pt_t)$$

$$\underline{T \simeq AM^{-2}=A'C^{-2}}$$ where A and A' are constants

This model could be used to approximate System Performance with a given Main Memory size and vice versa.

CHAPTER VIII

AN EVALUATION OF VIRTUAL MEMORY TECHNIQUES USING SUGGESTED MODELS

8.1 Introduction

Generally speaking, a virtual memory system is a cost-effective way of expanding main memory. The memory expansion made is completely transparent to the programmers. The cost-effectiveness of the method is achieved partly at the expense of time. All techniques investigated in Chapter IV, V and VI have a common goal of minimizing time or improving speed. In this Chapter a cost comparison and an evaluation of these techniques will be made. Advantages and disadvantages will be discussed to decide whether the use of a virtual memory system is indeed justified.

8.2 Cost

To evaluate the cost, the easiest way is to set up a cost comparison between a proposed virtual memory system and the ordinary way of expanding main memory. As an example, the IMP-16 in the Department of Electrical Engineering (University of Adelaide) will be considered.

The IMP-16 is a 16 bit-word micro-computer of the Department of Electrical Engineering. It has 256 words of RAM internally and another 512 words of ROM can be added. A main memory is obviously needed. For a virtual memory system, an auxiliary memory and electronic interfaces are also required.

Searching through the market for main memory at this time, we have three choices : Bipolar semiconductor, MOS semiconductor or core. Economic reason eliminates the first choice. MOS memory is smaller than core memory in size, approximately comparable cost and slightly faster. However, it is volatile and probably has shorter life than core memory. Data taken in the second quarter of 1976 is shown in Table 8.1 below.

4Kx16 Bit Main Memory	CORE	MOS
Company	Litton Memory Products	National Semiconductor
Model	LM-416N	MOSRAM DR104
Access time	350-550 nsecs	500 nsecs
Cycle time	0.95-1.4 usec	750 nsecs
Price (Single Unit)	$850.00	$850.00

Table 8.1 Data of some products of main memory.

For auxiliary memory, commercially at a moment, the floppy disk seems to be the most attractive because it is much cheaper than conventional disk and much faster than tape (order of thousand times faster). The floppy disk probably provides enough mass storage medium for a mini or micro-computer. Table 8.2 contains some data of one of the floppy disk drive unit, the Memorex 651.

DATA RETRIEVAL TIMES	
Rotational Speed	375 rpm
Access Time	10 ms track to track 10 ms settle @ track
Data Transfer Rate	250 kilobits/sec
DISC CHARACTERISTICS	
Records/Track (max)	32
Number of Tracks	64
Records/Disc (max)	2048
Recording Density	3100 bits/inch (max)
Record Length Sectorized (32 per track)	1056 bits
Record Length Indexed (1 per track)	38.5 kilobits
Disc Capacity Sectorized	2.2 megabits
Disc Capacity Indexed	2.5 megabits
DATA RECORDING FORMAT	
Recording Mode	Frequency modulation
Sectors per Track	32
Index per Track	1

Table 8.2 Summary of Memorex 651 Disc Drive Data

The floppy disk can store 128K of 16-bit words. It can provide 64K or 128K of "virtual memory" depending on whether segmentation is used. (Chapter IX) Main Memory capacity is 4K x 16 bit-words. Table 8.3 shows a cost comparison between a virtual memory system and systems with equal capacity of main memory.

Item	Virtual Memory System 64K or 128K	System with 64K Main Memory	System with 128 K Main Memory
Main Memory	~ $850	~ $13,000	~ $26,000
Floppy Disk Unit	~ $2,500		
Interfaces	~ $300		
Total	~ $3,650	~ $13,000	~ $26,000

Table 8.3 Approximate cost comparison of a virtual memory system and systems with equal capacity of main memory.

On the cost side of the "game", Table 8.3 shows that the virtual memory system is extremely cost-effective and rational. Even one could afford the main memory with equal capacity, he would think twice about buying a memory which costs 20 times more than the micro-computer itself. If time is not a critical factor, the virtual memory system therefore, would be a better and sensible approach. Note that this cost comparison is neither confined to this particular example nor the current price of memories. The

cost ratio will rather stay almost the same in different situations. In any computer system, cost is always one of the most important factors. This is the reason why most computers are preferred to have back-up disks or magnetic-tapes rather than to expand main memory to full capacity. The disks or tapes in a virtual memory system are not only back-up media but also the "virtual" part of the main memory. To show that cost is really important, recall that before the first virtual memory computer came to existence, the problem of expanding main memory had been tried by a group in M.I.T. in 1961. They proposed the construction of a computer with several million words of main memory (an amount then considered vast). Economic reasons had prevented this proposal from actually being realized (1).

8.3 Time

As mentioned before, memory expansion at substantially low cost of a virtual memory system is achieved usually at the expense of time. This is the price of the virtual memory system. However, with the techniques investigated in Chapters IV, V and VI, the average access time of the system can be greatly reduced. Using the model suggested in Chapter VIII, it is now possible to evaluate these techniques on the absolute scale : time.

Recall that the average access time of a virtual memory system is:

$$T = \alpha f(t_s + Pt_t)$$

In this expression, Page Size P is not treated as a variable, but rather as a constant. Without loss of generality, P=128 is chosen (Chapter IV). For a floppy disk as an auxiliary memory, t_s=10ms, t_t=64μsec. $(t_s+Pt_t) \simeq 18$ msec.

8.3.1 LRU Stack Replacement Algorithm

Let T_o be the average access time of a Virtual Memory System employing Direct Mapping with no other techniques and T_1 be that of the LRU Stack Algorithm. Since T is directly proportional to fault rate f, improvement of f or F is also improvement of time. Table 8.4 shows the T_o/T_1 ratio for fixed P=128 at M=2K and M=4K.

	2K	4K
ASSEMBLY	28	1.7
FORTRAN	10	25
ALGOL	10	44

Table 8.4 Improvement in Time (T_o/T_1) of LRU Stack over Direct Mapping.

8.3.2 Modified - Page Checking Scheme

Let T_2 be the average access time of the Modified Page Checking System with LRU Stack.

The improvement on $\propto$ is also the improvement on T. Table 8.5 shows the percentage of improvement of T_2 over T_1.

	2K	4K
ASSEMBLY	11%	11%
FORTRAN	38%	42%
ALGOL	43%	40%

Table 8.5 Improvement percentage of T_2 over T_1

8.3.3 Second Prediction Method

Employing Second Prediction Method, the average access time becomes T_3. Table 8.6 shows the improvement percentage over T_2.

	2K	4K
ASSEMBLY	4.7%	23%
FORTRAN	4.3%	20%
ALGOL	11.8%	25%

Table 8.6 Improvement percentage of T_3 over T_2

8.3.4 Bottom Page Avoiding Scheme

With Bottom-Page Avoiding Scheme applied for k=1, the improvement percentage over T_2 is shown in Table 8.7

	2K	4K
ASSEMBLY	-0.4%	.9%
FORTRAN	0.6%	15%
ALGOL	8.7%	6.4%

Table 8.7 Improvement percentage of Bottom page Avoiding Scheme over T_2

8.3.5 Observation

As far as time is concerned, the improvement of LRU Stack is the most important. Speed can be increased up to 45 times and 20 times on the average. The second important improvement is made by the Modified-Page Checking Scheme. The effectiveness of these two techniques is unconditional, i.e. it is always effective irrespective of system parameters. Second Prediction Method and Bottom-Page Avoiding Scheme also give smaller positive improvement and could be implemented if the operating point is known to be at the far right hand part of the parachore curve.

In case of the core/floppy-disk virtual memory system employing LRU, Modified-Page checking scheme and Second Prediction Method, the average access time T is -

$$T = \alpha f(t_s + Pt_t)$$

For the 8K ALGOL Program with M=4K, α =1.2 and $f \simeq 0.001$ (page C-6, Appendix C)

$T \simeq 1.2 \times 0.001 \times 18 \simeq 0.02$ msec $(t_1 \simeq 0.001$ ms$)$

Although it is larger than the access time of core (1μsec), it is very much smaller than the time for swapping a page from core to disk and back (2x18=36msec), or even the settling time of disk (10msec). Recall that this is the case where the program size (8K is twice larger than the main memory size (4K).

8.4 Limitations

8.4.1 Thrashing

This is the excessive overhead and severe degradation caused by too much paging. Thrashing turns a shortage of main memory space into a surplus of processor time, i.e. the processor is idle most of its time waiting for paging. Thrashing usually occurs in multiprogramming situation when two or more programs are stealing pages from one another (10). In single-user environment, thrashing can occur in case of Direct Mapping when pages are competing for one memory frame.

Employing LRU Stack in a single-user environment, the problems mentioned above are completely avoided. The only way that thrashing can occur is when there are program loops which are larger than main memory size. This problem can be solved by the Designer who would ensure sufficient main memory space for most "typical" programs.

8.4.2 Under-cored Programs

If the program is under-cored, i.e. it can fit into main memory, the use of virtual memory system would be a waste of time through address translations, etc. Usually a design of a virtual memory system should include a bypass switch which can be used by programmers to "turn off" the virtual memory when the program is under-cored.

8.5 Conclusion

After comparing cost and time of a virtual memory system and a system with equal main memory capacity, one can realize that the only disadvantage of virtual memory is time due to the slower auxiliary memory. However this disadvantage can be outweighed by the following advantages:

- extremely cost-effective
- transparent to programmers
- much faster than the auxiliary memory
- various techniques can be applied to increase speed considerably

In situation where time is not a critical factor, the virtual memory approach is no doubt one of the best methods of expanding main memory. For the techniques investigated, LRU Stack and Modified-Page Checking Scheme are most important techniques which can be applied with any system parameters. Prediction Methods and Bottom-Page Avoiding Scheme can also be applied at the right hand part of the Parachore curve .

CHAPTER IX

SUGGESTIONS FOR FUTURE WORK

9.1 Introduction

Although investigation into three basic techniques of virtual memory have been completed, the research in this field does not stop here. The next question to ask is which direction one should follow to continue on this project. At this stage, there seems to be no shortage of work for future investigation. Two possible extensions are suggested here: The Restructuring of Programs and the Expansion of the Virtual Address Space.

9.2 Restructuring of Programs

Unlike the three approaches discussed so far, which attempt to change the virtual memory system to fit the programs, this approach tries to change the programs to fit the system. Although there are good reasons for studying the way to change the system, there are equivalently good reasons for studying the way to change the programs. This is an approach which is totally independent of the previous approaches.

For any program, the sequence of virtual page requests (i.e. reference pattern) is an absolute measure of the page requirements of the program. Anything that both reduces the length of the overall sequence and the number of distinct pages used in subsequences should result in reducing the Program's page faults for almost all memory sizes and page replacement algorithms.

In a program, there are always parts that can be placed anywhere in the program. They are called relocatable sectors by D.J. Hatfield and J. Gerald in their paper (16). Examples of these are subroutines or arrays in FORTRAN COMMON. The size of sectors must be smaller than page size and typically about one tenth to one third of the page size. Suppose there are m sectors in the program, all communication between these sectors can be recorded in a m by m matrix (mxm) called nearness matrix. Usually the program must be run once to obtain all information to construct the nearness matrix. Obviously, the matrix will be different with different sets of data, however, there are always sectors which are data-independent and a "typical set of data" can be chosen from a collection of samples.

The key to this technique is the reordering so that sectors which communicate more with one another

are put in the same page. In addition, each page should be filled with sectors that are used with the same frequency. The technique attempts to maximise the use of a page by avoiding the situation when only a small part of a page is used. For example, 3 sectors which communicate with one another 10,000 times would surely perform better if are put in the same page rather than in 3 different pages which tend to be in main memory at the same time.

Since sectors are all different in sizes, there is no guarantee that an integer number of sectors will fit nicely in one page. For this reason, sectors should be allowed to cross page boundaries, otherwise "holes" will be created in each page.

As an extension to this reordering, Read-only sectors which are needed by many other sectors can be duplicated and distributed to bring about the best structure of programs.

Although it was claimed that this technique is more sensitive than replacement algorithm (16), the main criticism (15) is that the amount of work, software and hardware used for the preprocessing of a particular program is far too much. However, it can be applied for programs that will be used many times in future or programs that are not highly data-dependent.

D.J. Hatfield and J. Gerald (16), have used hardware to find program traces in order to detect sectors communication. However, hardware may not be the best approach because of the cost of equipments involved. The suggestion here is that in order to retain the cost-effectiveness, software may be a better method and to retain programmer transparency, a processing program can be written (at either compiler or assembler level) to restructure other programs. The processing program effectively "runs" the being-restructured program first and detects all communication between relocatable sectors. It then uses the data fetched to restructure the program. In search of this approach, the author has faced an interesting problem: The 32K Main Memory of the Nova Computer used becomes insufficient to store all data fetched back. This means that as far as Program Restructuring goes, the Virtual Memory concept is indeed self-fulfilled. After designing and building a virtual memory system, one can use this very facility to do the restructuring of programs. There will no longer be a shortage of memory space.

9.3 Expansion of the Virtual Address Space

One of the problems that faces designers of virtual memory system for small computers is the short word length. For example, a minicomputer which uses 16-bit word is capable of addressing only 64K of memory. For the Nova computers in this Deparment,

since one bit is used for continuing indirect addressing, the address space is reduced to 32K. On the other hand, a small disk is usually at least 128K of storage, for example Floppy Disk Memorex 651. If virtual memory is implemented, the disadvantage is that half or three quarters of the disk is not used. Furthermore, a 32K or 64K address space may not satisfy programmers using high-level languages. For these reasons, some technique to expand the virtual address space and maximize the use of the disk is definitely needed.

One way to solve the problem is introducing the segment concept. A segment is a group of pages whose size is exactly the same as the address space. In case of the Nova computer and the Floppy Disk, the address space is 32K, therefore 4 segments can be introduced to cover the whole 128K disk storage.

Since the address space is limited, only one segment can be engaged in operation at a time. We must now add a segment register of n bits to tell the CPU which of the 2^n segments we are currently referencing. Conceptually, the segment register is an extension of the instruction's address field. The combination of the segment register and the address register now contains the effective address and the address space has virtually been expanded.

To change the content of the segment register, which may be a software construction (7), a special instruction is needed. This kind of instruction can be executed through a service call which just changes the course of addressing on disk. (Cross-segment jump)

Doing it in this way, the whole disk can be used and the virtual address space is expanded, say, from 32K to 128K with a 2-bit segment register.

CHAPTER X

CONCLUSION

Based on the results of the overall evaluation (Chapter VIII), the virtual memory approach can be considered to be one of the best methods of expanding main memory so far in situations where time is not critical. The advantages of a virtual memory system can be summarized again as follows:

1. extremely cost-effective
2. transparent to programmers
3. much faster than the auxiliary memory used
4. speed can be increased considerably by various techniques

The techniques used to increase speed are interesting topics for research. For LRU Stack Algorithm, results in Chapter IV indicated that it does perform better than Direct Mapping. This improvement is very important because it reduces the risk of thrashing and increases speed up to 45 times faster (Chapter VIII). Besides, for typical small computers, a good page

size, if feasible, would be 128 words in a main memory of 4K.

The use of the Modified-Page Checking Scheme seems advantageous since it reduces the number of pushes considerably. In case of high-level languages, 80% of the pushes are prevented and as a result, the average access time can be reduced by 40% (Chapters VI and VIII).

It is also noted that the advantages of the two methods above are unconditional, that is, with these methods, the performance is always improved irrespective of the system parameters (M, P, Program). For this reason, they are highly recommended.

Simulation results of Prediction Methods in Chapter V indicates that Second Prediction Method is the best among the three methods considered. It can be used when the operating point is known to be at the right-hand part of the parachore curve. In this region, it becomes particularly useful because further increase in main memory size would have little effect on fault rate at this point.

The Bottom-Page Avoiding Scheme is a modified version of the Modified-Page Checking Scheme. It does have positive improvement over the latter (Chapter VI).

Like the Second Prediction Method, it can only be used when LRU Stack is employed and the operating point is on the right-hand half of the parachore curve.

Besides the four techniques discussed above, research on this project can be extended further into different areas. Two topics have been suggested in Chapter IX. Firstly, the restructuring of programs is a promising area in which large amounts of software will be involved. A processing program may be written to restructure other programs before they are run under a Virtual Memory System. Secondly, the segment concept which expands Virtual address space also needs further investigation before it can become a workable solution technically.

Finally, one must repeat that the biggest advantage of a virtual memory system is cost. Results in Chapter VIII have shown that the cost of a virtual memory system is about 14% of the cost of an equal-capacity main-memory. For this reason, the Virtual Memory approach is still the most cost-effective method of expanding main memory for small computers so far.

REFERENCES

1. P.J. Denning, "Virtual Memory", Computing Surveys 2, No. 3, 153-189, September 1970.

2. L.A. Belady, "A study of replacement algorithms for a virtual-storage computer" IBM Systems Journal 5, No. 2, 78-101, 1966.

3. A.V. Aho, P.J. Denning and J.D. Ullman, "Principles of optimal page replacement" Journal of the ACM 18, No. 1 80-93, Jan 1971.

4. T. Kilburn, et al. "One level Storage System". IRE Trans. EC-11, 2, 223-235, April 1962.

5. J. Fothering, "Dynamic storage allocation in the Atlas Computer, including an automatic use of a backing store", Comm. ACM 4, No.10, 435-436, October 1961.

6. R.P. Parmalee, et al. "Virtual storage and virtual machine concepts" IBM Systems Journal, 11, No.2, 99-130, 1972.

7. M. Joseph, "An analysis of paging and program behaviour" Computer Journal 13, No.1, 48-54, Feb. 1970.

8. W.W. Chu and H. Opderback. "The Page Fault Frequency Algorithm" AFIPS Conference Proceedings, 41, Part 1, 597-609, December 1972.

9. P.J. Denning, "The working set model for program behaviour" Comm. ACM 11, No.5, 323-333, May 1968.

10. P.J. Denning, "Thrashing: Its causes and prevention" AFIPS Conference Proceedings, Fall Joint Computer Conference, 33, 915-922, 1968.

11. R.M. Glorioso, T.D. Chase, "Design of Virtual Memory for small computers", Computer Design Vol 12, 67-72, 1973.

12. S. Davis, "Update on Magnetic Tape Memories" Computer Design Vol. 13, 127-140, August 1974.

13. J.H. Saltzer, "A Simple Linear Model of Demand Paging Performance", M.I.T. Comm. of the ACM, April 1974,

14. E. Gelenbe, "The distribution of a program in Primary and Fast Buffer storage". Comm. of the ACM, July 1973, Vol. 16, No. 7, p. 431.

15. J.L. Baer and G.R. Sager, "Dynamic Improvement of Locality in Virtual Memory Systems". IEEE Transactions on Software Engineering, Vol. SE-2 No. 1, March 1976, pp. 54-62.

16. D.J. Hatfield and J. Gerald, "Program Restructuring for Virtual Memory" IBM System Journal, Vol. 10, No. 3, pp. 168-192, 1971.

17. J.L. Baer, "On Program Placement in a Directly Executable Hierarchy of Memories" IEEE Transactions on Computers, Vol. C-23, No. 8, August, 1974.

18. C.K. Chow "Determination of Cache's Capacity and its Matching Storage Hierarchy". IEEE Transactions on Computers, Vol. C-25 No.2, February, 1976.

APPENDIX A

FLOW CHARTS OF THE SIMULATION PROGRAM

NOTES

1. Special Locations

INFLG:	- Interrupt flag; set to 1 when an interrupt occurs
PPC:	- Pseudo Program Counter
SPPC:	- Pseudo location 0
INSR:	- Pseudo location 1
ITN:	- Location used for executing 'INTEN' (interrupt enable) instruction, allows for execution to be delayed one 'instruction cycle' (INT) = INTEN when set = JMP .+1 when reset
PGSZ:	- page size P
TN:	- Number of page frames C in main memory
MELB:	- Main Memory lower bound
MEUB:	- Main Memory upper bound

2. Abbreviation

M.R.I.:	- Memory referenced Instruction
E.A.:	- Effective address
Bit 0:	- left most bit

RBYTE:	- right byte
LBYTE:	- left byte
V.P.:	- Virtual Page
V.A.:	- Virtual Address
R.A.:	- Real Address

3. Errors

ERROR:00	- The Program under Simulation has attempted to READ or WRITE into a memory location occupied by the Simulation Program
ERROR:01	- The Program under Simulation has attempted to jump into the Simulation Program
ERROR:02	- An instruction outside main memory is fetched
ERROR:03	- Virtual Address is larger than maximum allowable address. Not translatable

4. Subroutines

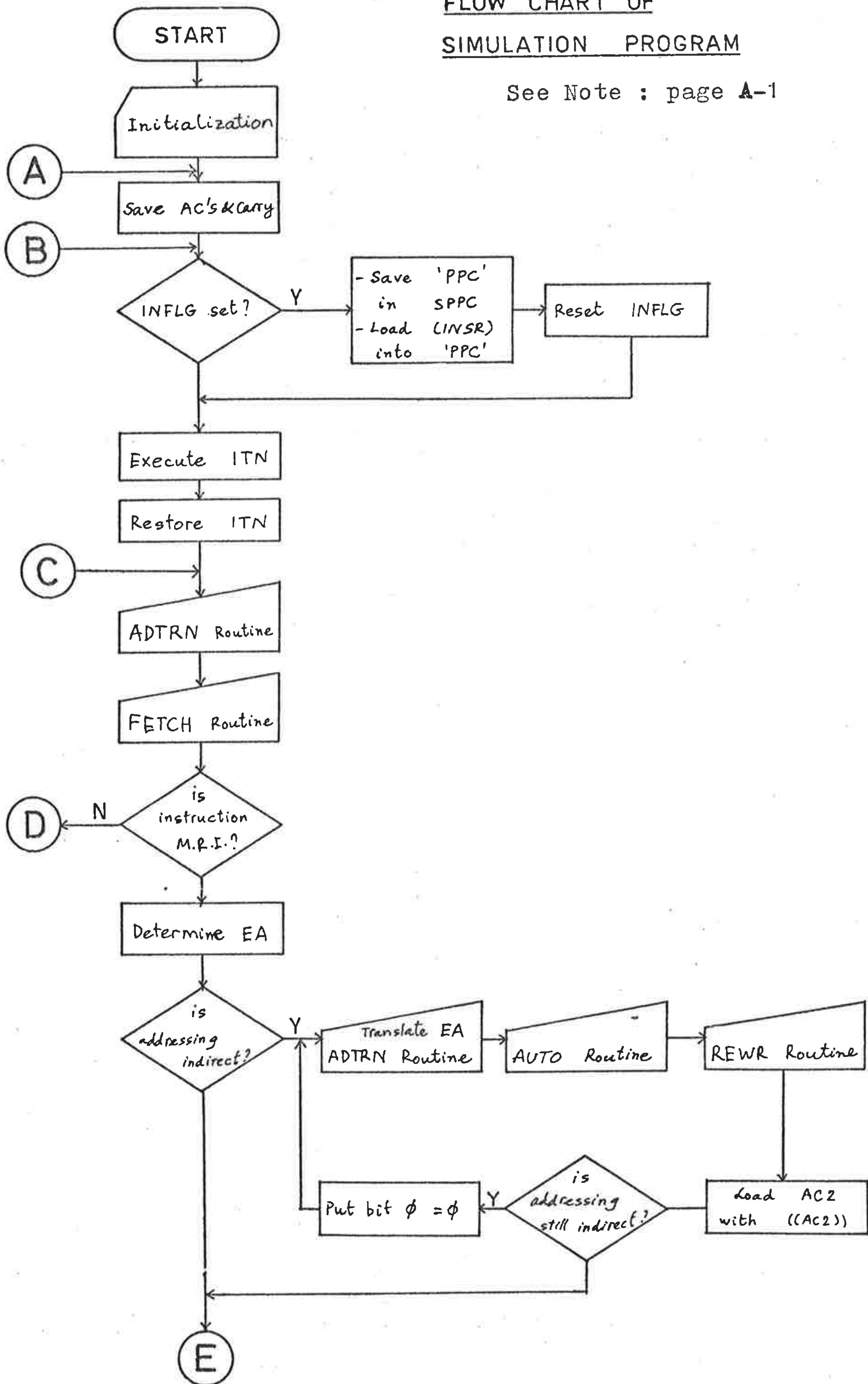
FETCH Routine	- Subroutine for fetching instruction from main memory. See page A-10
REWR Routine	- Subroutine for fetching data from main memory. See page A-11

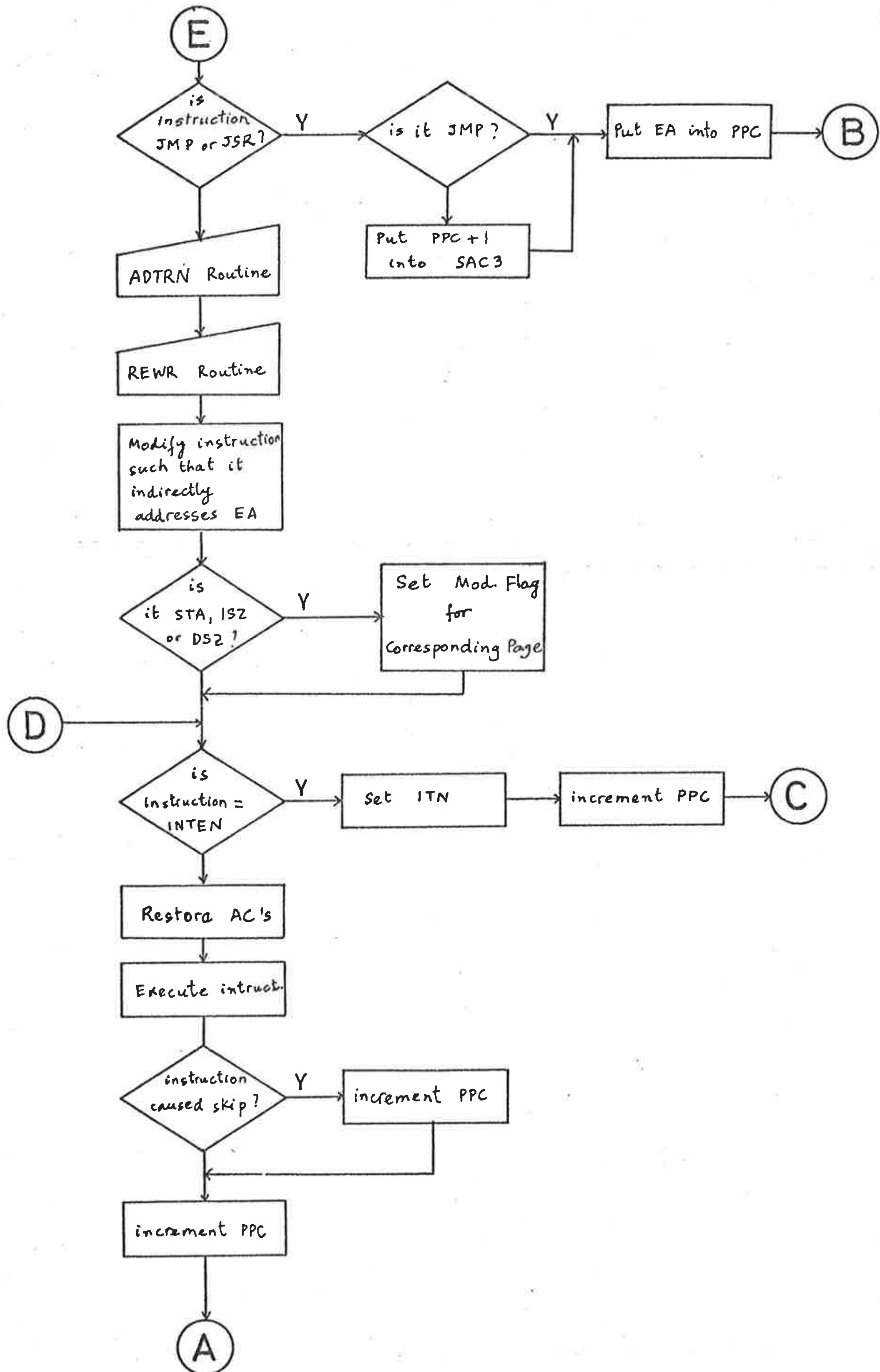
AUTO Routine	- Subroutine which takes care of auto-incrementing and decrementing locations. See page A-12
ERR Routine	- Subroutine to print errors
TXOP Routine	- Subroutine to print a text
SMSR Routine	- Interrupt Service Routine. Subroutine to take care of interrupts. See page A-12
RESET Routine	- Subroutine used to reset all counters in the simulation program for initialization purpose
MFLAG Routine	- Subroutine to set a flag to ON when the corresponding page is being modified.
ADTRN Routine	- Subroutine to translate Virtual Address to Real Address. See page A-7
IBR Routine	- Subroutine to load C pages into main memory for initialization purpose.
BRING Routine	- Subroutine to bring a requested page into main memory when a page fault occurs. See page A-8

XFER Routine	- Subroutine to transfer a page between 2 memories
IPRI Routine	- Subroutine to initialize priority numbers
PRI Routine	- Subroutine to update priority of pages in Main Memory.

FLOW CHART OF SIMULATION PROGRAM

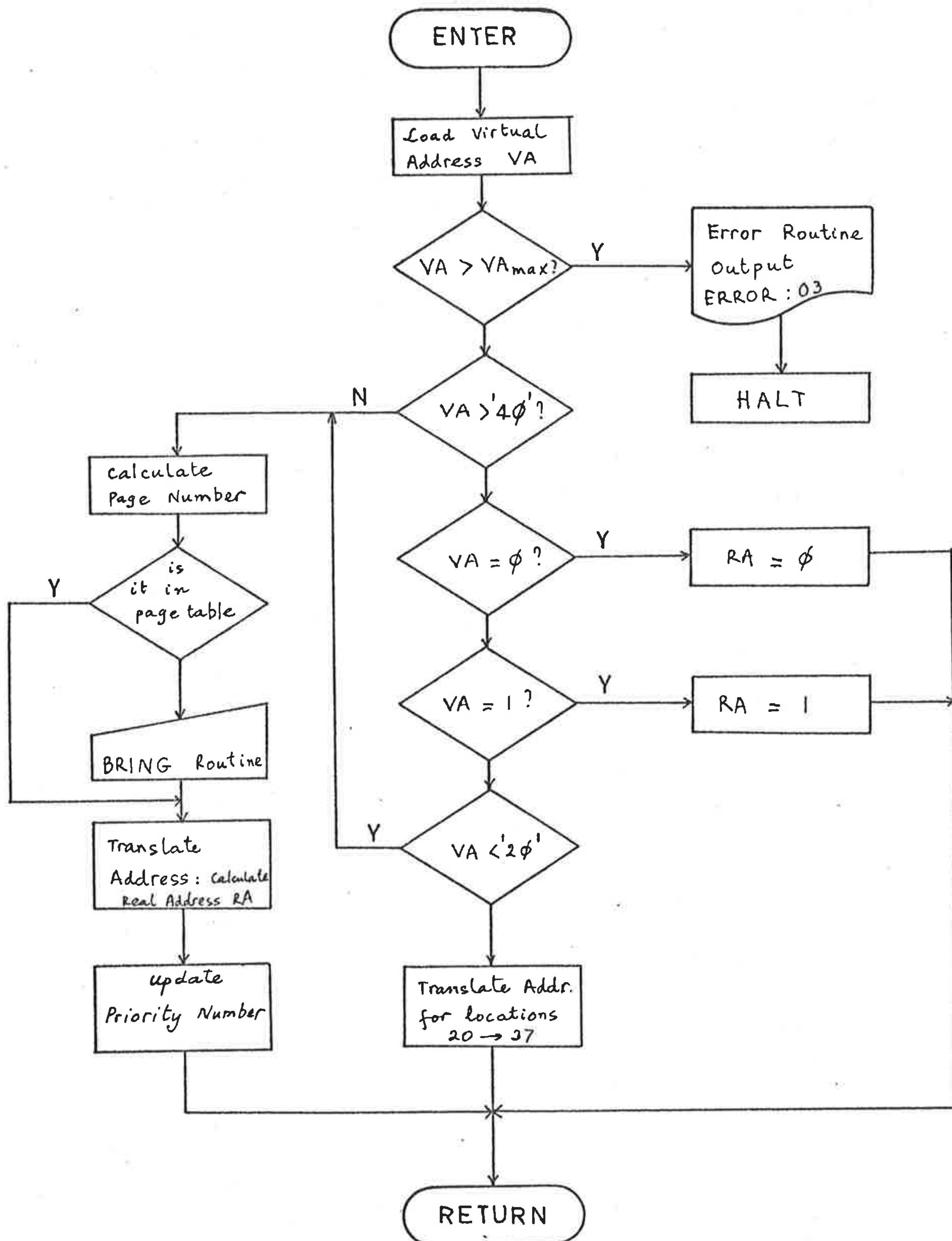
See Note : page A-1




E
is instruction JMP or JSR?
Y
is it JMP?
Y
Put EA into PPC
B
Put PPC + 1 into SAC3
ADTRN Routine
REWR Routine
Modify instruction such that it indirectly addresses EA
is it STA, ISZ or DSZ?
Y
Set Mod. Flag for Corresponding Page
D
is Instruction = INTEN
Y
Set ITN
increment PPC
C
Restore AC's
Execute intruct.
instruction caused skip?
Y
increment PPC
increment PPC
A

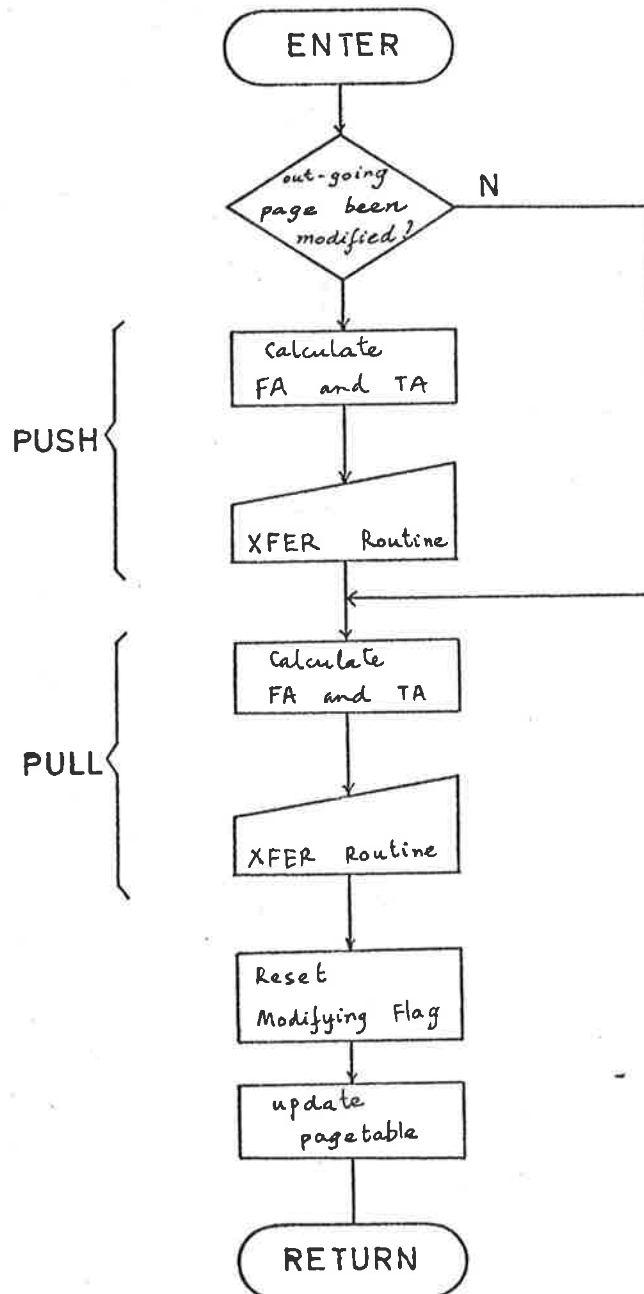
ADDRESS TRANSLATOR SUBROUTINE
ADTRN

Function: Translate Virtual Address VA to Real Address RA.



BRING SUBROUTINE

Function: Bring a new page into main memory whenever a page fault occurs.



Note

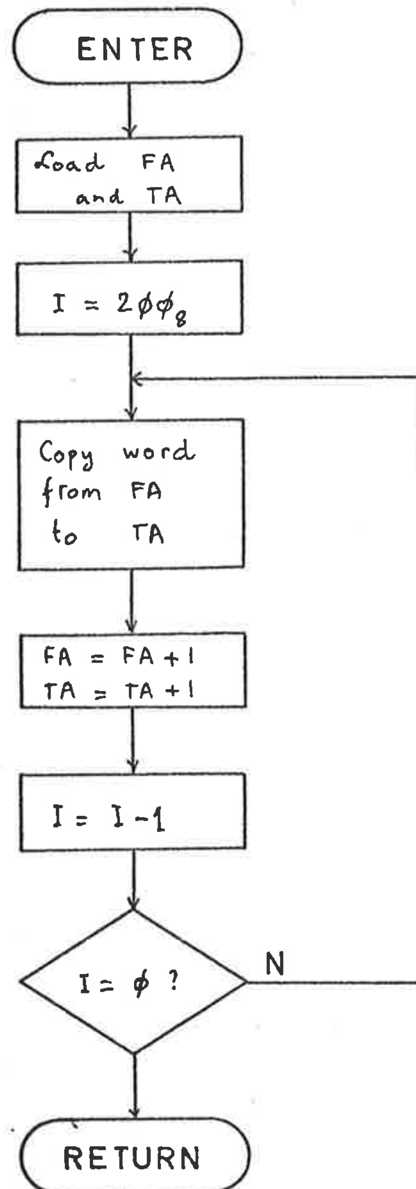
FA: - From-address, address where data is taken from.

TA: - To-address , address where data is put into.

TRANSFER SUBROUTINE

XFER

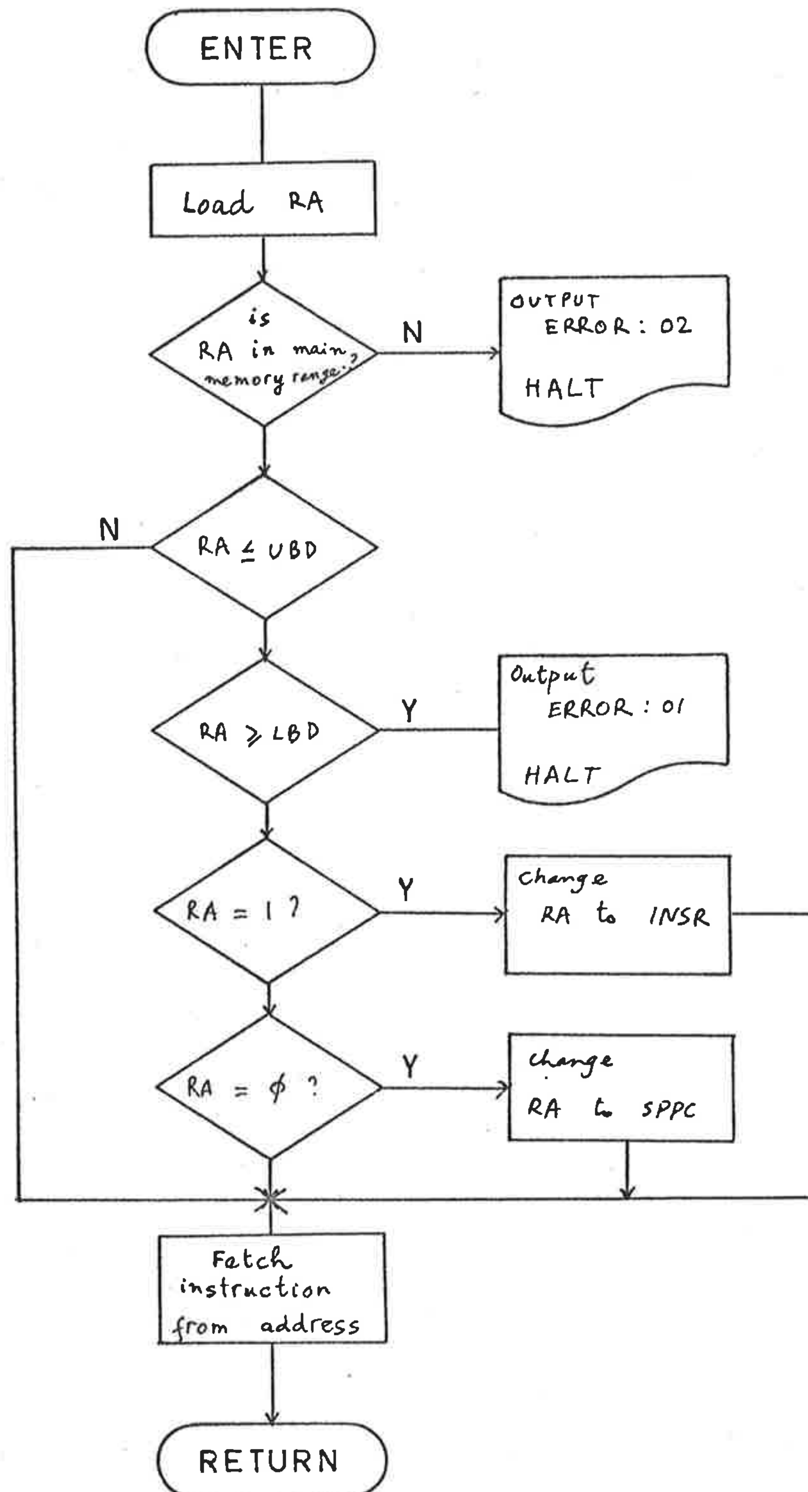
Function: Transfer a page =128 words between main memory and backing store.



Note FA : - From- address
TA : - To-address

FETCH SUBROUTINE

Function: Fetch instruction from Real Address RA in main memory.



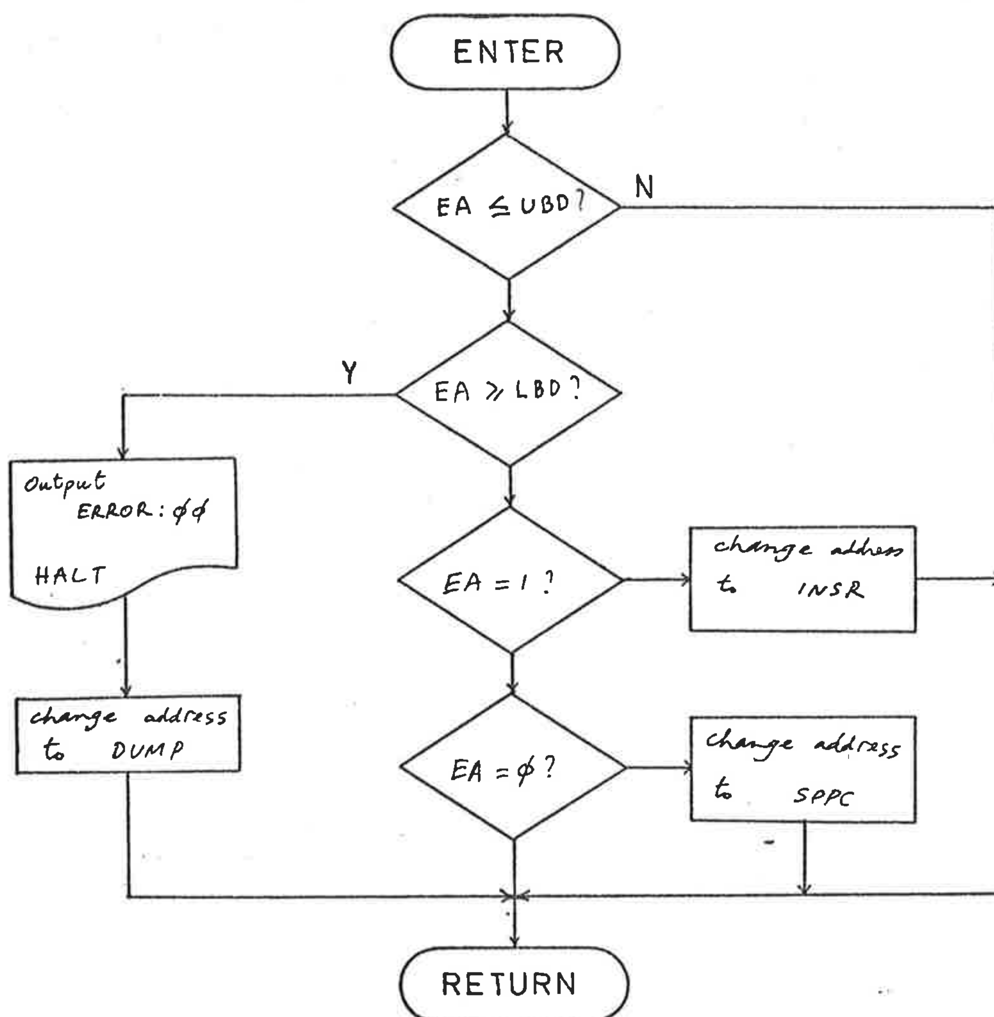
Note

UBD : Upper bound of Simulation Program

LBD : Lower bound of Simulation Program

READ-WRITE SUBROUTINE
REWR

Function: Fetch data from main memory.

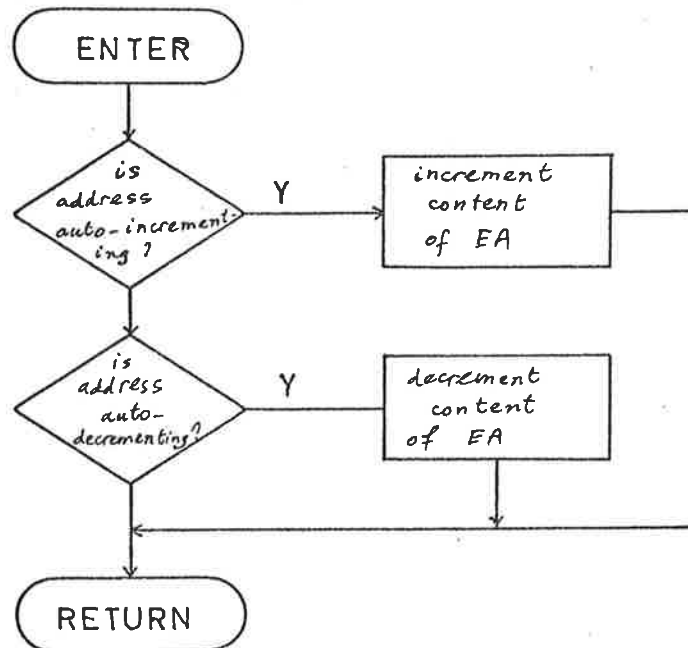


Note

DUMP: a location reserved for dumping , it has no significance.

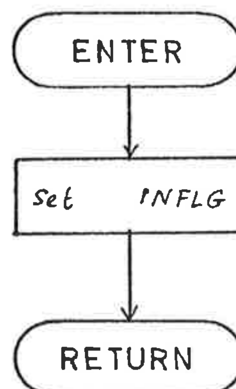
AUTO SUBROUTINE

Function: Take care of auto-incrementing and decrementing locations.



INTERRUPT SERVICE ROUTINE

Function: Take care of the interrupt service of the machine. Transfer an interrupt of the machine to a simulated interrupt.



A P P E N D I X   B

SIMULATION PROGRAM LISTING

```
---
0001   .MAIN
                                  ;VIRTUAL MEMORY SYSTEM SIMULATION PROGRAM
        076000                    .LOC 076000
        076000                    LBD=.
 76000 000000 STADD:    0                              ;ADDR. OF FIRST INSTR.
 76001 042447           STA      0,@ASACO              ;SAVE ACO
 76002 020001           LDA      0,1                   ;STORE ADDR. OF INTERR.
 76003 040532           STA      0,INSR                ;SERV. ROUT. IN INSR
 76004 020535           LDA      0,ASMSR               ;STORE ADDR. OF SMSR
 76005 040001           STA      0,1                   ;IN LOC. 1
 76006 102400           SUB      0,0
 76007 042537           STA      0,@ANFLG              ;RESET INTERR. FLAG
 76010 020537           LDA      0,DUM                 ;SET ITN TO
 76011 040465           STA      0,ITN                 ;JMP .+1
 76012 020766           LDA      0,STADD               ;PUT STARTING ADDRESS
 76013 040535           STA      0,PPC                 ;IN PPC
 76014 006432           JSR      @ARST                 ;RESET ALL COUNTERS
 76015 020440           LDA      0,TWN                 ;BRING IN LOC.'S
 76016 040441           STA      0,CTSS                ;20 TO 37 TO
 76017 040441           STA      0,IFA                 ;SPECIAL LOCATIONS
 76020 020442           LDA      0,MELB                ;IN MAIN MEMORY
 76021 040440           STA      0,ITA                 ;
 76022 022436           LDA      0,@IFA                ;
 76023 042436           STA      0,@ITA                ;
 76024 010434           ISZ      IFA                   ;
 76025 010434           ISZ      ITA                   ;
 76026 014431           DSZ      CTSS                  ;
 76027 000773           JMP      .-5                   ;
 76030 020424           LDA      0,TN
 76031 040421           STA      0,CT
 76032 030516 INI:      LDA      2,PPC
 76033 052511           STA      2,@AVA                ;BRING IN 1ST 32 PAGES
 76034 006417           JSR      @ABRG                 ;INTO MAIN MEMORY
 76035 032507           LDA      2,@AVA
 76036 020413           LDA      0,PGSZ
 76037 113000           ADD      0,2
 76040 052504           STA      2,@AVA
 76041 014411           DSZ      CT
 76042 000772           JMP      INI+2
 76043 006404           JSR      @AIPRI                ;INITIALIZE PRIORITY
 76044 063077           HALT
 76045 000416           JMP      START-1
 76046 076520 ARST:     RESET
 76047 077255 AIPRI:    IPRI
 76050 076163 ASACO:    SACO
 76051 000200 PGSZ:     200                            ;PAGE SIZE
 76052 000000 CT:       0
 76053 077051 ABRG:     IBR
 76054 000037 TN:       37                             ;NUMBER OF PAGE FRAMES
 76055 000020 TWN:      20
 76056 000000 CTS:      0
 76057 000000 CTSS:     0
 76060 000000 IFA:      0
 76061 000000 ITA:      0
 76062 065760 MELB:     65760                          ;MEMORY LOWER BOUND
 76063 020500           LDA      0,SACO                ;RESTORE ACO
 76064 040477 START:    STA      0,SACO                ;SAVE ACCUMULATORS
 76065 044477           STA      1,SAC1
 76066 050477           STA      2,SAC2
 76067 054477           STA      3,SAC3
```

```
---
0002  .MAIN
 76070 101200          MOVR     0,0                ;SAVE CARRY
 76071 040476          STA      0,SACY
 76072 022454 BEGIN:   LDA      0,@ANFLG
 76073 101004          MOV      0,0       SZR      ;SKP UNLESS INFLG IS SET
 76074 000455          JMP      INT                ;SERVICE INTERRUPT
 76075 020452          LDA      0,DUM
 76076 000401 ITN:     JMP      .+1                ;DUMMY LOC. USED FOR
                                                   ;EXEC. OF INTEN INSTR.
 76077 040777          STA      0,ITN              ;RESTORE JMP.+1
 76100 030450          LDA      2,PPC
 76101 052443          STA      2,@AVA
 76102 006441          JSR      @AAT               ;TRANSLATE ADDRESS
 76103 006442          JSR      @AFET              ;FETCH INSTRUCTION FROM
                                                   ;REAL ADDRESS
 76104 020464          LDA      0,TEST1
 76105 106433          SUBZ#    0,1       SNC      ;SKIP UNLESS M.R.I.
 76106 002434          JMP      @AMR
 76107 020471 EXEC:    LDA      0,TEST9
 76110 106415          SUB#     0,1       SNR      ;SKP UNLESS INSTR.=INTEN
 76111 000447          JMP      INTN
 76112 044407          STA      1,.+7              ;STORE INSTR. IN P.I.R.
 76113 020454          LDA      0,SACY
 76114 101100          MOVL     0,0                ;RESTORE CARRY
 76115 020446          LDA      0,SAC0             ;RESTORE ACCUMULATORS
 76116 024446          LDA      1,SAC1
 76117 030446          LDA      2,SAC2
 76120 034446          LDA      3,SAC3
 76121 000000 PIR:     0                           ;PSUEDO-INSTR. REGISTER
                                                   ;EXECUTES INSTRUCTION
 76122 000402          JMP      .+2                ;INSTR. DIDNT CAUSE SKP
 76123 010425          ISZ      PPC                ;INSTR. CAUSED SKIP-SO
                                                   ;INCREMENT PPC
 76124 010424          ISZ      PPC                ;INCREMENT PPC
 76125 010411          ISZ      ICT1
 76126 000736          JMP      START
 76127 010410          ISZ      ICT2
 76130 000734          JMP      START
 76131 010407          ISZ      ICT3
 76132 000401          JMP      .+1
 76133 000731          JMP      START              ;RETURN
 76134 000000 EA:      0                           ;EFFECTIVE ADDRESS
 76135 000000 INSR:    0                           ;PSUEDO-LOCATION 000001
 76136 000000 ICT1:    0                           ;1ST INSTRUCTION COUNTER
 76137 000000 ICT2:    0                           ;2ND INSTRUCTION COUNTER
 76140 000000 ICT3:    0                           ;3RD INSTRUCTION COUNTER
 76141 076511 ASMSR:   SMSR
 76142 076211 AMR:     MRS
 76143 076664 AAT:     ADTRN
 76144 077041 AVA:     VA
 76145 076325 AFET:    FETCH
 76146 076517 ANFLG:   INFLG
 76147 000401 DUM:     401
 76150 000000 PPC:     0
 76151 020777 INT:     LDA      0,PPC              ;SAVE PPC IN
 76152 040436          STA      0,SPPC             ;PSUEDO-LOC.0
 76153 020762          LDA      0,INSR             ;MODIFY PPC TO CONTENTS
 76154 040774          STA      0,PPC              ;OF PSUEDO-LOC.1
 76155 102400          SUB      0,0
 76156 042770          STA      0,@ANFLG           ;CLEAR INTERRUPT FLAG
```

```
---
0003  .MAIN
 76157 000716             JMP     BEGIN+3
 76160 040716 INTN:       STA     0,ITN                  ;PUT 'INTEN' IN ITN
 76161 010767             ISZ     PPC                    ;INCR. PROGRAM COUNTER
 76162 000716             JMP     ITN+2                  ;FETCH NEXT INSTR.
 76163 000000 SAC0:       0
 76164 000000 SAC1:       0
 76165 000000 SAC2:       0
 76166 000000 SAC3:       0
 76167 000000 SACY:       0
 76170 060000 TEST1:      060000
 76171 001400 TEST2:      001400
 76172 001000 TEST3:      001000
 76173 000400 TEST4:      000400
 76174 000200 TEST5:      000200
 76175 002000 TEST6:      002000
 76176 010000 TEST7:      010000
 76177 004000 TEST8:      004000
 76200 060177 TEST9:      060177
 76201 040000 TES10:      040000
 76202 007400 MSCS:       7400
 76203 001400 MASK1:      001400
 76204 000377 MASK2:      000377
 76205 174000 MASKA:      174000
 76206 002413 MODFY:      002413
 76207 177400 MINUS:      177400
 76210 000000 SPPC:       0                              ;PSUEDO-LOCATION 000000
 76211 034773 MRS:        LDA     3,MASK2
 76212 137400             AND     1,3                    ;LOAD DISPL. INTO AC 3
 76213 030770             LDA     2,MASK1
 76214 133400             AND     1,2                    ;LOAD INDEX INTO AC 2
 76215 020754             LDA     0,TEST2
 76216 112415             SUB#    0,2       SNR          ;SKIP UNLESS INDEX = 3
 76217 000470             JMP     INDX3
 76220 020752             LDA     0,TEST3
 76221 112415             SUB#    0,2       SNR          ;SKIP UNLESS INDEX = 2
 76222 000467             JMP     INDX2
 76223 020750             LDA     0,TEST4
 76224 112415             SUB#    0,2       SNR          ;SKIP UNLESS INDEX = 1
 76225 000470             JMP     INDX1
 76226 171000 INDX0:      MOV     3,2                    ;PUT E.A. IN AC2
 76227 020746             LDA     0,TEST6
 76230 107415             AND#    0,1       SNR          ;SKIP IF INDIRECT BIT =1
 76231 000420             JMP     DIR
 76232 054542 TR:         STA     3,SAC3S                ;SAVE AC3
 76233 044542             STA     1,SAC1S                ;SAVE AC1
 76234 052710             STA     2,@AVA
 76235 006706             JSR     @AAT                   ;TRANSLATE ADDRESS
 76236 032533             LDA     2,@ARA
 76237 034535             LDA     3,SAC3S                ;RESTORE AC3
 76240 024535             LDA     1,SAC1S                ;RESTORE AC1
 76241 004560             JSR     AUTO                   ;TEST FOR AUTO-INC/DEC
 76242 004534             JSR     REWR                   ;READ/WRITE ROUTINE
 76243 031000             LDA     2,0,2                  ;ADDRESSING IS INDIRECT-
                                                         ;LOAD AC2 WITH (AC2)
 76244 151103             MOVL    2,2       SNC          ;SKIP IF BIT( O )=1
 76245 000403             JMP     .+3
 76246 151220             MOVZR   2,2                    ;PUT BIT( O )=0
 76247 000765             JMP     TR+2                   ;RETURN
 76250 151220             MOVZR   2,2                    ;RESTORE E.A.IN AC2
```

```
---
0004  .MAIN
 76251 020725 DIR:     LDA      0,TEST7
 76252 106433          SUBZ#    0,1       SNC       ;SKIP UNLESS M.R.I. IS
                                                    ;A 'JMP' OR 'JSR'
 76253 000420          JMP      JI
 76254 044521          STA      1,SAC1S             ;SAVE AC1
 76255 052667          STA      2,@AVA
 76256 006665          JSR      @AAT                ;TRANSLATE ADDRESS
 76257 024516          LDA      1,SAC1S             ;RESTORE AC1
 76260 032511          LDA      2,@ARA
 76261 004515          JSR      REWR                ;READ/WRITE ROUTINE
 76262 020723          LDA      0,MASKA             ;REMOVE ADRRESSING INFO.
 76263 107400          AND      0,1                 ;FROM INSTRUCTION
 76264 020722          LDA      0,MODFY             ;MODIFY INSTR. SO IT
 76265 107000          ADD      0,1                 ;ADDRESSES EA
 76266 151100          MOVL     2,2                 ;ENSURE BIT 0 OF
 76267 151220          MOVZR    2,2                 ;EFF. ADDR. =0
 76270 050644          STA      2,EA                ;STORE EFF. ADDR. IN EA
 76271 006412          JSR      @AMF                ;SET FLG IF MODIFIED
 76272 002412          JMP      @AEXEC              ;EXECUTE INSTRUCTION
 76273 020704 JI:      LDA      0,TEST8
 76274 106433          SUBZ#    0,1       SNC       ;SKIP IF INSTR. IS 'JSR'
 76275 000404          JMP      .+4                 ;INSTR. IS 'JMP'
 76276 034652          LDA      3,PPC
 76277 175400          INC      3,3                 ;AC 3 CONTAINS PPC+1
 76300 054666          STA      3,SAC3              ;RETURN ADDR. INTO SAC3
 76301 050647          STA      2,PPC               ;MODIFY PPC TO E.A.
 76302 002403          JMP      @ABEG               ;RETURN
 76303 076626 AMF:     MFLAG
 76304 076107 AEXEC:   EXEC
 76305 076072 ABEG:    BEGIN
 76306 077777 CBIT0:   77777
 76307 030657 INDX3:   LDA      2,SAC3
 76310 000402          JMP      .+2
 76311 030654 INDX2:   LDA      2,SAC2
 76312 020774          LDA      0,CBIT0
 76313 113400          AND      0,2                           ;IGNORE ADDRESSB
 76314 000402          JMP      .+2
 76315 030633 INDX1:   LDA      2,PPC
 76316 020656          LDA      0,TEST5
 76317 116433          SUBZ#    0,3       SNC       ;SKIP IF DISPL.<0
 76320 000403          JMP      .+3
 76321 020666          LDA      0,MINUS
 76322 117000          ADD      0,3                 ;EXTEND SIGN
 76323 157000          ADD      2,3                 ;ADD APPROPRIATE POINTER
                                                    ;TO DISPLACEMENT
 76324 000702          JMP      INDX0

                                ;FETCH ROUTINE
 76325 054436 FETCH:   STA      3,FTSV              ;SAVE RETURN ADDR.
 76326 032443          LDA      2,@ARA              ;ADDR. OF NEXT INSTR.
 76327 102520          SUBZL    0,0                 ;PUT +1 IN AC0
 76330 142414          SUB#     2,0       SZR       ;SKIP IF ADDRESS=1
 76331 000403          JMP      .+3
 76332 030466          LDA      2,AINSR             ;ADDR.=PSEU.LOC.1
 76333 000426          JMP      FOUND
 76334 151014          MOV#     2,2       SZR       ;SKIP IF ADDR.=0
 76335 000403          JMP      .+3
 76336 030432          LDA      2,ASPPC             ;ADDR.=PSEU.LOC.0
 76337 000422          JMP      FOUND
```

```
---
0005  .MAIN
 76340 022432            LDA      0,@AMELB
 76341 112433            SUBZ#    0,2       SNC
 76342 000414            JMP      EX
 76343 020430            LDA      0,MEUB
 76344 142433            SUBZ#    2,0       SNC          ;SKP IF MEM. NOT EXEEDED
 76345 000411            JMP      EX
 76346 020421            LDA      0,AUBD
 76347 142433            SUBZ#    2,0       SNC          ;SKIP IF ADDR.<UBD
 76350 000411            JMP      FOUND
 76351 020415            LDA      0,ALBD
 76352 112433            SUBZ#    0,2       SNC          ;SKIP IF ADDR.>LBD
 76353 000406            JMP      .+6
 76354 020410            LDA      0,ZRON                 ;LOAD AC0 WITH ASCII 01
 76355 000402            JMP      .+2
 76356 020407 EX:        LDA      0,ZRTO
 76357 004466            JSR      ERR                    ;JMP TO ERROR ROUTINE
 76360 063077            HALT
 76361 025000 FOUND:     LDA      1,0,2                  ;LOAD INSTR. AT (PPC)
 76362 002401            JMP      @FTSV                  ;RETURN
 76363 000000 FTSV:      0
 76364 030460 ZRON:      030460
 76365 031060 ZRTO:      031060
 76366 076000 ALBD:      LBD
 76367 077327 AUBD:      UBD
 76370 076210 ASPPC:     SPPC
 76371 077042 ARA:       RA
 76372 076062 AMELB:     MELB                            ;MEM .LOWER BOUND ADDR.
 76373 075777 MEUB:      75777                           ;MEM. UPPER BOUND
 76374 000000 SAC3S:     0
 76375 000000 SAC1S:     0

                                  ;READ/WRITE ROUTINE
 76376 054765 REWR:      STA      3,FTSV                 ;SAVE RETURN ADDR.
 76377 020767            LDA      0,ALBD                 ;START ADDR.OF SIM
 76400 112433            SUBZ#    0,2       SNC          ;SKIP IF E.A.>LBD
 76401 000407            JMP      .+7
 76402 020765            LDA      0,AUBD                 ;END ADDR. OF SIM
 76403 142433            SUBZ#    2,0       SNC          ;SKIP IF E.A.<UBD
 76404 000404            JMP      .+4
 76405 020412            LDA      0,ZRZR                 ;LOAD AC0 WITH ASCII 00
 76406 004437            JSR      ERR                    ;JMP TO ERROR ROUTINE
 76407 030407            LDA      2,ADUMP                ;PUT EFF.ADDR.=ADUMP
 76410 102520            SUBZL    0,0                    ;PUT +1 IN AC0
 76411 142415            SUB#     2,0       SNR          ;SKIP UNLESS E.A.=1
 76412 030406            LDA      2,AINSR                ;E.A.=PSUEDO-LOC. 1
 76413 151005            MOV      2,2       SNR          ;SKIP UNLESS ADDR.=0
 76414 030754            LDA      2,ASPPC                ;ADDR.=PSUEDO-LOC.0
 76415 002746            JMP      @FTSV                  ;RETURN
 76416 077326 ADUMP:     DUMP
 76417 030060 ZRZR:      030060
 76420 076135 AINSR:     INSR

                                  ;AUTO ROUTINE
 76421 054420 AUTO:      STA      3,AUSV                 ;SAVE RETURN ADDRESS
 76422 020422            LDA      0,FO
 76423 142033            ADCZ#    2,0       SNC          ;SKIP IF E.A.<40
 76424 000414            JMP      RTN                    ;RETURN
 76425 022415            LDA      0,@TW
 76426 112433            SUBZ#    0,2       SNC          ;SKIP IF E.A.>=20
```

```
---
0006   .MAIN
 76427 000411             JMP      RTN                ;RETURN
 76430 020413             LDA      0,TH
 76431 142033             ADCZ#    2,0      SNC       ;SKIP IF E.A.<30
 76432 000404             JMP      .+4
 76433 011000             ISZ      0,2                ;INCREMENT LOCATION
 76434 000401             JMP      .+1                ;ALLOWS FOR SKIP
 76435 000403             JMP      .+3
 76436 015000             DSZ      0,2                ;DECREMENT LOCATION
 76437 000401             JMP      .+1
 76440 002401 RTN:        JMP      @AUSV              ;RETURN
 76441 000000 AUSV:       0
 76442 076062 TW:         MELB
 76443 065770 TH:         65770
 76444 066000 FO:         66000

                                   ;ERROR ROUTINE
 76445 054417 ERR:        STA      3,ERRET            ;STORE RETURN ADDR.
 76446 044417             STA      1,SAV1             ;SAVE ACC'S 1&2
 76447 050417             STA      2,SAV2
 76450 040406             STA      0,.+6              ;REPLACE BLANKS AT END
                                                      ;OF TXT WITH ERR. CODE
 76451 004416             JSR      TXOP               ;OUTPUT THIS TEXT
                          .TXT     /<15><12>ERROR:/
 76452 005015
 76453 051105
 76454 047522
 76455 035122
 76456 000000
 76457 000000 BLANK:      0                           ;BLANKS TO END TEXT
 76460 063077             HALT
 76461 024404             LDA      1,SAV1             ;RESTORE ACC'S 1&2
 76462 030404             LDA      2,SAV2
 76463 002401             JMP      @ERRET             ;RETURN TO PROG.
 76464 000000 ERRET:      0
 76465 000000 SAV1:       0
 76466 000000 SAV2:       0

                                   ;TEXT OUTPUT ROUTINE
 76467 054421 TXOP:       STA      3,PNTR             ;ADDR. OF TXT
 76470 022420 NEXT:       LDA      0,@PNTR            ;LOAD WORD OF TEXT
 76471 010417             ISZ      PNTR               ;INCREMENT PNTR
 76472 105300             MOVS     0,1                ;LOAD LEFT BYTE INTO AC1
 76473 030414             LDA      2,RBYT
 76474 143400             AND      2,0                ;MASK OUT RIGHT BYTE
 76475 147400             AND      2,1
 76476 063511             SKPBZ    TTO
 76477 000777             JMP      .-1
 76500 061111             DOAS     0,TTO              ;OUTPUT RIGHT BYTE
 76501 125005             MOV      1,1      SNR       ;SKIP IF L.B. IS'NT NULL
 76502 002406             JMP      @PNTR              ;RETURN
 76503 063511             SKPBZ    TTO
 76504 000777             JMP      .-1
 76505 065111             DOAS     1,TTO              ;OUTPUT LEFT BYTE
 76506 000762             JMP      NEXT
 76507 000377 RBYT:       377
 76510 000000 PNTR:       0

                                            ;INTERRUPT SERVICE ROUTINE
 76511 040405 SMSR:       STA      0,ISAC0            ;SAVE AC0
```

```
---
 0007  .MAIN
  76512 102520            SUBZL    0,0              ;PUT +1 IN AC0
  76513 040404            STA      0,INFLG          ;SET INTERRUPT FLAG
  76514 020402            LDA      0,ISAC0          ;RESTORE AC0
  76515 002000            JMP      @0               ;RETURN
  76516 000000 ISAC0:     0
  76517 000000 INFLG:     0                         ;INTERRUPT FLAG

                                         ;RESET ROUTINE
  76520 102440 RESET:     SUBO     0,0
  76521 042427            STA      0,@APS1
  76522 042427            STA      0,@APS2
  76523 042427            STA      0,@APL1
  76524 042427            STA      0,@APL2
  76525 042427            STA      0,@AXF1
  76526 042427            STA      0,@AXF2
  76527 042427            STA      0,@AMVA
  76530 042427            STA      0,@AICT1
  76531 042427            STA      0,@AICT2
  76532 042427            STA      0,@AICT3
  76533 042427            STA      0,@AACT1
  76534 042427            STA      0,@AACT2
  76535 042427            STA      0,@AACT3
  76536 026410            LDA      1,@KK
  76537 044410            STA      1,KKT
  76540 030425            LDA      2,AMP0
  76541 041000            STA      0,0,2
  76542 151400            INC      2,2
  76543 014404            DSZ      KKT
  76544 000775            JMP      .-3
  76545 001400            JMP      0,3
  76546 076054 KK:        TN
  76547 000000 KKT:       0
  76550 077163 APS1:      PSCT1
  76551 077164 APS2:      PSCT2
  76552 077165 APL1:      PLCT1
  76553 077166 APL2:      PLCT2
  76554 077210 AXF1:      XFCT1
  76555 077211 AXF2:      XFCT2
  76556 076763 AMVA:      MAXVA
  76557 076136 AICT1:     ICT1
  76560 076137 AICT2:     ICT2
  76561 076140 AICT3:     ICT3
  76562 076774 AACT1:     ADCT1
  76563 076775 AACT2:     ADCT2
  76564 076776 AACT3:     ADCT3
  76565 076566 AMP0:      MP0
  76566 000000 MP0:       0
  76567 000000 MP1:       0
  76570 000000 MP2:       0
  76571 000000 MP3:       0
  76572 000000 MP4:       0
  76573 000000 MP5:       0
  76574 000000 MP6:       0
  76575 000000 MP7:       0
  76576 000000 MP8:       0
  76577 000000 MP9:       0
  76600 000000 MP10:      0
  76601 000000 MP11:      0
  76602 000000 MP12:      0
```

```
---
0008  .MAIN
 76603 000000 MP13:    0
 76604 000000 MP14:    0
 76605 000000 MP15:    0
 76606 000000 MP16:    0
 76607 000000 MP17:    0
 76610 000000 MP18:    0
 76611 000000 MP19:    0
 76612 000000 MP20:    0
 76613 000000 MP21:    0
 76614 000000 MP22:    0
 76615 000000 MP23:    0
 76616 000000 MP24:    0
 76617 000000 MP25:    0
 76620 000000 MP26:    0
 76621 000000 MP27:    0
 76622 000000 MP28:    0
 76623 000000 MP29:    0
 76624 000000 MP30:    0
 76625 000000 MP31:    0

                                                   ;MODIFIED FLAG ROUTINE
 76626 054427 MFLAG:   STA      3,MRA
 76627 034430          LDA      3,MASC
 76630 137400          AND      1,3
 76631 020425          LDA      0,TES11
 76632 116415          SUB#     0,3       SNR      ;SKIP IF NOT EQUAL "LDA"
 76633 002422          JMP      @MRA
 76634 020610          LDA      0,F0
 76635 112433          SUBZ#    0,2       SNC      ;SKIP IF EA>=F0
 76636 002417          JMP      @MRA
 76637 022423          LDA      0,@AMEUB
 76640 142433          SUBZ#    2,0       SNC      ;SKIP IF EA<=MEUB
 76641 002414          JMP      @MRA
 76642 020416          LDA      0,SCON             ;FIND CORR. PAGE
 76643 112400          SUB      0,2
 76644 022415          LDA      0,@AMSC
 76645 143400          AND      2,0
 76646 101120          MOVZL    0,0
 76647 101300          MOVS     0,0
 76650 030715          LDA      2,AMP0
 76651 113000          ADD      0,2
 76652 102520          SUBZL    0,0
 76653 041000          STA      0,0,2
 76654 002401          JMP      @MRA
 76655 000000 MRA:     0
 76656 020000 TES11:   20000
 76657 160000 MASC:    160000
 76660 006000 SCON:    6000
 76661 077161 AMSC:    MSC
 76662 076373 AMEUB:   MEUB
 76663 076445 GERR:    ERR

                                                   ;ADDRESS TRANSLATION ROUTINE
 76664 054500 ADTRN:   STA      3,RAD
 76665 010507          ISZ      ADCT1
 76666 000405          JMP      .+5
 76667 010506          ISZ      ADCT2
 76670 000403          JMP      .+3
 76671 010505          ISZ      ADCT3
```

```
---
0009  .MAIN
 76672 000401          JMP      .+1
 76673 030546          LDA      2,VA
 76674 022475          LDA      0,@C
 76675 142432          SUBZ#    2,0      SZC      ;SKIP IF VA>65760
 76676 000405          JMP      .+5
 76677 020473          LDA      0,ZRTH
 76700 006763          JSR      @GERR
 76701 012460          ISZ      @APPC
 76702 002460          JMP      @ABEGS
 76703 020460          LDA      0,MAXVA
 76704 142432          SUBZ#    2,0      SZC      ;SKIP IF VA>MAXVA
 76705 000402          JMP      .+2
 76706 050455          STA      2,MAXVA
 76707 020456          LDA      0,FRTY
 76710 112432          SUBZ#    0,2      SZC      ;SKIP IF VA<40
 76711 000415          JMP      NORM
 76712 151015          MOV#     2,2      SNR      ;SKIP UNLESS VA=0
 76713 000411          JMP      SPE
 76714 102520          SUBZL    0,0               ;CREATE +1
 76715 142415          SUB#     2,0      SNR      ;SKIP UNLESS VA=1
 76716 000406          JMP      SPE
 76717 020447          LDA      0,TNTY
 76720 112433          SUBZ#    0,2      SNC      ;SKP UNLESS VA<20
 76721 000405          JMP      NORM
 76722 020446          LDA      0,CO              ;TRANSLATING
 76723 113000          ADD      0,2               ;ADDRESS
 76724 050516 SPE:     STA      2,RA
 76725 002437          JMP      @RAD
 76726 020445 NORM:    LDA      0,MINU            ;MASK OUT RBYTE
 76727 113400          AND      0,2               ; TO GET PAGE NUMBER
 76730 022437          LDA      0,@TEN            ;CHECK PAGETABLE
 76731 040507          STA      0,KOUNT
 76732 126440          SUBO     1,1
 76733 044514          STA      1,MPT
 76734 034443          LDA      3,APO
 76735 021400          LDA      0,0,3
 76736 112415          SUB#     0,2      SNR      ;SKIP UNLESS VP=PI
 76737 000406          JMP      OUT
 76740 010507          ISZ      MPT
 76741 175400          INC      3,3
 76742 014476          DSZ      KOUNT
 76743 000772          JMP      .-6
 76744 004531          JSR      BRING             ;NOT IN, CALL BRING
 76745 020502 OUT:     LDA      0,MPT             ;TRANSLATING
 76746 101300          MOVS     0,0
 76747 101220          MOVZR    0,0
 76750 026475          LDA      1,@CON
 76751 107000          ADD      0,1
 76752 030467          LDA      2,VA
 76753 034470          LDA      3,MSKL            ;MSK OUT LEFT BYTE
 76754 157400          AND      2,3
 76755 167000          ADD      3,1
 76756 044464          STA      1,RA
 76757 006471          JSR      @APRI             ;PRIORITY UPDATING
 76760 002404          JMP      @RAD

 76761 076150 APPC:    PPC
 76762 076072 ABEGS:   BEGIN
 76763 000000 MAXVA:   0                          ;MAXIMUM VIRTUAL ADDRESS
```

```
---
0010  .MAIN
 76764 000000 RAD:      0
 76765 000040 FRTY:     40
 76766 000020 TNTY:     20
 76767 076054 TEN:      TN
 76770 065740 CO:       65740
 76771 076062 C:        MELB
 76772 031460 ZRTH:     31460
 76773 177600 MINU:     177600
 76774 000000 ADCT1:    0
 76775 000000 ADCT2:    0
 76776 000000 ADCT3:    0
 76777 077000 AP0:      P0
 77000 000000 P0:       0
 77001 000000 P1:       0
 77002 000000 P2:       0
 77003 000000 P3:       0
 77004 000000 P4:       0
 77005 000000 P5:       0
 77006 000000 P6:       0
 77007 000000 P7:       0
 77010 000000 P8:       0
 77011 000000 P9:       0
 77012 000000 P10:      0
 77013 000000 P11:      0
 77014 000000 P12:      0
 77015 000000 P13:      0
 77016 000000 P14:      0
 77017 000000 P15:      0
 77020 000000 P16:      0
 77021 000000 P17:      0
 77022 000000 P18:      0
 77023 000000 P19:      0
 77024 000000 P20:      0
 77025 000000 P21:      0
 77026 000000 P22:      0
 77027 000000 P23:      0
 77030 000000 P24:      0
 77031 000000 P25:      0
 77032 000000 P26:      0
 77033 000000 P27:      0
 77034 000000 P28:      0
 77035 000000 P29:      0
 77036 000000 P30:      0
 77037 000000 P31:      0
 77040 000000 KOUNT:    0
 77041 000000 VA:       0
 77042 000000 RA:       0
 77043 000177 MSKL:     177
 77044 007777 MSK:      7777
 77045 076444 CON:      FO
 77046 000000 IBRA:     0
 77047 000000 MPT:      0
 77050 077277 APRI:     PRI

                                              ;INITIAL BRING ROUTINE
 77051 054775 IBR:      STA      3,IBRA
 77052 030767           LDA      2,VA
 77053 024506           LDA      1,MSC
 77054 147400           AND      2,1
```

```
---
0011  .MAIN
 77055 020417          LDA      0,C200
 77056 106400          SUB      0,1
 77057 022766          LDA      0,@CON
 77060 123000          ADD      1,0
 77061 040474          STA      0,TA
 77062 020711          LDA      0,MINU
 77063 143400          AND      2,0
 77064 040472          STA      0,FA
 77065 125120          MOVZL    1,1
 77066 125300          MOVS     1,1
 77067 034710          LDA      3,APO
 77070 137000          ADD      1,3
 77071 041400          STA      0,0,3
 77072 004475          JSR      XFER
 77073 002753          JMP      @IBRA
 77074 000200 C200:    200

                                          ;BRING ROUTINE
 77075 054465 BRING:   STA      3,BRA
 77076 024514          LDA      1,POINT
 77077 131000          MOV      1,2
 77100 125300          MOVS     1,1
 77101 125220          MOVZR    1,1
 77102 022743          LDA      0,@CON
 77103 107000          ADD      0,1
 77104 044453          STA      1,MARK
 77105 026453          LDA      1,@GMPO
 77106 133000          ADD      1,2
 77107 021000          LDA      0,0,2
 77110 101004          MOV      0,0       SZR      ;SKIP UNLESS MFLG SET
 77111 000402          JMP      PUSH
 77112 000415          JMP      PULL               ; BYPASSING PUSH
 77113 024477 PUSH:    LDA      1,POINT            ;TAKE POINTER
 77114 030663          LDA      2,APO
 77115 133000          ADD      1,2
 77116 021000          LDA      0,0,2              ;FIND LOWEST PRIOR. PAGE
 77117 040436          STA      0,TA
 77120 024437          LDA      1,MARK
 77121 044435          STA      1,FA
 77122 004445          JSR      XFER
 77123 010440          ISZ      PSCT1              ;COUNTING
 77124 000403          JMP      .+3
 77125 010437          ISZ      PSCT2
 77126 000401          JMP      .+1
 77127 024430 PULL:    LDA      1,MARK
 77130 044425          STA      1,TA
 77131 020710          LDA      0,VA
 77132 024641          LDA      1,MINU
 77133 107400          AND      0,1                ;FIND   TOP OF PAGE
 77134 044422          STA      1,FA
 77135 030455          LDA      2,POINT
 77136 050711          STA      2,MPT
 77137 034640          LDA      3,APO
 77140 157000          ADD      2,3
 77141 045400          STA      1,0,3              ;UPDATING PAGETABLE
 77142 004425          JSR      XFER
 77143 010422          ISZ      PLCT1
 77144 000403          JMP      .+3
 77145 010421          ISZ      PLCT2
```

```
---
 0012  .MAIN
  77146 000401            JMP      .+1
  77147 024443            LDA      1,POINT
  77150 032410            LDA      2,@GMP0
  77151 133000            ADD      1,2
  77152 102440            SUBO     0,0
  77153 041000            STA      0,0,2               ;CLEAR MFLAG
  77154 002406            JMP      @BRA

  77155 000000 TA:        0                            ;TO-ADDRESS
  77156 000000 FA:        0                            ;FROM-ADDRESS
  77157 000000 MARK:      0
  77160 076565 GMP0:      AMP0
  77161 007600 MSC:       7600
  77162 000000 BRA:       0
  77163 000000 PSCT1:     0                            ;1ST PUSH COUNTER
  77164 000000 PSCT2:     0                            ;2ND PUSH COUNTER
  77165 000000 PLCT1:     0                            ;1ST PULL COUNTER
  77166 000000 PLCT2:     0                            ;2ND PULL COUNTER

                                               ;PAGE TRANSFER ROUTINE
  77167 054420 XFER:      STA      3,XRA
  77170 010420            ISZ      XFCT1               ;INCREMENT XFER COUNTER
  77171 000403            JMP      .+3
  77172 010417            ISZ      XFCT2
  77173 000401            JMP      .+1
  77174 022411            LDA      0,@CONST
  77175 040411            STA      0,COUNT
  77176 022760 LOOP:      LDA      0,@FA               ;TRANSFER 128 WORDS
  77177 042756            STA      0,@TA               ;BETWEEN 2 MEMORIES
  77200 010755            ISZ      TA
  77201 010755            ISZ      FA
  77202 014404            DSZ      COUNT
  77203 000773            JMP      LOOP
  77204 002403            JMP      @XRA

  77205 076051 CONST:     PGSZ
  77206 000000 COUNT:     0
  77207 000000 XRA:       0
  77210 000000 XFCT1:     0                            ;1ST XFER COUNTER
  77211 000000 XFCT2:     0                    ;2ND XFER COUNTER
  77212 000000 POINT:     0                            ;LOWEST PRIOR.PAGE PNTR
  77213 077047 AMPT:      MPT                          ;REFERENCED PAGE POINTER
  77214 077215 APP0:      PP0
  77215 000000 PP0:       0
  77216 000000 PP1:       0
  77217 000000 PP2:       0
  77220 000000 PP3:       0
  77221 000000 PP4:       0
  77222 000000 PP5:       0
  77223 000000 PP6:       0
  77224 000000 PP7:       0
  77225 000000 PP8:       0
  77226 000000 PP9:       0
  77227 000000 PP10:      0
  77230 000000 PP11:      0
  77231 000000 PP12:      0
  77232 000000 PP13:      0
  77233 000000 PP14:      0
  77234 000000 PP15:      0
```

```
---
 0013  .MAIN
  77235 000000 PP16:    0
  77236 000000 PP17:    0
  77237 000000 PP18:    0
  77240 000000 PP19:    0
  77241 000000 PP20:    0
  77242 000000 PP21:    0
  77243 000000 PP22:    0
  77244 000000 PP23:    0
  77245 000000 PP24:    0
  77246 000000 PP25:    0
  77247 000000 PP26:    0
  77250 000000 PP27:    0
  77251 000000 PP28:    0
  77252 000000 PP29:    0
  77253 000000 PP30:    0
  77254 000000 PP31:    0

                                                ;INITIAL PRIORITY ROUTINE
  77255 054416 IPRI:    STA     3,IPRA
  77256 034736          LDA     3,APPO
  77257 030416          LDA     2,K
  77260 050416          STA     2,KT
  77261 030415          LDA     2,KT
  77262 051400          STA     2,0,3
  77263 175400          INC     3,3
  77264 014412          DSZ     KT
  77265 000774          JMP     .-4
  77266 152440          SUBO    2,2
  77267 051400          STA     2,0,3
  77270 030405          LDA     2,K
  77271 050721          STA     2,POINT
  77272 002401          JMP     @IPRA
  77273 000000 IPRA:    0
  77274 000000 PRA:     0
  77275 000036 K:       36
  77276 000000 KT:      0

                                                ;PRIORITY ROUTINE
  77277 054775 PRI:     STA     3,PRA
  77300 036713          LDA     3,@AMPT
  77301 030713          LDA     2,APPO
  77302 157000          ADD     2,3
  77303 021400          LDA     0,0,3
  77304 024771          LDA     1,K
  77305 106415          SUB#    0,1     SNR      ;SKIP IF NOT HIGHEST PRI
  77306 002766          JMP     @PRA
  77307 125400          INC     1,1
  77310 044766          STA     1,KT
  77311 045400          STA     1,0,3
  77312 176440          SUBO    3,3
  77313 025000 LOP:     LDA     1,0,2
  77314 122432          SUBZ#   1,0     SZC
  77315 000404          JMP     .+4
  77316 015000          DSZ     0,2
  77317 000402          JMP     .+2
  77320 054672          STA     3,POINT
  77321 175400          INC     3,3
  77322 151400          INC     2,2
  77323 014753          DSZ     KT
```

```
---
0014  .MAIN
 77324 000767              JMP     LOP
 77325 002747              JMP     @PRA
 77326 000000 DUMP:        0
 77327 077327 UBD:         .
                                   .END
```

```
---
 0015   .MAIN
AACT1   076562
AACT2   076563
AACT3   076564
AAT     076143
ABEG    076305
ABEGS   076762
ABRG    076053
ADCT1   076774
ADCT2   076775
ADCT3   076776
ADTRN   076664
ADUMP   076416
AEXEC   076304
AFET    076145
AICT1   076557
AICT2   076560
AICT3   076561
AINSR   076420
AIPRI   076047
ALBD    076366
AMELB   076372
AMEUB   076662
AMF     076303
AMP0    076565
AMPT    077213
AMR     076142
AMSC    076661
AMVA    076556
ANFLG   076146
AP0     076777
APL1    076552
APL2    076553
APP0    077214
APPC    076761
APRI    077050
APS1    076550
APS2    076551
ARA     076371
ARST    076046
ASAC0   076050
ASMSR   076141
ASPPC   076370
AUBD    076367
AUSV    076441
AUTO    076421
AVA     076144
AXF1    076554
AXF2    076555
BEGIN   076072
BLANK   076457
BRA     077162
BRING   077075
C       076771
C200    077074
CBIT0   076306
CO      076770
CON     077045
CONST   077205
COUNT   077206
```

```
---
  0016   .MAIN
CT       076052
CTS      076056
CTSS     076057
DIR      076251
DUM      076147
DUMP     077326
EA       076134
ERR      076445
ERRET    076464
EX       076356
EXEC     076107
FA       077156
FETCH    076325
FO       076444
FOUND    076361
FRTY     076765
FTSV     076363
GERR     076663
GMP0     077160
IBR      077051
IBRA     077046
ICT1     076136
ICT2     076137
ICT3     076140
IFA      076060
INDX0    076226
INDX1    076315
INDX2    076311
INDX3    076307
INFLG    076517
INI      076032
INSR     076135
INT      076151
INTN     076160
IPRA     077273
IPRI     077255
ISAC0    076516
ITA      076061
ITN      076076
JI       076273
K        077275
KK       076546
KKT      076547
KOUNT    077040
KT       077276
LBD      076000
LOOP     077176
LOP      077313
MARK     077157
MASC     076657
MASK1    076203
MASK2    076204
MASKA    076205
MAXVA    076763
MELB     076062
MEUB     076373
MFLAG    076626
MINU     076773
MINUS    076207
```

```
---
  0017   .MAIN
MODFY    076206
MP0      076566
MP1      076567
MP10     076600
MP11     076601
MP12     076602
MP13     076603
MP14     076604
MP15     076605
MP16     076606
MP17     076607
MP18     076610
MP19     076611
MP2      076570
MP20     076612
MP21     076613
MP22     076614
MP23     076615
MP24     076616
MP25     076617
MP26     076620
MP27     076621
MP28     076622
MP29     076623
MP3      076571
MP30     076624
MP31     076625
MP4      076572
MP5      076573
MP6      076574
MP7      076575
MP8      076576
MP9      076577
MPT      077047
MRA      076655
MRS      076211
MSC      077161
MSCS     076202
MSK      077044
MSKL     077043
NEXT     076470
NORM     076726
OUT      076745
P0       077000
P1       077001
P10      077012
P11      077013
P12      077014
P13      077015
P14      077016
P15      077017
P16      077020
P17      077021
P18      077022
P19      077023
P2       077002
P20      077024
P21      077025
P22      077026
```

```
---
  0018   .MAIN
P23      077027
P24      077030
P25      077031
P26      077032
P27      077033
P28      077034
P29      077035
P3       077003
P30      077036
P31      077037
P4       077004
P5       077005
P6       077006
P7       077007
P8       077010
P9       077011
PGSZ     076051
PIR      076121
PLCT1    077165
PLCT2    077166
PNTR     076510
POINT    077212
PP0      077215
PP1      077216
PP10     077227
PP11     077230
PP12     077231
PP13     077232
PP14     077233
PP15     077234
PP16     077235
PP17     077236
PP18     077237
PP19     077240
PP2      077217
PP20     077241
PP21     077242
PP22     077243
PP23     077244
PP24     077245
PP25     077246
PP26     077247
PP27     077250
PP28     077251
PP29     077252
PP3      077220
PP30     077253
PP31     077254
PP4      077221
PP5      077222
PP6      077223
PP7      077224
PP8      077225
PP9      077226
PPC      076150
PRA      077274
PRI      077277
PSCT1    077163
PSCT2    077164
```

```
---
  0019   .MAIN
PULL    077127
PUSH    077113
RA      077042
RAD     076764
RBYT    076507
RESET   076520
REWR    076376
RTN     076440
SAC0    076163
SAC1    076164
SAC1S   076375
SAC2    076165
SAC3    076166
SAC3S   076374
SACY    076167
SAV1    076465
SAV2    076466
SCON    076660
SMSR    076511
SPE     076724
SPPC    076210
STADD   076000
START   076064
TA      077155
TEN     076767
TES10   076201
TES11   076656
TEST1   076170
TEST2   076171
TEST3   076172
TEST4   076173
TEST5   076174
TEST6   076175
TEST7   076176
TEST8   076177
TEST9   076200
TH      076443
TN      076054
TNTY    076766
TR      076232
TW      076442
TWN     076055
TXOP    076467
UBD     077327
VA      077041
XFCT1   077210
XFCT2   077211
XFER    077167
XRA     077207
ZRON    076364
ZRTH    076772
ZRTO    076365
ZRZR    076417
R
```

APPENDIX C

BACK-UP NUMERICAL RESULTS

a/ ASSEMBLY

No. of frames	1	2	4	8	16
Memory size	$\frac{1}{4}$K	$\frac{1}{2}$K	1K	2K	4K
References	1,436,265	1,520,129	1,557,941	1,595,753	1,607,503
Page faults	359,440	72464	40919	18530	657
Fault rate f	25%	4.77%	2.63%	1.16%	.04%
Hit rate HR	75%	95.23%	97.37%	98.84%	99.96%

b/ ALGOL

No. of frames	2	4	8	16
Memory size	$\frac{1}{2}$K	1K	2K	4K
References	2,221186	2,099,456	2,345,507	2,479,224
Page faults	834,186	382,777	242,746	141,711
Fault rate f	37.5%	18%	10%	5.7%
Hit rate HR	62.5%	82%	90%	94.3%

c/ FORTRAN

No. of frames	2	4	8	16
Memory size	$\frac{1}{2}$K	1K	2K	4K
References	281,630	287,848	335,398	387,503
Page faults	103,732	37,728	28,121	23,036
Fault rate f	36.8%	13%	8.4%	6%
Hit rate HR	63.2%	87%	91.6%	94%

Table I Direct Mapping with fixed P=256 words

a/ ASSEMBLY

No. of frames Memory size	2 ¼K	4 ½K	8 1K	16 2K	32 4K
References	1,546,415	1,555,657	1,572,864	1,581,093	1587,061
Page faults	141,256	50,443	31651	28090	419
Fault rate f	9.2%	3.3%	2%	1.5%	.03%
Hit rate HR	90.8%	96.7%	98%	98.5%	99.97%

b/ ALGOL

No of frames Memory size	2 ¼K	4 ½K	8 1K	16 2K	32 4K
References	2,159,065	2,085,940	2,116,416	2119,376	2,155,592
Page faults	962,943	577,266	185,882	108,403	76,682
Fault rate f	44.6%	27.7%	8.78%	5.1%	3.56%
Hit rate HR	55.4%	72.3%	91.2%	94.9%	96.4%

c/ FORTRAN

No of frames Memory size	2 ¼K	4 ½K	8 1K	16 2K	32 4K
References	276,263	299,310	491,654	535,180	610,990
Page faults	114,059	60,691	25,480	27,279	6017
Fault rate f	41.3%	20%	5.2%	5.1%	1%
Hit rate HR	58.7%	80%	94.8%	94.9%	99%

Table II Direct Mapping with fixed P=128 words

a/ ASSEMBLY

No. of frames Page size P	128 16	64 32	32 64	16 128	4 512
References	1,433,333	1,495,141	1,548,021	1,573,831	1,576,745
Page faults	7454	7892	9319	14074	31040
Fault rate f	.52%	.53%	.6%	.89%	2%
Hit rate HR	99.48%	99.47%	99.4%	99.1%	98%

b/ ALGOL

No of frames Page size P	128 16	64 32	32 64	16 128	4 512
References	2,146,914	2,113,187	2,149,806	2,196230	2,373,938
Page faults	30,980	52,792	86,268	108,411	457,591
Fault rate f	1.44%	2.5%	4%	5%	19%
Hit rate HR	98.5%	97.5%	96%	95%	81%

c/ FORTRAN

No of frames Page size P	128 16	64 32	32 64	16 128	4 512
References	346,219	405,420	471,879	417,661	299,361
Page faults	6451	7419	9611	14978	34848
Fault rate f	1.86%	1.83%	2%	3.6%	11.6%
Hit rate HR	98.1%	98.1%	98%	96.4%	88.2%

Table III Direct Mapping with Fixed M = 2K

a/ASSEMBLY

Page size P Memory size	8 K	16 ¼K	32 ½K	64 1K	128 2K
References	1600,735	1,561,605	1,494,337	1545,459	1,581,045
Page faults	156,860	41623	30435	24272	14074
Fault rate f	9.8%	2.66%	2%	1.57%	.9%
Hit rate HR	90.2%	97.3%	98%	98.4%	99.1%

b/ALGOL

Page size P Memory size	8 K	16 ¼K	32 ½K	64 1K	128 2K
References	2134,726	2,137,286	2,121,329	2,127,980	2127,814
Page faults	458,282	343,258	253,998	135,440	108,409
Fault rate	21.5%	16%	12%	6.4%	5%
Hit rate HR	78.5%	84%	88%	93.6%	95%

c/ FORTRAN

Page size P Memory size	8 K	16 ¼K	32 ½K	64 1K	128 2K
References	520,721	460,084	476,911	474,272	408,736
Page faults	134,290	58,260	30,619	19,605	14,977
Fault rate f	25.8%	12.7%	6.4%	3.2%	3.7%
Hit rate HR	74.2%	87.3%	93.6%	96.8%	96.3%

Table IV Direct Mapping with fixed C=16 frames

a/ ASSEMBLY

No. of frames Memory size M	2 ½K	4 1K	8 2K	16 4K
References	1582,363	1,535,095	1,490,761	1,493,919
Page faults	44,260	15,785	865	132
Fault rate f	2.8%	1%	.06%	.01%
Hit rate HR	97.2%	99%	99.94%	99.99%

b/ ALGOL

NO. of frames Memory size M	2 ½K	4 1K	8 2K	16 4K
References	2094,159	2120,951	2,313,724	2124,098
Page faults	621,334	120,410	23,854	3088
Fault rate f	30%	5.7%	1%	.15%
Hit rate HR	70%	94.3%	99%	99.85%

c/ FORTRAN

No. of frames Memory size M	2 ½K	4 1K	8 2K	16 4K
References	266,885	395,113	412,265	375,872
Page faults	69,705	19,164	4282	278
Fault rate f	26%	5%	1%	.1%
Hit rate HR	74%	95%	99%	99.9%

<u>Table V</u> LRU Stack with fixed P=256 words

a/ ASSEMBLY

No. of frames Memory size M	2 $\frac{1}{4}$K	4 $\frac{1}{2}$K	8 1K	16 2K	32 4K
References	1533,603	1574,755	1,503,997	1,464,371	1403,349
Page faults	143,340	23,309	1583	1008	248
Fault rate f	9.4%	1.4%	.1%	.07%	.02%
Hit rate HR	90.6%	98.6%	99.9%	99.93%	99.98%

b/ ALGOL

No of frames Memory size M	4 $\frac{1}{2}$K	8 1K	16 2K	32 4K
References	2098,817	2112,530	2121,002	2252,858
Page faults	264,425	47,625	10,879	1726
Fault rate f	12.6%	2.25%	.5%	.08%
Hit rate HR	87.4%	98%	99.5%	99.92%

c/ FORTRAN

No. of frames Memory size M	2 $\frac{1}{4}$K	4 $\frac{1}{2}$K	8 1K	16 2K	32 4K
References	289,932	312,627	381,476	369,592	311,599
Page faults	76131	34690	10497	2713	235
Fault rate f	26%	11%	2.7%	.7%	.08%
Hit rate HR	74%	89%	97.3%	99.3%	99.92%

Table VI LRU Stack with fixed P=128 words

a/ ALGOL

No. of frames Page size P	64 32	32 64	4 512	
References	2,076,488	2,109,232	2,113,047	
Page faults	9177	10,382	68,004	
Fault rate f	.4%	.5%	3.2%	
Hit rate HR	99.6%	99.5%	96.8%	

b/ FORTRAN

No. of frames Page size P	64 32	32 64	4 512	
References	279,373	329,348	420,341	
Page faults	1668	1754	4000	
Fault rate f	.6%	.5%	1%	
Hit rate HR	99.4%	99.5%	99%	-

Table VII LRU Stack with fixed M=2K

a/ ALGOL

Page size P Memory size	16 ¼K	32 ½K	64 1K	
References	2095,304	2,102,382	2,107,585	
Page faults	150,311	77,745	34,413	
Fault rate	7.2%	3.7%	1.6%	
Hit rate HR	92.8%	96.3%	98.4%	

b/ FORTRAN

Page size P Memory size	16 ¼K	32 ½K	64 1K	
References	358,057	336,901	344,686	
Page faults	33,103	20,867	9647	
Fault rate	9.2%	6.2%	2.8%	
Hit rate HR	90.8%	93.8%	97.2%	

Table VIII LRU Stack with fixed C=16 frames

a/ ASSEMBLY

No. of frames Memory size M		4 1K	8 2K	16 4K
Pushes		9743	716	94
Pulls		15785	865	132
Ratio		.617	.827	.712

b/ ALGOL

No. of frames Memory size M	2 ½K	4 1K	8 2K	16 4K
Pushes		43,222	7204	526
Pulls		121,033	23,753	3088
Ratio		.357	.303	.170

c/ FORTRAN

No. of frames Memory size M		4 1K	8 2K	16 4K
Pushes		8048	881	70
Pulls		19294	4287	278
Ratio		.417	.205	.25

Table IX Modified-Page Checking system with LRU stack at fixed P=256 words

a/ ASSEMBLY

No. of frames / Memory size M	4 / ½K	8 / 1K	16 / 2K	32 / 4K
Pushes	16365	1187	792	194
Pulls	23309	1583	1008	248
Ratio	.70	.75	.785	.782

b/ ALGOL

No. of frames / Memory size M	4 / ½K	8 / 1K	16 / 2K	32 / 4K
Pushes	108,662	13,971	1567	349
Pulls	265,524	47,705	10,959	1726
Ratio	.41	.3	.14	.20

c/ FORTRAN

No. of frames / Memory size M	4 / ½K	8 / 1K	16 / 2K	32 / 4K
Pushes	11,411	3466	642	36
Pulls	34,790	10,869	2728	235
Ratio	.33	.32	.23	.153

Table X Modified-page checking system with LRU stack and fixed P=128 words

a/ ASSEMBLY

No. of frames Page size	32 64	16 128	8 256	4 512
Pushes	1038	792	716	5846
Pulls	1474	1008	865	9438
Ratio	.7	.785	.827	.62

b/ ALGOL

No. of frames Page size	32 64	16 128	8 256	4 512
Pushes	1289	1567	7204	22251
Pulls	10374	10,959	23,753	166339
Ratio	.124	.14	.30	.133

c/ FORTRAN

No. of frames Page size	32 64	16 128	8 256	4 512
Pushes	330	642	881	1477
Pulls	1743	2728	4287	4149
Ratio	.19	.23	.20	.36

Table XI Modified-checking system with LRU stack at fixed M=2K

APPENDIX D.

AMENDMENT

Chapter 1. Space-time product

When the amount of main memory can be varied in a process, one of the criteria is the cost of the storage. The space-time product C during a realtime interval (O,t) of a process is defined as :

$$C = \int_{0}^{t} S(z)dz$$

Where S(z) is the amount of storage occupied by the process at time z.

The space-time product can be considered as being proportional to the cost of storage.

Chapter 11. The replacement of algorithms in this chapter are of 2 types: static memory allocations (RAND, FIFO, CRU) where the main memory size is constant and the dynamic memory allocations (WS, PFF) where the main memory size can be varied.

The latter usually appears in multi-programming. LRU, WS, PFF, MIN have stack structure and can be called stack algorithms. In the discussion of prediction methods (page 18) it is not indicated that which pages are replaced because it can be used with many page-replacement algorithms.

Chapter 111. In the simulation of the virtual memory concept, for practical purposes, only small main memory size is considered. There are three test programs. However it is believed that they are typical programs for a small computer and the results from these programs are quite explainable.