SLAC-197
UC-32

EMPIRICAL AND ANALYTICAL STUDIES OF PROGRAM REFERENCE BEHAVIOR*

ABBAS RAFII

STANFORD LINEAR ACCELERATOR CENTER

STANFORD UNIVERSITY

Stanford, California 94305

PREPARED FOR THE ENERGY RESEARCH AND DEVELOPMENT ADMINISTRATION
UNDER CONTRACT NO. E(04-3)-515

July 1976

Printed in the United States of America. Available from National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Road, Springfield, VA 22161. Price: Printed Copy $8.00; Microfiche $2.25.

*Ph.D. dissertation.

ABSTRACT

Several aspects of program reference behavior, page reference behavior modeling and evaluation of multiprogramming paging systems have been considered.

In the first chapter, an experimental treatment of the generated working set size and LRU stack distance strings of actual programs is given. The effects of page size and other parameters on the distribution, serial correlation and frequency domain behavior are studied. A number of working set size models are discussed and the ability to capture the observed degree of the serial dependency in the working set size string is examined.

In the hierarchical design of memory systems, the effectiveness of paging algorithms can be measured by their performance, the processing overhead and the implementation cost. In the second chapter, some new results on the performance and cost of many practical paging systems are presented, and the related evaluation techniques are discussed. The independent reference model is used to find some analytical results for the expected page fault rate of some algorithms. The potential of program behavior models in predicting performance of different page replacement algorithms is demonstrated.

The development of a useful, simple and analytically tractable program page reference model is pursued in the third chapter. A new technique to estimate the parameters of an independent reference model is proposed. It is shown that by inverting the Aϕ optimal fault rate expressions and substituting the observed MIN fault rates of an actual program for the optimal fault rates, one can obtain a model with predictive capabilities. The Aϕ inversion model is capable of accurately predicting the LRU and FIFO page fault rate of programs for different main memory sizes. The model is also successful in predicting the average working set size and working set fault rates of programs for a wide range of window sizes. A comparison of the LRU stack model with the Aϕ inversion model is included. The potential of expanding the model into the areas of simple program restructuring techniques and evaluation of memory hierarchies with unequal read/write cost is discussed.

In the last chapter, some basic relations between the interaction of device scheduling and page scheduling in a multiprogramming virtual memory system are investigated. Queueing analysis and trace driven simulations are used to show the effect of memory allocation policies and various service disciplines on the resource utilization and job waiting times. Some interesting implications of partitioning memory among competing jobs are explored.

ACKNOWLEDGMENTS

I am deeply indebted to my dissertation advisor, Professor Forest Baskett, for his understanding and constant encouragement during the course of this work, and for his guidance and suggestions which have contributed immeasurably in the development of this research.

I would like to thank my readers, Professor Vinton C. Cerf and Professor Allen M. Peterson, for their comments and prompt attention.

I am also grateful to the staff of the SLAC Computation Research Group who have created an excellent research and development environment. Special thanks go to Leonard Shustek for generating the program address traces used in this work, Roger Chaffee and Robert Beach for the graphics system, and Harriet Canfield for her excellent typing.

I would also like to thank my colleagues in the Digital Systems Laboratory and the Computer Science Department for the stimulating discussions, and the staff of the SLAC Computer Center for providing a fine computing facility.

Finally, I wish to express my appreciation to Dr. Aziz Rafii and Mrs. Iran Rafii for their encouragement and constant support throughout my studies.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS(Contd.)

INTRODUCTION

The development of automated memory management techniques [11,14] has created interest in studying the factors which are important in the performance of these systems [2,6,8,12,17,18]. In this paper, several aspects of program reference behavior, page reference behavior modeling, and evaluation of multiprogramming paging systems have been considered.

Insight into the reference behavior of programs is essential in the architecture of new memory systems, computer modeling efforts [9], and the design of effective and efficient algorithms for some fundamental aspects of an operating system, such as scheduling and resource allocation policies. In order to study the address reference characteristics of the actual programs, the execution of several programs have been monitored. The first chapter is an experimental treatment of the page reference behavior of these programs. Statistical and time series analyses are performed on the generated working set size [10] and LRU stack distance [9] strings. The effects of page size and other parameters on the distribution, serial correlation and frequency domain behavior are studied. A number of working set size models are discussed, and the ability to capture the observed degree of the serial dependency in the working set size string is examined. A discussion on the observed accuracy of a number of algorithms to predict the working set size of a program, based on the past observations, is included.

In the second chapter, we address the problem of memory management policies in a paging system [11], and we specifically consider the performance efficiency of the page replacement algorithms. In the literature, many useful results in this area have been presented [1,4,7,12,15]. However, the problems regarding the performance and complexity of the strategies have not received enough treatment. In this chapter, we consider a number of practical algorithms which are easy to implement and which do not require much processing time. We compare the performance of these algorithms with more elaborate schemes and discuss the advantages of simple approaches. In studying the paging algorithms, we discuss several useful evaluation techniques [5,17].

The program reference behavior models [9] are valuable tools to study and predict the operation of paging techniques. In Chapter 2, we use the independent reference model to obtain analytical results for the expected page fault [15] and the average memory usage of several algorithms.

The development of a useful, simple, and analytically tractable program page reference model is pursued in the third chapter. We propose a new technique to estimate the parameters of an independent reference model. We show that by inverting the Aϕ [1] optimal fault rate expressions and substituting the observed MIN [4] fault rates of an actual program for the optimal fault rates, one can obtain a model with predictive capabilities. The Aϕ inversion model is capable of accurately predicting the LRU [9] and FIFO page fault rate of programs for different main memory sizes. The model is also successful in predicting the average working set size and working set fault rates of programs for a wide range of window sizes. We also consider another program reference model, namely, the LRU stack model [9]. We compare the behavior of the models under similar environments. Chapter 3 is concluded with a discussion of the limitation and the possible applications of the Aϕ inversion model in the areas of simple program restructuring techniques [13], and evaluation of memory hierarchies with unequal page read/write operation costs.

In Chapter 4, the modeling effort is extended to include the CPU and its paging device. We investigate some basic relations between the interaction of device scheduling and page scheduling in a multiprogramming virtual memory system. Queueing analysis [3,16] is used to show the effect of memory allocation policies and various service disciplines on the resource utilization and job waiting times. Some interesting implications of sharing memory among competing jobs are explored. Trace driven simulations are used to verify the results under alternative assumptions.

BIBLIOGRAPHY

1. Aho, A.V., Denning, P.J., Ullman, J.D., "Principles of optimal page replacement," Journal of ACM, 18, 1 (1971).

2. Bard, Y., "Experimental evaluation of system performance," IBM Systems Journal, 12, 3 (1973).

3. Baskett, F., Chandy, J.M., Muntz, R.R., "Open, closed and mixed networks of queues with different classes of customers," Journal of ACM, 22, 2 (1975).

4. Belady, L.A., "A study of replacement algorithms for a virtual storage computer," IBM Systems Journal, 5, 2 (1966).

5. Belady, L.A., Palmero, F.P., "On line measurement of paging behavior by multivalued MIN algorithm," IBM J. of Res. and Develop. 18, 1 (Jan. 1974).

6. Chamberlin, D.D., Fuller, S.H., Liu, L.Y., "An analysis of page allocation strategies for multiprogramming systems with virtual memory," IBM J. of Res. and Develop., 17, 5 (Sept. 1973).

7. Chu, W.W., Opderbeck, H., "The page fault frequency replacement algorithm," AFIPS Conf. Proc., Fall Joint Computer Conference (1972).

8. Chiu, W., Dupont, D., Wood, R., "Performance analysis of a multiprogramming computer system," IBM J. of Res. and Develop. (May 1975).

9. Coffman, E.G. Denning, P.J., "Operating system theory," Prentice Hall, Englewood Cliffs, New Jersey (1973).

10. Denning, P.J., "The working set model for program behavior," CACM 11, 5 (May 1968).

11. Denning, P.J., "Virtual memory," Computing Surveys, 2, 3 (1970).

12. Hatfield, D.J., "Experiments on page size, program access patterns, and virtual memory performance," IBM J. of Res. and Develop. 16, 1 (1972).

13. Hatfield, D.J., Gerald, J., "Program restructuring for virtual memory," IBM Systems Journal 10, 3 (1971).

14. Kilburn, T., Edwards, D.B.G., Lanigan, M.J., and Sumner, F.H., "One level storage system," IRE Trans. on Elec. Computers, EC-11, No. 2 (1962).

15. King, W.F., "Analysis of demand paging algorithms," Proc. IFIPS Conf., Ljubliana, Yugoslavia (1971).

16. Kleinrock, L., "Queueing systems," Vol. 1, II, John Wiley and Sons, New York (1975, 1976).

17. Mattson, R.L., Gecsei, D.R., Slutz, D.R. and Traiger, I.L., "Evaluation techniques for storage hierarchies," IBM Systems Journal, 9, 2 (1970).

18. Rodriguez, Rosell J., Dupuy, J-P., "The evaluation of a time sharing page demand system," AFIPS Conf. Proc., Spring Joint Computer Conference (1972).

CHAPTER 1

EMPIRICAL STUDIES OF PROGRAM REFERENCE BEHAVIOR

1.1 INTRODUCTION

The execution reference behavior of programs in general can be studied with several different objectives. A hardware designer can choose a cost effective instruction stack size by measuring the mean number of instructions between successive jumps on programs written for some earlier models. In the design of new instruction sets, the statistics of the successive occurrences of groups of codes on some typical programs can be used to determine the scope and the function for the instructions. In a similar way, the insight into the behavior of programs [4] is essential in the architecture of new memory systems, the computer modeling efforts, and the design of effective and efficient algorithms for some fundamental aspects of an operating system complex, such as scheduling and resource allocation policies.

In this chapter, we are mostly concerned with studying the dynamic (page) reference characteristics of programs, particularly those aspects which are independent of the identity of pages. The traces of several actual programs are used throughout the chapter. The pattern of generated references are investigated by considering the corresponding working set size [4] and stack distance sequences [1]. Some tools from the analysis of time series [8] are used to demonstrate the stochastic properties of each sequence and compare the effect of some parameters, such as the page size, on the behavior of these sequences. The success of some proposed models for the working set size sequence are examined in terms of the ability of these models to capture the time and frequency domain properties of the actual sequence. The algorithms to predict

the working set size, based on the past execution behavior of the programs, are compared.

1.2 SIMULATION

In this paper, the address trace of five programs are used for most of the experiments. A trace program monitors the execution of each program and records the address references, except those generated by the 360 privileged instructions, on a tape. Each entry of the address trace consists of a 22 bit address field and 2 type bits. The address field covers an address space of 4000K bytes, and the type bits indicate if the referenced address is for an instruction fetch, data read, or data write. A paging environment is simulated by putting a page grid on the address space of each program. Thus, each address reference is the page number in which the referenced location falls.

These programs are selected to cover a broad range of application programs. However, we acknowledge the fact that we have omitted a distinctive category of programs, namely, list processing routines. These kinds of programs usually probe a large address space by following linked lists or tree structures which can easily go across many page boundaries. The use of these routines are not very common in general computing environments; however, they may be the most important elements for some particular centers for some dedicated applications. In the following, a brief description of each monitored program is given.

WATFIV Compiler: This program is an incore, one pass, load and go processor to compile FORTRAN programs. It was monitored while compiling a FORTRAN program named WATEX.

WATEX: This program is written in FORTRAN and is supposed to find a minimum of a multi-parameter function. Given a function, execution proceeds until a convergence condition is satisfied or an iteration count is exhausted. It requires little I/O and consists of many small loops and few cases of repeated subroutine calls.

FFT1: This program is a Fast Fourier Transform using the Cooley-Tukey algorithm. It was monitored while computing the finite Fourier Transform of n=2084 data points. In Figure 1.1, we show how the data points are referenced with this algorithm. For n=32 data points, Figure 1.3 shows the pattern of references. There are $\log_2 n$ basic iterations. In the first iteration, pairs $d_1 = n/2 = 16$ words apart are referenced, namely, pairs (1,16), (9,25), (2,18), ... , (16,32). In the second iteration, pairs $d_2 = \frac{n}{2^2} = 8$ words apart are referenced, namely, (1,9), (5,13), (17,25), (21,29), (2,10), ... (24,32). Similarly, in the third iteration, pairs $d_3 = \frac{d_2}{2} = \frac{n}{2^3} = 4$ words apart are referenced. In the final step, data points are referenced sequentially from 1 to n. The reference sequence is schematically shown in Figure 1.1.

<table>
<tr><td colspan="8">1</td></tr>
<tr><td colspan="4">2</td><td colspan="4">3</td></tr>
<tr><td colspan="2">4</td><td colspan="2">5</td><td colspan="2">6</td><td colspan="2">7</td></tr>
<tr><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td></tr>
</table>

FIGURE 1.1

Notice the analogy between this and traversing a binary tree **breadth first.**

FFT2: This is also a Fast Fourier Transform Program which is based on the Cooley-Tukey algorithm and uses the method proposed in [10]. The main difference between this program and FFT1 is the pattern which the data points are referenced. For n=32, Figure 1.3 shows the pattern of references in this program. In the first iteration, data pairs d_1=16 words apart are referenced, namely, pairs (16,32), 15,31), (14,30), (13,29), ... (1,17). In the second iteration, Figure 1.2, data is divided into two parts, (2) and (9), each containing $\frac{n}{2}$=16 points. The following references in each part, and subsequent subdivisions, are carried out independently. In region (2), data points 8 words apart are referenced. Part (2) is divided into parts (3) and (4). In part (3), points distance d=4 apart are referenced. This procedure continues until all data in part (2) have been referenced. Then references are directed to data points in part (9) and the same reference pattern is repeated here.

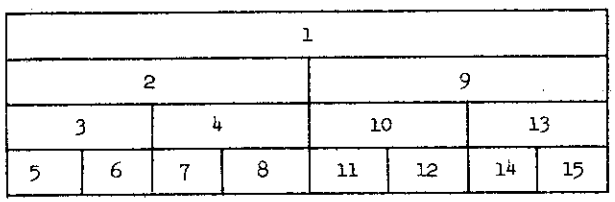


FIGURE 1.2

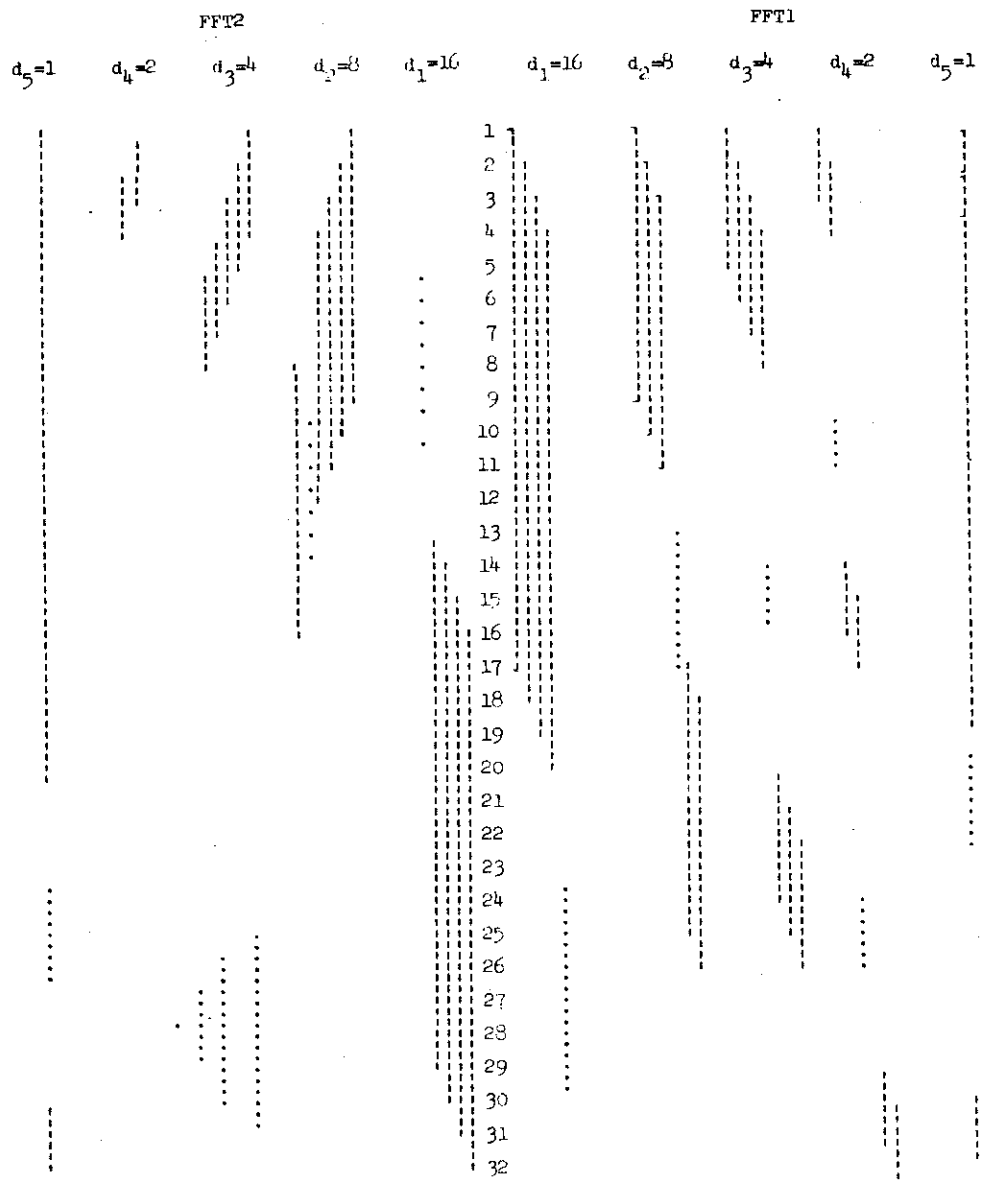


FIGURE 1.3

There is an analogy between the order of references to data here, and traversing a binary tree depth first.

Programs FFT1 and FFT2 are written in ALGOL W language.

APL: This is a trace of an interactive session with APL processor. The session begins by giving some initial commands; then a computation is performed and finally an output (plot of a graph) is obtained.

Table 1.1 gives a summary statistics about each program.

Program	Total Ref's	Type (percent)			Size in Pages			
		read	write	fetch	512	1K	2K	4K
WATFIV	1048661	23.4	15.7	60.9	168	95	54	32
WATEX	2748339	25.0	7.6	67.3	70	41	23	14
FFT1	2954786	30.8	6.2	63.0	82	48	29	18
FFT2	2256197	29.0	6.7	64.3	84	49	29	18
APL	2670920	21.7	10.0	68.3	205	115	67	41

TABLE 1.1 Size and reference type statistics of the monitored programs

1.3 ADDRESS REFERENCE RELATED SEQUENCES

The page reference sequence $r_1, r_2, r_3, \ldots, r_k \ldots$ is a string of page addresses generated by the execution of a program. The page names are from the set of n program pages $[1,2,3,\ldots,n]$ which constitute the address space of the program. Except with some varying degree of sequentiality and repetitive patterns which are observed in page reference sequences, programs tend to generate page addresses in a fairly non-honogenous way. Some weak underlying stochastic structures may be recognized

in each program segment which are tightly related to the identity of the pages. However, as we remove the page identity constraint, many distinct festures emerge from the sequences related to the page reference string. Among such sequences, we consider the working set size [4] and stack distance [1] strings.

The working set size sequence can describe the time dependent locality properties (the tendency to reference in the vicinity of the location addressed in the recent past) of the page references, and the stack distance string gives a space characterization of the references.

1.3.1 Working Set Size Sequence

The working set size sequence on a page reference string is a sequence of working set sizes defined at each reference. The working set [4] at time t, WS(t,T), is the set of pages addressed in the past T references. The working set size at the same time, ws(t,T), is the number of pages referenced at the same interval. T is the working set parameter; thus, the working set size sequence ws(t,T), t=T,T+1,T+2,... can be visualized as the number of distinct pages which are contained in a window of size T, which slides over the page reference string (T and W are used interchangeably to denote the working set parameter in the figures of this chapter).

One can also define the working sets over the data references exclusively. Denote this set by data working set. Thus, the data working set is the set of distinct data pages which are referenced in the last T data references.

The working set size sequence of several programs are sampled and plotted in Figures 1.4 to 1.10. In each figure, the horizontal axis is

the reference count (time) where each unit is 1280 references and the vertical axis is the working set size in number of pages of each size, as indicated in the plots. From these graphs, we can see that the working set size variations can be very significant during the execution of a program. However, there is a high degree of serial dependence between the neighboring points which can be revealed by further investigation. The peaks correspond to the so-called locality changes, such as jumps with large offset, subroutine calls, and similar changes in the concentration points of the references. In such cases, the working sets include the pages from both adjacent localities. Frequently, the working set size drops sharply, which indicates that the references are heavily clustered on a small segment of the program (e.g., execution of a small loop). The peak points, as well as the sharp low points, can create difficulties for a process which should conform itself with the working set size requirements of programs.

The page size and the window size parameters can have significant effect on the shape of the sequence. In Figures 1.4, 1.6 and 1.8, the effect of page size on the working set size sequence of each program are shown. As we expect, large page sizes have a damping effect on the amplitude of variation of the sequence. The mean and coefficient of variation of the sampled points are shown for different page sizes in Table 1.2. We note that the variance decreases as we increase the page size. but the coefficient of variation remains fairly uniform for the range of selected page sizes.

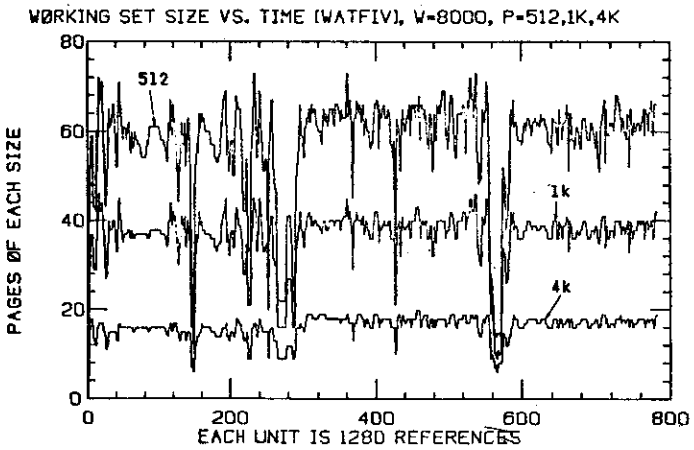


Fig. 1.4

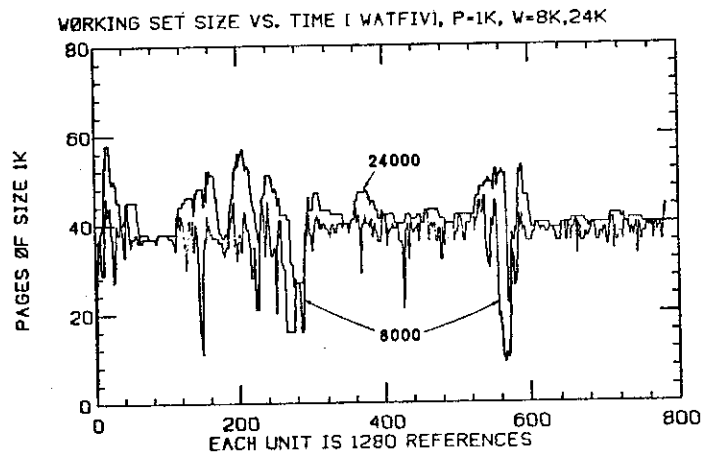


Fig. 1.5

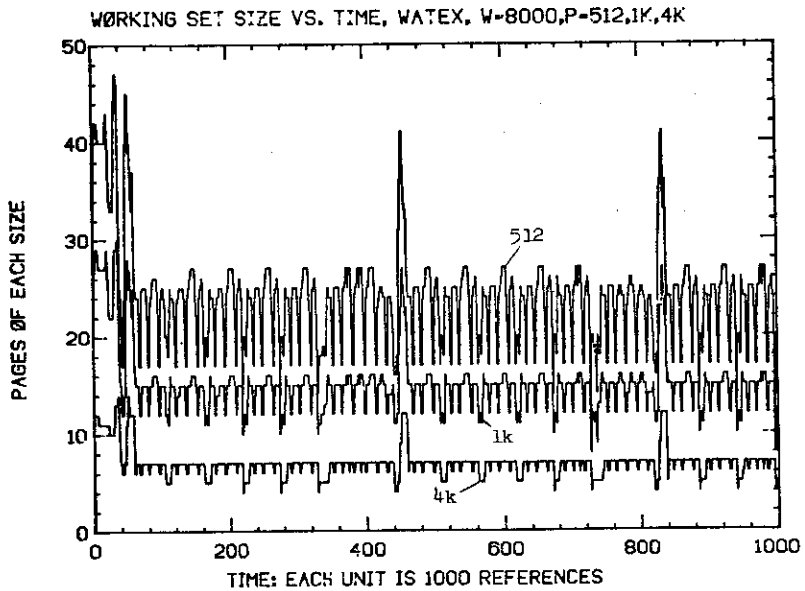


Fig. 1.6

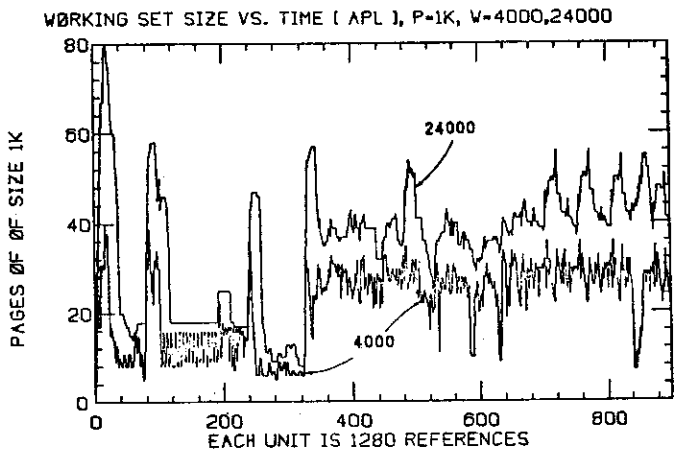


Fig. 1.9

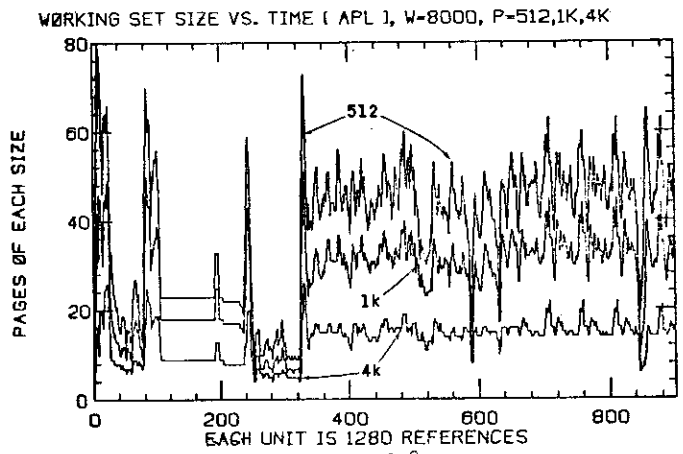


Fig. 1.8

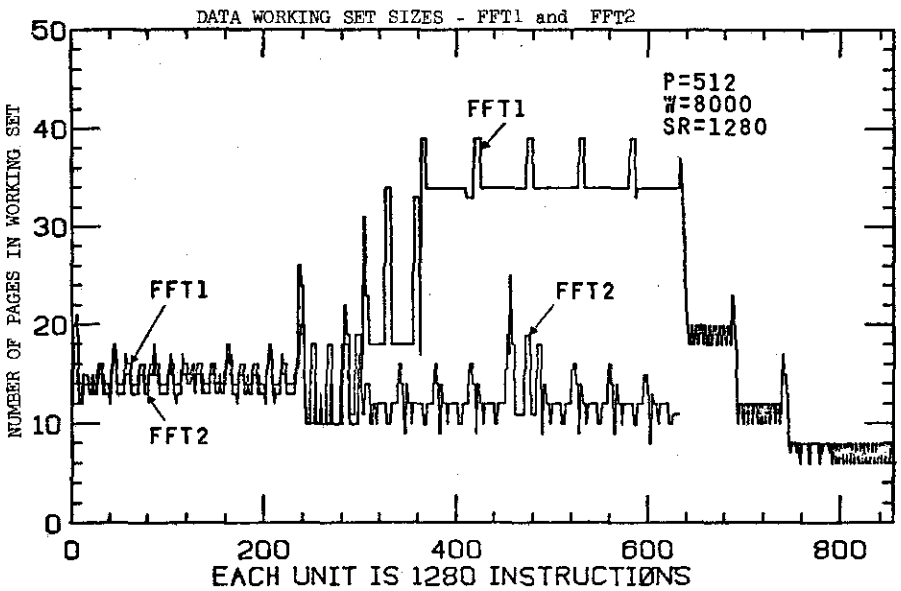


Fig. 1.10

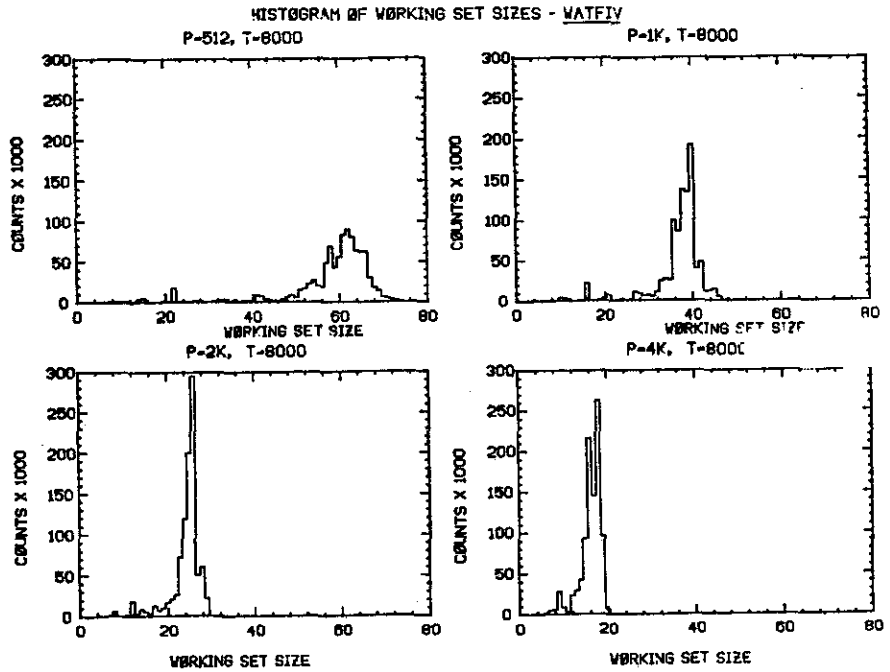


Fig. 1.11

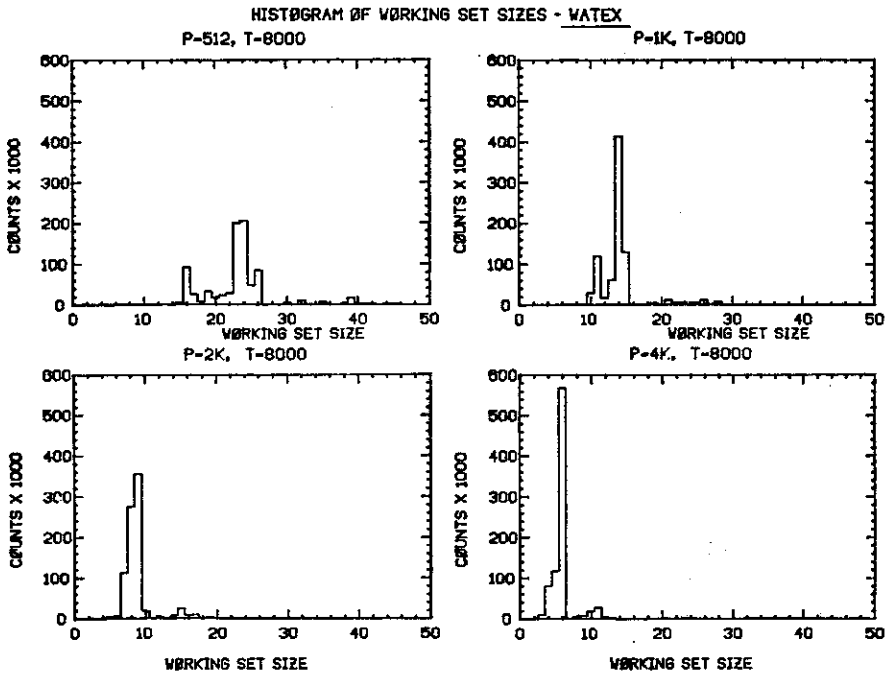


Fig. 1.12

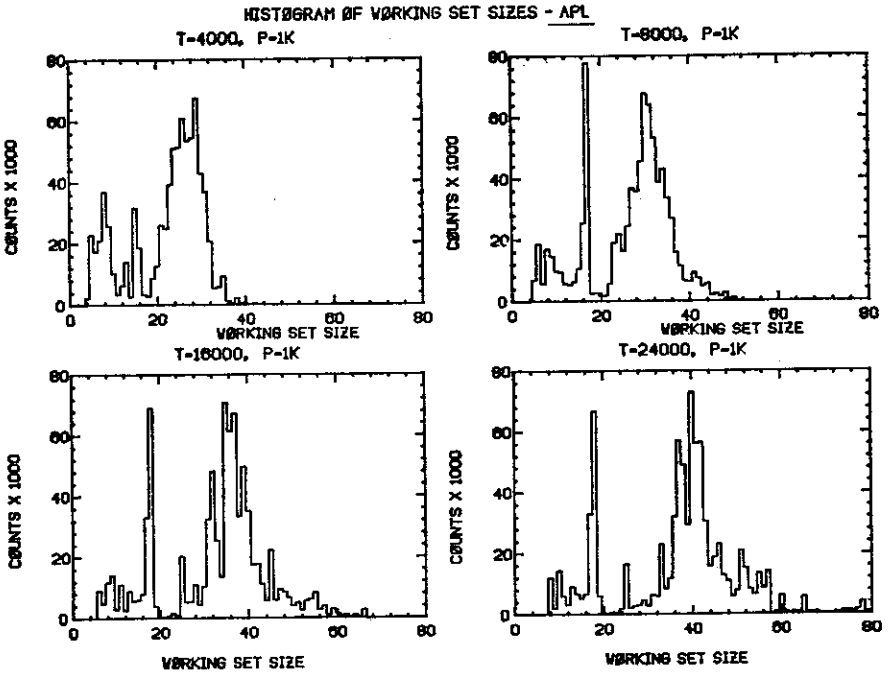


Fig. 1.13

WATFIV 1000569 References	Window Size →	4000			8000			16000			24000		
	Page Size	Mean	Var.	C.V.	Mean	Var.	C.V.	Mean	Var.	C.V.	Mean	Var.	C.V.
	512	50.1	136.4	0.23	58.0	112.7	0.18	65.3	82.7	0.14	69.5	73.2	0.12
	1k	33.2	50.4	0.21	36.7	38.5	0.17	40.3	27.8	0.13	42.5	27.1	0.12
	2k	22.5	19.7	0.20	24.1	14.7	0.16	26.1	9.4	0.12	27.2	7.6	0.10
	4k	14.8	8.4	0.20	16.2	6.4	0.16	17.3	3.9	0.11	17.8	2.7	0.09

WATEX 1000569 References	Window Size →	4000			8000			16000			24000		
	Page Size	Mean	Var.	C.V.	Mean	Var.	C.V.	Mean	Var.	C.V.	Mean	Var.	C.V.
	512	19.5	30.6	0.28	23.8	28.3	0.22	28.6	33.1	0.20	30.8	52.4	0.23
	1k	13.0	10.7	0.25	15.0	10.6	0.22	17.4	16.3	0.23	18.5	24.1	0.23
	2k	8.6	3.9	0.23	9.8	4.5	0.22	11.2	7.3	0.24	11.9	10.1	0.27
	4k	6.2	2.3	0.24	6.9	2.3	0.22	7.5	3.2	0.24	7.8	4.4	0.27

APL 1201152 References	Window Size →	4000			8000			16000			24000		
	Page Size	Mean	Var.	C.V.	Mean	Var.	C.V.	Mean	Var.	C.V.	Mean	Var.	C.V.
	512	31.0	143.7	0.37	39.7	211.8	0.37	48.7	353.9	0.38	57.2	479.4	0.39
	1k	22.5	74.4	0.38	28.0	97.6	0.35	33.1	144.4	0.36	37.6	185.9	0.36
	2k	16.7	41.3	0.38	18.8	43.4	0.35	21.7	60.9	0.36	24.5	77.3	0.36
	4k	11.9	15.5	0.33	13.3	17.8	0.31	14.9	24.2	0.39	15.6	29.3	0.35

TABLE 1.2

The effect of the window size parameter is not as obvious as page size. In Figures 1.5 and 1.9, for a given page size, the working set size sequence is plotted under different window sizes. A large window size can encompass several localities. On the other hand, a small window size can come short of containing a single locality in most cases. Therefore, the effect of window size on the working set size sequence is very dependent on the characteristics of each program. For the APL program, Figure 1.9, many high frequency components in the waveform of the working set size for small window sizes have been eliminated when a larger window size is used. This effect is especially notable in the beginning of the program. In Table 1.2, we note that the variance of the sampled points for a given page size does not always decrease as we increase the window size.

In Figure 1.10, the data working set size for FFT1 and FFT2 programs are shown. We note that FFT2 maintains a fairly uniform data working set size compared with FFT1. More localized data references in FFT2 may partly account for this behavior.

The histogram of working set sizes for a number of programs are plotted in Figures 1.11, 1.12, and 1.13, under different page and window sizes. The distribution of the sampled points are generally clustered around one or more points. For the APL program, we can see three peaks. The peaks demonstrate the distinctive locality regions in this program. The frequent locality transitions can also contribute to the creation of peaks in the histogram and may give spurious large locality sizes.

The distribution of working set sizes around the respective peaks can hardly be considered as being normal. In the programs considered here, the test of normality for the points which are sampled far apart

(to get fairly independent samples) are strongly rejected.

The serial dependence of successive working set size samples and the frequency domain features of this sequence can be evaluated by computing the estimated autocorrelation function and spectral density of this sequence.

Let $x(t), t=..,-1,0,1,2,3,...$ be the observations from a stationary process in the wide sense. The covariance function over this sequence with lag h is defined as:

$$\text{cov}[x(t),x(t+h)] = Ex(t)x(t+h) - Ex(t).Ex(t+h)$$

and $[x(t),x(t+h)] = c(h)$. An unbiased estimator for the covariance function is:

$$c(h) = c(-h) = \left(\frac{1}{N-h}\right) \sum_{t=1}^{n-h} [x(t)-\bar{x}]\,[x(t+h)-\bar{x}]$$

$$h=0,\ 1,2,3,\ldots,N-1$$

where N is the sample size and $\bar{x}$ is the sample mean. This function has a local maximum at the periods of the frequencies present in the data.

The sample autocovariance function, c(h), is normalized by the variance of the sequence to get the estimated <u>autocorrelation</u> function:

$$-1 \le R(h) = \frac{c(h)}{c(0)} \le 1$$

The autocorrelation function R(h) is a measure of serial direct or reverse dependence between the observations h units apart. In fact, under certain conditions, the test for $R(1) = 0$ is the most powerful test for the independence of a stationary time series [5].

For two different sequences of observations x(t) and y(t), $t=..-1,0,1,2,3..$, a similar function can be defined. Denote the cross-covariance function of x(t) and y(t) by:

$$c_{xy} = \left(\frac{1}{N-h}\right) \sum_{t=1}^{N-h} [x(t)-\bar{x}]\,[y(t+h)-\bar{y}]$$

$$h=0,1,2,3,\ldots.N-1$$

where $\bar{x}$ and $\bar{y}$ are respective sample means for x(t) and y(t). From this function, the estimated <u>crosscorrelation</u> coefficients are obtained by:

$$-1 \le R_{xy}(h) = \frac{c_{xy}(h)}{c_{xx}(0).c_{yy}(0)} \le 1$$

The crosscorrelation coefficients are measures of serial dependence between two different sets of observation. Some care should be taken when interpreting the values of this estimator. A large and spurious crosscovariance can be obtained if x(t) and y(t) are highly correlated within themselves [7].

The frequency domain formulation of a discrete time observation, x(t), by its spectral density f(w), is obtained by taking the Fourier transform of the covariance function:

$$f(w) = \frac{1}{2\pi} \sum_{h=-\infty}^{\infty} \text{cov}[x(t),x(t+h)]e^{-ihw}$$

An unstable estimator for f(w) is

$$p(w) = \frac{1}{2\pi} \sum_{h=-N}^{N} c(h)e^{-ihw} = \frac{1}{2\pi N} \sum_{t=1}^{N} [x(t)e^{-itw}]^2$$

where $w=2\pi N$. p(w) is called the raw periodgram. This is not a statistically consistent estimator and its variance around the true value does not decrease as the sample size N increases. To stabilize the estimator, one takes a local average over the frequencies surrounding the frequency

for which the power is desired. If a linear trend exists in data, it causes high power in the lowest frequency range of the computed power spectrum.

The estimate of autocorrelation coefficients and the spectral density for the sample points are computed by the package described in [11]. For estimating the spectral density, a cosine window of 10% taper is applied on data, and an equally weighted moving average is applied in the frequency domain.

Before we proceed to evaluate these functions on our sampled working sets, we perform a test for trend in the data. A test for the existence of trend in the sample points consists of comparing the centroid of observed values with the mid-point of the sum of observations [3]. The test statistics for the sampled working set sizes are:

$$u = \frac{(\frac{1}{n}) \Sigma W_i - 0.5 W_n}{W_n \sqrt{\frac{1}{12n}}}$$

where $W_i = \sum_{t=1}^{i} ws(t,T)$ and n is the number of observations.

When W_i's are independent random variables, u has a normal distribution with mean zero and variance one. In Table 1.3 for the WATFIV and APL programs, the u statistics for different number of samples of 100 differences apart show that the data from WATFIV is fairly consistent with the null hypothesis that there is no significant trend in this data, while data from the APL program clearly shows the existence of trend.

Program	No. of Samples	T Window	ws(T) Mean WS	u
WATFIV	10005	1000	27	- 0.327
	4985	1000	27	1.468
	2463	1000	28	0.592
WATFIV	10005	4000	49	- 2.919
	4985	4000	48	0.668
	2463	4000	50	0.671
APL	11084	1000	17	-23.898
	5513	1000	13	-16.425
	2697	1000	11	9.429
APL	11084	4000	30	-22.587
	5513	4000	23	-10.022
	2697	4000	21	9.300

TABLE 1.3 u-statistics for the trend in the string of working set sizes of WATFIV and APL programs

The autocorrelation function of working set size samples for WATFIV, WATEX and APL programs in Figures 1.14, 1.15 and 1.16 respectively, show a high serial correlation in data with small lags. In each figure, a unit lag is equal to 64 actual references. Thus, for instance, in the WATFIV program working set size samples of up to 6400 references show significant serial correlation.

The power spectrum of sampled working set sizes for three different programs are computed, and the results are shown in Figures 1.17 to 1.20. The data is sampled at every NS points as indicated in each figure. The horizontal axis is the frequency coordinate. The period, in terms of the number of references at each point in this axis, can be found by evalu-

sting NS*16384/N where N is read from the horizontal axis in each figure (16384 samples have been considered in each case).

We generally note that significant power lies in the low frequencies. In the case of the APL program, the trend in the data might have also contributed to the power in this region. The power drops sharply as we move toward higher frequencies.

In Figures 1.17 and 1.19, the power spectrum of working set size samples under different page sizes are plotted. We note that the general periodical patterns of the string are fairly invariant with respect to changes in page sizes. However, as we increase the page size, the power at the corresponding frequencies decrease and the high frequencies are smoothed out. Unlike the previous case, changing the window size results in different periodical patterns, as we see in Figures 1.18 and 1.20. We cannot make a strong statement on the effect of window size on these patterns because most of the observed changes are very much related to the characteristics of each program. In Figure 1.20 for the APL program, we notice that the drop in power in the higher frequencies is much more significant when we increase the window size. This shows that in this program many high frequency components are filtered out by increasing the window size.

As we have mentioned earlier, FFT1 and FFT2 programs use different arrangements for referring to their data. As we see in Figure 1.21, the power spectrum of sampled working set sizes, when all type of references are considered, are uneffected by the dissimilarity in the way these data are handled in each program. However, when we compute the power spectrum of only data references, the difference is more significant (Figure 1.22). The more localized data references in FFT2 is reflected in its estimates

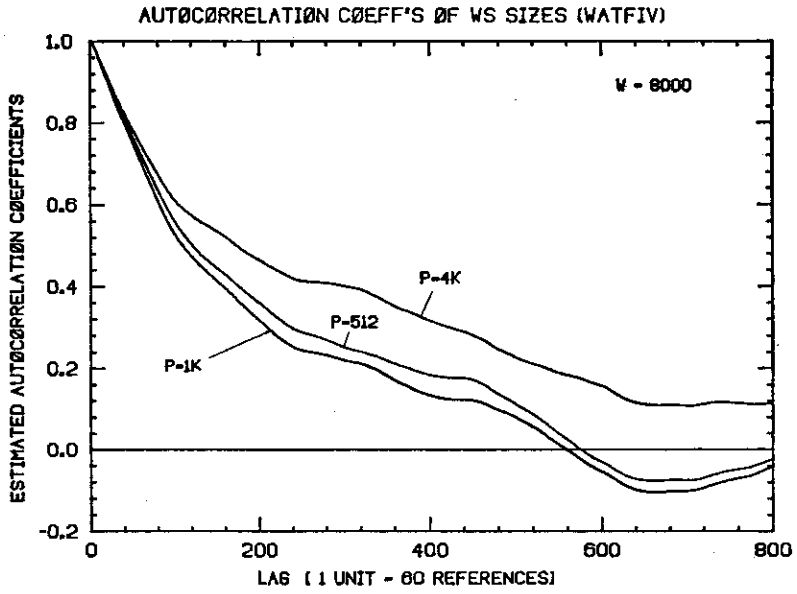


Fig. 1.14

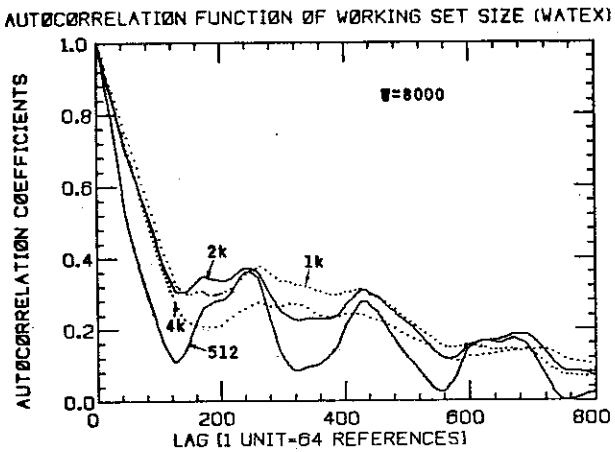


Fig. 1.15

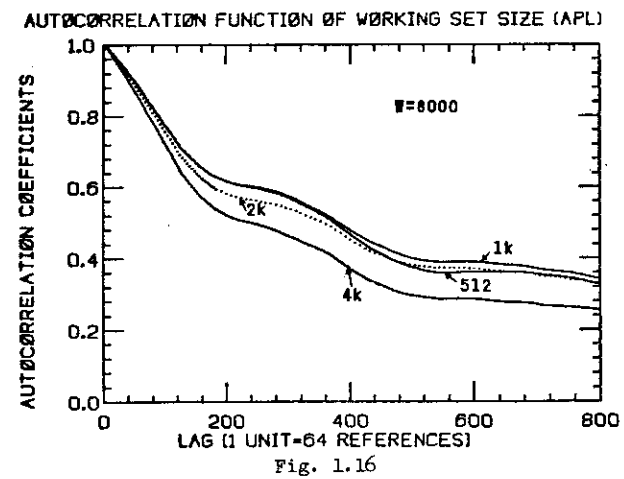


Fig. 1.16

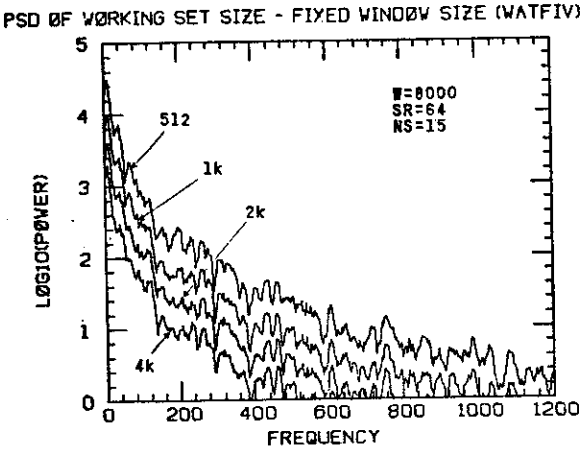


Fig. 1.17

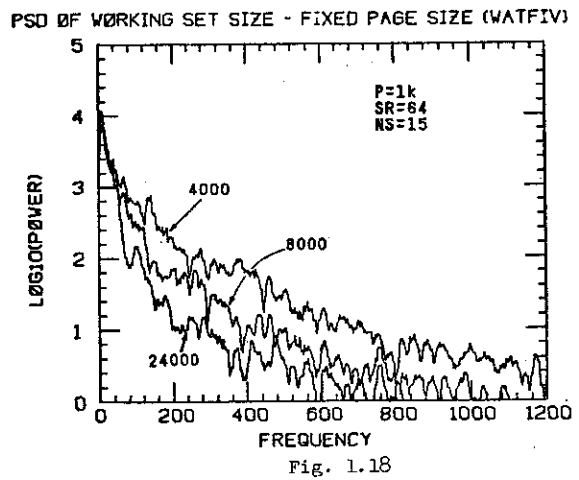


Fig. 1.18

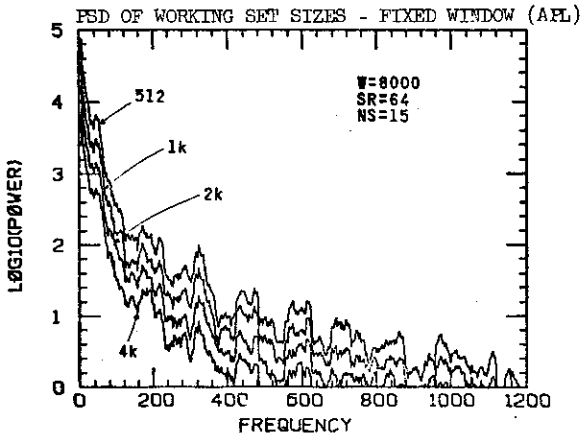


Fig. 1.19

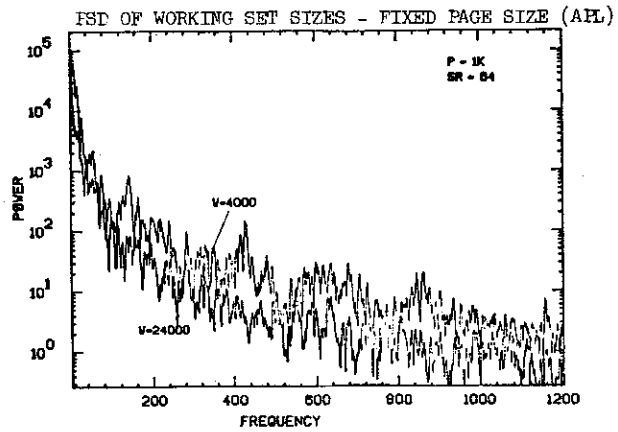


Fig. 1.20

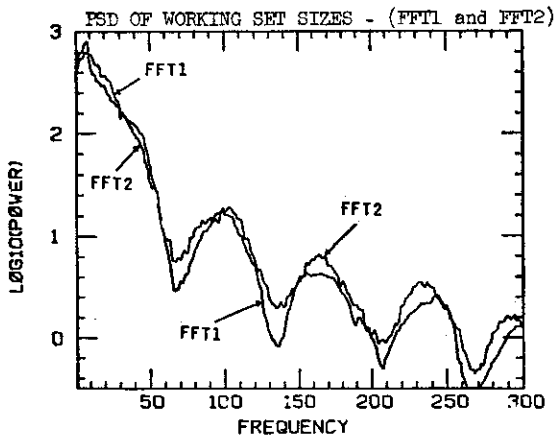


Fig. 1.21

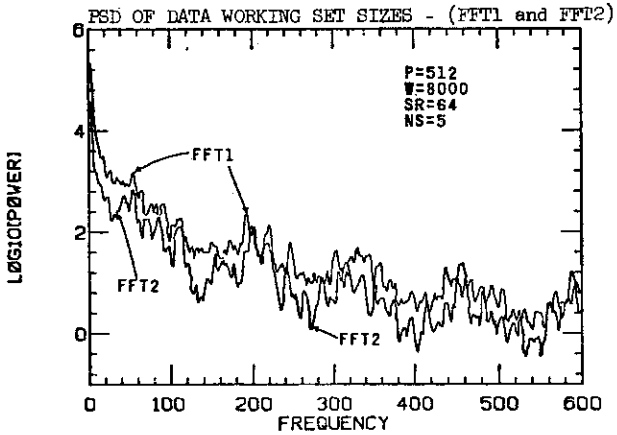


Fig. 1.22

power spectrum by obtaining lower power values compared with those of the FFT1 program.

1.3.2 Working Set Size Models

In this section, we consider a number of models for the working set size sequence and examine the capability of one model to capture the stochastic properties of the actual string, and particularly its correlation estimates.

A working set size, w_t, at time t, can be expressed in terms of w_{t-1} and another element δ_t, by the following relation:

$$w_t = w_{t-1} + \delta_t$$

where δ_t can assume integer values -1, 0 and +1. The boundary condition is $1 \leq w_t \leq \min(T,n)$ where n is the program size in pages, and T is the window size. Since T is usually greater than n, we can simply require that $1 \leq w_t \leq n$.

Now we can assume different probability structures on the value of δ_t. As the first step to simplify the model, we assume that the value of δ_t only depends on δ_{t-1}, and this holds for all values of t. In Tables 1.4 and 1.5, using the actual program traces of the WATFIV and WATEX programs, we have shown the frequency transition matrix of δ using the observed working set sizes. We note that δ has three states, -1, 0 and +1, and each entry, jk in the matrices, is the frequency of being at state j and going to state k. We can see that there is a great tendency that δ remains in or returns to state 0. We also note that we seldom see the transition from state +1 directly to -1 and vice versa.

Next, we assume that the value of δ_t is independent of δ_{t-1} and, therefore, we have a fixed probability description for δ as follows:

$$\text{pr}[\delta = -1] = p$$

$$\text{pr}[\delta = +1] = q$$

$$\text{pr}[\delta = 0] = 1-p-q$$

The model of working set size sequence becomes a finite state Markov chain with n states. In the boundaries, we can assume that $\text{pr}[\delta=+1] = 0$ and $\text{pr}[\delta=0] = 1-p$ when $w_t = n$ and $\text{pr}[\delta=-1] = 0$ and $\text{pr}[\delta=0] = 1-q$ when $w_t = 1$.

By adding up the columns of the matrices in Tables 1.4 and 1.5 and normalizing, we can get estimates for p and q from actual programs. In doing so, we note that from both WATFIV and WATEX programs, p and q are estimated very close together. Therefore, we can further simplify the model by assuming

$$\text{pr}[\delta=-1] = \frac{p}{2}$$

$$\text{pr}[\delta=+1] = \frac{p}{2}$$

$$\text{pr}[\delta=0] = 1-p$$

The covariance coefficients of the Markov chain can be found from the transition probabilities. Let $p_{ij}^{(m)}$ be the probability of going from state i to state j in exactly m steps. Assume that the chain starts from state 1 and $p_{ij}^{1} = p_{ij}$. Then

$$\text{cov}(w_t, w_{t+h}) = E\, w_t w_{t+n} - E\, w_t E_{t+h}$$

we have:

$$E\, w_t = \sum_j j\, p_{1j}^{(t)}$$

and

$$E(w_t w_{t+h}) = E[E(w_t w_{t+h}|w_t)]$$

$$= \sum_j E[w_t w_{t+h}|w_t = j]\ pr[w_t=j]$$

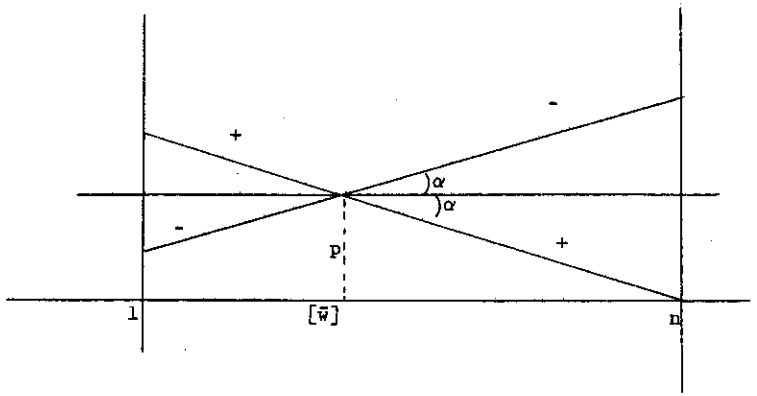
$$= \sum_j E[j\ w_{t+h}|w_t=j]\ pr[w_t=j]$$

$$= \sum_{jk} jk\ p_{jk}^{(h)}\ p_{1j}^{(t)}$$

Therefore:

$$cov(w_t, w_{t+h}) = \sum_{jk} jk\ p_{jk}^{(h)}\ p_{1j}^{(t)} - \sum_j j\ p_{ij}^{(t)} \sum_j j\ p_{1j}^{(t+h)}$$

Unfortunately, this model does not necessarily give an average working set size which is equal or close to the average working set size of the actual program from which the transition probabilities were obtained. Another model which partially alleviates this problem is to parameterize a Markov chain, which has some degree of central tendency, to a state which is equal to the observed working set size $\bar{w}$,[6]. One approach is to formulate the transition probabilities such that the central attraction linearly increases as the function of the distance of the chain from the state $[\bar{w}]$. A possible scheme is shown in the following sketch:



The probability transition at each state is, therefore, a function of the state and can be found from the following expressions:

$$pr[\delta=+1] = p - \frac{p}{n-\bar{w}}\ (w-\bar{w})$$

$$pr[\delta=-1] = p + \frac{p}{n-\bar{w}}\ (w-\bar{w})$$

$$pr[\delta=0] = 1-2p.$$

As before, we can estimate p from the actual observations (Tables 1.4 and 1.5).

WATFIV: T = 1000 $\bar{w}$ = 27

	-1	0	+1
-1	602	12499	130
0	12488	949293	12309
+1	142	12297	808
	13232	974089	13247

T = 4000 $\bar{w}$ = 50

	-1	0	+1
-1	103	3180	14
0	3181	987591	3150
+1	14	3150	185
	3298	993921	3349

T = 8000 $\bar{w}$ = 57

	-1	0	+1	
-1	48	1301	2	
0	1298	995142	1354	
+1	6	1350	67	
	1352	997793	1423	
	0.0013	0.9973	0.0014	p=0.0013

TABLE 1.4 The one-step transition frequencies of observed successive working set sizes of the WATFIV program

WATEX:

T = 1000 $\bar{w}$ = 18

	-1	0	+1
-1	241	4373	2
0	4374	1090491	4377
+1	2	4377	247
	4617	1099241	4626

T = 4000 $\bar{w}$ = 19

	-1	0	+1
-1	44	1348	0
0	1348	1102974	1362
+1	1	1361	46
	1393	1105683	1408

T = 8000 $\bar{w}$ = 23

	-1	0	+1	
-1	33	961	0	
0	961	1104527	990	
+1	1	989	22	
	995	1106477	1012	
	0.0009	0.9982	0.0009	p = 0.0009

TABLE 1.5 The one-step transition frequencies of observed successive working set sizes of the WATEX program

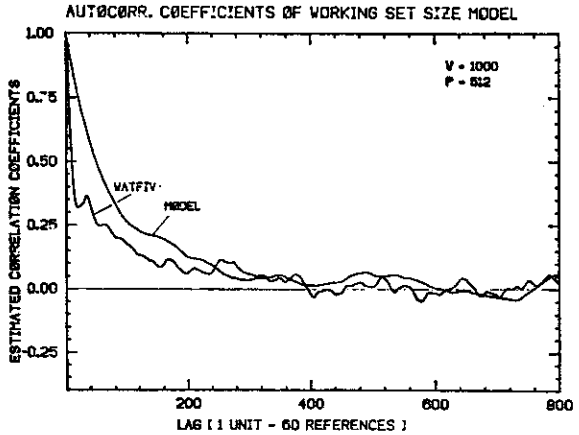


Fig. 1.23

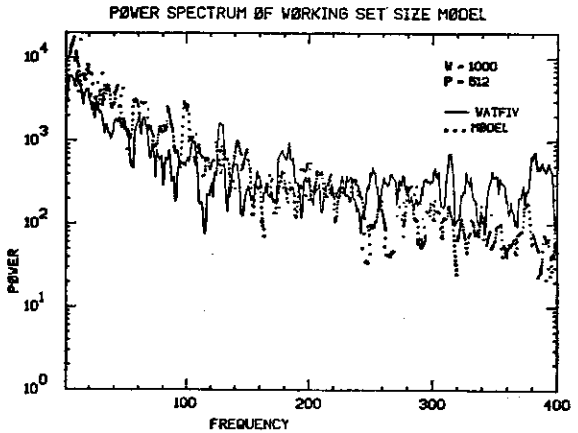


Fig. 1.24

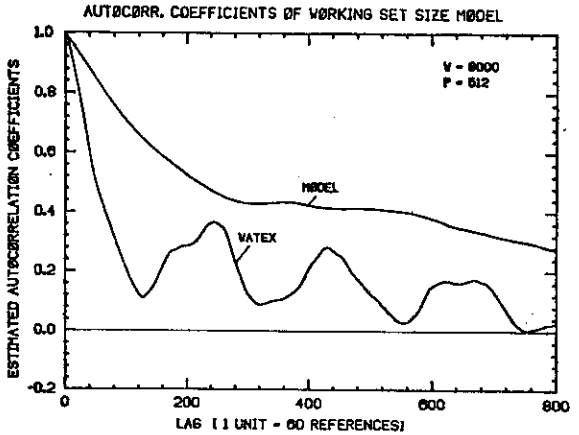


Fig. 1.25

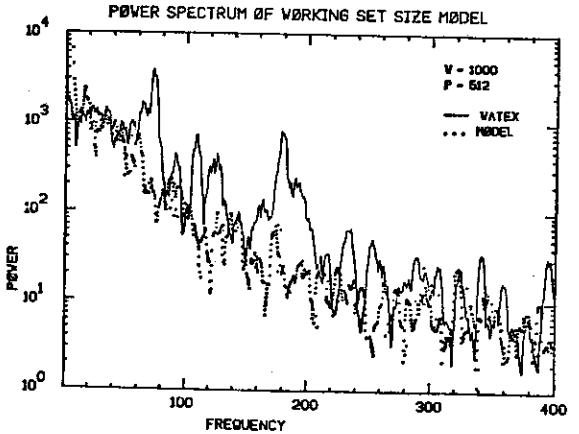


Fig. 1.26

In Figures 1.23 and 1.25, we compare the estimated correlation and estimated power spectrum of the working set size string of the WATFIV and WATEX programs and their respective models, based on the central tendency assumption. We note that the model is fairly successful in capturing the amount of serial correlation and the general frequency domain behavior in the case of WATFIV programs. However, it overestimates the correlation coefficients in the WATEX program. Other tests suggest that the behavior of the model is very sensitive to the choice of p. Considering the small value of p as estimated by the actual programs, this sensitivity can be very undesirable. Therefore, the usefulness of this model may be limited to the cases where the behavior of the model can be validated by the actual observations for any chosen value of p.

1.3.3 Predicting Working Set Sizes

In a multiprogramming system where programs compete for main memory resources, a way of sharing memory among the processes is to allocate memory space according to each job's working set requirements. The motivation behind this is that when the working set of a program is loaded into the memory, the program can efficiently run in this environment without putting heavy demands on the paging facility and other resources of the computer. This approach to sharing memory is used by a working set dispatcher and its variations. Successful implementations of these dispatchers require a good estimation of the working set size demand of the programs in the near future. This estimation is most possible by observing the past behavior of the programs. In this subsection, we will examine a number of approaches to estimating the working set size of a single program, and will present the results of case studies concerning the success of each method.

Earlier we have seen that the working set sequence of programs generally exhibit a high serial correlation in short range intervals. Therefore, in any estimation method which predicts the future working set size of a program based on its past occurrences, we would like to put the highest weight on the most recent observations.

A convenient point to stop the processing and estimate the next working set size value is after the elapse of an interval which is equal to the chosen window size parameter. These points are the inspection points where a sample of the working set size sequence is taken. The choice of a sample interval equal to the window size also facilitates the implementation of a kind of WS dispatcher. In the next chapter, we will see that a high performance modified WS dispatcher, which measures the working set size at the same points, can be implemented with the available facilities in many recent computers.

We have used three algorithms to predict the working set sizes for the next execution interval. Let w_i, i=1,2,3,... be a sequence of working set sizes generated by a program, and which are sampled at the inspection points which are T references apart. T is also equal to the window size parameter. Denote by $\hat{w}_i$ the predicted value of the true working set size at point i. We want to estimate $\hat{w}_{i+1}$ based on the past observations $w_i, w_{i-1}, w_{i-2}, \ldots, w_1$. We are not assuming a particular model for the working set sizes and, therefore, the following procedures are chosen purely empirically.

a) Exponential smoothing algorithm:

Let r be the parameter of the algorithm. The working set set size w_{i+1} is estimated by:

$$\hat{w}_{i+1} = r\, w_i + (1-r)\, \hat{w}_i \qquad 0 < r \leq 1$$

$$= r \sum_{k=0}^{i-1} (1-r)^k\, w_{i-k} + (1-r)^i\, \hat{w}_i$$

This predictor places the highest weight on the most recent observation and the weight given to the other observations decreases geometrically with age. When r is close to one, the value $\hat{w}_{i+1}$ is heavily influenced by w_i. Therefore, in this case, the prediction mechanism becomes very responsive to the immediate changes in the sequence. As r approaches to zero, the time required by the estimated value to respond to the changes in the past observations, increases.

b) First order auto-regressive algorithm:

Let the working set size difference at time, i, be defined by

$$dw_i = w_i - w_{i-1} .$$

For the actual programs, the observed average value of the differences is close to zero. For instance, for WATEX and APL programs with window size equal to 4000 references, the respective average differences are 0.032 and -0.043. Thus, we can estimate $\hat{w}_{i+1}$ using the last difference value by:

$$\hat{w}_{i+1} = r\, w_i + (r-1)\, \hat{w}_i + dw_i$$

where again $0 < r \leq 1$.

c) Moving average algorithm:

The estimate $\hat{w}_{i+1}$ is obtained by an equal weighted average of the past N observations. N is the parameter of the algorithm. Thus,

$$\hat{w}_{i+1} = \frac{1}{N} \sum_{k=i-N+1}^{i} w_i = \hat{w}_i + \frac{(w_i - w_{i-N})}{N}$$

The rate of response of the estimated value is controlled by the choice of the parameter N. It also determines the amount of past information that must be retained in order to estimate the future working set size.

The predictive power of each algorithm is measured by the computed mean square error, MSE, and the relative error, E. Let k be the number of observations. Define

$$MSE = \frac{1}{k} \sum_{i=1}^{k} [w_i - \hat{w}_i]^2$$

and

$$E = \frac{(MSE)^{1/2}}{\bar{w}}$$

where $\bar{w}$ is the observed average working set size.

The performance of each algorithm tested on actual programs is shown in Tables 1.6, 1.7 and 1.8, and also in Figures 1.27 and 1.28.

Algorithm (a) is the most successful among the other algorithms in predicting the working set sizes. The best relative error for this algorithm ranges between 17% to 28% in the experiments. The algorithm does not seem to be very sensitive to the value of the parameter. However, in most cases r = 0.5 is a good choice and gives an estimation error

comparable with the optimum performance of the algorithm in each case.

The inclusion of the difference quantity in algorithm (b) decreases the accuracy of the prediction compared with the first algorithm. The best result with algorithm (b) is obtained when the parameter of the algorithm is kept small. The best relative error in our tests with this algorithm ranges between 19% to 42%.

The performance of algorithm (c) is very close to algorithm (a). The best prediction relative error under this algorithm ranges between 18% to 27%. In our experiments, a good choice of N varies between 2 to 4. The change in the relative error is fairly insignificant within this range of the parameter.

Since the fluctuation of the working set size string usually reduces as the window size increases, the relative prediction error improves for large window sizes. However, we can see instances in which this argument does not hold. For example, in the case of APL programs, the relative error with T = 8000 is higher than the relative error with T = 4000 under all three algorithms. This indicates the inappropriate choice of a window size which frequently goes across the boundaries of the program localities.

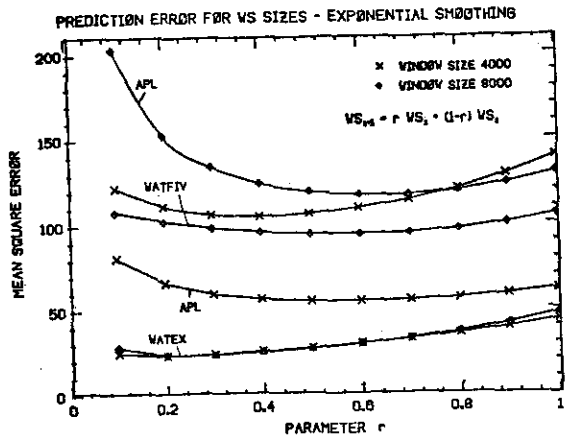


Fig. 1.27

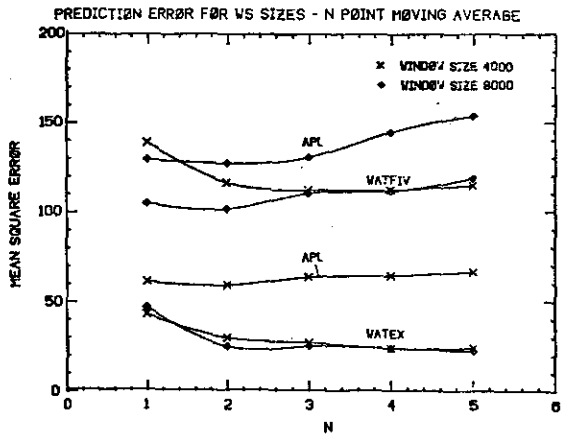


Fig 1.28

Window size=4000 Average working set size=49

exp. smoothing (a)			auto-regressive (b)			moving average (c)			
r	MSE	E	r	MSE	E	N	MSE	E	
0.1	122.9	0.23	0.1	154.6	0.25	1	139.0	0.24	
0.2	111.6	0.22	0.2	169.2	0.26	2	116.3	0.22	
0.3	106.5	0.21	0.3	185.8	0.28	3	112.2	0.22	
0.4	105.2	0.21	0.4	204.5	0.29	4	112.2	0.22	
0.5	106.3	0.21	0.5	225.8	0.31	5	114.8	0.22	
0.6	109.5	0.21	0.6	254.0	0.33				
0.7	114.2	0.22	0.7	277.9	0.34				
0.8	120.6	0.22	0.8	310.4	0.36				
0.9	128.7	0.23	0.9	348.6	0.38				
1.0	139.0	0.24	1.0	394.5	0.40				
0.4	105.2	0.21	0.1	154.6	0.25	3	112.2	0.22	Min

Window size=8000 Average working set size=57

exp. smoothing (a)			auto-regressive (b)			moving average (c)			
r	MSE	E	r	MSE	E	N	MSE	E	
0.1	108.4	0.18	0.1	118.8	0.19	1	105.2	0.18	
0.2	102.7	0.18	0.2	127.7	0.19	2	101.5	0.18	
0.3	98.6	0.17	0.3	138.2	0.21	3	110.1	0.18	
0.4	95.9	0.17	0.4	149.9	0.21	4	111.5	0.19	
0.5	94.4	0.17	0.5	163.0	0.22	5	118.9	0.19	
0.6	94.1	0.17	0.6	177.9	0.23				
0.7	95.0	0.17	0.7	195.0	0.24				
0.8	96.9	0.17	0.8	215.1	0.26				
0.9	100.2	0.18	0.9	239.1	0.27				
1.0	105.2	0.18	1.0	268.4	0.29				
0.6	94.1	0.17	0.1	118.8	0.19	2	101.5	0.18	Min

TABLE 1.6 The prediction errors in working set size estimation for WATFIV program

Window size=4000 Average working set size=18

exp. smoothing (a)			auto-regressive (b)			moving average (c)			
r	MSE	E	r	MSE	E	N	MSE	E	
0.1	25.6	0.28	0.1	57.7	0.42	1	43.2	0.36	
0.2	23.7	0.27	0.2	58.5	0.42	2	29.9	0.30	
0.3	24.4	0.27	0.3	62.9	0.44	3	27.2	0.29	
0.4	25.7	0.28	0.4	68.8	0.46	4	23.8	0.27	
0.5	27.5	0.29	0.5	75.9	0.48	5	24.3	0.27	
0.6	29.7	0.30	0.6	84.4	0.51				
0.7	32.2	0.31	0.7	94.4	0.54				
0.8	35.3	0.33	0.8	106.4	0.57				
0.9	38.9	0.35	0.9	120.7	0.61				
1.0	43.2	0.36	1.0	138.4	0.65				
0.2	23.7	0.27	0.1	57.7	0.42	4	23.8	0.27	Min

Window size=8000 Average working set size=23

exp. smoothing (a)			auto-regressive (b)			moving average (c)			
r	MSE	E	r	MSE	E	N	MSE	E	
0.1	28.9	0.23	0.1	71.0	0.37	1	47.2	0.30	
0.2	24.2	0.21	0.2	67.5	0.36	2	25.0	0.22	
0.3	24.0	0.21	0.3	70.6	0.37	3	25.2	0.22	
0.4	25.1	0.22	0.4	76.3	0.38	4	23.9	0.21	
0.5	27.0	0.23	0.5	83.9	0.40	5	22.6	0.20	
0.6	29.5	0.24	0.6	93.4	0.42				
0.7	32.6	0.25	0.7	105.1	0.45				
0.8	36.4	0.26	0.8	119.1	0.47				
0.9	41.2	0.28	0.9	137.4	0.51				
1.0	47.2	0.30	1.0	159.9	0.55				
0.3	24.0	0.21	0.2	67.5	0.36	5	22.6	0.20	Min

TABLE 1.7 The prediction errors in working set size estimation for WATEX program

Window size=4000 Average working set size=30

exp. smoothing (a)			auto-regressive (b)			moving average (c)		
r	MSE	E	r	MSE	E	N	MSE	E
0.1	81.6	0.30	0.1	71.9	0.28	1	61.5	0.26
0.2	66.5	0.27	0.2	76.0	0.29	2	59.2	0.26
0.3	59.9	0.26	0.3	81.9	0.30	3	63.6	0.26
0.4	56.6	0.25	0.4	88.6	0.31	4	64.4	0.27
0.5	55.0	0.25	0.5	96.8	0.33	5	66.7	0.27
0.6	54.7	0.25	0.6	105.8	0.34			
0.7	55.2	0.25	0.7	116.3	0.36			
0.8	56.4	0.25	0.8	128.4	0.38			
0.9	58.5	0.25	0.9	142.7	0.40			
1.0	61.5	0.26	1.0	160.2	0.42			
0.6	54.7	0.25	0.1	71.9	0.28	2	59.2	0.26

Window size=8000 Average working set size=38

exp. smoothing (a)			auto-regressive (b)			moving average (c)		
r	MSE	E	r	MSE	E	N	MSE	E
0.1	203.7	0.37	0.1	229.5	0.40	1	129.6	0.30
0.2	153.3	0.33	0.2	206.5	0.38	2	127.1	0.27
0.3	134.5	0.30	0.3	207.6	0.38	3	130.4	0.30
0.4	124.6	0.29	0.4	215.9	0.39	4	144.4	0.32
0.5	119.4	0.29	0.5	228.6	0.40	5	153.9	0.33
0.6	117.2	0.28	0.6	244.9	0.41			
0.7	117.5	0.28	0.7	264.8	0.43			
0.8	119.7	0.29	0.8	288.5	0.45			
0.9	123.7	0.29	0.9	316.6	0.47			
1.0	129.6	0.30	1.0	349.7	0.49			
0.6	117.2	0.28	0.2	206.5	0.38	2	127.1	0.27

TABLE 1.8 The prediction errors of working set size estimation for APL program

1.3.4 Stack Distance String

For a page reference string, $r_1, r_2, r_3, \ldots, r_t, \ldots$, the stack distance at time t, d_t, is the number of distinct pages addressed since the last reference to page r_t. When r_t is referenced for the first time, then the distance can be set to some special value, say, infinity. Then the stack distance string (or simply distance string) is the sequence $d_1, d_2, d_3, \ldots, d_t, \ldots$ associated with the string $r_1, r_2, r_3, \ldots, r_t, \ldots$

The distance string provides most of the information in the reference string and the reference string, up to the identity of pages, can be reconstructed from this sequence.

The distance string of actual programs tend to exhibit some common properties which are readily verified. A majority of distances have values close to one. We can see this point by examining the histogram of stack distance string of a number of programs in Figures 1.29, 1.30 and 1.31. A large number of distances with values one and two indicate that many successive reference addresses are in the same page, and there are frequent occurrences of reference patterns like instruction-data-instruction-data-.... where instruction and the respective data are in different pages. Although the distribution of distances are heavily biased toward very small values, the more interesting information in the level of memory management problems lies in the region of relatively higher distances.

The estimated autocorrelation function of distance string, Figures 1.32 and 1.33, show a sharp drop to values close to zero for lags greater than zero. These estimates are computed by taking successive 16384 distances from different parts of the programs. This behavior suggests that there is no correlation of practical significance in the successive stack distances generated by these programs.

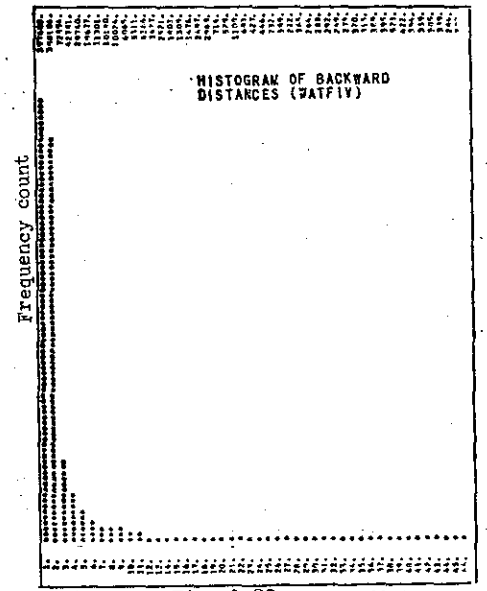


Fig. 1.29

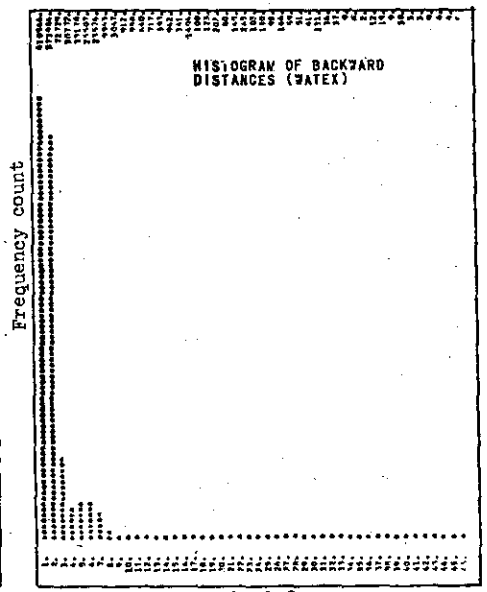


Fig 1.30

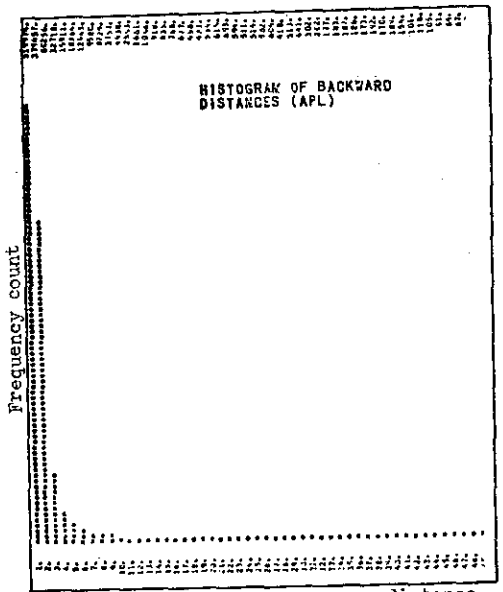


Fig. 1.31

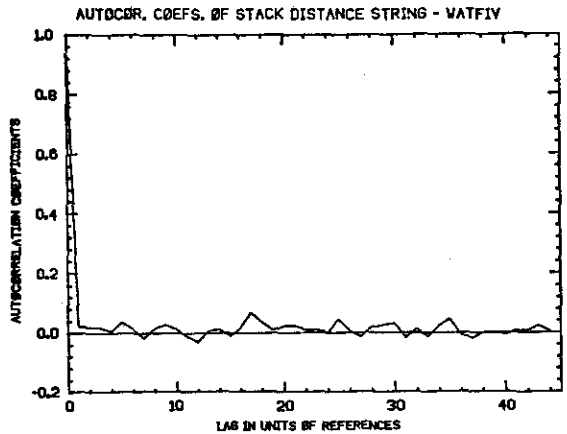


Fig. 1.32

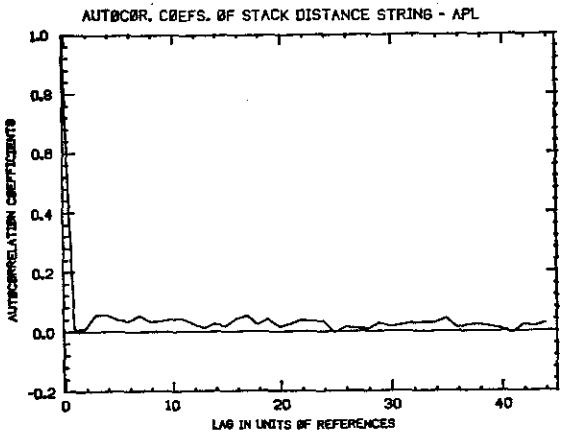


Fig. 1.33

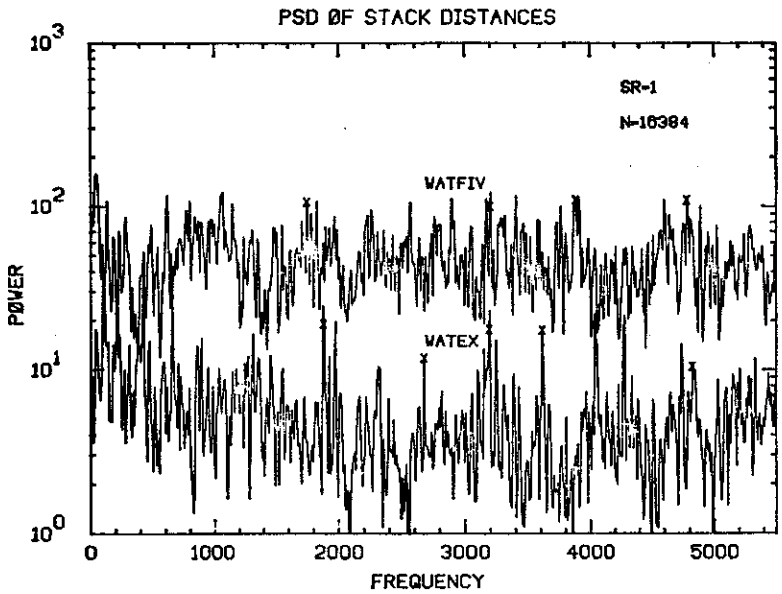


Fig. 1.34

When we plot the power spectrum of stack distances, Figure 1.34, we obtain a fairly flat spectrum which supports the previous observation of the existence of a degree of randomness in the generated distances. In a closer inspection of the estimated power spectrum, we can, however, see that in both programs in the high frequency range, relatively significant power lie on periods less than ten instructions. Each period determines the more frequently observed time it takes for the access mechanism of each program to address the same page in two successive references.

Because of its weak stochastic properties, the exact modeling of this string does not seem to justify the effort. However, the observed distribution of the stack distances can be a basis for strong program models (Chapter 3).

1.4 CONCLUSION

In this chapter, we studied the properties of working set size and stack distance strings of some actual programs.

We saw the working set size of the programs vary significantly during the execution of a program. However, there are high correlation in the neighboring samples and the extent of this serial dependence can be measured by computing the estimated autocorrelation coefficients. It is inaccurate to assume that the distribution of the independent working set size samples fit a normal distribution. The change of page size parameter retains most of the periodical patterns of working set

size waveform, and as we increase the page size, we can see a gradual reduction of the power in corresponding frequencies. The window size parameter can drastically change the periodical characteristics of this string. A notable effect, as we increase the window size, is the damping of the high frequencies in the working set size string as it is observed in some programs. The change in the data reference pattern in the high level language formulation of an algorithm is reflected in the actual machine references only after the other type of the references are filtered out. In other words, the effect on the overall reference pattern is very small.

The working set size models were considered and their ability to capture the degree of the serial dependence of the string were examined. The Markov model with reflecting barriers and central attraction can come close to mimicing the working set sizes in this respect; however, the accuracy of the model is very sensitive to the choice of the values for the parameters.

The algorithms for predicting the working set requirement of a program in the near future, based on the past observations, were examined. The exponential smoothing algorithm seems to give the best result. The accuracy of the prediction when the window size changes, were discussed.

The analysis of the generated stack distances shows that the distribution of the distances is heavily biased toward small values. There is no significant serial correlation between the successive distances. A measure of the page locality transition periods were obtained by inspecting the observed dominant high frequencies in the data.

1.5 BIBLIOGRAPHY

1. Coffman, E.G.Jr., Denning, P.J., "Operating system theory," Prentice-Hall, Inc., New Jersey (1973).
2. Coffman, E.G. Jr., Ryan, T.A. Jr., "A study of storage partitioning using the mathematical model of locality," CACM 15, 3, (March 1972), pp 185-190.
3. Cox, D.R., Lewis, P.A.W., "The statistical analysis of series of events," Mathuen and Co., Ltd., London (1966).
4. Denning, P.J., "The working set model for program behavior," CACM 11, 5, (May 1968).
5. Feller, W., "An introduction to probability theory and its application," Vol. II, John Wiley and Sons, Inc., N.Y. (1966).
6. Ghanem, M.Z., and Kobayashi, H., "A parametric representation of program behavior in a virtual memory system," IBM Research Report RC-4560 (October 1973).
7. Jenkin, G.M. and Watts, D.G., "Spectral analysis and its applications," Holden-Day, Inc., San Francisco (1958).
8. Kendall, M.G., "Time Series", Griffin and Co., Ltd., London (1973).
9. Singleton, R.C., "On computing the Fast Fourier Transform," CACM 10, 10 (October 1967), pp 647-654.
10. Singleton, R.C., Algorithm 338, "Algol procedure for the Fast Fourier Transform," CACM 11, 11 (November 1968).
11. Webb, C., "Practical use of the Fast Fourier Transform (FFT) in time series analysis," Applied Research Lab., University of Texas at Austin, ARL-TR-70-22 (June 1970).

CHAPTER 2

COMPARATIVE STUDY OF PRACTICAL PAGING ALGORITHMS

2.1 INTRODUCTION

The idea of virtual memory computers have received great acceptance since the concept of a one-level memory store was introduced and effectively implemented on the ICT Atlas Computer [10]. The architecture of many computers has been changed accordingly, to provide a suitable host for the implementation of the mechanisms and algorithms regarding the address interpretation and management of the information in the storage hierarchies (e.g., IBM 360/85, Burroughs B6500, GE 645). Because of this development, the need for a careful study of the problems relating to the efficiency of the operations in virtual memory computers soon emerged. The study of paging algorithms has been one of the major concerns in designing the policies for the dynamic organization of program pages [1, 3, 6, 13, 15]. The overhead involved in the paging operations, due to the processing of the paging algorithm and the time to place and replace a page in the memory hierarchy, necessitates the development of the efficient paging algorithms which result in the speedy execution of the jobs and the least amount of paging traffic.

In all levels of current and newly emerging memory technologies, we can see the need for data management techniques similar to paging operations. One example is the control of data movement between very fast (cache type) memories and the slower main memory devices. This technology seems destined to stay around, especially if we look at the ever-increasing speed of processors, while the speed of economical memories increases but still lags behind. The exchange of data blocks between direct access de-

vices and mass storage facilities is another example which shows the need for the scheduling of data between relatively slow memory devices.

Since the performance evaluation of the paging algorithms requires processing of large amounts of data, valuable efforts have been directed in developing efficient evaluation techniques [2, 12, 13]. In this chapter, some of the important techniques will be discussed.

The paging algorithms which we are going to consider in this paper can be categorized into three different groups, based on their complexity and the implementation costs. Some algorithms require large amounts of processing and bookkeeping. Using these algorithms can be very costly in terms of the space they need to keep the data, and in terms of the execution overhead.

If we specify the performance of a paging algorithm with the number of page faults it causes, then we can see that not all of the expensive algorithms yield a performance level which can justify their costs. On the other hand, we will show that some simple and efficient algorithms give performances which are competitive with more elaborate algorithms.

We will consider three groups of algorithms. In the first group, we put some well known algorithms such as LRU, WS, CLIMB and MIN. The amount of storage required for bookkeeping and the processing cost of these algorithms are set to be in the middle of our comparison scale.

The algorithms in the second group require the most amount of processing compared to their counterparts in the first group. Members of this group include PFF (counterpart of LRU), WS with variable window size (counterpart of WS). We will attempt to see whether the extra work involved in the execution of these algorithms is justified in the final result.

The more practical algorithms fall into the third group. All the algorithms in this group require the least amount of processing compared to their counterparts in the other groups. They can use the hardware facilities available in most virtual memory computers very effectively to manage their bookkeeping requirements. The use of these algorithms becomes more advantageous by keeping down the overhead when the central processor is slow, or there is a need to have a small and efficient memory manager as part of a supervisor. Within this group, we discuss FIFO, CLOCK (two versions) and Modified WS (MWS).

Inclusion of the program model into the study of the paging algorithms is important because the use of the model can broaden the scope of the study. We will use the independent reference model which is tractable for analysis and simulation.

In this chapter, we will start by giving the description and properties of a number of paging algorithms. Whenever an important evaluation technique is available, it will be presented. In some cases, the analytical results of the independent reference model will be given too. This chapter concludes by giving the performance result of most of these algorithms on actual program traces, and independent reference models constructed, based on actual programs.

2.2 THE VIRTUAL MEMORY COMPUTER

Consider a two-level memory system where at each level the memory is partitioned into equal sized consecutive words, generally called page frames. The transfer of the information between the two levels is done in a unit of a page. The address space of the program running in this hierarchy is also partitioned into pages. Program pages are initially mapped to the page frames in the secondary level. A program, with some

of its pages loaded into the first level memory, issues references to its logical address space which, in turn, is interpreted and mapped to actual addresses in the primary level. When the program addresses a page which is not in the main memory, a page fault occurs. At this time, the supervisor halts the execution of the program and starts loading in the requested page from the secondary storage. If the allowable space of the program (memory capacity or buffer capacity) in the first level is exhausted, one of the pages of the program must be copied back (if it had been modified) to the secondary level to make room for the incoming page. The decision to select a page to be pushed out of the memory is taken by a paging or replacement algorithm. When the missing page is brought into the main memory, the delayed program can resume its execution. If the replacement selection is always done from the pages of the program which has caused the page fault, the algorithm is said to organize the program pages on a local basis. In a multiprogramming system where more than one program may be active and they compete for main memory resources, a paging algorithm is said to work on a global basis if it considers all the pages of the active programs as a common pool and, at the time of a page fault, it may replace a page from a program which has not necessarily generated the fault.

The allocated space of a program in the main memory can either remain fixed during the entire execution of the program (fixed memory size) or it can vary according to the requirements of the program (variable memory size).

The objective of the paging algorithm is to organize the program pages in such a way that the likelihood that a page is found in the main memory (or first level memory) when it is referenced, is increased. The

common criterion to compare the performance of paging algorithms is the number of page faults that a program experiences during its execution for a given memory capacity. Therefore, for a fixed memory size, m, and under the replacement algorithm, A, define the page fault rate or miss ratio, $F_A(m)$, as the ratio of the number of page faults over the total number of references. Denote the success function by $1 - F_A(m)$.

For the variable size memory algorithms, we substitute for m the average memory size over the total execution time. The average memory size is obtained by summing up the memory sizes at the time of each reference and dividing the total by the number of references. This criterion is called virtual space-time product versus the page fault rate. Another alternative criterion to compare paging algorithms is the real space-time product versus the page fault rate. This criterion takes into account the cost of page transfer time in terms of the main memory space occupied and idle during the transfer. This can be summarized as the curve of

$$\sum_{i=1}^{k} [I(i)m_i + I'(i)*C*m_i]/(k+F*C)$$

versus page fault, where m_i is the memory size at the time of reference i, C is the time it takes to read a page in terms of the number of references, k is the total number of references, and I(i) and I'(i) are indicator functions such that

$$I(i) = \begin{cases} 0 & \text{if reference i causes page fault} \\ 1 & \text{otherwise} \end{cases}$$

and

$$I'(i) = \begin{cases} 0 & \text{if } I(i) = 1 \\ 1 & \text{if } I(i) = 0 \end{cases}$$

Some authors use a virtual space-time product versus page fault rate

which is similar to the earlier one except the memory sizes are normalized to the number of actual program references k.

We will use the first representation because it does not involve the parameter, C , which can assume different values in different components.

2.3 INDEPENDENT REFERENCE MODEL

For some of the paging algorithms in this chapter, we will give some analytical results for the performance, and the average amount of main memory used by a program which generates reference sequences according to the independent reference model. We will extensively study this model in the next chapter. For our purpose here, it suffices to mention that this model can be a very good predictor of the performance of actual programs under many replacement algorithms [Ch. 3].

In the independent reference model, the sequence of page references $r_1, r_2, \ldots, r_k$ are iid random numbers with probability density function $[p_1, p_2, \ldots, p_n]$ where,

$$Pr[r_t = i] = p_i$$

with n being equal to the total number of model program pages. Without loss of generality, we label the page names so that $p_1 \geq p_2 \geq \ldots \geq p_n$.

2.4 DEMAND PAGING ALGORITHMS

In the remaining part of this chapter, we will do a comparative study of a variety of paging algorithms and measure their performances and the relative amount of processing and bookkeeping which is involved in each of them. The most important evaluation techniques will be mentioned in each case.

The following algorithms are considered:

RR (random replacement)
FIFO (first in first out)

LRU (least recently used)
CLIMB
CLOCK
MIN
WS (working set)
WSVT (working set with variable window size)
PFF (page fault frequency)
MWS (modified working set)
WMIN (variable memory size MIN)

For the following discussion, we denote by n the total number of program pages and by m the main memory size ($m \leq n$). All algorithms are demand paging; i.e., a page will be brought into the memory when it is requested.

2.4.1 RR (Random Replacement)

At the time of a page fault, this algorithm will replace, at random, one of the pages from the content of the main memory. Therefore, it is not required to keep any information about the past paging activity of the program. We can expect poor performance from this algorithm because it replaces the pages indiscriminately and, therefore, it might remove a page which is going to be referenced again in the near future, or else it will retain an idle page for an unnecessarily long time.

For the independent reference model, the long run page fault rate is equal to (see Appendix 2.7 in this chapter):

$$F_{RR}(m) = G^{-1} \cdot \sum_{s \in Q} \prod_{j \in s} p_j \sum_{i \notin s} p_i \qquad \text{(IRM)}$$

where

$$G = \sum_{s \in Q} \prod_{i \in s} p_i$$

and Q is the set of all combination of m numbers out of n.

2.4.2 FIFO (First In First Out)

The program pages are ordered based on the time of their first arrival into the main memory. At the time of a page fault, the page which was first referenced is removed.

This algorithm has some unusual behavior. The function of page fault rate with respect to increasing values of memory sizes is not always non-increasing. It is also known that the initial buffer content (the set of pages which are initially loaded into the main memory prior to the execution of the program) may have a significant effect on the final number of faults.

For the independent reference model, the FIFO fault rate is equal to the RR fault rate at equal memory sizes [5].

It is interesting to see that even for the actual programs the performance of FIFO is close to RR [1].

2.3.4 LRU (Least Recently Used)

This is by far one of the most discussed replacement algorithms. The algorithm maintains a stack of the main memory page names based on the time of last reference. The most recently used page is on the top of the stack and the least recently used page is at the bottom of the stack. At each reference to a page in main memory, the stack is updated by moving the page name to the top of the stack and pushing the rest of the page names one position down. At the time of a page fault, the page which has not referenced for the longest time (the page in the bottom position in the stack) is removed and the new page is placed on the top of the stack.

The LRU algorithm is an important member of a class of paging algorithms called stack algorithms [13]. An algorithm is a stack algorithm if the buffer content of size m is a subset of the buffer content of size m+1 at each reference time and for all values of m. In [13], an efficient algorithm is presented which can be used to evaluate the performance of stack algorithms for the entire range of memory capacities. For each reference, this algorithm generates a number d, called the stack distance, which has the property that if d > m for any memory capacity less than or equal to m, this reference will cause a page fault. Therefore, for a sequence of references to evaluate the number of page faults for all memory capacities, one needs only to generate stack distances. Then the number of faults for a memory of size m is simply the total number of stack distances greater than m.

The performance of LRU is generally superior to FIFO (Figures 2.1, 2.2, 2.3, 2.4) because it uses the recent past history of the references to predict the future page requirements of the program. The implementation of LRU in its ideal form requires maintaining stacks and the stack operation can be costly. We will see later that by using a much less complex algorithm (CLOCK), we can approximate the operation of the LRU and get performance which is very close to LRU.

A similar algorithm is used in IBM 3850 Mass Storage Facility [4] to schedule blocks of data between the direct access devices and mass storage system.

The long run page fault rate of the independent reference model under LRU is obtained in [11] and [5] and is equal to

$$E_{LRU}(m) = \sum_{s \in Q} D_1^{\,2}(s) \cdot \prod_{i=1}^{m} \frac{p_{j_i}}{D_i(s)} \qquad \text{(IRM)}$$

where

$$D_i(s) = 1 - \sum_{k=1}^{m-i+1} p_{j_k}$$

and $s = (j_1, j_2, \cdots, j_m)$ and Q is the set of all permutations of m out of n. The size of the set Q is equal to $m!\binom{n}{m}$.

Another result obtained for LRU on the independent reference model is given in [8]. Motivated by actual observations, they show the independence of the LRU miss ratio from the page size by giving an upper limit to the difference of approximate LRU fault rates with two different page sizes.

2.4.4 CLIMB

This algorithm works as follows: A stack of all pages in the main memory is maintained. At the time of a reference to a page inside the memory, the position of that page is interchanged with the page immediately above it (if any) in the stack. Therefore, if a page is frequently referenced, it will migrate to the top of the stack. At the time of a page fault, the page on the bottom of the stack is removed and the new page goes to the bottom of the stack. This algorithm was proposed in [16] for self-organizing files. The idea is that the files (or pages, in our case) that are frequently used will be likely to remain in the faster memory. CLIMB, as a paging algorithm for actual programs, has the disadvantage of making the wrong decision on removing a page for traces which have long runs of page reference sequences like ...xyxyxyxy.... (e.g., x=instruction reference and y=data reference). In such cases, the algorithm replaces a page which is going to be referenced again immediately

after it is removed. The performance of CLIMB is compared with other algorithms in Figures 2.7 and 2.8.

For the independent reference model, CLIMB does generally better than LRU (Figure 2.8). Among all the algorithms we have considered, this is the only algorithm for which the independent reference model gives an incorrect estimation of the relative performance of the actual programs under two (CLIMB and LRU) algorithms.

2.4.5 CLOCK

This algorithm is most widely known as the one used in the Multics operating systems. It is basically a very simple algorithm with a good performance. Associated with each page in main memory is a reference bit (or use bit). A list of the names of the pages present in main memory is maintained. A pointer scans through the list in a circular path. When a page is addressed, its reference bit is set to one. When a page fault occurs, the pointer starts searching the list from its current position. As the pointer moves, it turns off the use bit of all pages on its way. The first page with the zero flag (reference bit) is removed and is replaced by the new page. The reference bit of the new page is set to one and the pointer stays there until the time of the next page fault. This algorithm approximates the behavior of LRU in a very efficient way. The implementation of CLOCK is generally very easy using the capabilities of many small and large virtual memory computers. The overhead of processing this algorithm is much less than LRU. The rate of paging activity can be effectively monitored by measuring the average speed of the pointer. Clearly a high pointer speed indicates heavy paging activity. This information can be used by the memory allocator

to tune the memory requirement of active programs in a multiprogramming system.

In Figure 2.7, the performance of CLOCK is compared with LRU. We notice that the performance of CLOCK is very close to LRU for most of the memory sizes.

There are other variations to the CLOCK algorithm. In one scheme, which we refer to as second change CLOCK, after a page is brought into the memory the flag is set to one and the pointer moves one position forward without turning the flag off. Therefore, as the results of upcoming faults, the pointer has to pass over this page at least twice until it can be removed by an incoming page. In our experiments, the measured performance of this algorithm is just about the same as the usual CLOCK algorithm.

2.4.6 MIN

The significance of this algorithm is that it gives the minimum number of page faults among all fixed memory size algorithms. In one of the earliest papers about paging algorithms, Belady [1] describes this algorithm. Later in [13], it is proved to be optimal.

At the time of a page fault, this algorithm removes a page which is not going to be referenced for the longest time in the future. To process this algorithm, we, therefore, need a complete knowledge of future references and clearly it is not realizable in actual situations. However, the MIN algorithm is valuable when we want to see how well other algorithms perform with respect to the optimal strategy.

The MIN algorithm is a stack algorithm. Therefore, for a reference trace, we can efficiently measure the performance of this algorithm for

the entire range of memory sizes. In [13] an evaluation technique is given which requires two passes over the reference string. In the first pass, a sequence of forward distances (forward distance is the number of distinct pages referenced until the next reference to the current page) is generated. In the second pass, the reference string and forward distance string are scanned to generate the sequence of MIN stack distances. The MIN stack distance (like the LRU stack distance for LRU) is the minimum memory capacity associated with the referenced page, such that no page fault occurs if the optimal strategy is used up to that point. In [2], it was shown that the MIN stack distance at each reference time is a function of past references and, hence, to generate the MIN stack distances no look-ahead is necessary. In other words, once the next reference is determined, it is possible, by using the past history of references, to find the associated MIN stack distance. Based on the algorithm presented in [2], Lewis and Nelson [12] give an algorithm which generates the MIN stack distances and LRU stack distances in one pass over the reference string. In the evaluation of MIN and LRU algorithms, we have mostly used their algorithms.

The fault rate of the MIN or OPT algorithm for the independent reference model is given by [5]:

$$F_{OPT}(m) = \sum_{i=m}^{n} p_i - \frac{\sum_{i=m}^{n} p_i^2}{\sum_{i=m}^{n} p_i} \qquad \text{(IRM)}$$

The remaining algorithms are variable space paging algorithms.

2.4.7 WS (Working Set) [6,7]

This algorithm removes a page only if it hasn't been referenced in

the past T references. The parameter T is called the window size. The working set at time, t, WS(t,T), is the set of distinct pages which has been referenced in the interval (t-T+1,t). The working set size at time t, ws(t,T), is the number of pages in the working set. The average working set size is defined by:

$$ws(T) = (1/k) \sum_{t=1}^{k} [w(t,T)]$$

A multiprogramming virtual memory system which uses a WS dispatcher, only dispatches the program which has all its working set in the main memory.

In the MANIAC II computer [14], the working set mechanism is actually built into the hardware. Each page has a register associated to it. When a page is referenced, the value T is loaded into the corresponding register. At each general reference, all registers are decremented by one. The first register to become zero will have its page removed from memory.

The working set dispatcher in its ideal form is not simple to implement. Nonetheless, it has very useful properties which make it desirable as a paging algorithm. By using the WS strategy, the memory allotment of the programs become responsive to the locality (loosely, the set of pages which are heavily referenced in a time interval) changes of the program.

The performance of this algorithm is compared with other algorithms in Figures 2.1, 2.2, 2.3, 2.4, 2.5 and 2.6. There is an efficient technique [5,17] to evaluate the performance of WS for the entire range of the parameter T. The inter-reference statistics can be used to find the average memory size and the fault rate of the WS algorithm: the inter-reference period, x_t, at time t is the number of references since the last reference to the page addressed at time t. If a page is ref-

erenced for the first time, we set the inter-reference at that time equal to infinity. Let I(.) be an indicator function which assumes the value 1 if the expression inside the parenthesis is ture and 0 otherwise. For a reference string $r_1, r_2, \ldots, r_k$, define g(x) as

$$g(x) = (1/k) \sum_{t=1}^{k} I(x_t=x).$$

A page fault can occur if $x_{t+1} > T$; therefore,

$$F_{WS}[ws(T)] = 1 - \sum_{x=1}^{T} g(x).$$

The average working set size is obtained from

$$ws(T) = T - \sum_{t=1}^{T-1} \sum_{x=1}^{t} g(x).$$

For the independent reference model, the fault rate probability is equal to the product of the probability that a page hasn't been referenced in the past T references and the probability that it will be addressed in the next reference, summed over all pages, i.e.,

$$F_{WS}[ws(T)] = \sum_{i=1}^{n} (1-p_i)^T p_i \qquad \text{(IRM)}$$

The average working set size is the probability that a page is in main memory summed over all pages. A page is in memory if it has been referenced at least once in the pat T units of time (or is equal to one minus the probability that it has never been referenced):

$$ws(T) = \sum_{i=1}^{n} [1 - (1-p_i)^T] \qquad \text{(IRM)}$$

2.4.8 WSVT (WS with Variable Window Size)

The choice of the working set parameter T for WS strategy has always been a problem. In [6], Denning originally suggested that T should be equal to twice the time required to load a page block. This choice of T, of course, does not take into account the paging characteristics of the program and, therefore, cannot be considered a right choice in all cases. The tradeoff in choosing the size of T is between inefficient use of the memory (for T too large) or suffering a high page fault rate (for T too small). Thus, if the overall paging behavior of the program is initially unknown, the natural thing to do is to use the observed recent past paging behavior of the program to determine dynamically the size of T for the near future. In doing so, one can specify an acceptable range for the paging activity of the program by setting lower and upper limits on the page fault rate of the program. At the decision points, if the page fault rate had been higher than the upper limit, one increases T, and if it had been lower than the lower limit, one decreases T [7]. Part of the problem with this approach is the extra processing time at the decision points and the choice of the lower and upper page fault rates. The most significant problem, however, is that it is not usually known how sensitive the paging activity is to changes in the window size and whether observed high paging activity occurs because of a locality change (where increasing T is the incorrect decision) or occurs because of an actual increase in the locality size.

We tried a version of variable size WS to see how the performance of this algorithm compares with fixed WS strategy (Figure 2.9). In our algorithm, we start with a given T and follow a normal WS algorithm until a page fault occurs. At this point, the page fault rate (obtained by

measuring the time between the last two faults) is compared with the given lower and upper limits, and if the observed rate falls outside the acceptable range, we increase or decrease the window size accordingly with the amount equal to some fraction of the window size.

In [16], a page partitioning algorithm is presented which assumes a window size parameter for each page, T_{page}. This is a generalization to the idea of having two different window size parameters for data references and instruction references. The parameter T_{page} is the maximum permissable unreferenced interval for which a page can still be kept in the memory. T_{page} is obtained by taking into account the cost of keeping a page in main memory until its next reference and the cost of processing a page fault.

2.4.9 PFF (Page Fault Frequency) [3]

The objective of this algorithm is, like WSVT, to respond to the page demand of the programs dynamically. PFF adapts itself to the paging characteristics of programs by using the measured page fault frequency to change the memory capacity directly. Whenever a page fault occurs, if the fault frequency lies above the given critical value P, then the memory capacity is increased, and if it lies below this value, the memory capacity is decreased. The critical fault frequency P is the parameter of this algorithm. When shrinking the memory, only those pages are removed which have not been referenced since the last page fault time.

The performance of this algorithm is compared with WS in Figure 2.9. We found this algorithm sometimes too responsive to the changes of the program paging characteristics. In the same way, WSVT makes incorrect decisions during the locality changes.

2.4.10 MWS (Modified WS)

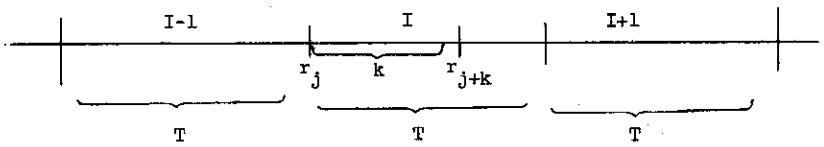
One of the disadvantages of the WS algorithm in its idealized form is the implementation complexity and the overhead involved in processing the algorithm. MWS is intended to reduce the overhead to a great extent while maintaining the positive points of the actual WS algorithm (like capability to respond to the locality changes of the program).

This algorithm uses the reference bit (use bit or flag) associated with each page. This feature is commonly available in many small and large virtual memory computers. Unlike the WS algorithm, MWS does not require the information about the last time a page was used. A window size parameter, T, is defined in this algorithm similar to the same parameter in the WS algorithm. As the program is running, only after the elapse of T time units does the algorithm interrogate the reference bits to determine which pages have been referenced in the past interval. These pages are kept in the memory and constitute the working set for the next interval. The steps of the algorithm are as follows:

- Initialize the reference bits to zero (off).
- For an interval of length T, repeat 1 to 3
  1) Next reference (page)
  2) If it is a new page, increase memory capacity by one
  3) Set reference bit to one (on)
- After T is elapsed, do 4 to 5
  4) Remove all the pages which have not been referenced (use bit = 0) in the last interval
  5) Set the use bit of the remaining pages to zero (now the main memory size is equal to the number of pages which have been referenced in the last interval).

The performance of MWS is compared with WS in Figure 2.11. We can see that MWS does as well as WS for most memory sizes and is significantly superior to LRU. We also notice that for a given window size T, the MWS algorithm gives a larger average working set size compared to WS with the same parameter.

For an independent reference model, assume that the page reference string is divided into intervals of length T (see the sketch). Let the first reference of an interval I be



r_j. Denote the kth reference of this interval by r_{j+k}. The probability that a page i is present in the memory at time j+k is equal to the probability that it has been referenced at least once in the past T+k units of time. This probability is equal to

$$1 - \text{pr}\left\{\begin{array}{l}\text{page i hasn't been referenced in the past}\\ \text{T+k references}\end{array}\right\}$$

$$= [1 - (1-p_i)^{T+k}]$$

Therefore, the expected number of pages present in the memory is the above expression summed over all pages, i.e.,

$$\sum_{i=1}^{n} [1 - (1-p_i)^{T+k}] .$$

The average number of pages in the memory at the end of each interval is denoted by mws(T) and is equal to:

$$\begin{aligned}
\text{mws}(T) &= \frac{1}{T} \sum_{k=1}^{T} \sum_{i=1}^{n} \left[1 - (1-p_i)^{T+k}\right] \\
&= \frac{1}{T} \sum_{i=1}^{n} \sum_{k=1}^{T} \left[1 - (1-p_i)^{T+k}\right] \\
&= \frac{1}{T} \sum_{i=1}^{n} \left[T - (1-p_i)^{T} \sum_{k=1}^{T} (1-p_i)^{k}\right] \\
&= \frac{1}{T} \sum_{i=1}^{n} \left[T - (1-p_i)^{T} \left(\frac{1 - (1-p_i)^{T+1}}{1 - (1-p_i)} - 1\right)\right] \\
&= n - \frac{1}{T} \sum_{i=1}^{n} \frac{(1-p_i)^{T}}{p_i} + \frac{1}{T} \sum_{i=1}^{n} \frac{(1-p_i)^{2T+1}}{p_i} . \qquad \text{[IRM]}
\end{aligned}$$

In the interval I, the first reference to page i causes a page fault only if this page hasn't been referenced in the last interval I-1. Therefore, the number of page faults during interval I is the product of the probability that a page has not been referenced in an interval of length T by the probability that the same page is referenced, at least once, in an interval of the same length, summed over all pages. Thus, the page fault rate of the MWS algorithm is obtained by the following expression:

$$\begin{aligned}
F_{\text{MWS}}[\text{mws}(T)] &= \frac{1}{T} \left\{\sum_{i=1}^{n} [1 - (1-p_i)^{T}](1-p_i)^{T}\right\} \\
&= \frac{1}{T} \left\{\sum_{i=1}^{n} (1-p_i)^{T} - \sum_{i=1}^{n} (1-p_i)^{2T}\right\} .
\end{aligned}$$

As we will show shortly, F_{WS} and F_{MWS}, with the same average memory usage, are very close together for the independent reference models which are constructed, based on the paging behavior of an actual program.

2.4.11 VMIN (Variable Memory Size MIN) [20]

Like MIN for the class of fixed memory size algorithms, VMIN gives the least fault rate among variable size memory algorithms in the plane of average virtual space-time/page fault rate. This algorithm removes a page which is not going to be referenced in the next T units of time. By changing the T parameter, the plot of average memory size versus page fault rate is obtained. This algorithm is proved to be optimal by the following argument. Let U be the cost associated with keeping a page in memory per reference. Let R be the cost of processing a page fault. After a reference to page i, a decision must be made by comparing the cost of keeping this page in the memory until the time of next reference to the same page, with the cost of immediately removing the page from the memory and facing a page fault processing cost at the next reference to this page. The cost associated with the former decision is equal to $C_1 = d.U$, where d is the time until the next reference and the cost associated with the latter decision is equal to $C_2 = R$. Let $R/U = T$, then $C_1 = d.U$ and $C_2 = U.T$. Regardless of the value of U, the optimal algorithm keeps page i if $d \geq T$ and removes it immediately after it is referenced if $d < T$.

For a given T, the fault rate of the VMIN algorithm is equal to the fault rate of the WS algorithm with the same window size. The average memory usage of the VMIN algorithm can be obtained from the average working set size for any window size if we note that, at the time of each page fault using a WS algorithm, the VMIN algorithm, having been used instead, would have kept one less page during the past T references. This is because VMIN removes a page immediately if it is not going to be referenced again in the next T time units. Let ws(T) and F_{WS} be the

average working set size and the fault rate of the WS algorithm for window size T, respectively. Denote vmin(T) the average space time product of the VMIN algorithm, then

$$vmin(T) = ws(T) - T \cdot F_{WS}.$$

For the independent reference model the average memory size and page fault rate are obtained by [19]:

$$F_{VMIN}\,[vmin(T)] = \sum_{i=1}^{n} p_i (1 - p_i)^T$$

$$vmin(T) = n - \sum_{i=1}^{n} (1 - p_i)^T (1 - Tp_i). \qquad [IRM]$$

2.5 TEST RESULTS

The performance of the paging algorithms described in the previous section was obtained using the trace of four computer programs. The testing environment is a simulated virtual memory system with page sizes equal to 512 bytes.

The programs used in the experiments are:

WATFIV: Trace of Watfiv compiler while it is compiling a Fortran program called WATEX (1048662 references).

WATEX: Trace of a Fortran program to find a minimum of a multi-parameter function (1108485 references).

APL: Trace of an interactive session with the APL processor (1108485 references).

FFT1: The trace of Fast Fourier Transform using the Cooley-Tukey algorithm (1108485 references).

In the presentation of the results the initial page faults are excluded.

In Tables 2.1, 2.3 and 2.5 the page faults generated by WATFIV, WATEX, and APL programs under a number of fixed memory size replacement algorithms are shown (the results for WS are also shown for comparison.) In Figures 2.1, 2.2, 2.3, and 2.4 the performances of LRU, FIFO, MIN, and WS on four different traces are shown. We note that the relative performances of these algorithms with respect to each other are roughly the same. The FIFO algorithm gives the worst performance among them. The performance curve of the WS lies between LRU and MIN in most cases. However, in Figure 2.4 we note that, on the average, for certain memory sizes, the WS algorithm gives a smaller page fault rate for a given average memory usage compared to the MIN algorithm.

In Figures 2.5 and 2.6 the performances of LRU (a fixed memory size algorithm) and WS (a variable space algorithm) are compared with the optimal algorithms corresponding to each class using the traces of the WATFIV and APL programs. VMIN, which is optimal among the variable space algorithms, gives significantly better performance.

In Figure 2.7, the performances of CLOCK and CLIMB algorithms are compared with LRU, using the WATFIV and WATEX programs. It is interesting to observe that the CLOCK algorithm, with its simple structure, gives a performance which is very close to the more elaborate LRU algorithm.

CLIMB does not seem to be very suitable as a paging algorithm. Its performance is comparable with FIFO, and it is usually worse than LRU on actual traces.

In Figure 2.8, the performances of LRU and CLIMB on two independent reference models are shown. As it was expected for independent

reference models, CLIMB gives lower page faults than LRU.

The fault rate of variable space algorithms is shown in Tables 2.2, 2.4, and 2.6 for WATFIV, WATEX, APL, and FFT1 programs.

In Figure 2.9, the performance of PFF and WSVT (working set with variable window size) is compared with the WS algorithm on the traces of the WATFIV and WATEX programs. The PFF algorithm does slightly worse than WS. In Figure 2.10, the memory demand of PFF versus time is shown. We can see that this algorithm constantly changes the main memory space of the program. Each sharp drop in the memory size is followed by a high demand for a larger memory size in the coming references.

The slight improvement which is obtained by using WSVT over the WS algorithm does not justify the extra work which is required to change the window size according to the paging characteristics of the program (Figure 2.9).

In Figure 2.11, the page fault rate of MWS (modified WS), LRU and WS are plotted for the WATFIV and APL programs. We note that on the average MWS usually does better than LRU. The performance of MWS is only slightly worse than WS. Considering the efficiency of MWS, the use of this algorithm is definitely more advantageous than WS.

In Figure 2.12, the average memory usage of the MWS and WS algorithms with respect to the window size parameter T is shown. For a given T, MWS gives a larger average memory size. If we let mws(T) and ws(T) be the average memory size with window size T for MWS and WS algorithms, respectively, then $ws(T) \leq mws(T) \leq ws(2T)$.

We basically get the same results when we compare the performance of MWS and WS on two specially constructed independent reference models.

IRM1 and IRM2 are two independent reference models which are parametrized, based on the actual trace of the WATFIV and APL programs using the AØ inversion technique described in the next chapter. In Figures 2.13 and 2.14, the performance and average memory (buffer) size of these two models under the MWS and WS algorithms are shown. The compatibility of these results with those we obtained using the actual traces shows that there is a potential for using models instead of actual traces to predict the behavior of certain algorithms.

2.6 CONCLUSION

In this chapter, we have presented some new results on the performance of paging algorithms, and have reviewed many major findings in this area.

Because of the ever-increasing demand for on-line direct accesibility of very large data spaces and the current gap between the processing speed and the speed of cost-effective memory components, the use of paging algorithms in different echelons of memory hierarchies is inevitable.

We have examined two major categories of paging algorithms with fixed or variable size buffers. Considering the uncertainty of the exact pattern of future references, the algorithms which, in their page selection process and buffer adjustment decisions, use the least recently referenced criterion, seem to be most effective. Among such algorithms, however, the implementation of the exact LRU and WS schemes in a typical processing environment is costly and involves provision of sizable hardware and software machinery. In an alternative approach, one can significantly bring down the overhead by using simpler and more practical algorithms, like CLOCK and MWS, and still achieve comparable performance.

One of the reasons which makes the variable size buffer algorithms

more effective than the fixed size memory algorithms is the capability to respond rapidly to the changes in the program reference pattern. But being too responsive can sometimes degrade the performance. Nevertheless, it is important to have this flexibility to a certain degree. We have seen that if an algorithm has total freedom to change its buffer size, it will do so very frequently. Since it is not conceivable that this freedom can be provided in an actual, say, multiprogramming environment, we should expect the performance of such algorithms under more realistic conditions to degrade accordingly. In this respect, the use of algorithms like MWS, particularly on smaller computers, seems more reasonable. This algorithm, despite its simplicity, can respond sufficiently to the dynamic buffer requirements of programs and meanwhile moderate the demand on the service of the memory manager.

The results of the study of fixed size memory algorithms may be applicable to the overall performance of systems which use a global replacement scheme. In such a system, the collection of active programs can can be considered as a single larger program running in a fixed size buffer. The dispatching of jobs is then analogous to the locality transitions in a single program.

We have presented some examples of the potential of program models in predicting the realistic performance of some of the paging algorithms. The development of this idea is pursued in the next chapter.

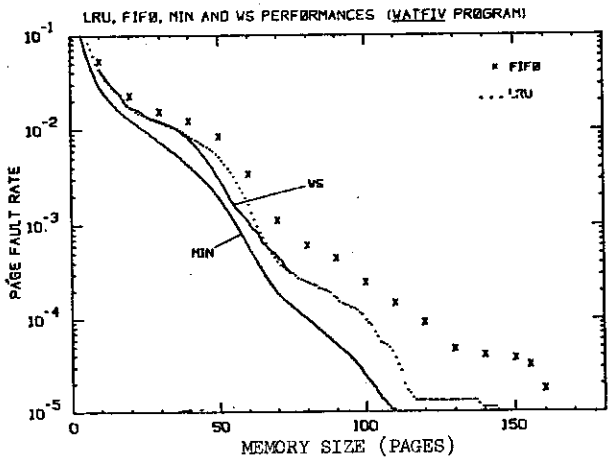


Fig 2.1

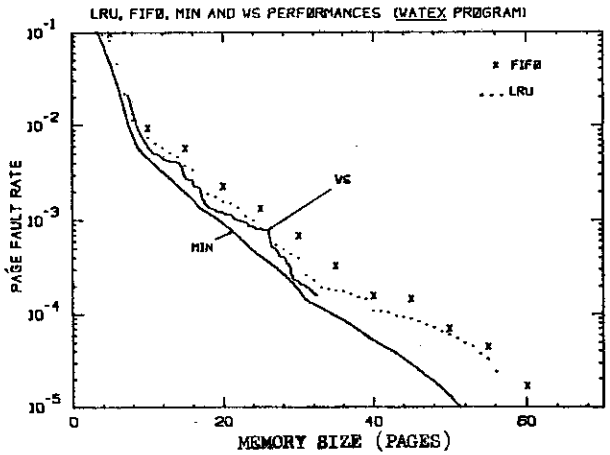


Fig. 2.2

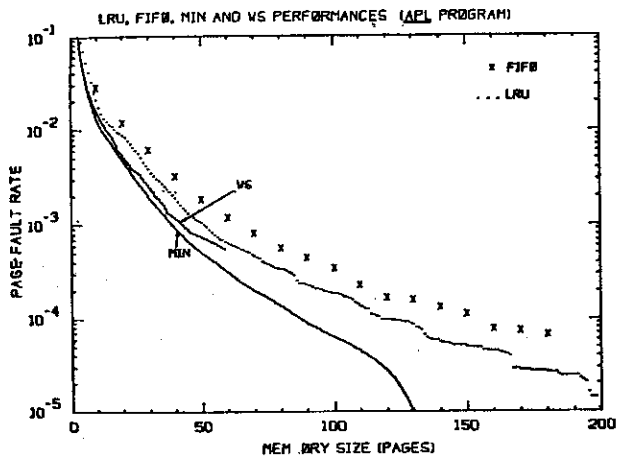


Fig. 2.3

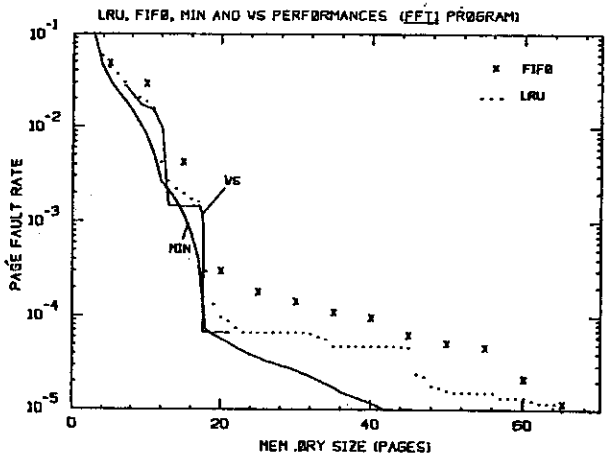


Fig. 2.4

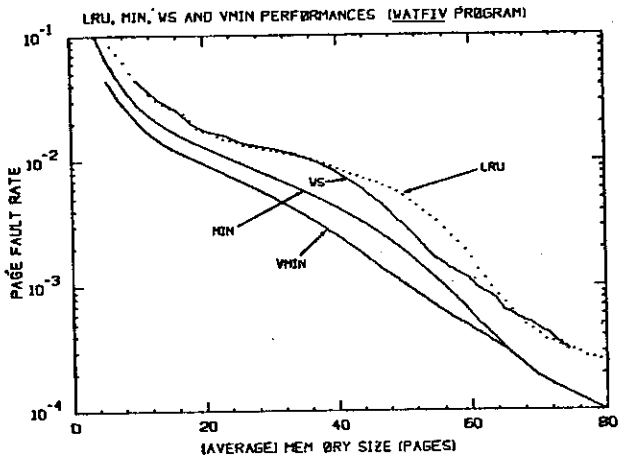


Fig. 2.5

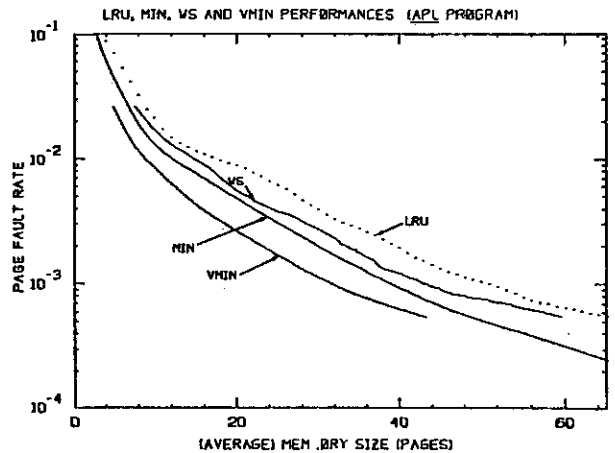


Fig. 2.6

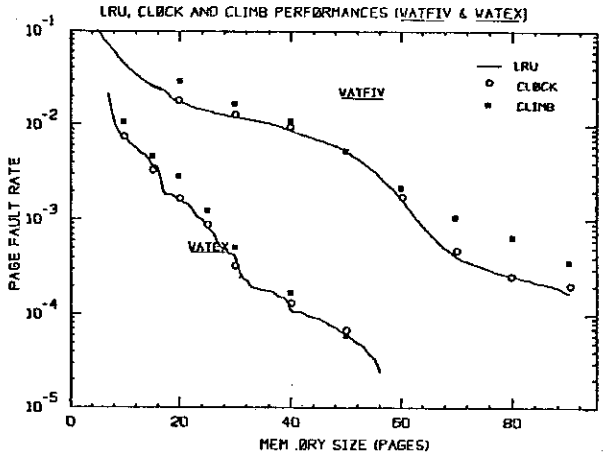


Fig. 2.7

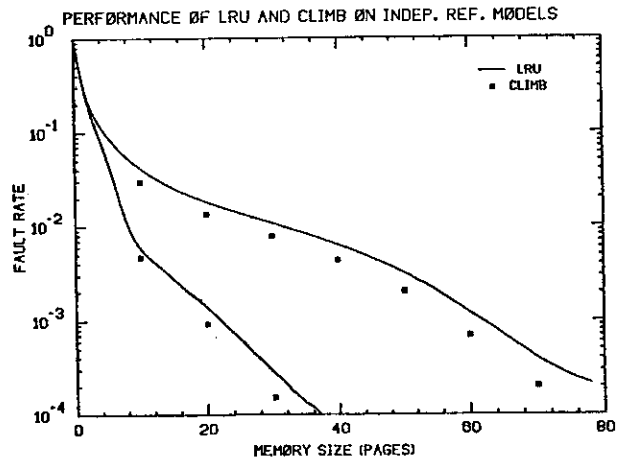


Fig. 2.8

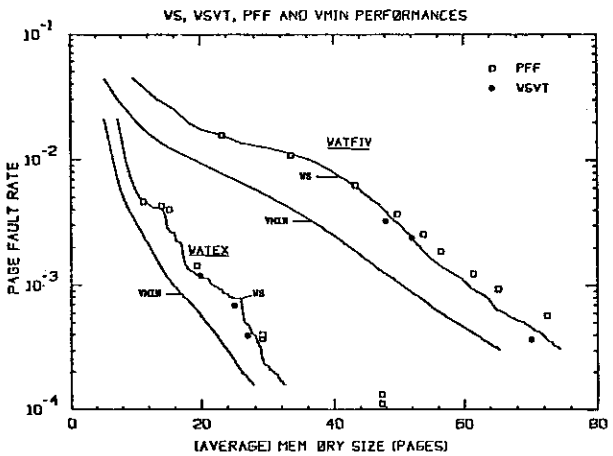


Fig. 2.9

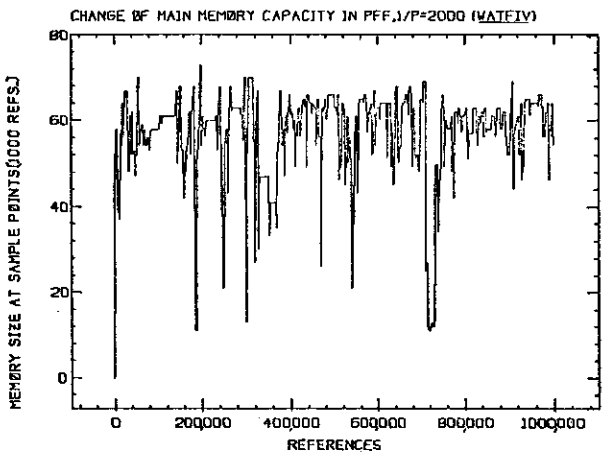


Fig. 2.10

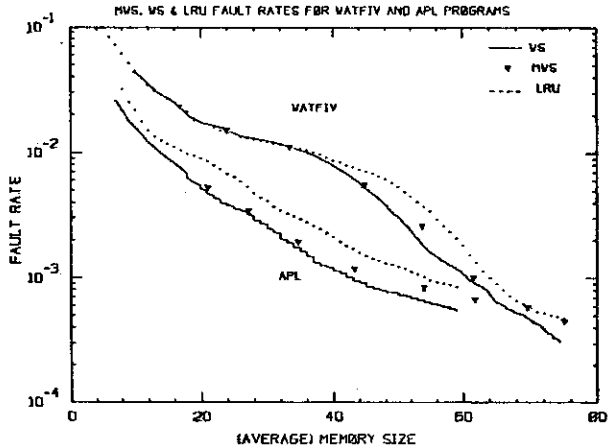


Fig. 2.11

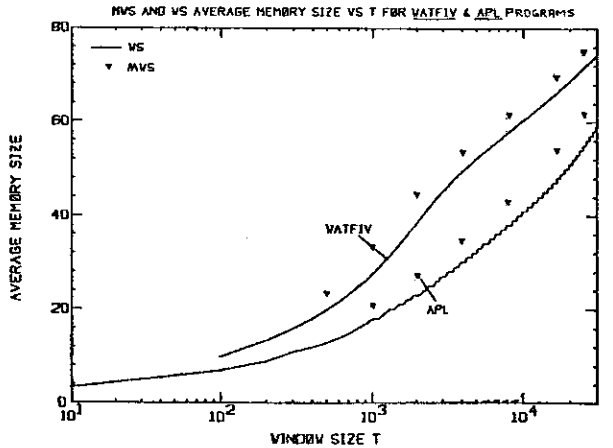


Fig. 2.12

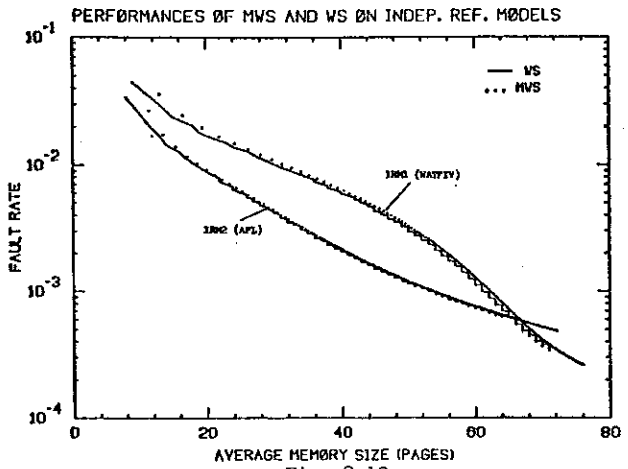


Fig. 2.13

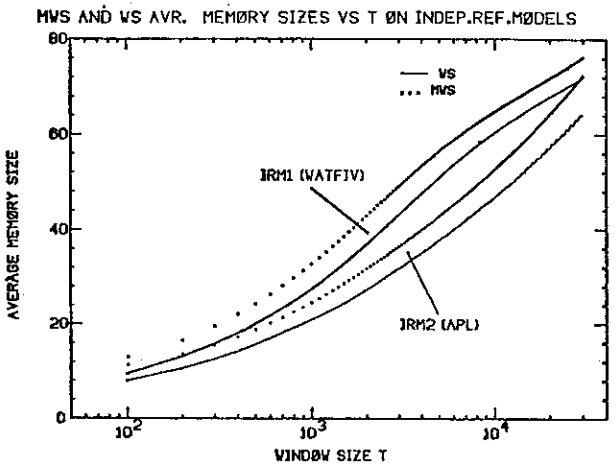


Fig. 2.14

M	FIFO	LRU	CLIMB	CLOCK	MIN	WS	
						Window Size	Fault Rate
5	.121616	.102273			.069809		
10	.053024	.043416			.027715	100	.043991
15	.031814				.017306		
16		.024703			.016139	300	.024823
20	.022607	.016972	.029459	.017815	.012690	500	.017220
23		.014686			.010765	700	.015488
24		.014272			.010210		
25	.017710	.013829			.009678	800	.014256
30	.001561	.011864	.016727	.012854	.007337	1200	.012235
33		.011030			.006246	1400	.011411
35	.014279	.010452			.005594	1600	.010465
40	.012360	.008463	.010332	.009433	.004237	2200	.007766
43		.007404			.003378	2600	.006191
45	.010588	.006809			.002924	2900	.005228
50	.008447	.005160	.005306	.005379	.001891	4100	.003070
54		.003545			.001236	5800	.001787
55	.005761	.003237			.001101	6300	.001603
56		.002787			.000979	6900	.001471
60	.003407	.001645	.002208	.001173	.000590	9700	.001068
61		.001341			.000518	10600	.000948
62		.001150			.000460	11500	.000878
65	.002037	.000729			.000317	14700	.000631
70	.001128	.000406	.001063	.000459	.000186	21900	.000456
72		.000357			.000162	25300	.000387
75	.000824	.000308			.000134		
80	.000611	.000246	.000643	.000252	.000099		
90	.000443	.000166	.000353	.000199	.000052		
100	.000247	.000096			.000023		
110	.000148	.000038	1		.000009		
120	.000092	.000013			.0		
130	.000047	.000013					
140	.000041	.000011					
150	.000038	.000004					
160	.000018	.000001					
164		.0					

TABLE 2.1 Performance of Different Paging Algorithms on WATFIV Program

	WS		MWS		VMIN		PFF	
M	Window Size	Fault Rate	Window Size	Fault Rate	Window Size	Fault Rate	1/P Parameter	Fault Rate
5					100	.043991		
10	100	.043991			400	.019423		
15					1100	.012644		
16	300	.024823			1300	.011833		
20	500	.017220			1900	.009125		
23	700	.015488			2200	.007698	150	.015487
24			500	.015068	2300	.007260		
25	800	.014256			2400	.006888		
30	1200	.012235			2900	.005228		
33	1400	.011411	1000	.011045	3400	.004156	300	.010661
35	1600	.010465			3700	.003595		
40	2200	.007766			4700	.002498		
43	2600	.006191			5500	.001902	500	.006320
45	2900	.005228	2000	.005553	6300	.001603		
50	4100	.003070			9900	.001037	1000	.003752
54	5800	.001787	4000	.002578	13500	.000734	1500	.002589
55	6300	.001603			14100	.000670		
56	6900	.001471			14300	.000617	2000	.001894
60	9700	.001068			21900	.000456		
61	10600	.000948			23500	.000422	3000	.001234
62	11500	.000878	8000	.001004	25000	.000393		
65	14700	.000631	16000	.000591	29700	.000308	4000	.000960
70	21900	.000456						
72	25300	.000387					6000	.000584
75			24000	.000349				
80								
90								
100								
110								
120								
130								
140								
150								
160								
164								

TABLE 2.2 Performance of Different Variable Space Paging Algorithms on WATFIV Program

						WS	
M	FIFO	LRU	CLIMB	CLOCK	MIN	Window Size	Fault Rate
1		.589950					
2		.216472					
3		.144052					
4		.113409					
5	095367	.080320					
10	.009328	.007337	.010883	.007513			.005717
11		.006355				400	.004893
14		.004632				1300	.003964
15	.005706	.003707	.004651	.003454		1400	.002908
19		.001758				3700	.001251
20	.002286	.001566	.002936	.001677		4500	.001151
25	.001317	.000804	.001123	.000885		9400	.000788
29		.000433				16700	.000272
30	.000686	.000394	.000512	.000332		20000	.000223
32		.000226			.000127	30000	.000157
35	.000324	.000177					
40	.000159	.000109	.000171	.000131			
45	.000145	.000088					
47		.000076					
47		.000076					
47		.000076					
50	.000071	.000060	.000060	.000065			
55	.000045	.000033					
57		.0					
60	.000017						
65	.000009						

TABLE 2.3 Performance of Different Paging Algorithms on WATEX Program

	WS		MWS		VMIN		PFF	
M	Window Size	Fault Rate	Window Size	Fault Rate	Window Size	Fault Rate	1/P Parameter	Fault Rate
1								
2								
3								
4								
5					100	.021189		
10	400	.005717			1400	.003486		
11	600	.004893			1800	.002643	150	.004720
14	1300	.003964			3000	.001336	300	.004223
15	1400	.002908			5200	.001137	500	.004091
19	3700	.001251			10800	.000645	1000	.001494
20	4500	.001151			11100	.000556		
25	9400	.000788			17500	.000239		
29	16700	.000272					1500	.000404
30	20000	.000223						
32	30000	.000157						
35								
40								
45								
47							3000	.000130
47							4000	.000111
47							5000	.000111
50								
55								
57								
60								
65								

TABLE 2.4 Performance of Different Variable Space Paging Algorithms on WATEX Program

M	FIFO	LRU	MIN	WS	
				Window Size	Fault Rate
5		0.068350	0.043446		
10	0.028335	0.021101	0.021996	200	0.017631
11		0.017098	0.011242	300	0.013939
15		0.011442	0.007508	600	0.009887
20	0.011886	0.008771	0.004746	1300	0.005462
21		0.008289	0.004330	1500	0.004968
25		0.006109	0.003025	2400	0.003719
27		0.005120	0.002541	2900	0.003373
30	0.006156	0.003916	0.001962	3800	0.002704
35		0.002738	0.001327	6000	0.001745
40	0.003272	0.001919	0.000923	9200	0.001196
43		0.001519	0.000756	11500	0.001003
45		0.001318	0.000664	13300	0.000900
50	0.001877	0.001042	0.000507	18300	0.000757
54		0.000835	0.000421	22800	0.000661
55		0.000779	0.000401	24000	0.000638
60	0.001201	0.000645	0.000315	30000	0.000520
65		0.000553	0.000246		
70	0.000816	0.000367	0.000199		
80	0.000570	0.000339	0.000187		
90	0.000453	0.000231	0.000089		
100	0.000346	0.000187	0.000063		
110	0.000231	0.000139	0.000045		
120	0.000169	0.000098	0.000027		
130	0.000161	0.000086	0.000009		
140	0.000135	0.000055	0.		
150	0.000114	0.000050			
160	0.000077	0.000045			
170	0.000074	0.000028			
180	0.000067	0.000026			
190		0.000024			
195		0.0			

TABLE 2.5 Performance of Different Paging Algorithms on APL

M	WS		MWS		VMIN	
	Window Size	Fault Rate	Window Size	Fault Rate	Window Size	Fault Rate
10	200	0.017631			700	0.008995
15	600	0.009887				
20	1300	0.005462			4000	0.002598
21	1500	0.004968	1000	0.005277	4400	0.002378
25	2400	0.003719			6300	0.001659
27	2900	0.003373	2000	0.003484	7300	0.001413
30	3800	0.002704			10000	0.001116
35	6000	0.001745	4000	0.001983	15900	0.000800
40	9200	0.001196			24600	0.000629
43	11500	0.001003	8000	0.001192	29500	0.000551
45	13300	0.000900				
50	18300	0.000757				
54	22800	0.000661	16000	0.000833		
60	30000	0.000520				
62			24000	0.000673		

TABLE 2.6 Performance of different variable space algorithms on APL

2.7 APPENDIX

In the following, we derive the long run page fault rate for the independent reference model with parameters $[p_1, p_2, \ldots p_n]$ under the random replacement (RR) algorithm for a buffer size $m \leq n$.

Define a buffer state s as the collection of m pages present in the buffer at any reference time. There are a total of $\binom{n}{m}$ distinguishable states. The long run page fault rate is equal to the probability of being in a state s by the probability of referencing a page outside the buffer summed over all the states, i.e.,

$$F(m) = \sum_s \Pr(s) \sum_{i \notin s} p_i. \qquad (1)$$

Under our replacement algorithm, the buffer state transitions can be modeled by a discrete finite state Markov chain with the state space $Q = \{s_i\}$, $i=1,2,\ldots\binom{n}{m}$.

Define a neighbor of a state s by $s(-i,+j)$ which is identical to s except for one page, namely, $i \in s$, but $i \notin s(-i,+j)$ and $j \notin s$ but $j \in s(-i,+j)$.

Let $\Pr(s,s')$ be the state transition probability of going from state s to s' in one reference. With the random replacement of the pages in the buffer, we have

$$\Pr(s,s') = \begin{cases} \sum_{i \in s} p_i & \text{for } s=s' \\ \frac{1}{m} p_j & \text{for } s'=s(-i,+j) \\ 0 & \text{otherwise} \end{cases}$$

One can find the steady buffer state probabilities by solving the equilibrium equations of the Markov chain. Let $\pi = (\pi_s)$ be the vector of steady state probabilities and $P = [\Pr(s,s')]$ be the state transition matrix.

We can verify that the normalized solution for the equilibrium equations $\pi = \pi P$ is given by:

$$\pi_s = G^{-1} \prod_{j \in s} p_j \qquad (3)$$

where $G = \sum_{s \in Q} \prod_{k \in s} p_k$.

For a state s, the equilibrium equation becomes:

$$\pi_s = \pi_s \sum_{j \in s} p_j + \sum_{j \in s} \sum_{i \notin s} \frac{p_j}{m} \pi_{s(-j,+i)}$$

Substituting for $\pi_{s(-j,+i)}$ from (3) above we get:

$$\pi_s = \pi_s \sum_{j \in s} p_j + \sum_{j \in s} \sum_{i \notin s} \frac{p_j}{m} \frac{\prod_{k \in s(-j,+i)} p_k}{G}$$

$$= \pi_s \sum_{j \in s} p_j + \frac{1}{m} \sum_{j \in s} \sum_{i \notin s} \frac{p_i p_j \prod_{k \in s} p_k}{p_j \cdot G}$$

$$= \pi_s \sum_{j \in s} p_j + \frac{1}{m} \cdot m \sum_{i \notin s} p_i \cdot \pi_s$$

$$= \pi_s \left(\sum_{j \in s} p_j + \sum_{i \notin s} p_i \right) = \pi_s$$

We just showed that (3) is the solution for the normalized equilibrium equations. We can now substitute for the state probabilities in (1) the solution from (3) to get the page fault rate of the independent reference model under random replacement algorithm:

$$F_{RR}(m) = \sum_{s \in Q} \frac{\prod_{j \in s} p_j}{G} \sum_{i \notin s} p_i$$

2.8 BIBLIOGRAPHY

1. Belady, L.A., "A study of replacement algorithms for virtual-storage computer," IBM Systems J. 5,2 (1966).
2. Belady, L.A., Palmero, F.P, "On line measurement of paging behavior by multivalued MIN algorithm," IBM J. Res. Develop (Jan. 1974).
3. Chu, W.W., Opderbeck, H., "The page fault frequency replacement algorithm," AFIPS Conf. Proc., Fall Joint Computer Conf. (1972) pp 597-609.
4. Clayton, J., "IBM 3850 - Mass storage system," IFIPS National Computer Conference (1975), pp 509-514.
5. Coffman, E.G., Denning, P.J., "Operating systems theory," Prentice-Hall, Inc., Englewood, N. J. (1973).
6. Denning, P.J., "The working set model for program behavior," CACM 11,5 (May 1968).
7. Denning, P.J., "On modeling program behavior," AFIPS Conf. Proc., Spring Joint Computer Conference (1972), pp 937-944.
8. Fagin, F., Easton, M.C., "The independence of LRU miss ratio on page size," IBM Res. Report RC5006 (August 1974).
9. Franaszek, P.A., "An algorithm for computing MIN fault statistics," IBM Res. Report RC5291 (February 1975).
10. Kilburn, T., et.al., "One level storage system," IRE Trans. Elec. Computers, EC-11, No. 2 (1962).
11. King, W.F., "Analysis of demand paging algorithms," Proc. IFIPS Conference, Ljubliana, Yugoslavia (1971).
12. Lewis, C.H., Nelson, R.A., "Some one pass algorithms for the generation of OPT distance strings," IBM Watson Res. Center Report RC4758 (March 1974).

13. Mattson, R.L., Gecsei, D.R., Traiger, I.L., "Evaluation techniques for storage hierarchies," IBM Systems J. 9,2 (1970).

14. Morris, J.B., "Demand paging through utilization of working sets on MANIAC II," CACM 15, 10 (October 1972).

15. Oliver, N.A., "Experimental data on page replacement algorithm," IFIPS National Computer Conf. (1974).

16. Rivest, R.L., "On self-organizing sequential search heuristics," CACM 19,2 (February 1976).

17. Slutz, D.R., Traiger, I.L., "A note on the calculation of average working set size," CACM 17,10 (October 1974).

18. Smith, A.J., "Analysis of optimal, look ahead demand paging algorithms," Computer Science Division, University of California, Berkeley, (March 1975).

19. Prieve, B.G., Fabry, R.S., "Evaluation of page partitioning replacement algorithm," Computer Science Division, University of California, Berkeley, (1975).

20. Prieve, B.G., Fabry, R.S., "VMIN - An optimal variable space replacement algorithm," Computer Science Division, University of California, Berkeley, (May 1975).

CHAPTER 3

Aϕ INVERSION MODEL

3.1 INTRODUCTION

In every computing system, programs are the basic entities which determine and control the dynamics of the system. The address reference behavior of programs is one of the fundamental factors affecting the design of useful and efficient control algorithms throughout a computing system.

In this chapter, some aspects of program page reference modeling techniques are considered. We will be particularly interested in the characterization of a compact process to generate a sequence of page references which can effectively replace the page reference sequence of the real programs. This effort has immediate application in the performance evaluation of virtual memory [9, 10, 21] systems and can be extended to the evaluation of high speed buffers (caches) [2, 8] for high speed CPUs as well as slower automated filing systems [3].

A new and effective technique will be introduced which takes the optimal fault rate [5] characteristics of a program and projects them back into an independent reference [7] model. The new model is referred to as Aϕ inversion model. This model maintains the simple probablistic structure of the independent reference model, yet it has excellent predictive power: the A0 inversion model can predict the paging performance of actual programs under several well-known replacement algorithms [5], and is capable of estimating the average working set sizes [11] of actual programs for a wide range of window sizes.

We begin this chapter with a discussion of the uses of program reference models in the performance evaluation of different aspects of com-

puting systems and we point out the goals and motivations for constructing simple and yet powerful models. A description of a number of proposed program reference models follows. The properties and features of the notable models are mentioned. Next, we define the AØ inversion model and demonstrate the technique for finding its parameters. The capabilities of the model are validated by presenting some simulation results and comparing them with the observed behavior of real programs. New experimental results with the LRU stack model are presented. The performance of both models are compared. A possible functional form is considered which can describe the parameters of the AØ inversion model.

Other approaches that yield similar models, possible extensions, and other applications are suggested. Following that, we discuss some of the problems which may be encountered when we construct the AØ inversion model. The execution characteristics of the model, which limits its use in certain applications, is discussed. Some concluding remarks will end this chapter.

3.2 PROGRAM REFERENCE MODELS

The sequence of page references (reference string) generated by a running program generally exhibit periods of concentrated references over a sub-space of the program address space, repetitive reference patterns, and some degree of random behavior. Our modeling efforts essentially are based on constructing a stochastic process which can characterize the address reference behavior of a program or other sequences which are extracted from the reference string. Examples are the modeling of the sequence of page exceptions generated by a program [19], and the modeling of the sequence of working set sizes [Chapter 1].

The modeling of a page reference sequence itself is of more interest for us because it provides a flexible tool for a more generalized set of applications. A model of page reference sequences, for example, can be used as an efficient tool in the evaluation and comparative study of several paging algorithms for a wide range of memory and page sizes in a virtual memory system.

Our motivation to study program reference models can be explained by considering the areas in which these models can be applied, and a set of qualifications which makes each model a convenient tool for the analysis of different aspects of a computer system.

The analysis of some of the algorithms which work on program address reference sequences is only possible if we can find a manageable and realistic formulation of the problem. The analytical approaches enable us to get a better insight into the subjects ranging from finding the performance bounds, the long run behavior, the relative efficiency of different mechanisms, etc. In this regard, we prefer program models which are analytically tractable and, meanwhile, posses a good degree of realism with respect to the real world.

The trace driven simulations are widely used computer performance evaluation techniques. We may resort to simulation when the analytical approaches are difficult to formulate or when the analytical expressions which describe the behavior of the system is inappropriate for the numerical evaluations. The trace driven simulation runs are usually costly operations in terms of CPU and other resource utilizations. Therefore, it is very desirable to have a simple program model which can be used efficiently in simulation packages, and which could free us of the need for the trace of real programs.

The generality and the predictive power of a model are important factors which enable us to get realistic results using the model. The models which can minic certain behavior of the real programs, but drastically fail in other aspects, are of limited use in certain applications. We say a model has predictive power if it can be used successfully under new circumstances which are different from those that are directly built into the model. For example, the LRU stack model [7] is constructed, based on the LRU stack depth distribution of actual programs. It is no surprise that it gives the same LRU fault rate as the original program. However, if on the average it generates the same number of faults as the actual program under another replacement algorithm, say MIN, we can claim that it has a certain degree of predictive power.

In the literature, a number of program reference models have been proposed. In each model, the program references are generated according to some mechanism which is built into the model. Let $r_1, r_2, \ldots, r_k$ be a sequence of consecutive page references which are generated by a program. At each instance t, r_t is a page name from the set of n program pages [1, 2, 3, ..., n]. This set constitutes the virtual address space of the program.

In the following, a number of proposed program page reference models are discussed.

3.2.1 Locality Model [12]

Programs tend to cluster their references on a (slowly) varying subset of pages during an execution sub-interval. Locality models are proposed in an attempt to formalize this behavior. The locality sets L, L', L", ... are defined where each is a subset of ℓ program pages. In a general locality model, P(L,L'), give the probabilities of inter-locality transitions and each locality has a holding time distribution $h_L(x)$ which determines the probability of staying in the locality L for x units of time. In a simple locality model, a page inside the current locality is referenced with the fixed probability (1-u), therefore, $[h_L(x) = (1-u)^{x-1}u]$. An interior page i is referenced with probability a(i), which can be a function of past locality transitions. The probability of referencing a page outside the locality is uniform for all those pages and is equal to $\frac{u}{n-\ell}$. One of the restrictions imposed on this model is that any interior page has a higher reference probability than any exterior page, i.e., $(1-u)a(i) \geq \frac{u}{n-\ell}$ for all i inside the current locality.

A restricted form of the LRU stack model can be considered a generalization of a simple locality model and we shall discuss this model in more detail.

3.2.2 Denning and Schwartz Model [7, 13]

In this model, it is assumed that the inter-reference intervals (i.e., the distance between two successive references to the same page) of any page i is distributed, independent of the other pages, with the probability density function $f_i(.)$. The mean inter-reference interval for each page is finite and is equal to:

$$\bar{x}_i = \sum_{x \geq 1} x\, f_i(x).$$

A useful closed form solution for the average working set size with window size T, ws(T), can be found for this model. Let $F_i(x)$ be the

cumulative probability distribution function of the inter-reference intervals of page i, i.e.,

$$F_i(x) = \sum_{j=1}^{x} f_i(j)$$

and define the overall inter-reference distribution as

$$F(x) = \sum_{i=1}^{n} \frac{F_i(x)}{x_i}$$

It can be shown that the mean working set size with parameter T is given by:

$$ws(T) = \sum_{x=0}^{T-1} [1 - F(x)].$$

Thus, when F(x) is known, this result can be used to find the average working set sizes and the average WS fault rate for all window sizes.

3.2.3 Markovian Models [4, 15]

For this class of models, the consecutive address references are generated by a Markov process. With first order Markov dependence, the probability that page i is referenced at time t only depends on the page name referenced at time t-1, i.e.,

$$Pr[r_t = i | r_1, r_2, \ldots, r_{t-1} = j] = Pr[r_t = i | r_{t-1} = j] = q_{ij}$$

The q_{ij}'s, $1 \leq i, j \leq n$, are the state transition probabilities of the model.

This model has a fairly simple probability structure. However, when n is large, using this model becomes practically impossible. There hasn't been much evidence supporting the capability of this model in simulating actual program behavior.

3.2.4 LRU Stack Model [7, 12]

The LRU stack model is based on the probability distribution of LRU stack depths. The LRU stack is a stack in which the pages are ordered according to their last references. The most recently referenced page is on the top of the stack, the next most recently referenced page is below that, and so on. In this way, the least recently referenced page resides at the bottom of the stack.

The LRU stack depth or distance at each instance is the stack position of the referenced page. After each reference, the stack is updated by moving the page to the top of the stack. The distribution of the LRU stack depths are constructed by counting the number of times a particular LRU depth is accessed. These counts can be used to approximate a discrete probability density function for an LRU stack model.

In the LRU stack model, the sequence of LRU distances are i.i.d. random numbers with probability density function $[d_1, d_2, \ldots, d_n]$ where:

$$Pr[\text{LRU distance at time } t=i] = d_i \quad .$$

This model, therefore, generates a sequence of LRU stack distances and, from this sequence, a unique sequence of page names can be reconstructed by maintaining an ordinary LRU stack of all the referenced pages.

We note that if the distribution of stack distances are biased toward the smaller distances, then a highly localized sequence of page names is generated by this model. Scattered references are generated depending on the spread of the probability weights over the stack positions.

In Figure 3.1, the relative frequency count of the observed LRU stack positions generated by a WATFIV compiler trace is shown. This histogram can be used to construct an LRU stack model based on the same program.

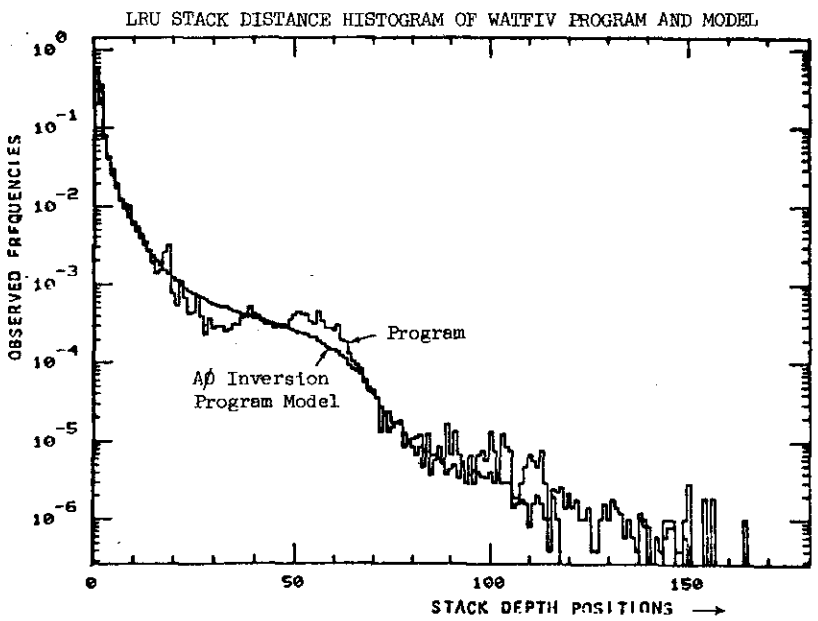


Fig. 3.1

The LRU stack model has a number of significant properties. Namely, in a model with n pages, the probability that any page, independent of its identity, is in any stack position is 1/n. To show this, let $[d_i]$, i=1,2,...,n be the probability density function of the LRU stack positions. Define a Markov chain $X_q(t)$ in which the states are the positions of some page q in the LRU stack. Let $M=(m_{ij})$ be the state transition matrix of this chain where m_{ij} is the probability that at the time of a page reference page q goes from stack position i to position j. In this model, the position of page q is updated according to the following probability transition matrix:

$$M = \begin{Bmatrix} d_1 & \sum_{i=2}^{n} d_i & 0 & \cdot & 0 \\ d_2 & d_1 & \sum_{i=3}^{n} d_i & \cdot & 0 \\ d_3 & 0 & \sum_{i=1}^{2} d_i & \sum_{i=4}^{n} d_i & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ d_{n-1} & \cdot & \cdot & \cdot & d_n \\ d_n & \cdot & \cdot & \cdot & \sum_{i=1}^{n-1} d_i \end{Bmatrix}$$

Let $\pi = (\pi_i)$, i=1,2,...,n be the steady state probability of the position of page q in the stack. When the chain is an irreducible ergodic process, the solution to $\pi.M=\pi$ and the normalization term $\sum_{i=1}^{n} \pi_i = 1$ give the steady state probabilities of the position of page q in the stack. Since matrix m is doubly stochastic (i.e., its rows and columns sum to one), the solution for the steady state probabilities is:

$$\pi_1 = \pi_2 = \ldots\ldots = \pi_n = \frac{1}{n}$$

This result holds for all pages and it implies that the relative reference frequency of all pages in a reference string generated by this model (a realization of the model) is the same for all pages. This behavior, of course, is against our intuition about the reference patterns in real program traces. This might be one of the reasons why the model has not received enough empirical treatment in the literature.

If the stack position probability density function $[d_i]$, i=1,2,...,n in an LRU stack model is such that $d_1 \geq d_2 \geq d_3 \geq \ldots. \geq d_n$, then the expected fault rate generated by this model under the LRU replacement rule is optimal (minimal) among all fixed memory size algorithms and for all memory sizes. This can be seen by noting that with memory size m, $(m \leq n)$, at the time of a page fault the LRU algorithm replaces a page which is least recently used (i.e., the page in the bottom of the stack of the main memory pages) and, in this model, among all pages in the memory, this page is the one which is least likely to be referenced in the future. The optimal fault rate probability is, therefore, $F_m[LRU] = \sum_{i=m+1}^{n} d_i$.

Expressions like the distribution of working set sizes can be found for this model; however, the results soon become computationally unattractive.

This model has good predictive power and we shall return to it later in this chapter.

3.2.5 Independent Reference Model

This model has a very simple probability structure, yet it is the most interesting and versatile among the models discussed so far. A set of page reference probabilities $[p_1,p_2,p_3,\ldots,p_n]$ is defined. The sequence of page references $r_1,r_2,r_3,\ldots,r_k$ are drawn independently from the set of reference probabilities, such that

$$Pr[r_t=i] = p_i$$

Among the fine properties of this model is its analytical tractability. This model has been extensively used in the literature [chapter 2, 7, 13, 14, 18], especially for the performance evaluation of a number of replacement algorithms. Since the sequence of actual program references do not exhibit the strict serial independence as dictated by the model, the analytical results are assumed to be of limited significance when carried over to actual situations. However, as we shall shortly see, the validity of this argument to a great extent depends on the method by which the parameters of this model are found.

The parameters of the independent reference model are its page reference probabilities. One can estimate a set of reference probabilities by using the trace of a real program. The trivial approach is to count the number of references to each page and assume that these references were generated by an independent reference model. Using the page refer-

ence frequency (counts), we then approximate an independent reference model which has the same relative reference probabilities. Let us call this method of determining the parameters as the frequency method. Now we ask ourselves how well this model can mimic the behavior of an actual program with respect to the average fault rate performance under different paging algorithms.

In Figure 3.2, the fault rate curves of a WATFIV compiler under LRU and MIN paging algorithms are shown with solid lines. In this figure, the horizontal axis is the memory capacity and the vertical axis is the fault rate, i.e., (number of faults)/(total number of references).

We use the trace of the WATFIV program to construct an independent reference model by the frequency method. The long run fault rate of this model, under the LRU and MIN algorithms, are shown by dotted line on the same figure. We notice that the model generates far more faults for a given memory size compared to the real program. These results lead us to believe that other features of the model also will be grossly off the observed values in the actual program. For example, this model significantly overestimates the observed working set sizes in the actual program.

In the next section, we shall introduce a new and important method which we use to find the parameters of an independent reference model. We will see that even though keeping the model as simple as the independent reference model, we can substantially improve its predictive power. This gives us more confidence in interpreting different types of results which are based on this model.

3.3 Aϕ INVERSION MODEL

The Aϕ inversion model (the choice of the name becomes clear shortly) is basically a member of the class of independent reference models. What distinguishes this model is the method by which the parameters are obtained. Earlier we showed that the frequency method produces an independent reference model which is a poor predictor of actual program behavior. Referring to Figure 3.2, we note, however, that although by the frequency method we obtain a model which grossly fails to estimate the fault rate of the actual program, the model is fairly successful in capturing the relative performance of LRU and MIN algorithms. This property of the independent reference model is important because it suggests that if we are able to come up with a model which can predict the fault rate behavior of a program correctly under one replacement algorithm, then the performance of the model under other algorithms might also be close to the performance of the actual program under the same algorithms.

We observe that one way of binding the model to the characteristics of a real program is to require that the lower bound on the page fault rate of both the model and the actual program under optimal replacement algorithms be close together. We can then be sure that enough structure is built into the model so that, at least in the long run, the model is capable of predicting the behavior of the actual program under the optimal paging algorithm.

For a given page reference sequence of an actual program, we know that MIN algorithm gives the least amount of faults among all fixed memory size algorithms. We can, in fact, measure the MIN fault rate of the program, $F_{MIN}(m)$, for different memory sizes m $(1 \leq m \leq n)$. For the independent reference model, the so-called Aϕ [4] algorithm gives the optimal

fault rate. At the time of a page fault, the Aϕ algorithm pushes out a page which is least likely to be referenced in the future.

Let $[p_1,p_2,p_3,\cdots,p_n]$ and $p_1 \geq p_2 \geq p_3 \cdots \geq p_n$ be the set of reference probabilities for an independent reference model. The Aϕ fault rate produced by this model, $F_{A\phi}(m)$, for a memory size m is equal to [7]:

$$F_m = F_{A\phi}(m) = \sum_{i=m}^{n} P_i - \frac{\sum_{i=m}^{n} P_i^2}{\sum_{i=m}^{n} P_i} \qquad 1 \leq i \leq n \qquad (1)$$

Therefore, if the reference probabilities are known, (1) can give the optimal fault rate for different values of m, $1 \leq m \leq n$. Conversely, if a set of n fault rate values are given, we should be able to find a set of reference probabilities which satisfy the relations in (1).

We observe that if our independent reference model is to capture the fault rate behavior of actual programs, then we expect that the fault rate of the model under the Aϕ algorithm should be close to the fault rate of the actual program under the MIN algorithm and for all memory sizes. This gives us a procedure to find p_i's from the relations in (1). In other words, we now substitute for F_m's in (1) the observed MIN fault rate values, and then we invert (1) to get a set of recurrence expressions for finding p_i's. The independent reference model which is obtained by this procedure is referred to henceforth as the Aϕ inversion model.

We carry out this procedure by letting $S_m = \sum_{i=m}^{n} p_i$ and $R_m = \sum_{i=1}^{m} p_i$. We then successively get:

$$F_m = \sum_{i=m}^{n} p_i - \frac{\sum_{i=m}^{n} p_i^2}{\sum_{i=m}^{n} p_i}$$

$$F_m = S_m - \frac{\sum_{i=m}^{n} p_i^2}{S_m}$$

$$F_m S_m = S_m^2 - \sum_{i=m}^{n} p_i^2$$

Similarly:

$$F_{m+1} S_{n+1} = S_{m+1}^2 - \sum_{i=m+1}^{n} p_i^2$$

Subtracting the above two expressions we get:

$$F_m S_m - F_{m+1} S_{m+1} = S_m^2 - S_{m+1}^2 - p_m^2$$

$$F_m(p_m+S_{m+1}) - F_{m+1} S_{m+1} = (p_m+S_{m+1})^2 - S_{m+1}^2 - p_m^2$$

$$F_m p_m + F_m S_{m+1} - F_{m+1} S_{m+1} = p_m^2 + S_{m+1}^2 + 2p_m S_{m+1} - S_{m+1}^2 - p_m^2$$

or

$$p_m = \frac{S_{m+1}(F_m - F_{m+1})}{2S_{m+1} - F_m} \qquad 1 \leq m \leq n \qquad (2)$$

If p_n is known, then (2) can be used successively to find p_{n-1}, p_{n-2} and so on. However, we can arrange (2) so that first we can find p_1, and having p_1, we can find p_2, and so on. Since p_i's are probabilities, we have

$$S_{m+1} = 1 - R_m = 1 - R_{m-1} - p_m.$$

Replacing this in (2), we find:

$$p_m = \frac{(1-R_{m-1}-p_m)\,(F_m-F_{m+1})}{2(1-R_{m-1}-p_m) - F_m} \qquad 1 \le m < n \qquad (3)$$

In (3), we assume that $R_0 = 0$. Each p_i, i=1,2,3,...,n-1 can be successively computed from (3) by solving a quadratic equation. Later in this chapter, we return to this derivation for more comments.

We now examine the ability of the AØ inversion model to predict the fault rate behavior of real programs. We expect to get much improvement over the previously mentioned frequency method. Indeed, by inspecting Figure 3.3, we can see the success of the model. In this figure, the solid lines represent the fault rate curves of WATFIV programs under MIN and LRU algorithms. Using the AØ inversion technique, we construct an independent reference model based on the same program. The MIN and LRU fault rate which are produced by the model are shown by dotted lines on the same figure. As we expected, the MIN fault rate curve belonging to the model closely follows the MIN fault rate curve of the actual program for a wide range of memory sizes. It is interesting, however, that even the LRU fault rate curves of both the model and the actual program are fairly close together. The success of the model becomes more significant if we compare Figure 3.3 with Figure 3.2 to see the amount of improvement over the frequency method. This demonstrates the fact that by using an appropriate method, we can build substantial predictive power into a simple independent reference model.

Another way of looking at the fault rate behavior of the model, with respect to the actual WATFIV program, is to compare the observed histogram of LRU stack distances generated by each of them. In Figure 3.1, both histograms are shown.

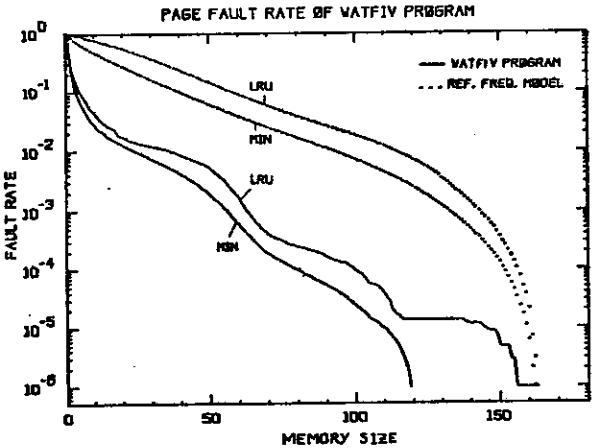


Fig. 3.2

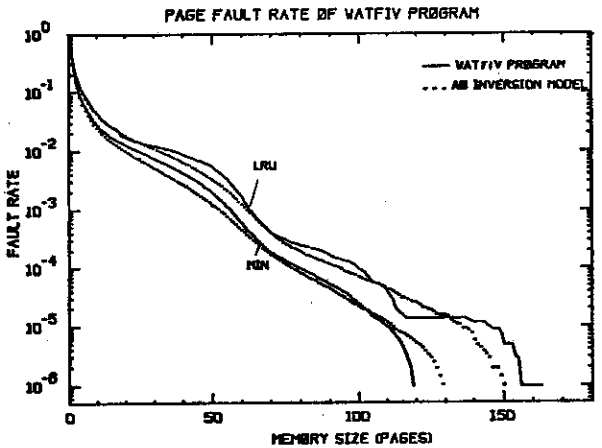


Fig. 3.3

It is interesting to inspect the set of reference probabilities which are obtained by the AØ inversion model. We can get a better insight into the structure of this model by comparing these reference probabilities with the reference probabilities which might have been obtained if we had used the simple frequency method. In Figure 3.4, the two sets of reference probability densities based on the WATFIV compiler are shown. The horizontal axis is the page number and the vertical axis is the probability weight.

In the frequency method, the reference probabilities are found by taking the global averages on the entire string. In the averaging process, most of the information about the regional characteristics of the string is lost. [Along the same lines, we have tried other approaches to get a better representative set of probabilities. One method we used was to divide the trace into intervals and find the relative reference frequencies in each interval, and order each set and combine over all intervals. The results, which are not reported here, showed only a slight improvement over the usual frequency method].

In the AØ inversion model, a completely different approach is taken and the reference probabilities which are obtained in this case bear no direct relation with the relative reference frequency of each page in the actual program. In Figure 3.4, we note that the AØ inversion model produces a reference probability mass distribution which has a distinctive resemblance to the fault rate curve of the program upon which the model is based. We can see that some important information, such as the memory sizes where the actual fault rate changes curvature, is precisely carried over to the corresponding page numbers in the reference probability curve.

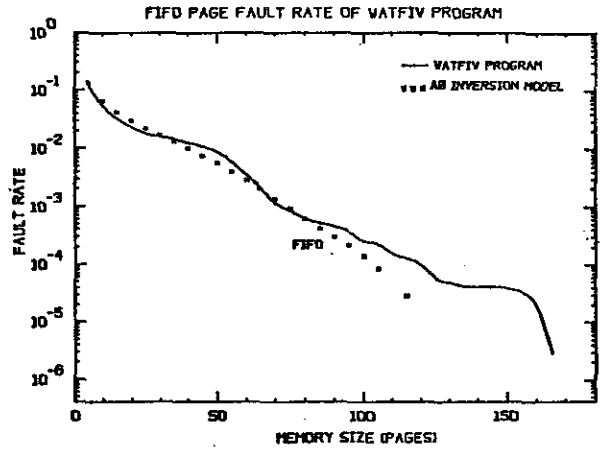


Fig. 3.5

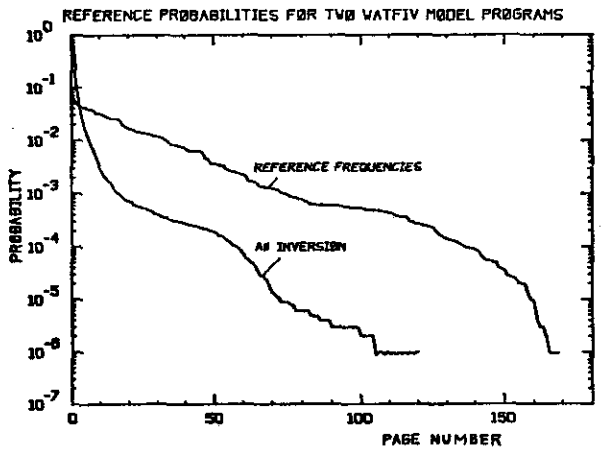


Fig. 3.4

Generally, the Aϕ inversion model assigns large probability mass to a few top pages (i.e., pages with the lowest subscript) and the remaining pages receive probability weights in sharply decreasing quantities. One can interpret the top pages (e.g., the first 20 pages in Figure 3.4) as the set of the current locality pages of the program. References to these pages are mostly favored in the reference string generated by the model. The pages which receive the least probability weights can be imagined to produce the instances corresponding to locality transitions in the actual program. The remaining pages which receive probability weights between the above two extremes can be considered to contribute to the small variation of the locality sizes in time.

We can support our claim about the predictive power of the Aϕ inversion model by presenting more evidences about the success of the model. For another replacement algorithm, we test the behavior of the model under the FIFO paging algorithm. In Figure 3.5, the solid line is the fault rate of the actual WATFIV program versus memory sizes under FIFO algorithm. In the same figure, the dotted line represents the fault rate curve of the model under the same algorithm. We can see that the model is capable of predicting the average fault behavior of the program on the lower range of memory capacities. For very large memory sizes the dotted line slightly drifts away from the solid line. The behavior of the model in this region can be partially accounted for by anyone of the following reasons. Since we simulate the model, in this case the sampling error becomes significant for large memory sizes. The other source of the error is the inaccuracy in defining the tail (i.e., the pages with the lowest subscripts) page reference probabilities. We shall return to the problem of finding the tail probabilities later in this chapter.

The performance of the model using some other programs and with different page sizes, and the ability of the model to predict the actual working set sizes are discussed in the next section.

3.4 TEST RESULTS

In a series of experiments, we are going to present more data for validation of the model. We have constructed an Aϕ inversion model based on the page reference trace of several real programs. These programs include a trace of a WATFIV compiler, a FORTRAN program called WATEX, an APL program, and the trace of a program to calculate the Fast Fourier Transform, called FFT, of a set of data points.

3.4.1 Fault Rate Prediction

In Figures 3.6, 3.7, 3.8 and 3.9, the fault rate curve of each model under the MIN and LRU algorithms are compared with those of the corresponding actual programs. In each figure, the solid lines belong to the actual program and the dotted lines represent the data points from the model. We note that in each case the model is able to predict the LRU fault rate of the actual program in a satisfactory way. All these models are especially successful in the range of lower memory sizes. It is significant, for instance, to note that the Aϕ inversion model has been able to capture the odd behavior of the FFT program as can be seen in Figure 3.8. We observe that the fault rate curves of the model breaks in exactly the right point (memory size) in this case. This is a rather promising result which shows that the technique can be used successfully to model highly structured program behaviors.

In the range of relatively large memory sizes, since the fault rate values are very small, the behavior of the model in these regions matches less well with the actual situation.

In Figure 3.9, the same results are demonstrated for the case of the WATFIV program with page sizes which are twice as large as the earlier case (Figure 3.3). It is notable that the model is robust to changes in page sizes.

3.4.2 Average Working Set Size Prediction

The degree of spread of the references over the program address space is of interest in efforts to characterize a computer program. A measure of the scattering of references can be obtained by observing the size of and changes of program working sets. The working set [11], WS(t,T), at time t, is the set of peges addressed in the past T references. The size of this set is denoted by ws(t,T). The window size T is the working set parameter. The measured working set sizes can be averaged over the entire program trace and lumped into one number, called the average working set size, ws(T).

The average working set size can be defined for the references generated by the model. Since the probabilistic structure of the model is known, the expected working set size can be readily obtained by a probalistic argument. Let $[p_1,p_2,p_3,\cdots,p_n]$ be the parameters of the Aϕ inversion independent reference model. The expected working set size with parameter T is equal to the probability that a page is in the working set summed over all pages. A page is in the working set if it has been referenced at least once in the past T units of time; therefore,

$$ws(T) = \sum_{i=1}^{n} [1 - (1-p_i)^T] \qquad (4)$$

We can now examine the capability of the model in predicing the average working set sizes of actual programs. In a series of experiments, we have measured the average working set sizes of a number of programs with different window sizes. For each actual program, the average working set sizes of the corresponding Aϕ inversion model is calculated from (4). The results are illustrated in Figures 3.12 and 3.13 for the WATFIV, APL and FFT programs. In each figure, the horizontal axis is the window size in terms of address reference units and the vertical axis is the average working set size. The solid lines are obtained from the measurements on the actual programs and the dotted lines are computed from the parameters of each model.

We can see that the predicted average working set size values derived from the model are strikingly close to those of the actual programs. This result demonstrates the capability of the Aϕ inversion model in capturing an important feature of the address reference behavior of real programs.

Once the average working set size is known, the fault rate values under WS algorithm can be obtained. For the independent reference model, the WS fault rate is equal to the probability that a page hasn't been addressed in the past T references and that it will be addressed in the next reference summed over all pages, i.e.,

$$E_{WS}[T] = \sum_{i=1}^{n} (1-p_i)^T p_i$$

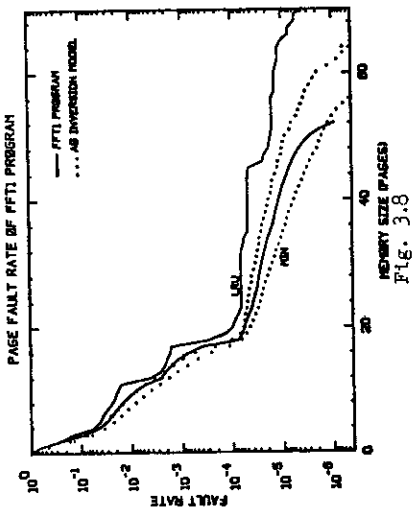


Fig. 3.8

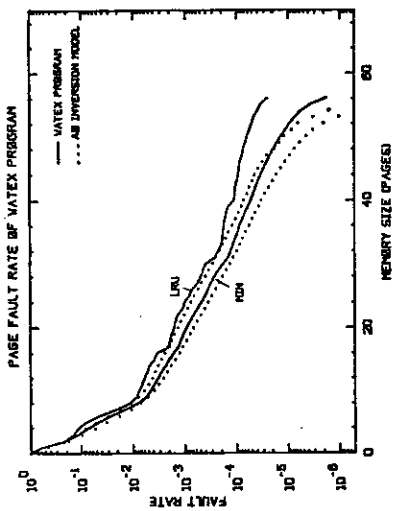


Fig. 3.6

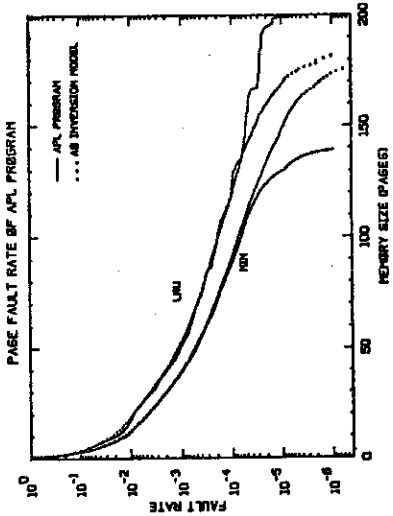


Fig. 3.7

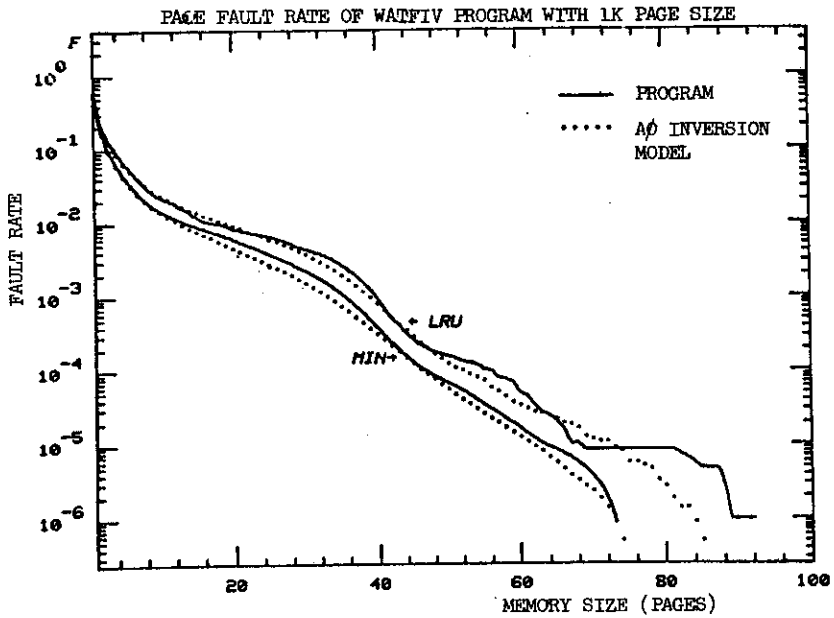


Fig. 3.9

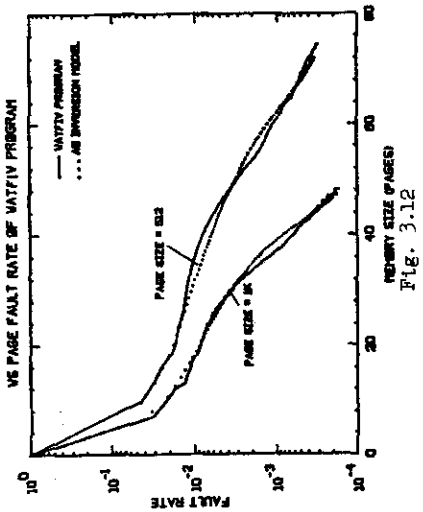


Fig. 3.12

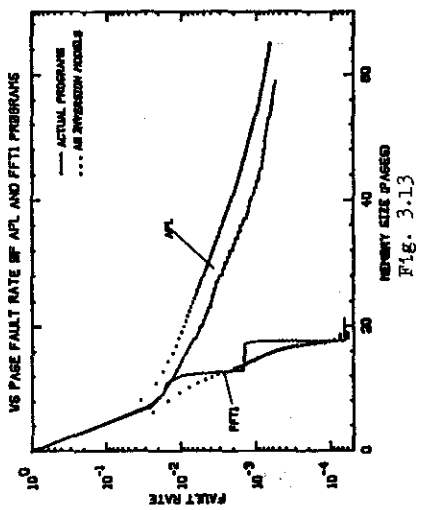


Fig. 3.13

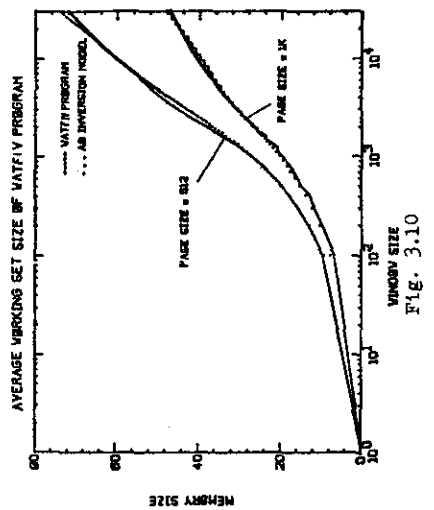


Fig. 3.10

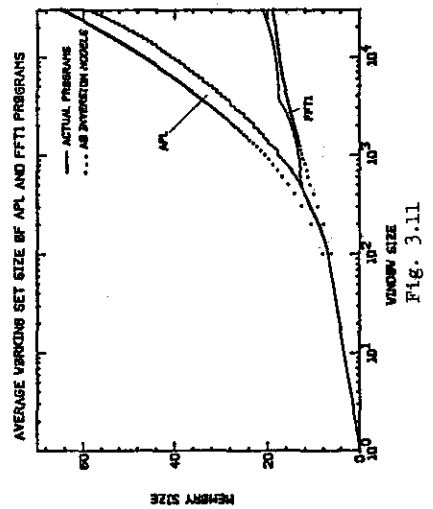


Fig. 3.11

In Figures 3.12 and 3.13, the WS fault rate of the WATFIV program with two different page sizes, and the WS fault rate of APL and FFT programs are shown. In each figure, the WS fault rate probability of the corresponding AØ inversion model is shown by dotted lines. The horizontal axis is the average working set size (or the average memory size used) and the vertical axis is the fault rate. The fit of the points obtained from the model to the points measured on the actual programs, basically reflects the results illustrated in Figures 3.10 and 3.11.

In Figure 3.11, we notice that the model slightly overestimates the working set size of the APL program. An explanation for this behavior will follow in the next part.

3.4.3 Comparison with LRU Stack Model

We have defined the LRU stack model for the sequence of page references earlier in this chapter. This model is strongly bound to the observed LRU stack depth distribution of the programs. The long run fault rate of LRU stack model, under LRU algorithm, converges to the LRU fault rate of the program upon which the model is based. This property is built into the LRU stack model by setting the stack depth hit probabilities $\{d_i\}$, i=1,2,...,n of the model equal to the relative observed stack distances generated by an actual program. It is interesting to investigate the behavior of LRU stack model under systems other than LRU.

Similar to our earlier set of experiments, the trace of several programs have been used to construct the empirical LRU stack distance distributions. In each case, an LRU stack distribution is used to construct the corresponding LRU stack model. In order to compare the optimal fault rate behavior of an actual program with the respective LRU stack model,

the MIN algorithm is used for both of them. We note that since the observed LRU stack distance densities are not monotone decreasing values, we don't expect that LRU would be optimal for the model.

In Figures 3.14 and 3.17, the result of the experiments on the WATFIV and APL programs using MIN and LRU algorithms are shown. The solid lines belong to the actual programs and the dotted lines represent data points from the corresponding LRU stack models. The LRU algorithm, as well as the MIN algorithm, were applied by a simulation run for the actual program and the model. Therefore, the discrepancy between the LRU fault rate curve of the model and the corresponding program gives a significant measure of the sampling error in the simulation of the model. The more interesting information in these figures is, of course, the behavior of LRU stack model under the MIN algorithm. We note that in both cases the models give a good prediction of the MIN fault rates of the actual programs. Like the A0 inversion model, the good fits are especially notable for lower range of memory sizes.

In Figures 3.15, 3.16, 3.18 and 3.20, the average working set sizes and the WS fault rate of WATFIV and APL programs are compared with the same terms in the respective LRU stack models. If we inspect Figures 3.11 and 3.18, we notice that both the Aϕ inversion model and the LRU stack model give up to about a 10% overestimation of the actual average working set sizes of the APL program for most values of window sizes. We can give an explanation for this by taking a closer look at the distribution of working set size of the APL program. In Figure 3.19, a histogram of the observed working set sizes for window size T=4000 units for this program is plotted. In this plot, we can distinguish three major peaks. Although this is not a typical working set histogram, nevertheless,

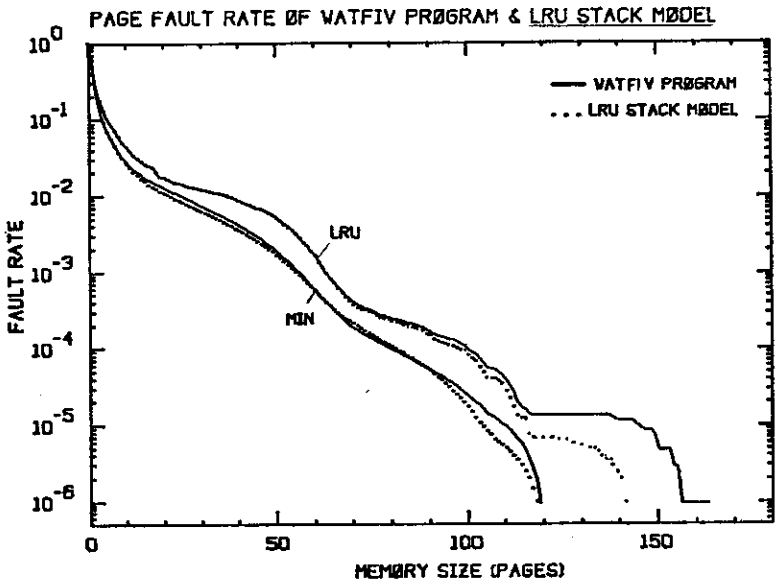


Fig. 3.14

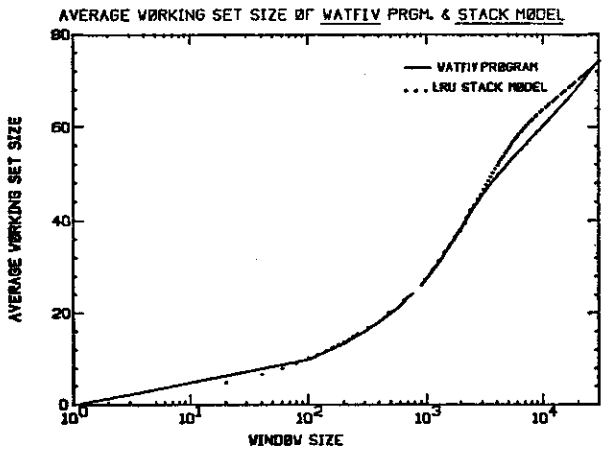


Fig. 3.15

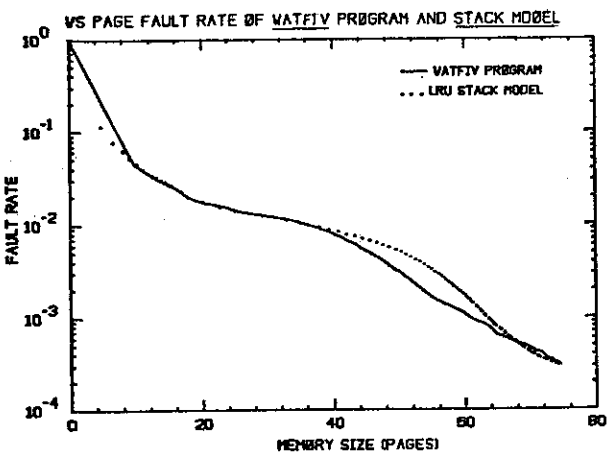


Fig. 3.16

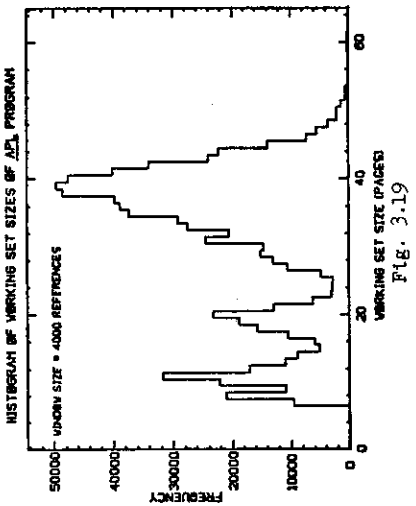


Fig. 3.19

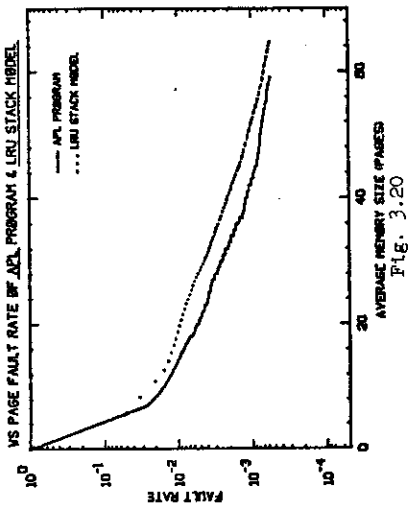


Fig. 3.20

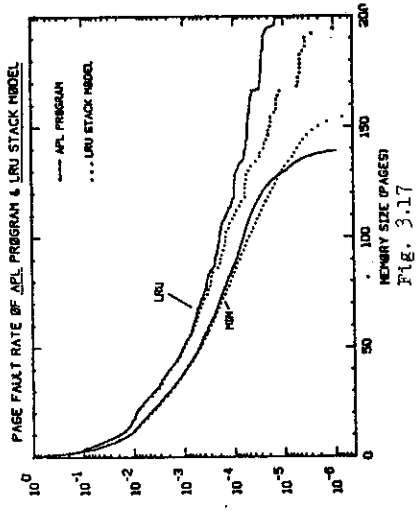


Fig. 3.17

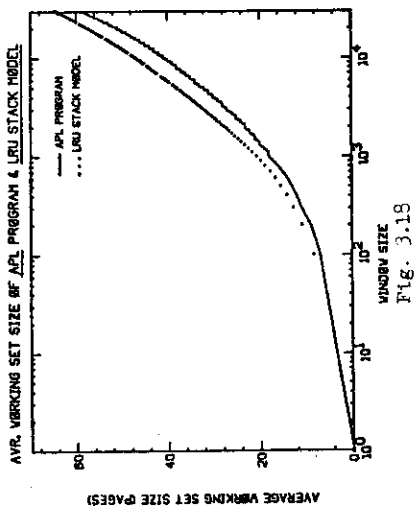


Fig. 3.18

programs sometimes do exhibit this behavior. Each peak can be associated with a large period of time which the program predominately spends in a locality which is different in size from other major localities. The frequent locality changes may also contribute to the clusters of fairly large working set sizes in the histogram.

Programs like APL which exhibit distinctive multiple locality regions give the illusion for the mechanisms which build the models (e.g., AØ inversion and LRU stack models) as being programs with fairly scattered reference patterns. The overestimation of the average working set sizes can be attributed to this mis-interpretation of the actual reference patterns.

3.4.4 Analytical Form for AØ Inversion Parameters

In this section, we examine a possible analytical function which fits the reference probabilities produced by the AØ inversion model. Our motivation here is to give an approximate compact form to describe the parameters of this model.

In Figure 3.21, three different sets of AØ inversion model reference probabilities are plotted on a log-log scale. In this graphical representation, we can see that large portions of each set of probabilities can be approximated by straight lines in the log-log domain. There is usually one major breakpoint in each curve. This breakpoint for the WATFIV model is quite distinct at page number 54. For the other program models, the breakpoints are not as obvious as in the case of the WATFIV model. For instance, for the WATFIV model we can roughly set this point at page number six.

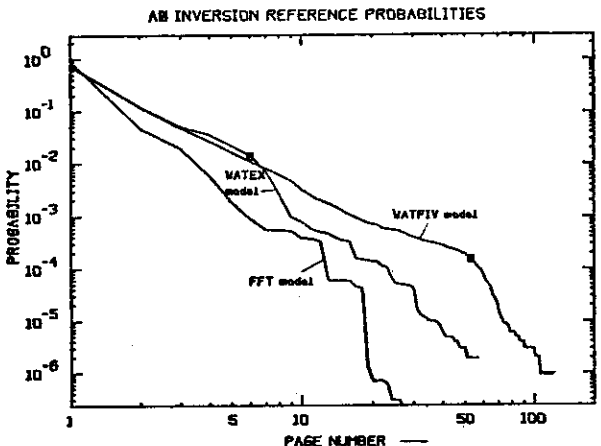


Fig 3.21

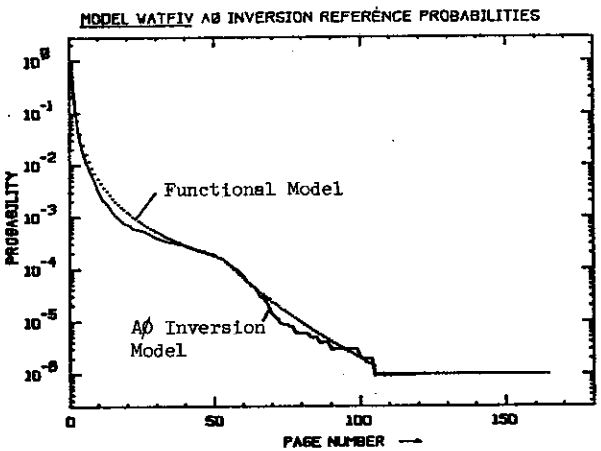


Fig. 3.22

Using the page number at the breakpoint, we can try to fit two or more power law functions for each segment. These functions and the page number at the breakpoint can approximately describe the parameters of themodel. For instance, the reference probabilities of the WATFIV AØ inversion model can be approximated by the following function:

$$p_k = \begin{cases} 0.7381k^{-2.135474} & 1 \le k \le 53 \\ 116x10^{6}k^{-6.8832} & 54 \le k \le 120 \\ 0.000001 & 121 \le k \le 168 \end{cases}$$

In Figure 3.22, the actual reference probabilities are compared with those found by the above function.

3.5 EXTENSIONS AND OTHER APPLICATIONS OF THE MODEL

In this section, we consider some possible extensions and applications of the AØ inversion model. We would like to point out here that inverting the AØ fault rate expression is not the only way to find the reference probabilities. For example, one might consider inverting the average working set size expression (4) and substituting for the ws(T) values the observed average working set sizes from an actual program.

One possible area for the extension of the AØ inversion model is the study of the management and organization of file systems in the storage hierarchies. A file can be thought of a block of data which is handled by I/O communication facilities as one integral part. In the context of earlier discussions, the file blocks can be considered as variable size pages.

A possible application of the model is the areas of program and data restructuring techniques. Restructuring can be used to increase some performance measure, such as access efficiency. Grouping of the

records which are referenced together is a common approach in restructuring efforts. The pages of a program can also be combined into one larger block, based on some frequency of reference criterion. For instance, assume there are n pages and we would like to combine each $k \le n$ pages into one block. One way of doing this is to combine the first k most referenced pages into the first block, the next k most referenced pages into the next block, and so on. We call this simple restructuring.

In order to find out the behavior of the model in this application, we have done some experiments on the trace of the WATFIV and APL programs. The pages of these programs are sorted in the order of the frequency of reference to each page. Then, each successive pair of pages in each list are combined together to create a larger page size twice the size of the original pages. For each program, before restructuring, we find the AØ reference probabilities $[p_1, p_2, p_3, \cdots, p_n]$. Next, we restructure each model so that first and second pages are combined, third and fourth are combined, and so on. (We recall that $p_1 \ge p_2 \ge p_3 \ge \cdots p_n$.) In this manner, we obtain new restructured models with $\frac{n}{2}$ parameters

$[p_1 + p_2,\ p_3 + p_4, \cdots,\ p_{(n/2)-1} + p_{n/2}]$ (n even).

We can now find the performance of the restructured programs under MIN and LRU replacement algorithms. These are shown by solid lines in Figures 3.23 and 3.25. Next, we find the performance of the restructured models under the same algorithms. We don't expect to get similar results because the parameters of the AØ inversion model are not directly related to the reference probabilities in each actual program. The dotted lines in Figures 3.23 and 3.25 demonstrate the performance of each model. We note that the models underestimate the fault rate of the restructured

programs by some factor. It is interesting to note that the restructured models underestimate the performance of restructured programs by a fixed factor over all memory sizes. For both the WATFIV and APL programs, these factors are 0.5. This is shown in Figures 3.24 and 3.26 where we multiply the fault rates of the restructured models by two, and plot the new curves by dotted lines. We note that for both programs now, the models follow the fault rate curves of restructured programs reasonably well and for APL we have a particularly good fit. The significance of this result is that we can now work with the model and obtain numbers which are off from the actual results by a fairly constant proportionality factor over all memory sizes. Determining this factor probably needs more experiments with different programs. It seems that its value should not have great variation among fairly homogenous representative programs.

In Figures 3.27 and 3.28, we have compared the effect of simple program restructuring on the fault rate behavior of WATFIV and APL programs. The solid lines show the performance of the restructured programs, as described earlier, and the dotted lines give the performance of unaltered programs with the natural doubling of the page size of each program. We can see that simple restructuring can have significant effect on the memory utilization in the case of the APL program, and has mixed positive and negative effect on the performance of the WATFIV program over different range of memory sizes.

Another feasible application area of this model is the study of read/write characteristics of the programs and the paging algorithms which take these operations into consideration. In the remaining part of this section, we point out the significance of the problem and give some suggestions for dealing with the problem.

In a paging system, an important task of the supervisor is to take note of the modified pages (i.e., write pages). This information becomes important when a page needs to be pushed out of the memory to make room for an incoming page. Clearly, if the content of the page has not been modified since it was first brought into the main memory, it is generally not necessary to copy it back on, say, the paging drum. This, of course, can save the I/O channel time and reduce the channel traffic. In this respect, in the study of paging algorithms the number of transfers produced under an algorithm can be considered a measure of performance. Thus, we can associate a cost or weight of one for a page which is brought into the memory and is not modified by the time it has to leave. Similarly, a cost of two is associated with a page which is brought into the memory and is modified by the time it has to be copied back to the backup memory. The performance of the algorithm is the total cost (total number of transfers) associated with each memory capacity.

The significance of the subject can also be seen when we study the operational specifications of some of the newly evolving memory technologies. At least in two of the promising technologies, namely, the bubble domain memory [22] and the electron beam addressed memory (EBAM) [17] systems, the write operation can be slower than read operation. For instance, in one prototype of an EBAM system the write speed is ten times slower than read. Both systems are aimed at providing reliable and large storage media and direct block access capability. Therefore, the need for data management similar to paging operation is feasible for a memory hierarchy which include any of the devices.

We now proceed to give some data on the number of transfers caused by some actual programs under two important replacement algorithms,

namely, LRU and MIN. We are also interested in the relation between the transfer rate (i.e., number of transfers/number of references) and the page fault rate at different memory sizes.

In Figures 3.29 and 3.30, the fault rate and transfer rate of the WATFIV and WATEX programs under LRU and MIN algorithms are shown. The fault rate curves are the same as those presented earlier. The transfer rate curves (dotted lines) are obtained using an efficient stack algorithm. For each memory size, when a page is brought into the memory, one transfer is counted. When the same page has to be pushed out, the modify bit of that page is examined to determine if that page has been modified. If it is modified, another transfer is counted. The dotted lines in the figures actually represent 0.5 x Transfer Rate Curves.

On the upper corner of the plots, the ratio of 0.5 x Transfer Rate/ Page fault rate for both algorithms are shown. We note that in each figure, these curves follow a fairly straight line for a major range of lower memory sizes. This indicates that a constant relationship can be approximated between the transfer rate and the fault rate for relatively small memory sizes. As the memory size becomes large, the ratio moves slightly upward in each figure, showing higher transfer rates relative to the fault rates. This behavior is expected because when the size of the main memory is large, the program pages tend to spend more time in the memory and, thus, the likelihood that a page is modified during its prolonged stay increases.

The AØ inversion model can be extended in this application in two ways:

a) A single new parameter q is defined which indicates the probability that a page reference is of modify (or write) type.

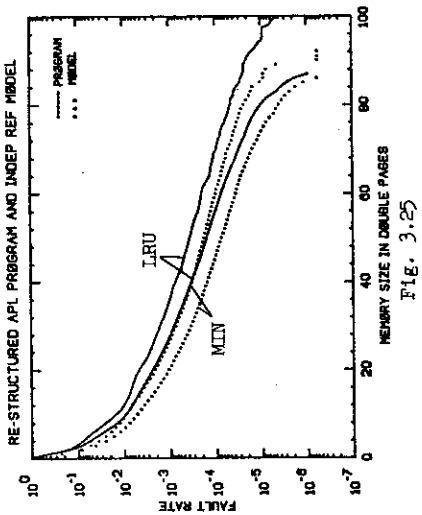


Fig. 3.25

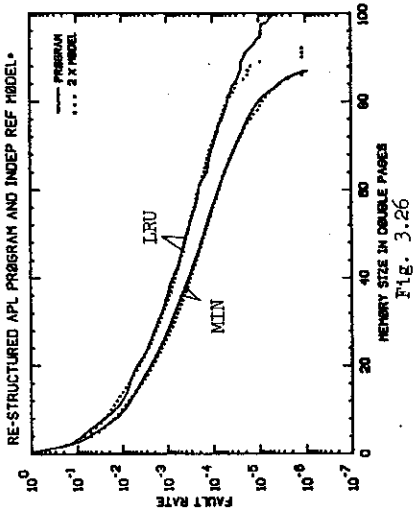


Fig. 3.26

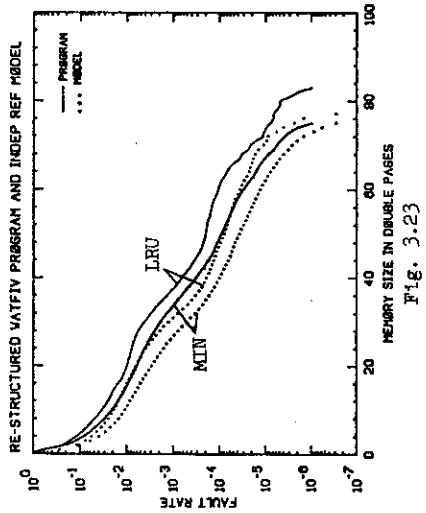


Fig. 3.23

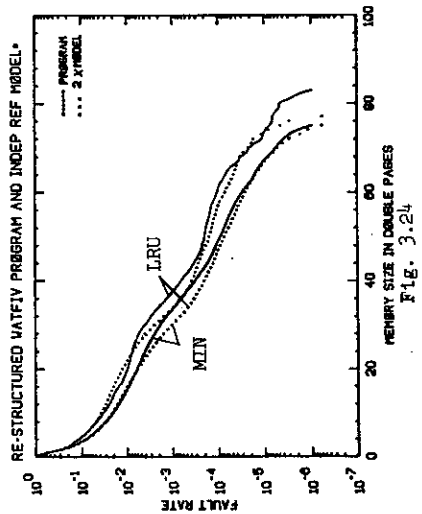


Fig. 3.24

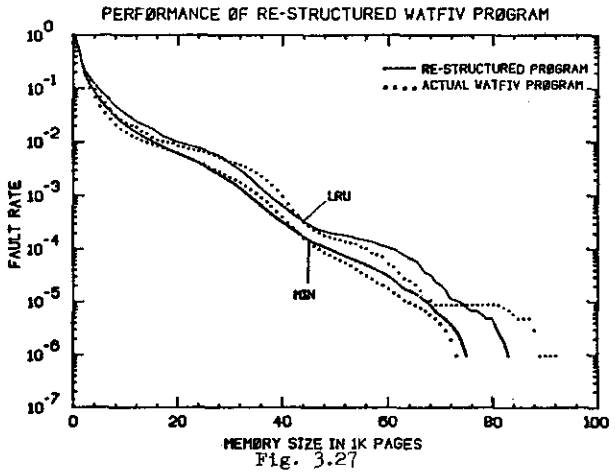


Fig. 3.27

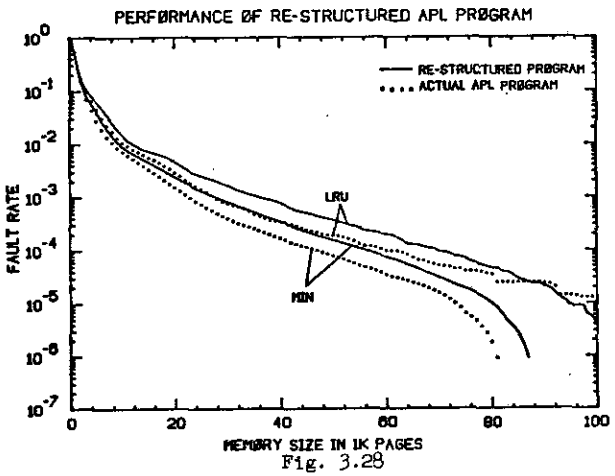


Fig. 3.28

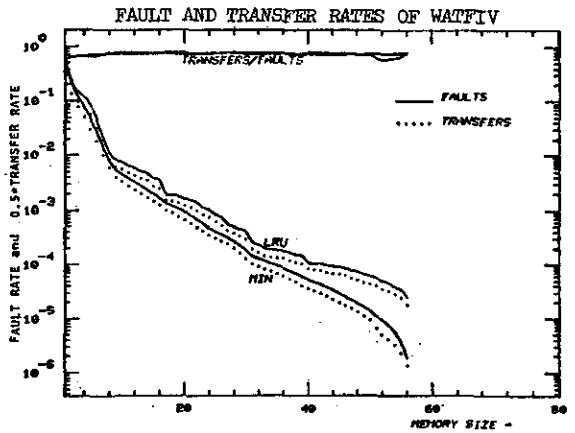


Fig. 3.29

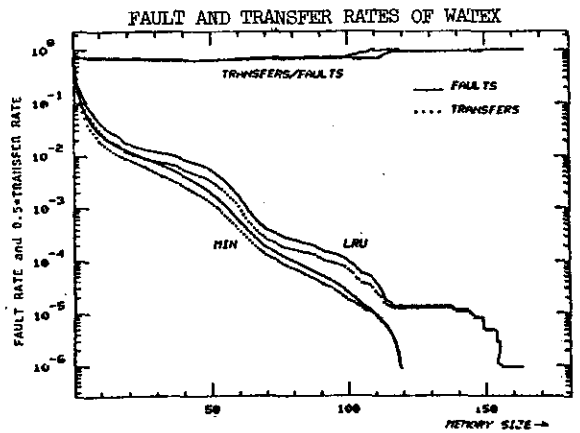


Fig. 3.30

Therefore, the AØ inversion model is characterized by n reference probabilities and one modify probability, as:

$$[p_1, p_2, p_3, \cdots, p_n, q] .$$

This approach can capture the constant relationship between the transfer rate and the fault rate in the region of smaller memory sizes.

b) The MIN transfer rate function can be used in much the same way as the MIN fault rate function was used, and we can obtain a set of n new parameters

$$[q_1, q_2, q_3, \cdots, q_n] .$$

These parameters, when added to the reference probabilities $[p_1,p_2,\ldots,p_n]$, give a comprehensive model which can characterize the read/write and the fault rate behavior of the actual programs.

3.6 PROBLEMS AND LIMITATIONS OF AØ INVERSION MODEL

In this section, we return to the subject of finding the parameters of AØ inversion model. Essentially, we may encounter two kinds of problems in finding the reference probabilities. The first problem deals with solving the recurrence relations (3) and the second problem is related to the tail probabilities.

3.6.1 Problems with Finding the Reference Probabilities

We recall that the MIN fault rate of an n page program are substituted for F_i's in the relations (3) and, subsequently, the equations are solved for p_i's. It is theoretically quite possible that a set of F_i's, $1 \le i < n$ and $F_i > F_j$ for $i < j$, be defined, for which there is no real

valued solution for p_i's. In fact, it is much harder to come up with some empirical values for F_i's where we can solve for p_i's.

The case where we can't solve the equations signifies the situation where there is no independent reference model with its fault rate under optimal algorithm exactly equal to those values that we have substituted for F_i's.

Our experiments in using the actual program traces show that for traces of reasonable length, we usually can find fairly accurate values for p_i's. However, when the measured MIN fault rate values are such that the equations (3) cannot be solved for all values of p_i's, we can find approximate values for these parameters by using the relations:

$$p_i = F_i - F_{i+1} \qquad (5)$$

Once a p_i is found in this way, we can try to use relations (3) to find the successive parameters. For instance, in the FFT1 program, p_1 was found using (5) and the remaining probabilities were obtained by (3). The model seems to function properly even with approximate reference probabilities obtained from the above procedure.

3.6.2 Problems with Tail Probabilities

Consider a program with n pages. Denote by F_m the fault rate of the program with memory size m under the MIN algorithm. When m becomes large, it is possible that for some memory size n' the observed F_i, $i=n', n'+1,\ldots,n$ will become zero. Here we assume that initial faults, due to the initial loading of the memory, are excluded from the total fault counts. Since F_m is the minimum fault rate with memory size m, then for any other fixed memory size paging algorithm the lower bound on the maximum memory size, n", for which it produces non-zero fault rate,

is equal or greater than n'. For instance, for the WATFIV program, n'=120 and n"=164 (under LRU) and for the WATEX program, n'=n"=57 under MIN and LRU.

The point is that the Aϕ inversion method, which uses the MIN fault rate of the programs, can give us only n'-1 non-zero reference probabilities. Therefore, we get a model with n'-1 parameters and, clearly, when we use the model as it is, the pages n' through n never get referenced. For the lower range of memory sizes, the model with n'-1 parameter still gives satisfactory results. This is because in the practical cases, the reference probabilities close to the tail of the model are relatively very small. However, the behavior of the model can greatly be degraded for large memory sizes if we don't extend the tail probabilities to get a full size n parameter model.

Extending the tail probabilities to get n non-zero reference probabilities is still an open question here. We have chosen an ad-hoc method to get around the problem; we have simply extended the last non-zero reference probability so that $p_{n'-1} = p_{n'} = \cdots = p_n$. Then we need to normalize to get a consistent set of probabilities. This solution has almost no effect on the performance of the model for small memory sizes, but it has greatly improved its performance in the region of large memory sizes.

3.6.3 Execution Characteristics of the Model

Thus far in this paper, we have demonstrated the capabilities of the Aϕ inversion model to capture most of the long run characteristics of actual programs. However, we point out the fact that this model, and in this matter none of the models which are mentioned in this report,

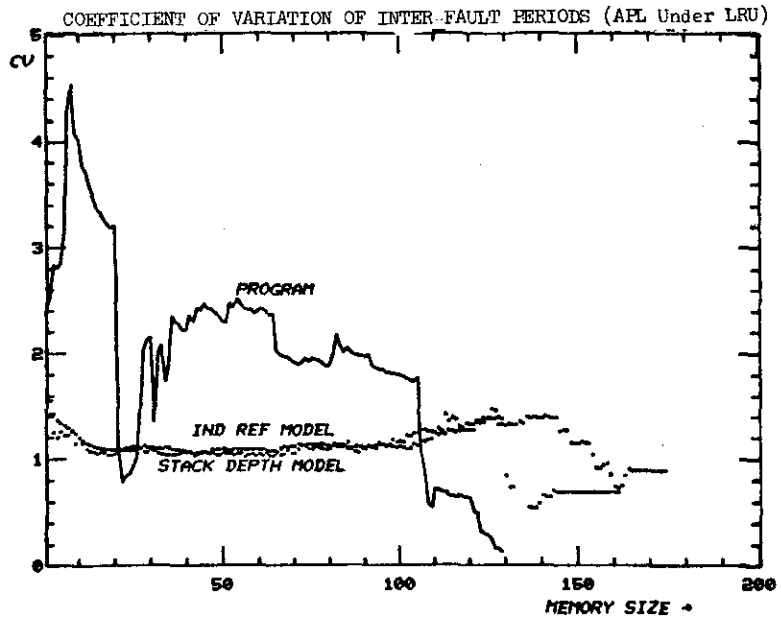


FIGURE 3.31

are able to capture some of the complex non-stationary features of actual programs.

In the applications where the variance of quantities, like the inter-fault distances, are critical, the Aϕ inversion model would be of limited use. To illustrate this point, we compute the coefficient of variation of actual inter-fault distances for the WATFIV program under the LRU algorithm, and for different memory sizes (coefficient of variation, C.V., is equal to standard deviation/mean and is a scale of departure from geometric or exponential distribution). In Figure 3.31, the horizontal axis is the memory size and the vertical axis is the C.V. of the actual LRU inter-fault values for the WATFIV program. We can see that, in this case, C.V. is generally greater than one. The fluctuation of the C.V. of inter-faults with different memory sizes is notable. When the memory becomes very large, the computed values become erratic because of insufficient samples.

The coefficient of variation of the inter-fault distances for the Aϕ inversion model, as well as the LRU stack model, is equal to one. This is because at each reference there is a fixed probability of a fault independent of previous references. Thus, the inter-fault periods are geometrically distributed for both models and the C.V. of this distribution is equal to one.

3.7 CONCLUSION

Constructing program models can be a compact way of characterizing the page reference behavior of actual computer programs. In this chapter, we have presented the technique to build an Aϕ inversion independent reference model, based on the actual MIN fault rate of a page reference

trace. We noted that the independent reference model preserves the relative fault rate of actual program traces under MIN and LRU algorithms. Thus, the Aϕ inversion model should be capable of predicting the true LRU and FIFO fault rates of real programs for different main memory sizes. We presented the results of experiments on several programs to validate the model.

The Aϕ inversion model is also successful at predicting the average working set size and the WS fault rate of programs for a wide range of window sizes.

We have seen that when an LRU stack model is constructed, based on the actual LRU distribution of a reference string, it can reasonably predict the MIN fault rate of the same program. The performance of the LRU stack model and the Aϕ inversion model have been compared by presenting experimental data.

The analytical tractability and the simple probability structure of the Aϕ inversion model make this model a convenient tool for the analysis and evaluation of virtual memory systems and the performance of CPU's with high speed buffers.

We have shown the potential of expanding the model into the areas of simple program restructuring techniques and the evaluation of memory hierarchies with unequal read/write costs.

When a program has several very distinctive locality regions, the Aϕ inversion model, as well as the LRU stack model, overestimates the average working set size by a small percentage. However, the prediction accuracy of the average fault rate under fixed memory size algorithms are virtually uneffected.

The problem of finding the tail probabilities has been dealt with here in an ad-hoc manner. More elaborate treatment of this subject should justify the desired accuracy of the model under very large memory sizes, where the effect of these probabilities are most noticeable.

The independent reference assumption on the successive references of a program is against our intuition and the actual observations. However, we have demonstrated that by putting enough structure into the model, we can obtain a powerful model which produces realistic results, and can be used effectively in the analysis, simulation and evaluation of several problems in the area of memory management techniques.

3.8 BIBLIOGRAPHY

1. "The Cray-1 computer preliminary reference manual," CRAY Research, Inc.
2. "IBM System 360 Model 85 functional characteristics," Form A22-6916.
3. "Introduction to the IBM 3850 Mass Storage System (MSS)" Form GA32-0028-1.
4. Aho, A.V., Denning, P.J., Ullman, J.D., "Principles of optimal page replacement, J. of ACM 18, 1 (January 1971), pp 80-93.
5. Belady, L.A., "A study of replacement algorithms for virtual storage computer, IBM System J., 5, 2 (1966), pp 79-101.
6. Boyce, J.W., "Execution characteristics of programs in a page-on-demand system, Comm. ACM 17, 4 (April 1974).
7. Coffman, E.G., Denning, P.J., "Operating system theory," Prentice Hall, Englewood Cliffs, New Jersey (1973).
8. Conti, C.J., "Concepts for buffer storage," Computer Group News (March 1969).
9. Denning, P.J., "Thrashing: its causes and prevention," AFIPS Conf. Proc., Fall Joint Computer Conference, 33, (1968), pp 915-922.
10. Denning, P.J., "Virtual memory," Computing Surveys, 2, 3, (1970).
11. Denning, P.J., "On modeling program behavior," AFIPS Conf. Proc., Spring Joint Computer Conference (1972), pp 937-944.
12. Denning, P.J., Savage, J.E., Spirn, J.R., "Models for locality in program behavior," TR 107, Computer Science Laboratory, Dept. of Electrical Engineering, Princeton University (1972).
13. Denning, P.J., Schwartz, S.C., "Properties of the working set model," Comm. of ACM, 15, 3 (March 1972).

14. Franaszek, P.A., Wagner, T.J., "Some distribution free aspects of paging algorithm performance," J. of ACM, 21, 1, (Jan. 1974), pp 31-39.

15. Franklin, M.A., Gupta, R.K., "Computation of page fault probability from program transition diagram," Comm. of ACM, 17, 4, (April 1974).

16. Ghanem, M.Z., Kobayashi, H., "A parametric representation of program behavior in virtual memory systems," IBM Research Report RC 4560 (October 1973).

17. Kelly, J., "The development of an experimental electron beam-address memory module," Computer IEEE 8, 2 (February 1975).

18. King, W.F., "Analysis of paging algorithms," IBM Watson Research Center, Report RC-3288 (March 1971).

19. Lewis, P.A.W., Shedler, G.S., "Empirically derived micromodels for sequence of page exceptions," IBM J. of Research and Development (March 1973).

20. Shedler, G.S., Tung, C., "Locality in page reference strings," SIAM J. on Computing 1, 3 (September 1972).

21. Watson, R.W., "Time sharing system design concepts," McGraw-Hill, N.Y. (1970).

22. Ypma, J.E., "Bubble domain memory systems," AFIPS Conf. Proc., (1975), pp 523, 528.

CHAPTER 4

QUEUEING ANALYSIS OF THE INTERACTION OF PAGE SCHEDULING AND DEVICE SCHEDULING

4.1 INTRODUCTION

Multiprogramming on virtual memory computers has been with us for more than a decade. It has become a common computing environment on many large and, recently, on small computers. Programs running in such systems compete with each other for resources, such as main memory, processing time and IO devices. The operating system has the complex task of managing the affairs of the system. They include the initiation of jobs, scheduling and assignment of resources, supervising the execution of the jobs, etc. Performance has always been a critical issue here. The performance goal is usually to have a balanced and efficient utilization of the resources under the constraint that an acceptable service level be provided for the diverse computing requirements of the community of users. A wide range of techniques have been used to study and, therefore, to improve the performance of these systems. They include direct measurements, modeling, simulation and a wide range of analytical techniques. In this regard, some questions have been answered and many others need to be explored further.

In this chapter, we are concerned with a part of the system which consists of the central processor with its memory and the paging device. These two fairly independent units interact strongly with each other to meet the paging requirements of active programs.

The basic assumption is that the main memory is partially loaded with a subset of the pages of each active program. The number of active programs determine the degree of multiprogramming. The remaining pages of

each program reside in fast auxiliary storage, like a drum. A program runs on part of its address space which is in the main memory. When a reference is issued to a location outside this region, the execution of the program will be delayed until the page which contains this address is fetched into the main memory.

A two-stage cyclic queue will be used to model the structure of the system. Using the actual and approximated paging behavior of programs, we will study the performance of the model with different memory allocation policies and scheduling disciplines. Quantities like utilization (efficiency) of the devices, job completion rates and job waiting times will be considered. The robustness of the results, with respect to different assumption, will be investigated.

4.2 MODEL

A multiprogrammed CPU with its paging device (here, mostly called IO device or IOD) is modeled by a two-stage cyclic queue with fixed number of customers. The number of customers is determined by the degree of multiprogramming (Figure 4.1).

Actual systems could be more complex than this scheme. For example, we can have more than one IO channel which is essentially a communication link between the CPU and the auxiliary storage media. In this case, the IO requests can be routed to different IO devices. Nonetheless, the above model captures the basic structure of the interaction of the CPU with its paging device. Using this compact model, we are able to interpret the results without getting involved in details and questionable assumptions. The goal of this study is to explore the underlying nature of the interaction of the page scheduling and device scheduling under fairly realistic typical program behavior assumption.

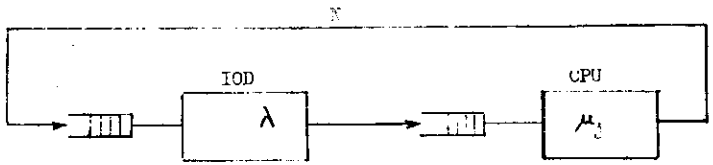


FIGURE 4.1 Two Stage Cyclic Queue Model with N Customers

Cyclic queueing models have been used by several other authors in this area. In [11], a Markov model for the sequence of LRU stack distances is given. Then a two-stage cyclic queue model with two statistically identical programs and equal memory partitioning is considered. By changing the parameters of the Markov model, CPU execution intervals with different variances are obtained. The numerical results suggest that a lower CPU utilization is obtained when the variance of execution intervals is increased. In [9], a single server queue with feedback and $N > 1$ customers (programs) is used to model a paging machine with variable partitioning memory. The customers are N statistically identical programs. These programs are characterized by the distribution of their LRU distance values. When a program experiences a page fault, one page is added to its main memory space at the expense of stealing one page from another program. The long run average execution intervals and the distribution of the number of page frames allocated to a program are sought. In [4], the authors consider a multiprogramming system with variable memory partitioning where queueing delays are neglected. Carrying out an average value analysis over the time epochs during which the memory allocation is fixed, they basically conclude that the mean processing efficiency is higher and the mean page fault rate is lower compared to the case where a fixed partitioning scheme is used. For N=2, the numerical results are presented. In [2], the page frame allocation strategies among competing processes in mtuliprograming systems are investigated. The basic performance measure is taken to be the cost of allocated page frames in main memory and the contention on the paging device. The average waiting times of the requests to the paging drum as a function of load on the drum and the number of active jobs is found by simulating a two-stage cyclic queue. The execution

intervals of the jobs on the CPU is drawn from a common exponential distribution model. A rate of accrual value for the whole system under each memory allocation scheme for N jobs is defined. This value gives a measure of how efficiently the available main memory is used taking into account the average delays of the paging device, the progress rate of the jobs, and page frame demand of each program present in the system. They conclude that the optimal allocation policy for two identical programs, when there is sufficient available space, is to divide the main memory evenly between them. For non-identical programs, they suggest that the main memory should be divided in such a way that the page fault rate of the jobs become equal. In [5] and [13], the authors use a two-stage cyclic queue to model the CPU and its IO device in their work. In [5], the average waiting time in a SLTF filing drum and the central processor utilization versus the ratio of transmission time to the computing time are illustrated. In [13], under the assumption that the mean execution time between page faults is a linear function of the allocated main memory size, the optimal memory partitioning, in terms of CPU utilization, is found. A processor sharing scheduling policy is assumed for the CPU. In [15], a two-level cyclic queue is used to study the effectiveness of the HASP Execution Task Monitor (OS/360) which gives the tasks a preemptive execution priority in the inverse order of their CPU usage history.

The model that we have considered here works as follows: There are N customers (programs, jobs) in the system. When a job is ready to receive CPU service (ready to run on the CPU), it will run until it experiences a page fault. The job with its request for the missing page will join the IOD queue. When the service of the job in IOD completes (equivalent of transferring the missing page from the drum to main memory), it will

join the CPU queue and is ready to resume its execution. CPU and IOD can work independently. The overhead of switching the jobs in the CPU is neglected. There is only one processor (server), the CPU. The main memory is divided into N fixed partitions $m_1, m_2, \ldots, m_N$, where m_i is the main memory space of job i and $\sum_{i=1}^{N} m_i = M$. Each program runs in its fixed allotment and no sharing of the pages is allowed (in one large computer installation, the shared pages account for less than 15% of all pages). By changing the amount of main memory which is assigned to each job, we can manipulate the mean CPU service time (between page faults) of that job on that processor.

We can either simulate the model or solve it directly. For simulation, we can use the reference traces of actual programs to drive the simulation. In the latter case, an LRU replacement rule will be used to manage the pages of the programs. For solving the model, we make some simplifying assumptions about the program behavior.

The extent of getting useful results from the model depends on how realistic we make the assumptions about the paging behavior of the programs, scheduling disciplines, the distribution of page transfer time, etc.

4.2.1 Program Paging Behavior

Program paging behavior has been extensively discussed in the previous chapters. We will extend some of the relevant points here.

Assuming a fixed partitioning of the memory, the mean service time for a job with memory allotment m is assumed to be equal to the mean interfault times (i.e., the time between two successive page faults). For each job, the mean interfault time is a function of the main memory space m and

the replacement rule which is in effect. We denote this function by g_m, and will assume a suitable replacement rule. For most replacement rules, this function is non-decreasing for increasing values of m. In Figure 4.2, g_m functions are plotted for the WATFIV and APL programs under the LRU algorithm. These curves are obtained by taking the reciprocal of page fault rate functions, f_m, for each case, i.e.,

$$g_m = \frac{1}{f_m}, \quad m=1,2,\ldots,M.$$

For our model, we need some mechanism to generate the interfault periods for a given memory allotment, m. In simulating the model, we can actually monitor real programs and measure the interfaults directly. This approach can sometimes be very costly. For analysis and more efficient simulation runs, we can use a model for program paging behavior. Later in this chapter, we will compare the results obtained from both methods.

Let $g_m(x)$ be the interfault density function for a given memory size m, i.e.,

$$\Pr[\text{interfault} = x] = g_m(x)$$

For the independent reference model, and under any replacement rule (e.g., OPT, LRU, FIFO, RR) where the page fault rate converges to some value f_m, the steady state interfault intervals have a geometric distribution with parameter f_m, i.e.,

$$\Pr[\text{interfault} = x] = f_m(1-f_m)^{x-1}.$$

For the LRU stack model, under the LRU algorithm, $g_m(x)$ is geometrically distributed with rate $f_m = \sum_{i=m+1}^{M} d_i$,

where M is the program size and

$$d_i = \Pr[\text{LRU distance}=i] .$$

These two models have been shown to be good models for program paging behavior [Ch. 2 and 3]. In both models, the distribution of $g_m(x)$ has a simple form. If the paging behavior of the programs are assumed to follow one of these models, then a natural assumption for the interfault distribution in continuous time is exponential.

For the geometric (or exponential) distribution, the coefficient of variation (C.V.) is equal to one. The C.V. of interfaults from measurements on actual programs has generally values higher than one. In Table 4.1, the coefficient of variation of interfaults for different memory sizes for two programs are shown. In Figure 4.3, the survivor functions of the actual interfault distribution with two different main memory sizes are shown. In the same figure, the survivor function of a fitted exponential distribution for each of the actual observations are also plotted.

The results show that the coefficient of variation of the actual interfault periods is generally higher than one, and the exponential fit is not a perfect choice in this case. We shall take this fact into consideration in simulation of the model when we use the actual trace of programs to run the simulation. Later in this chapter, we shall compare the results of the analytical approach, where we assume exponential service times, with the results of simulation where more realistic program behavior considerations are possible.

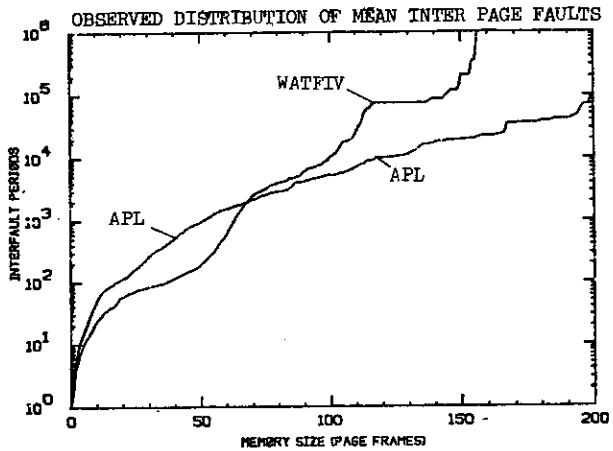


Fig. 4.2

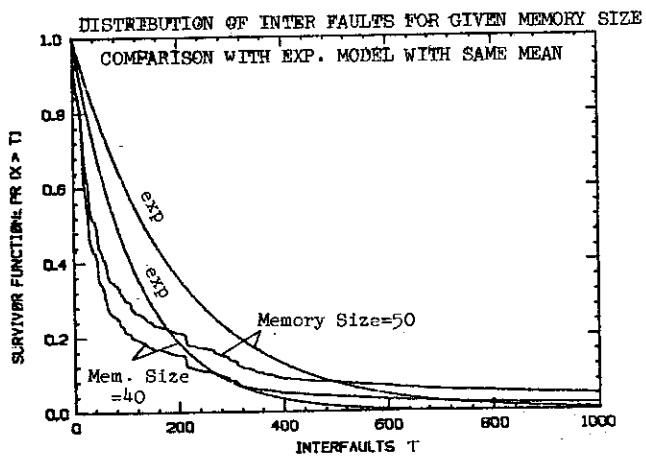


Fig. 4.3

Actual WATFIV Program				Exp. Model
Memory Size	Mean	C.V.	Standard Deviation	Standard Deviation
10	23	3.01	69	23
20	59	2.52	149	59
30	84	3.82	321	84
40	118	3.47	410	118
50	193	3.29	635	193
60	607	2.94	1785	607
70	2462	1.96	4846	2462
80	4062	1.20	4905	4062

(a)

Actual APL Program				Exp. Model
Memory Size	Mean	C.V.	Standard Deviation	Standard Deviation
10	38	4.00	152	38
20	92	3.19	293	92
30	188	2.11	398	188
40	390	2.15	841	390
50	741	2.18	1620	741
60	1161	2.20	2564	1161
70	1815	1.67	3040	1815
80	2623	1.56	4107	2623

(b)

TABLE 4.1 Mean and C.V. Of LRU Interfaults for Different Memory Sizes. The standard deviation is given for actual programs and exponential distribution with equal mean.

4.2.2 IO Device Model

The nature of the service time in IOD depends on the type of device which is used to store the program pages. With today's technology, the most common device is a paging drum. In a paging drum, each track is divided into equal size secotrs and each sector contains one page block. The service time of a request which arrives at this center is equal to the rotational latency until the read/write head reaches the beginning of the requested sector and the time to transfer one sector. Therefore, it seems that the distribution of the service time can be best formulated by a random component describing the rotational latency and a constant time for transferring one page block (see [5] for more detail). We should, however, realize that each request for this center can experience different rotational latency periods, depending on the utilization factor of the device. For instance, if there is more than one request waiting for service in this center, the completion of the transfer stage of the first request with a high probability, can leave the head on the beginning of the sector of the next request, and result in zero seek time for the latter request.

For the newer technologies that might replace paging drums, there is no reason to believe that each read/write request for the IOD needs a preparation period like the latency time described for the paging drum. To free ourselves from the internal organization of the IO devices, we will use a simple single exponential service time for the analysis of the model and, in order to find the effect of other device dependent service times on the performance of the model, we shall use a more accurate paging drum scheme for the simulation of the model.

4.2.3 Scheduling of the CPU and IOD Requests

The effects of the scheduling disciplines in both centers on the performance of the model is an important concern in this chapter. The decision to choose a request for service in each service station may be based on one of the following criterions:

- Requests are served in the order of arrivals, namely, first come first serve (FCFS) policy.
- Requests are divided into classes and each class has a service priority. The server chooses the request with the highest priority first. In this policy, the arrival of a higher priority request may preempt the execution of a lower priority job. In our case, the service priorities at a center are based on the service rate of the job at that center or on the service rate of the job at the other service center.
- The service discipline is processor sharing, i.e., when there are N jobs present in the center each job will receive service at the rate of $\frac{1}{N}$. Processor sharing is considered the limit of round robin scheduling when the quantum size approaches zero. It is a good approximation to the service policies in time sharing systems with small quantum size (e.g., 10 ms).

We will see later that under certain conditions the service disciplines have a significant effect on the resource utilization and the waiting times in the model.

4.2.4 Queueing Analysis of the Model

We assume that there are N jobs circulating in the queue where N is the degree of multiprogramming. The IOD service times for all jobs are

the same, and are exponentially distributed with rate λ. The CPU service time of each job j is exponentially distributed with the rate μ_j.

Each center may service the arrival requests based on its own service policy. Four types of service modes have been considered here:

- Type I (FCFS-FCFS): IOD and CPU both use FCFS policy.
- Type II (Independent Priority): IOD uses FCFS policy and CPU uses a preemptive priority scheme based on the service rate of each job at this center.
- Type III (CPU Priority): IOD and CPU both use priority discipline based on the service time of the job in CPU. The higher priority jobs in CPU preempt the lower priority jobs in this center. The priority ordering of the jobs in CPU is the reverse of the priority ordering of the jobs in IOD.
- Type IV (PS-FCFS): IOD uses FCFS and CPU uses processor sharing policy.

4.2.5 State Identification and General Solution of the Model

The states of the queue are elements of the set $[S_1, S_2, \ldots, S_n]$ where n is the total number of states. A state S is defined by a vector

$$S = (x_1, x_2)$$

where

$$x_s = (x_{s1}, x_{s2}, \ldots, x_{sn_s}),\ s=1,2 \text{ (or } s= \text{IOD, CPU)},$$

n_s is the number of customers (jobs) in service center s and x_{sj} is the class of customer who is jth in the FCFS or priority order. The customer undergoing service in center s is, therefore, identified by x_{s1}.

With the assumptions we have made, the state transitions constitute a positive recurrent Markov chain. Let the infinitesimal transition rate

matrix of this chain be $Q=(q_{ij})$, then the steady state probabilities are the solution for the system $\pi Q=0$, where $\pi=[P(S_1),P(S_2),.....P(S_n)]$ is the vector of steady state probabilities and n is the number of states.

The normalizing equation is $\sum_{i=1}^{n} P(S_i) = 1$.

The linear system of equations $\pi.Q=0$ can be set up by writing the so-called global balance equations. These are obtained by equating the rate the chain enters a state, to the rate it leaves that state.

This model becomes a special case of the general model analyzed in [1] for Type I scheduling when all service rates at each center are the same, and for Type IV scheduling. For other cases, one needs to write the balance equations and solve the equilibrium probabilities for each case.

4.2.6 Definition of Performance Measures

These terms will be used in the following discussion:

- CPU or IOD UTILIZATION (U_{cpu}, U_{iod}): The fraction of time the CPU or IOD is busy. These terms are defined by the equilibrium probability of each center being busy.
- WAITING TIME of job i at center s ($W_{s,i}$, s= IOD or CPU): The time period that job i spends at center s. This value consists of the queueing time plus the service time of the job.
- DILATION TIME of job i at center s ($D_{s,i}$, s= IOD or CPU): This quantity indicates the amount of time a job waits in a center relative to its service time, and is equal to $\frac{W_{s,i}}{R_{s,i}}$ where $R_{s,i}$ is the service time of job i at center s.

- COMPLETION RATE (denoted by C): The average number of jobs which terminate their execution and leave the system in a unit of time.
- PROGRESS RATE: The average rate jobs are processed. This quantity is equal to the CPU utilization because the CPU service is the real processing requirement of the jobs.

We begin solving the model starting with two special cases with N=2 jobs, and type I and type II scheduling policies. The explicit solution for the steady state probabilities, for these cases, is given. When the number of jobs increases, it is more convenient to use an algorithm to set up and solve the steady state equations with a computer program.

4.2.7 Case 1: Two Jobs and Type I (FCFS-FCFS) Scheduling

Service discipline: { CPU: FCFS; IOD: FCFS }

n_1 = number of jobs in center 1 (IOD)

n_2 = number of jobs in center 2 (CPU)

State identification:

n_1	n_2		
1	1	$S_1 = [(1),(2)]$	$S_2 = [(2),(1)]$
0	2	$S_3 = [-,(1,2)]$	$S_4 = [-,(2,1)]$
2	0	$S_5 = [(1,2),-]$	$S_6 = [(2,1),-]$

The balance equations:

$$(\lambda + \mu_2) \cdot P(S_1) = \mu_1 . P(S_3) + \lambda\ P(S_6)$$

$$(\lambda + \mu_1) \cdot P(S_2) = \mu_2 . P(S_4) + \lambda . P(S_5)$$

$$\mu_1 \cdot P(S_3) = \lambda \cdot P(S_2)$$

$$\lambda \cdot P(S_5) = \mu_2 . P(S_1)$$

$$\lambda \cdot P(S_6) = \mu_1 \cdot P(S_2)$$

$$\mu_2 \cdot P(S_4) = \lambda \cdot P(S_1)$$

The normalizing equation is:

$$\sum_{i=1}^{n} P(S_i) = 1.$$

The solution to the balance equations will give the steady state probabilities:

Let $D = 2\mu_1\mu_2(\lambda+\mu_1)(\lambda+\mu_2) + \lambda^2\mu_1(\lambda+\mu_1) + \lambda^2\mu_2(\lambda+\mu_2)$

Then:

$$P(S_1) = \lambda\mu_1\mu_2(\lambda+\mu_1)/D$$
$$P(S_2) = \lambda\mu_1\mu_2(\lambda+\mu_2)/D$$
$$P(S_3) = \lambda^2\mu_2(\lambda+\mu_2)/D$$
$$P(S_4) = \lambda^2\mu_1(\lambda+\mu_1)/D$$
$$P(S_5) = \mu_1\mu_2(\lambda+\mu_1)/D$$
$$P(S_6) = \mu_1^2\mu_2(\lambda+\mu_2)/D$$

The steady state probabilities can be used to find the equilibrium device utilization and job waiting times, as follows:

The device utilization is the probability that each device is busy, i.e.,

$$U_{iod} = Pr[IOD\ busy] = 1-Pr[IOD\ idle] = 1-P(S_3)-P(S_4)$$

$$U_{cpu} = Pr[CPU\ busy] = 1-Pr[CPU\ idle] = 1-P(S_5)-P(S_6) .$$

The waiting time of each job at each center can be found using Little's Theorm [Little, 1961]. We need to calculate the number and the arrival rate of each job class at the center.

Let $n_{s,i}$ denote the number of jobs of class i at center s. Then:

$$n_{IOD,1} = P(S_1) + P(S_5) + P(S_6)$$
$$n_{IOD,2} = P(S_2) + P(S_5) + P(S_6)$$
$$n_{CPU,1} = P(S_2) + P(S_3) + P(S_4)$$
$$n_{CPU,2} = P(S_1) + P(S_3) + P(S_4)$$

Since we have a cyclic queue, the arrival rate of a job at a center is the product of the probability that this job is under service in the other center by the service rate of the job in that center. Thus, the waiting times of the jobs at each center are:

$$W_{IOD,1} = \frac{n_{IOD,1}}{[\mu_1 \cdot (P(S_2)+P(S_3))]}$$

$$W_{IOD,2} = \frac{n_{IOD,2}}{[\mu_2 \cdot (P(S_1)+P(S_4))]}$$

$$W_{CPU,1} = \frac{n_{CPU,1}}{[\lambda \cdot (P(S_1)+P(S_5))]}$$

$$W_{CPU,2} = \frac{n_{CPU,2}}{[\lambda \cdot (P(S_2)+P(S_6))]}$$

The average waiting times at each service center are:

$$W_{IOD} = \frac{(W_{IOD,1} + W_{IOD,2})}{2}$$

$$W_{CPU} = \frac{(W_{CPU,1} + W_{CPU,2}}{2}$$

4.2.8 Case 2: Two Jobs and Type II (independent priority) Scheduling

Service discipline:
- CPU: preemptive priority: prio(job 2) > prio(job 1)
- IOD: FCFS

n_1 = number of jobs in center 1 (IOD)

n_2 = number of jobs in center 2 (CPU)

State identification:

n_1	n_2		
1	1	$S_1 = [(1),(2)]$	$S_2 = [(2),(1)]$
0	2	$S_3 = [-,(2,1)]$	
2	0	$S_4 = [(1,2),-]$	$S_5 = [(2,1),-]$

The balance equations:

$$(\lambda + \mu_2).P(S_1) = \lambda .P(S_5)$$

$$(\lambda + \mu_1).P(S_2) = \mu_2.P(S_3) + \lambda .P(S_4)$$

$$\mu_2 . P(S_3) = \lambda . [P(S_1)+P(S_2)]$$

$$\lambda . P(S_4) = \mu_2 . P(S_1)$$

$$\lambda . P(S_5) = \mu_1 . P(S_2)$$

$$\lambda . P(S_5) = \mu_1 . P(S_2)$$

The normalizing equation is:

$$\sum_{i=1}^{n} P(S_1) = 1 .$$

The solution to the balance equations will give the steady state probabilities:

Let $E = (\lambda+\mu_1)(\lambda+\mu_2)^2 + \mu_1\mu_2^2$

Then:

$$P(S_1) = \lambda\mu_1\mu_2/E$$

$$P(S_2) = \lambda\mu_2(\lambda+\mu_2)/E$$

$$P(S_3) = \lambda^2(\lambda+\mu_1+\mu_2)/E$$

$$P(S_4) = \mu_1\mu_2^2/E$$

$$P(S_5) = \mu_1\mu_2(\lambda+\mu_2)/E$$

As in the previous case, we can find terms like device utilizations and waiting times. For instance, the IOD and CPU utilization for this case is:

$$U_{IOD} = 1 - P(S_3)$$

$$U_{CPU} = P(S_1) + P(S_2) + P(S_3) .$$

4.2.9 An Algorithm to Set Up and Solve the Balance Equations

As the degree of the multiprogramming increases, the number of states grows very fast. For example, for N=4 and Type I scheduling, the number of states becomes 4!(4+1)=120. In such cases, we can use a computer program to solve the steady state equations. But first, we need a convenient way to set up the infinitesimal transition matrix, Q.

Let the states be the elements in the set $[S_1, S_2, \ldots, S_n]$. Denote by $X=(x_{ij})$ the matrix such that x_{ij} is equal to the rate state S_i goes to S_j. Thus, the balance equations become:

$$\begin{Bmatrix} \sum_{k=1}^{n} x_{1k} & & & \\ & \sum_{k=1}^{n} x_{2k} & & 0 \\ & & \ddots & \\ & 0 & & \sum_{k=1}^{n} x_{nk} \end{Bmatrix} \begin{Bmatrix} P(S_1) \\ P(S_2) \\ \vdots \\ P(S_n) \end{Bmatrix} = X^t \begin{Bmatrix} P(S_1) \\ P(S_2) \\ \vdots \\ P(S_n) \end{Bmatrix}$$

Re-writing it, we get:

$$Q \cdot \begin{Bmatrix} P(S_1) \\ P(S_2) \\ \cdot\ \cdot \\ \cdot\ \cdot \\ P(S_n) \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ \cdot \\ \cdot \\ \cdot \end{Bmatrix} \quad \text{where } Q=(q_{ij}) = \begin{cases} -x_{ii} + \sum_{k=1}^{n} x_{ik} & \text{if } i=j \\ -x_{ij} & \text{if } i=j \end{cases}$$

If we couple a normalization equation to the above system of linear equations, we can solve for the steady state probabilities. Therefore, we only need to find the X matrix in each case, which is much easier than constructing the Q matrix, and let a computer program find the Q matrix and solve the equations for the numeric solutions.

In the following sections, two more cases with N = 3 jobs are treated.

4.2.10 Case 3: Three Jobs and Type II (independent priority) Scheduling

Service discipline:
- CPU: preemptive priority: prio(job 3) > prio(job2) > prio(job 1)
- IOD: FCFS

The states are:

n_1	n_2	
0	3	$S_1 = [-,(3,2,1)]$
1	2	$S_2 = [(1),(3,2)]$, $S_3 = [(2),(3,1)]$, $S_4 = [(3),(2,1)]$
2	1	$S_5 = [(1,2),(3)]$, $S_6 = [(2,1),(3)]$, $S_7 = [(1,3),(2)]$, $S_8 = [(3,1),(2)]$, $S_9 = [(2,3),(1)]$, $S_{10} = [(3,2),(1)]$
3	0	$S_{11} = [(1,2,3),-]$, $S_{12} = [(1,3,2),-]$, $S_{13} = [(2,1,3),-]$, $S_{14} = [(2,3,1),-]$, $S_{15} = [(3,1,2),-]$, $S_{16} = [(3,2,1),-]$

In this case, the X matrix is:

$$X = \begin{Bmatrix}
0 & 0 & 0 & \mu_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\lambda & 0 & 0 & 0 & 0 & 0 & \mu_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\lambda & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mu_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\lambda & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mu_2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \lambda & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mu_3 & 0 & 0 & 0 & 0 & 0 \\
0 & \lambda & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mu_3 & 0 & 0 & 0 \\
0 & 0 & 0 & \lambda & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mu_2 & 0 & 0 & 0 & 0 \\
0 & \lambda & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mu_2 & 0 \\
0 & 0 & 0 & \lambda & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mu_1 & 0 & 0 \\
0 & 0 & \lambda & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mu_1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \lambda & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \lambda & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \lambda & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \lambda & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \lambda & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \lambda & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{Bmatrix}$$

4.2.11 Case 4: Three jobs and Type III (CPU priority) Scheduling

Service discipline:

CPU: preemptive priority:
prio(job 3) > prio(job 2) > prio(job 1)

IOD: priority:
prio(job 1) > prio(job 2) > prio(job 3)

The states are:

n_1	n_2	
0	3	$S_1 = [-,(3,2,1)]$
1	2	$S_2 = [(3),(2,1)]$, $S_3 = [(2),(3,1)]$, $S_4 = [(1),(3,2)]$
2	1	$S_5 = [(2,3),(1)]$, $S_6 = [(3,2),(1)]$, $S_7 = [(1,3),(2)]$, $S_8 = [(3,1),(2)]$, $S_9 = [(1,2),(3)]$, $S_{10} = [(2,1),(3)]$
3	0	$S_{11} = [(3,1,2),-]$, $S_{12} = [2,1,3),-]$, $S_{13} = [(1,2,3),-]$

And the X matrix for this case is:

$$X = \begin{Bmatrix}
0 & \mu_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\lambda & 0 & 0 & 0 & 0 & \mu_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\lambda & 0 & 0 & 0 & \mu_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\lambda & 0 & 0 & 0 & 0 & 0 & \mu_3 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \lambda & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mu_1 & 0 \\
0 & 0 & \lambda & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mu_1 & 0 & 0 \\
0 & \lambda & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mu_2 \\
0 & 0 & 0 & \lambda & 0 & 0 & 0 & 0 & 0 & 0 & \mu_2 & 0 & 0 \\
0 & 0 & \lambda & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mu_3 \\
0 & 0 & 0 & \lambda & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mu_3 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \lambda & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \lambda & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \lambda & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{Bmatrix}$$

4.2.12 Case 5: Type IV (processor sharing - FCFS) Scheduling

A closed form solution for the steady state probabilities can be found for this model when the scheduling policy at the CPU is processor sharing. In this case, we use a derivation of a more general model discussed in [1].

Service discipline is Type IV:

CPU: Processor sharing

IOD: FCFS

There are N job classes and each class has one job. The states of the queue are denoted by

$$S = (x_1, x_2)$$

where $x_s = (n_{s1}, n_{s2}, \ldots, n_{sN})$, s = 1(IOD), 2(CPU) and n_{sr} is the number of customers of class r present in service center s (we note that the state definition is different from earlier cases and n_{sr} = 0 or 1). Let $n_s = \sum_{n=1}^{n} n_{sr}$, s=1,2 be the total number of jobs present in station s, we then have $n_1 + n_2 = N$.

The joint equilibrium state probability distribution has a product form solution as follows:

$$Pr[S = (x_1,x_2)] = K\, g_1(x_1) \cdot g_2(x_2)$$

where

$$g_1(x_1) = n_1! \prod_{r=1}^{n_1} \left(\frac{1}{\lambda}\right) = n_1! \left(\frac{1}{\lambda}\right)^{n_1}$$

and

$$g_2(x_2) = n_2! \prod_{r=1}^{n_2} \left(\frac{1}{\mu_r}\right)^{n_{2r}}$$

The coefficient K is the normalization factor which makes the probabilities sum to one, and can be obtained from:

$$K^{-1} = \sum_{\substack{\text{for all} \\ \text{states } S}} g_1(x_1) \cdot g_2(x_2) =$$

$$= \sum_{n_{1,1}=0}^{1} \cdots \sum_{n_{1,N-1}=0}^{1} \sum_{n_{1,N}=0}^{1} n_2! n_1! \left(\frac{1}{\lambda}\right)^{n_1} \prod_{r=1}^{N} \left(\frac{1}{\mu_r}\right)^{n_{2r}}$$

The CPU and IOD utilizations can be found from the expression for the steady state probabilities:

$$U_{CPU} = 1 - Pr[\text{CPU is idle}] = 1 - Pr[n_2=0] =$$

$$= 1 - [K^{-1} N! \prod_{r=1}^{N} \frac{1}{\mu_r}]$$

$$U_{IOD} = 1 - Pr[\text{IOD is idle}] = 1 - Pr[n_1=0] =$$

$$= 1 - [K^{-1} N! \left(\frac{1}{\lambda}\right)^N]$$

The expressions for K and the device utilizations look awkward but, in fact, for the numerical evaluations, they can be conveniently evaluated with a program with recursive calls.

4.3 DISCUSSION OF THE NUMERICAL RESULTS

The numerical results obtained from the analysis of the model are best illustrated with a series of graphs. The objective is to do a comparative study of the behavior of the system under different loads and different structural assumptions.

The jobs circulating in the cycle are assumed to be identical programs. We have taken the WATFIV program and measured the fault rate function f_m under the LRU replacement rule. When m pages are allocated to a job in the main memory, f_m determines the service rate of that job

at the CPU. Since the paging behavior of the WATFIV program is typical of many programs we have measured, the numerical results should be representative for a large class of frequently used programs.

The size of the WATFIV program used here is 163 pages. The size of the main memory, M, is taken to be equal to this number. Therefore, only one program can be resident in the main memory with all its pages.

The service rate at the IOD is the same for all jobs. Unless specified, the IOD service distributions are assumed to be exponential.

The degree of multiprogramming is denoted by N. When N=2, the main memory is divided into two parts. Each program has m_i, i=1,2 pages and $m_1+m_2=M$. The service rate at the CPU for job i is equal to $\mu_i=f_{m_i}$, i=1,2. Referring to, say, Figure 4.4, the memory allocation (m_1,m_2) is specified by a point on the x axis which is $m_1(m_2)$ units from the right (left) vertical axis.

In a similar way, when N=3, the memory is divided into three parts; namely, m_1, m_2 and m_3 where $m_1+m_2+m_3=M$. To represent a memory partition (m_1,m_2,m_3), we use baricentric coordinates. In this coordinate system, we have an equilateral triangle with an altitude equal to M, the total main memory size. Each three way memory partitioning (m_1,m_2,m_3) is uniquely defined by a point inside this triangle. This point has vertical distances m_1, m_2 and m_3 from the three sides (faces) of the triangle.

For a general N way memory partitioning, one needs a tetrahedron in N-1 space to specifiy a point.

The performance of the model can be affected by any of the following factors:

(a) memory allocation policies

(b) degree of multiprogramming

(c) scheduling disciplines in IOD and CPU

(d) IOD (e.g., drum) speed

(e) assumption about program paging behavior

(f) assumption about IOD characteristics

4.3.1 Effect of Memory Allocation Policies on Device Utilizations

The way main memory is divided among active jobs has a very substantial effect on CPU and IOD utilization. Regardless of the service discipline, when memory is symmetrically divided into two parts (Figures 4.4 and 4.5) or into three parts (Figure 4.6), the CPU utilization is low. Conversely, asymmetric memory allocations give rise to higher CPU utilization. In Table 4.2, the numerical values of device utilizations for a number of different memory allocation schemes are given. For instance, for N=2, m_1=81 and m_2=82 (fairly symmetric allocation), the CPU utilization with Type I scheduling is 0.67. Under the same condition for the allocation m_1=40 and m_2=123 (asymmetric), the CPU utilization is 0.94. With N=3, we get more interesting results by referring to Figure 4.6a. In this figure, which shows the CPU utilization with Type I scheduling, we observe that in the region in the middle of the triangle, which corresponds to memory partitioning of fairly equal sizes, the CPU utilization is much lower than other regions.

In the same figure, consider a line parallel to one of the sides of the triangle. The variation of CPU utilization along this line corresponds to the class of memory allocations where the space for one job is fixed and the remaining space is divided between the other two jobs in a variable way, depending at which point along the line we stand. For instance, consider a conceptual line which is parallel to the base of a

triangle (Figure 4.6a). Along this line, m_1 is fixed. When this line lies on the top of the base line, m_1=0, we get a CPU utilization graph which is similar to Figure 4.4. As we move this line upward, i.e., increasing m_1, we eventually reach a point where from then on the utilization remains fairly constant along the line, and hence, it is fairly independent of the way we allocate the remaining space (not used by job 1) between job 2 and 3 ($m_1 > 102$).

The high CPU utilization for asymmetric allocation is due to the fact that in this scheme one job gets sufficient space at the expense of leaving a little room for the other jobs. This job will execute for a relatively long time without being interrupted by a page fault. Concersely, in a symmetric allocation, since all jobs are running in a relatively small area, the resulting frequent page faults cause contention in IOD, which in turn will lower the flow of the jobs in the system.

The IO device (IOD) utilization with Type I scheduling has an inverse relationship with CPU utilization. The IOD utilization is the highest when the main memory is divided into equal parts. Short CPU service times, due to symmetric allocation of main memory, increases the rate of the requests arriving at IOD and, therefore, keeps the device busy most of the time. These are illustrated in Figures 4.4 and 4.5 for N=2, and Figure 4.6a for N=3. Table 4.2 gives some of the numerical values.

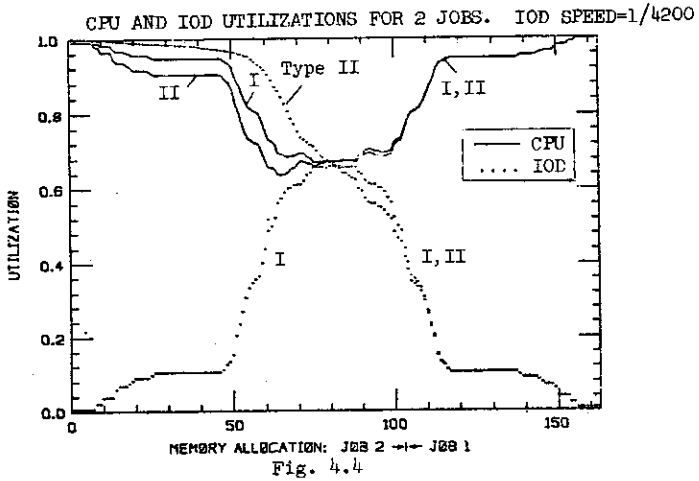


Fig. 4.4

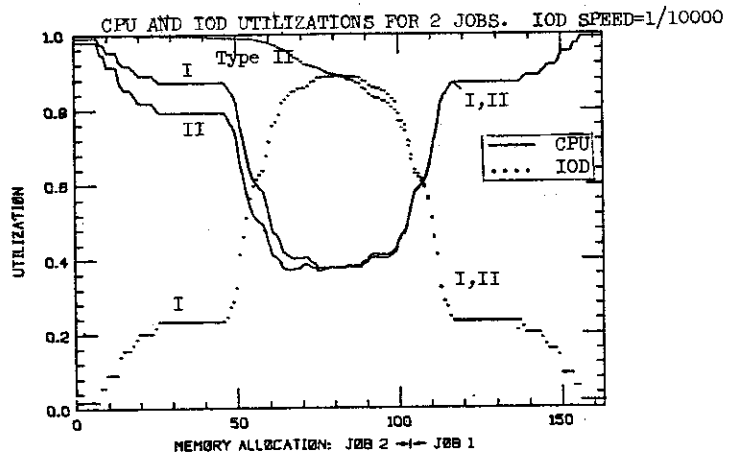
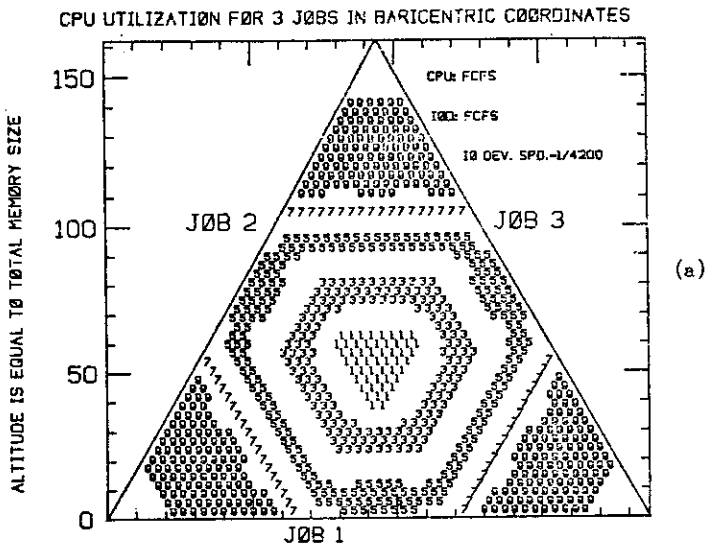
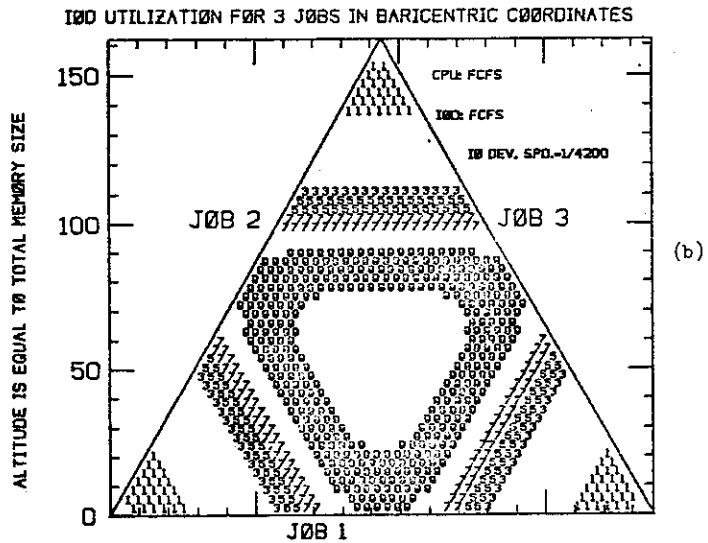


Fig. 4.5



(a)



(b)

Fig. 4.6

N	Memory Allocation			CPU Utlz.	IOD Utlz.
	m1	m2	m3		
2	40	123	-	0.94	0.10
2	60	103	-	0.76	0.10
2	81	82	-	0.67	0.66
2	103	60	-	0.76	0.44
2	123	40	0	0.94	0.10
3	30	32	101	0.62	0.68
3	41	42	80	0.29	0.94
3	53	54	56	0.07	0.99
3	32	75	56	0.28	0.95
3	20	87	56	0.38	0.89

TABLE 4.2 Effect of Memory Allocation Policies on CPU and IOD Utilization. Scheduling is Type I and IO Device Speed is 1/4200.

4.3.2 Effect of Page Scheduling on Completion Rate

We next define the completion rate as the number of jobs which can be pushed through the system in a unit of time. These jobs presumably complete their execution and leave the system. This definition is not complete until we specify the resource requirements of the jobs entering the system. In our next discussion, we only take the total CPU service requirement of each job into consideration. A slight external modification must be given to our model to permit the conceptual entering and leaving of the jobs in and out of the system. This modivication is shown in Figure 4.7. Here we sssume that, at each instance, the system can accommodate N jobs from N different classes. All jobs of class i have the same average total CPU service requirement R_i, i=1,2,...N. On the average, with each passage through the CPU, job i receives $\frac{1}{\mu_i}$ units of

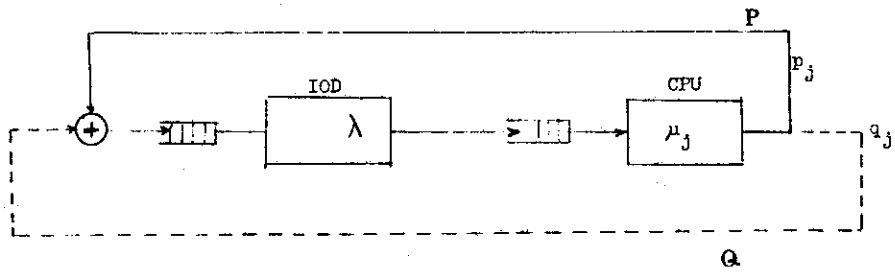


FIGURE 4.7 A Conceptual Modification of the Model to Permit the Computation of Completion Rates

service from this center where μ_i is the CPU service rate. Each job circulates in the cycle until its CPU service requirement is exhausted. Then it will leave the system and it is immediately replaced by a new job from the same class. Thus, we assume that the system is heavily loaded with all classes of jobs. Let R_i be a random number which describes the total CPU service requirement of a job from class i. Let N_i be the number of times job i must cycle through the system until it receives all its CPU time. It can be shown (Appendix 4.5) that

$$\bar{N}_i = E[N_i] = E[R_i]\ \mu_i = \bar{R}_i\ \mu_i \ .$$

Let p_i and q_i, $q_i+p_i=1$, i=1,2,...N be the branch probabilities at the departure point after the CPU (Figure 4.7). With the probability p_i, job i will return to the IOD queue via branch $\vec{P}$, and with probability q_i, it will leave the system through path $\vec{Q}$ to be subsequently replaced by an identical job. We can see that the internal structure of the model is untoughed except with this scheme we are not able to measure the rate jobs flow through path $\vec{Q}$ which determine the rate the jobs are leaving the system.

We would like to estimate p_i and q_i for i=1,2,...N such that a job from class i cycles $\bar{N}_i$ times on the average through path P before it leaves the system through branch Q.

We note that random variable N_i is geometrically distributed, i.e., $pr[N_i=k] = p_i^{\,k}\cdot q_i$, k=0,1,2,...

Then:

$$E[N_i] = p_i\cdot q_i(1+2p_i+3p_i^{\,2}+....) =$$

$$= \frac{p_i}{q_i}$$

Using the method of moments, we require that:

$$E[N_i] = \frac{p_i}{f_i} = \frac{1-q_i}{q_i} = \bar{N}_i$$

or

$$q_i = \frac{1}{1-\bar{N}_i}$$

or

$$q_i = 1-p_i = \frac{1}{1+\bar{R}_i\mu_i}$$

Let c_i be the rate jobs from class i leave the system through branch $\vec{Q}$ (Figure 4.7). Define the completion rate of the system, denoted by C, as:

$$C = \sum_{i=1}^{N} c_i$$

where

$$c_i = \mu_i q_i \sum_{\substack{\text{all states S}\\ \text{which job i is}\\ \text{under service in}\\ \text{CPU}}} p[S]$$

In Figure 4.8, for N=2 and Type I scheduling, the completion rate C for various memory partitionings is shown. The total CPU service requirement of each job class is assumed to be exponentially distributed with means $\bar{R}_1$ and $\bar{R}_2$, respectively. In this figure $\bar{R}_1=10^7$ time units and from top to bottom $\bar{R}_2=\bar{R}_1$, $\bar{R}_2=2\bar{R}_1$ and $\bar{R}_2=3\bar{R}_1$.

We can see that for symmetric allocation, we get a lower completion rate. This is mostly because of the low CPU utilization obtained from symmetric partitioning. When $\bar{R}_2 > \bar{R}_1$, the completion rate curves do not entirely follow the pattern of the CPU utilization curves. There are two advantages if we let job 1, with its shorter total execution time, get a

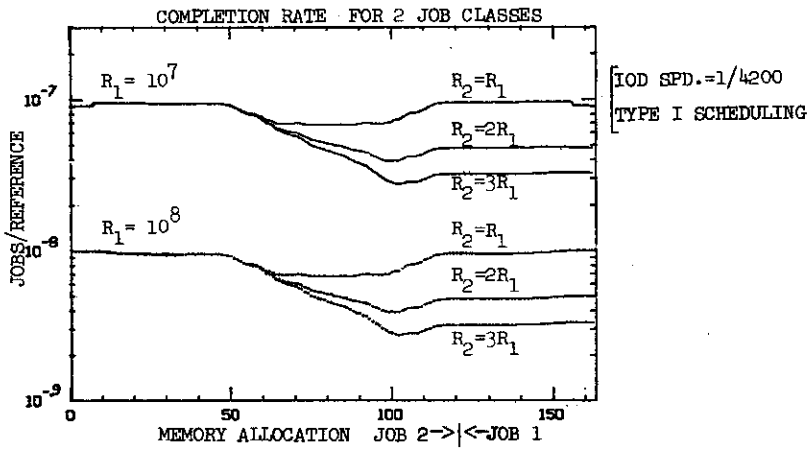


Fig. 4.8

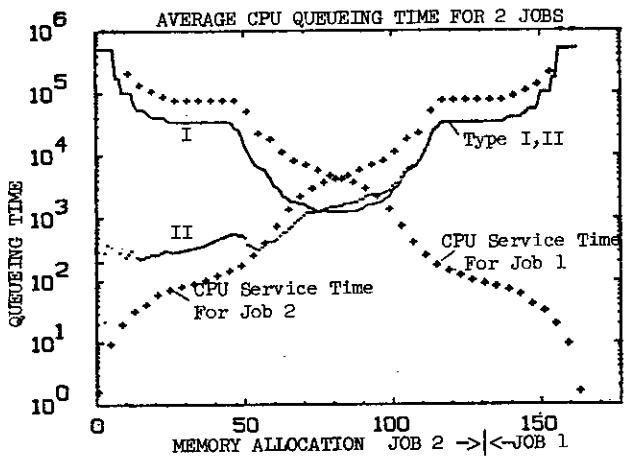


Fig. 4.9

a larger share of the main memory. First, we will have an asymmetric memory allocation which causes more efficient use of CPU; second, the jobs with shorter total execution times can finish faster and leave the system without being delayed behind the jobs with longer CPU service demand.

In all completion rate curves, we can see that a minimum is obtained in asymmetric memory partitioning $m_1=62$ and $m_2=101$. There are two reasons for this. First, this partitioning is close enough to a region where CPU utilization is low (see Figure 4.4). Second, with this uneven partitioning, class 2 jobs with their longer execution times receive most of the CPU service and, hence, reduce the total completion rate.

In our formulation of the completion rate, if we let $\bar{R}_i$ approach infinity for all i, then C will tend to zero. In this case, we can only talk about the rate the jobs progress toward their completion at infinity. This brings up the notion of the progress rate which indicates the rate jobs are receiving CPU service. This quantity, averaged over all the jobs present in the system, is equal to the CPU utilization.

4.3.3 Effect of Page Scheduling on Queueing Times

Efficiency measured as resource utilization is not the only concern in the performance of a computer system. There are other equally important factors which should be taken into account. Going back to the original model, we can study the waiting time of the jobs at each center. The significance of this study arises from the goal of most operating systems to allow all programs to advance in the system at a reasonable pace. We want to avoid the situations where the execution of one job is greatly delayed in favor of giving faster service to other programs.

We have two terms which are related to the waiting time of jobs at each center. When a job arrives at a center, its queueing time in that center is measured as the time it spends in the queue before it starts to receive service. For all jobs in the cycle, the average of the mean queueing time at a center is defined to be the average queueing time at that center.

For N=2 with Type I (and Type II), scheduling the average queueing time at the CPU are shown in Figure 4.9. In the same figure, the dotted lines give the average service time values for each job under different main memory space. We can see that a symmetric memory allocation results in lower average queueing times. This is partly because with this memory configuration, CUP is not heavily used and, therefore, when a job arrives there, it is more likely that it finds an empty queue. For Type I, scheduling the queueing time curves follow the pattern of CPU utilization curves, i.e., low values for symmetric and nearly symmeteric allocations and high values for asymmetric partitioning. In the next section, we will see that by introducing a suitable priority scheduling discipline we can maintain high CPU utilization and, meanwhile, keep the average queueing times low. Table 4.3 gives the queueing time of each job and the average queueing time at the CPU for N=2 and Type I scheduling.

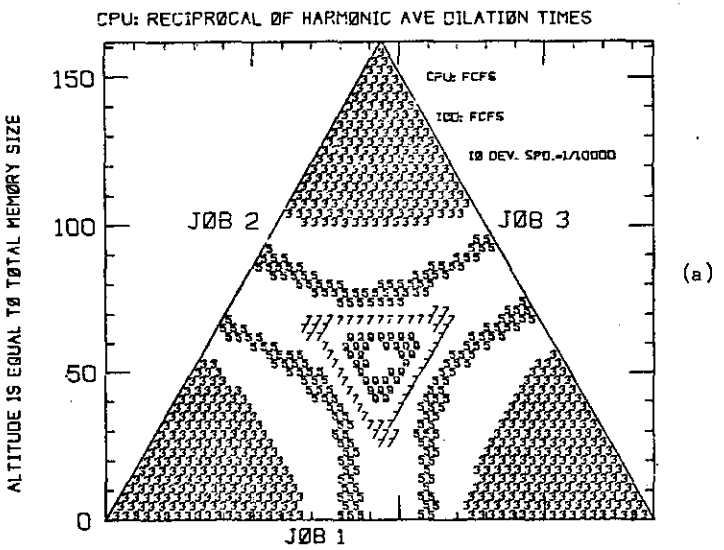
Memory Allocat'n		CPU	Ave. queueing time of		(W1+W2)/2	Ave CPU service time of	
m_1	m_2	Utlz.	job 1 W1	job 2 W2		job 1	job 2
40	123	0.87	65807	1	32904	118	74627
45	118	.087	65808	2	32905	147	74626
50	113	0.81	28270	4	14137	194	50000
55	108	0.61	12522	9	6265	309	20161
60	103	0.51	6370	34	3202	607	13793
65	98	0.41	3645	165	1905	1371	8665
70	93	0.40	2757	486	1621	2462	7037
75	88	0.38	1730	796	1263	3247	5269
80	83	0.37	1310	1174	1242	4065	4425
85	78	0.38	984	1538	1261	4766	3786

TABLE 4.3 Average Queueing Times at CPU. Scheduling is Type I and Drum Speed = 0.0001

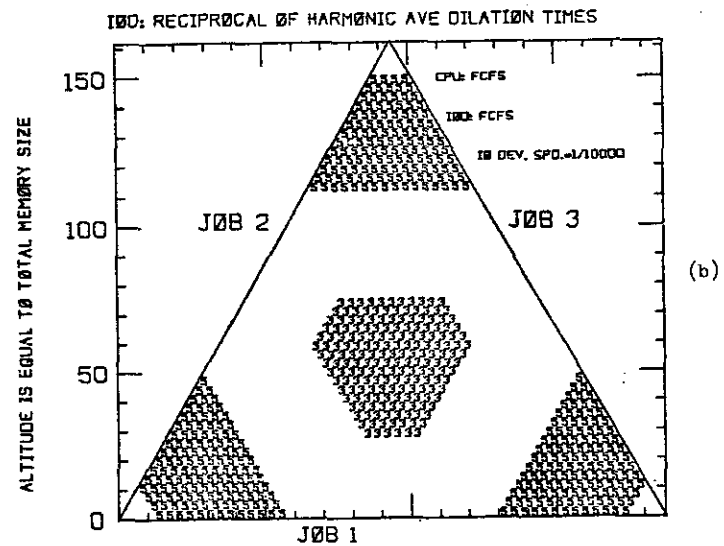
Another quantity which is also related to relative waiting times is is the dilation time of a job. It is the amount of time a job waits at a center relative to its processing time in that center. Therefore, the high job dilation time at a center indicates the situation where a job is waiting a long time to get a relatively short service from that center. This can be undesirable if the goal is to keep down the waiting time of the jobs with short service requirements. It can also be a deliberate penalty which is imposed on a job if the overhead of switching jobs is high.

The harmonic average dilation time, D_s, at each center s, reflects the contribution from all jobs in that center and is equal to

$$D_s = \frac{N}{\sum_{i=1}^{N} 1/D_{s,i}}$$. In Figure 4.10a and 4.10b, the reciprocal of D_s,



(a)



(b)

Fig. 4.10

i.e., $D_s^{-1} = 1/D_s$, which has real values between 0 and 1, for N=3 and Type I scheduling, and for s=CPU and IOD are shown. The high values of D_s^{-1} is indicative of low dilation times. We note that since we are dealing with the reciprocal of harmonic averages, the contribution from each term is between 0 and 1/3 for N = 3.

For symmetric allocations (regions around the center of the triangle in Figure 4.10a), all jobs spend very little extra time in CPU over their execution time [$D_s^{-1} > 0.9$]. With extremely asymmetric allocation, i.e., in the areas around the vertices in the figure, the job with the largest share of the memory causes a long delay in the execution of the other two jobs.

In the remaining areas, we can see the pattern according to which D_s^{-1} changes for other allocations. A descriptive picture of the relation between the dilation time and the utilization at CPU can be obtained by comparing Figure 4.11a with Figure 4.10a.

In Figure 4.10b, the reciprocal of the harmonic average dilation times in IOD, D_{IOD}^{-1}, is shown for different main memory configurations. Since the service time at this center is the same for all jobs, the variation of D_{IOD}^{-1} reflects the relative waiting times under various amounts of traffic which arrive at this center. Comparing Figures 4.10b with 4.11b (IOD utilization plot), we can see the relationship between the utilization and dilation times at IOD. The congested IO device [$U_{IOD} > 0.9$] gives rise to low D_{IOD}^{-1} which indicates high dilations times.

4.3.4 Effect of Degree of Multiprogramming on the Performance of the Model

In a general multiprogramming system, it has been observed that due

to the different IO activity in the system the CPU utilization reaches a maximum value for an optimal degree of multiprogramming. In our model, however, the only IO activity is generated by the paging demands of the programs. Therefore, by changing the degree of multiprogramming, we are going to investigate the effect of the number of active jobs which intervene with each other's execution behavior by occupying a part of the common memory.

In Table 4.4, the CPU and IOD utilization for N=1, 2 and 3 programs are given. For each value of N greater than one, we have multiple allocation possibilities. When we have symmetric allocation, the CPU utilization with N=2 is equal to 0.67, and with N=3 is equal to 0.07. Correspondingly, the IOD utilization is 0.66 for N=2 and 0.99 with N=3. Therefore, when N=3, we have very high paging activity and we are faced with a thrashing problem.

Another comparison point is the case when we fix, say, m_1 and allocate the remaining memory for one or two jobs. We notice that when m_1 is large enough, the CPU utilization is less sensitive to the degree of multiprogramming. For instance, with m_1=102 and N=2, the CPU utilization is 0.73, and for the same value for m_1 but with N=3, the CPU utilization is 0.63. Moreover, for the latter case the way we allocate the space for the second and third job does not have a substantial effect on the CPU utilization.

The result we quoted here were from Type I scheduling. In the next section, we investigate the subject of job scheduling at CPU, which brings up a number of interesting issues.

N	memory allocation			IOD spd.=1/4200		IOD spd.=1/10000	
	m_1	m_2	m_3	CPU utlz.	IOD utlz.	CPU utlz.	IOD utlz.
1	163	-	-	1.0	0.0	1.0	0.0
2	102	61	-	0.73	0.49	0.47	0.75
2	81	82	-	0.67	0.66	0.37	0.89
3	56	53	54	0.07	0.99	0.03	0.99
3	102	11	50	0.63	0.66	0.36	0.89
3	102	20	41	0.63	0.66	0.36	0.89
3	102	29	32	0.63	0.66	0.36	0.89
3	102	41	20	0.63	0.66	0.36	0.89
3	102	50	11	0.63	0.66	0.36	0.89

TABLE 4.4 CPU and IOD Utilization for Different Degree of Multiprogramming and IO Device Speed. Scheduling is Type I.

4.3.5 Effect of Device Scheduling on CPU and IOD Utilization

Another aspect of our study is related to the effect of various scheduling disciplines on the performance of the model. At the CPU, we consider FCFS, preemptive priority and processor sharing scheduling. In IOD, we also allow a priority discipline besides the usual FCFS policy.

In Figures 4.4 and 4.5, the CPU and IOD utilization is plotted for Type I [CPU: FCFS IOD: FCFS] and Type II [CPU: preemptive priority with PRIO (job 2) > PRIO (job 1), IOD: FCFS] scheduling disciplines. With the Type I policy, the CPU and IOD utilization curves are symmetric with respect to the equal memory partition point (i.e., $m_1=m_2=M/2$). When Type II scheduling is in effect, the left half of the plots indicates the region where the job with the higher fault rate (i.e., with shorter CPU execution bursts) has preemptive priority over the other job at the CPU. Conversely, the right half of the same figures indicate the region where the job with the lower fault rate (i.e., with longer CPU execution bursts) has a higher preemptive priority over the other job. In the former case, the CPU utilization with Type II scheduling is lower (about 10% with the slow IOD) than the CPU utilization of the corresponding memory partitioning using Type I scheduling. The CPU utilization of Type I and II scheduling are fairly equal in the right hand side of the plot.

IOD utilization with N=2 and Type I and II scheduling are also shown in Figures 4.4 and 4.5, with dotted lines. We get a very high IOD utilization when the job with the highest fault rate has a higher CPU priority than the other job. This is because the higher priority job with its short CPU service requests will reach the IOD without being blocked by the other job. The frequent visits by this job to the IOD will increase the utilization of this center.

The IOD utilization curve with Type II scheduling will lie on the top of the IOD utilization curve with Type I scheduling when the job with longer CPU execution bursts has a higher CPU priority over the other job (memory allocation regions corresponding to the right half of the plots).

With the degree of multiprogramming equal to 3, the CPU utilizations are shown in Figures 4.11a and 4.12a for Type I and Type II scheduling, respectively.

In Figure 4.12a, we can distinguish three important regions.

- Region (1) around the upper vertex of the triangle where job 1 with the longest average CPU service time has the least priority at the CPU.
- Region (2) around the right hand vertex where the job with the largest average CPU service time, namely, job 2, has less priority than job 3 and higher priority than job 1.
- Region (3) around the left vertex where job 3 with the longest CPU service time has the highest CPU priority.

Among these three regions, the lowest CPU utilization is obtained in Region 1 and the highest is obtained in Region 3. This implies that with priority discipline at the CPU, it is better, in terms of CPU utilization, that the least priority be given to the jobs with short CPU services, i.e., IO bound jobs. This seems to contradict the widely accepted statement that the best way to schedule a CPU is to give higher CPU priority to a job which will compute for the shortest time before issuing an IO request [12]. An explanation for this is as follows: in our model, when we use priority scheduling at the CPU and a FCFS policy at the IOD, the job flow of the system is mostly governed by the frequency

that the higher CPU priority jobs issue their IO requests. Therefore, when higher CPU priority is given to the job which has the shortest CPU execution intervals (IO bound jobs) a contention of IO requests takes place. This, in turn, increases the likelihood that the CPU remains idle. Therefore, the only way to increase the CPU utilization in this case is to prevent the IOD contention by providing multiple IO paths or increasing the IO processing speed.

Referring to Figure 4.12a, we can see that for fairly balanced memory allocation the scheduling does not have significant effect on the efficiency of CPU.

For N=3 in Figures 4.11a and 4.12a, we can compare the effect of priority scheduling and a FCFS policy on the utilization of the CPU. The result of the comparison is basically the same as the earlier case with N=2. Specifically, when higher CPU priority is given to the job with the longest average CPU service time (Region 3 in Figure 4.12a), the utilization is the same as in the FCFS case (the region around the lower left vertex in Figure 4.11a). In the other regions, the priority scheme does slightly worse.

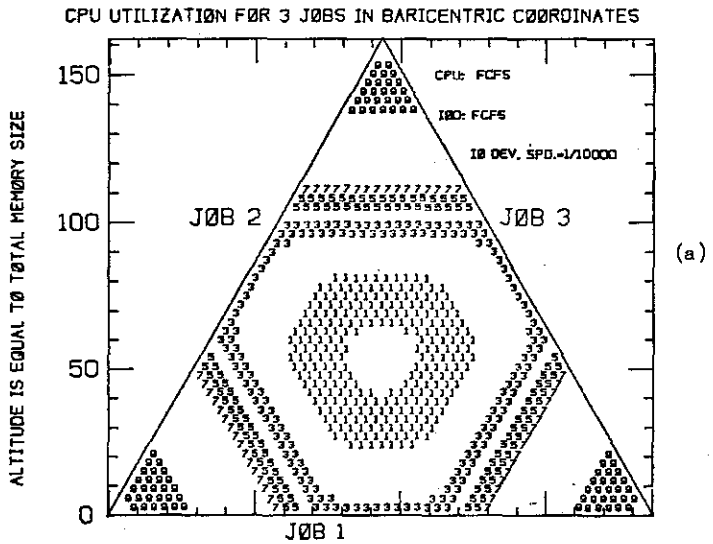
The IOD utilization for N=3, Type I and Type II scheduling is shown in Figures 4.11b and 4.12b, respectively. In both cases the IOD service policy is FCFS. As before, the utilization plot is given for each point corresponding to a memory partitioning. With Type II scheduling (Figure 4.12b), IOD is utilized heavily (more than 90% of the time) in most regions, except when the highest priority is given to a job which issues the least amount of IO requests.

From our earlier discussion, we concluded that we get a better CPU utilization if we give the highest priority to the jobs which are CPU

bound. In actual systems this is not a desirable priority assignment and usually the jobs which require short CPU service are given a higher priority. Moreover, with optimal CPU priority assignment, IOD utilization is very low which is an indication that jobs IO and processing demand are not balanced. The Type III scheduling policy is used to balance the processing and IO load, and increase the CPU utilization when the short jobs have higher priority. In this scheme jobs in IOD are serviced according to a priority ordering which is the reverse of each job's priority ordering at the CPU. Preemption is not allowed at the IOD. Therefore, with Type III scheduling, at the CPU we have PRIO(job 1) < PRIO(job 2) < PRIO (job 3) and at the IOD we have PRIO(job 1) > PRIO(job 2) > PRIO(job 3).

The CPU and IOD utilizations with Type III scheduling are shown in Figures 4.13a and 4.13b, respectively. We immediately notice that in almost all the allocation regions we obtain high IOD utilization. The highest CPU utilization is observed in Region 1, Figure 4.13a, where the job with least IO activity has the lowest CPU priority. This is, of course, in contract to Type II scheduling where in the same region we get the lowest CPU efficiency.

The low CPU utilization area (less than 10%) with Type III scheduling (Figure 4.13a) is larger than the similar area with Type I (Figure 4.11a), and Type II (Figure 4.12a) scheduling. In Figure 4.13a, this region is extended toward the lower left corner of the triangle (Region 3). The fluctuation of the CPU utilization as we approach the left vertex of the triangle is due to the mutual effect of the flat CPU service rate function



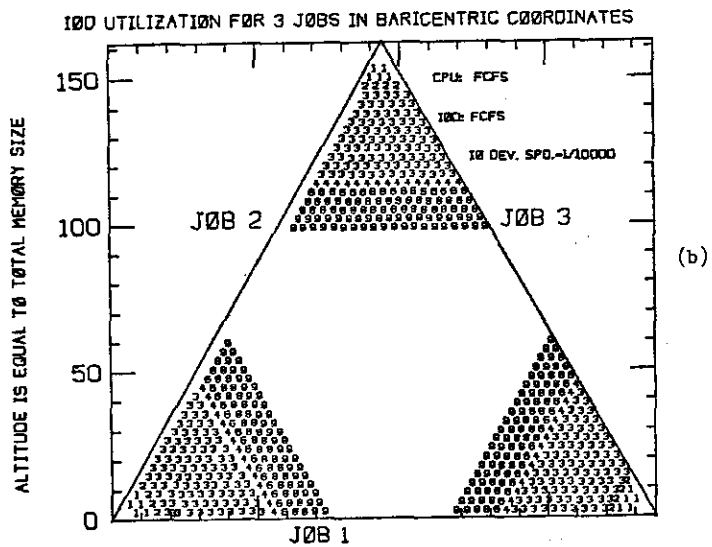
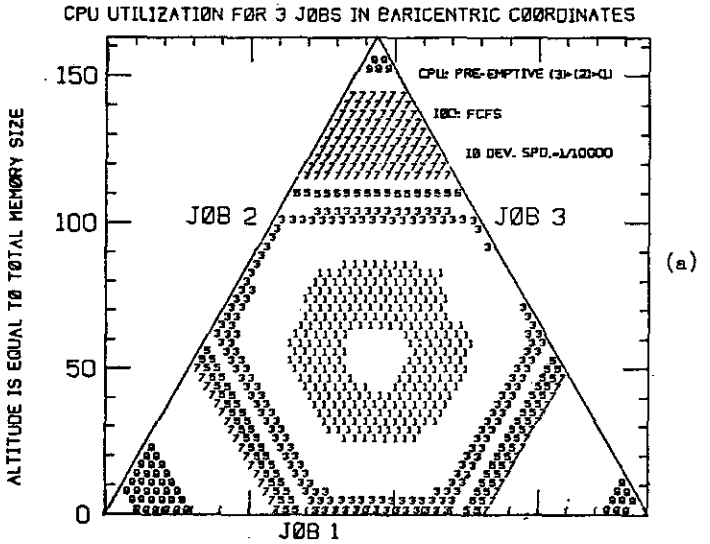


Fig. 4.11



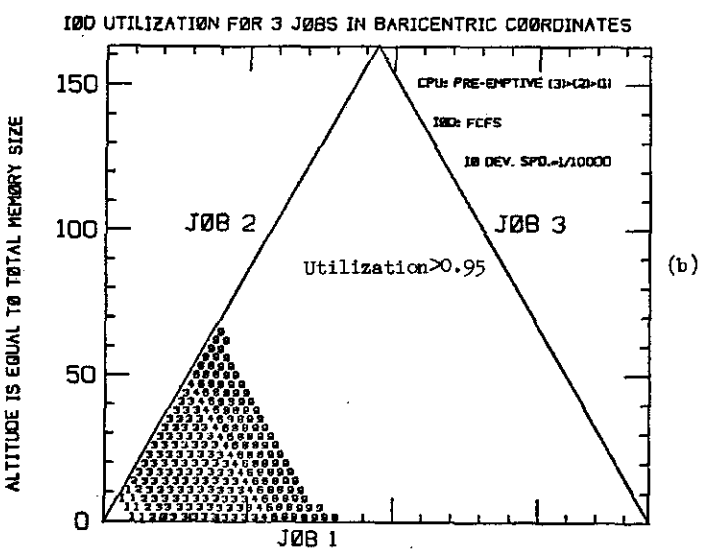


Fig. 4.12

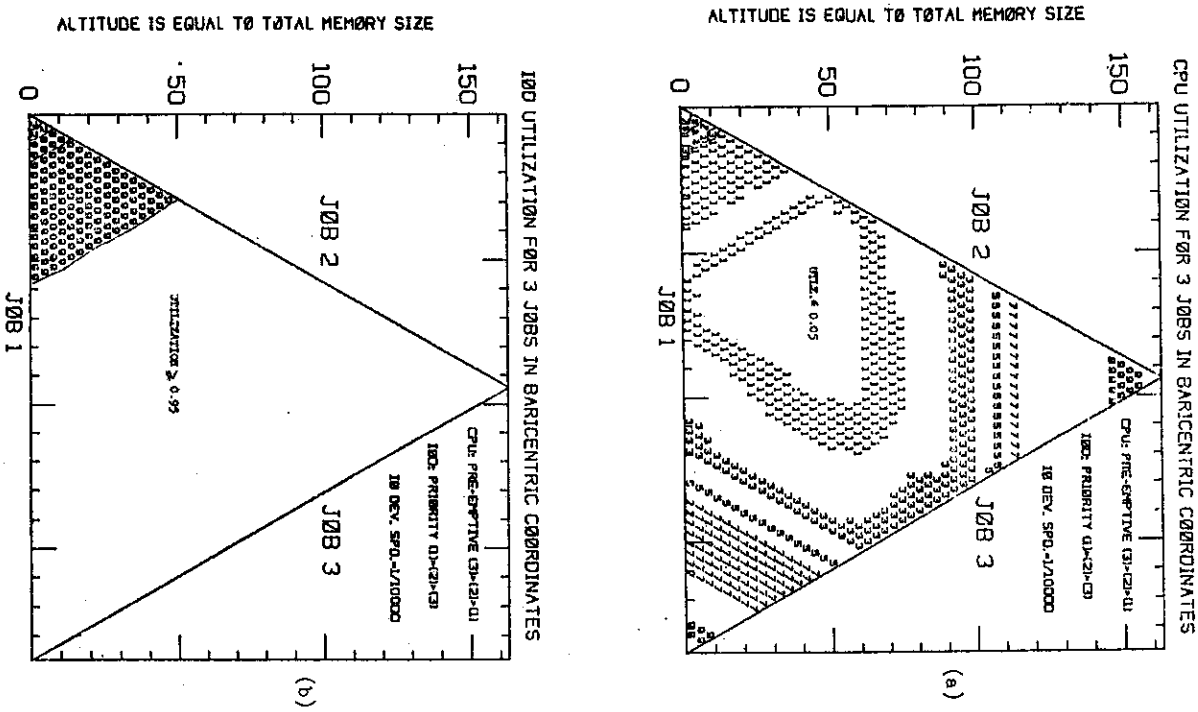


Fig. 4.13

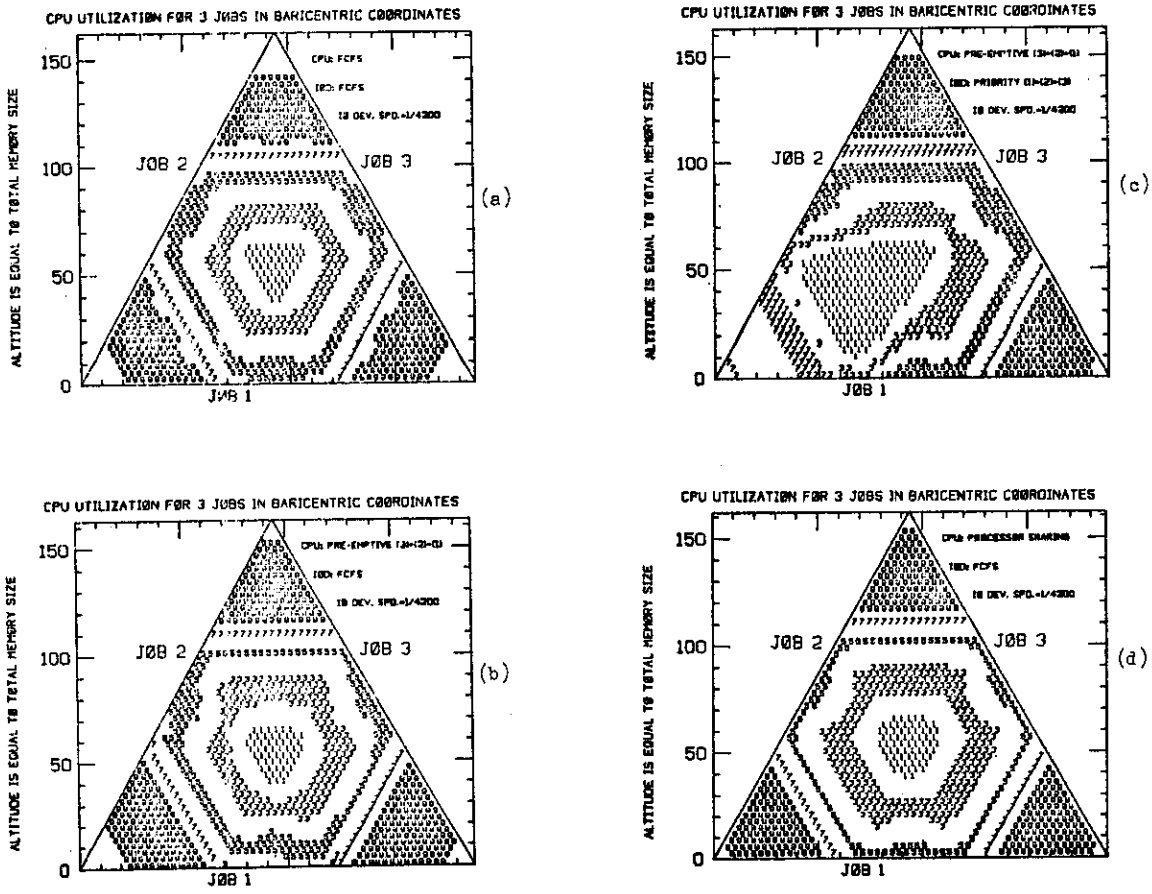


Fig. 4.14 CPU utilization with four types of scheduling
IOD speed=1/4200

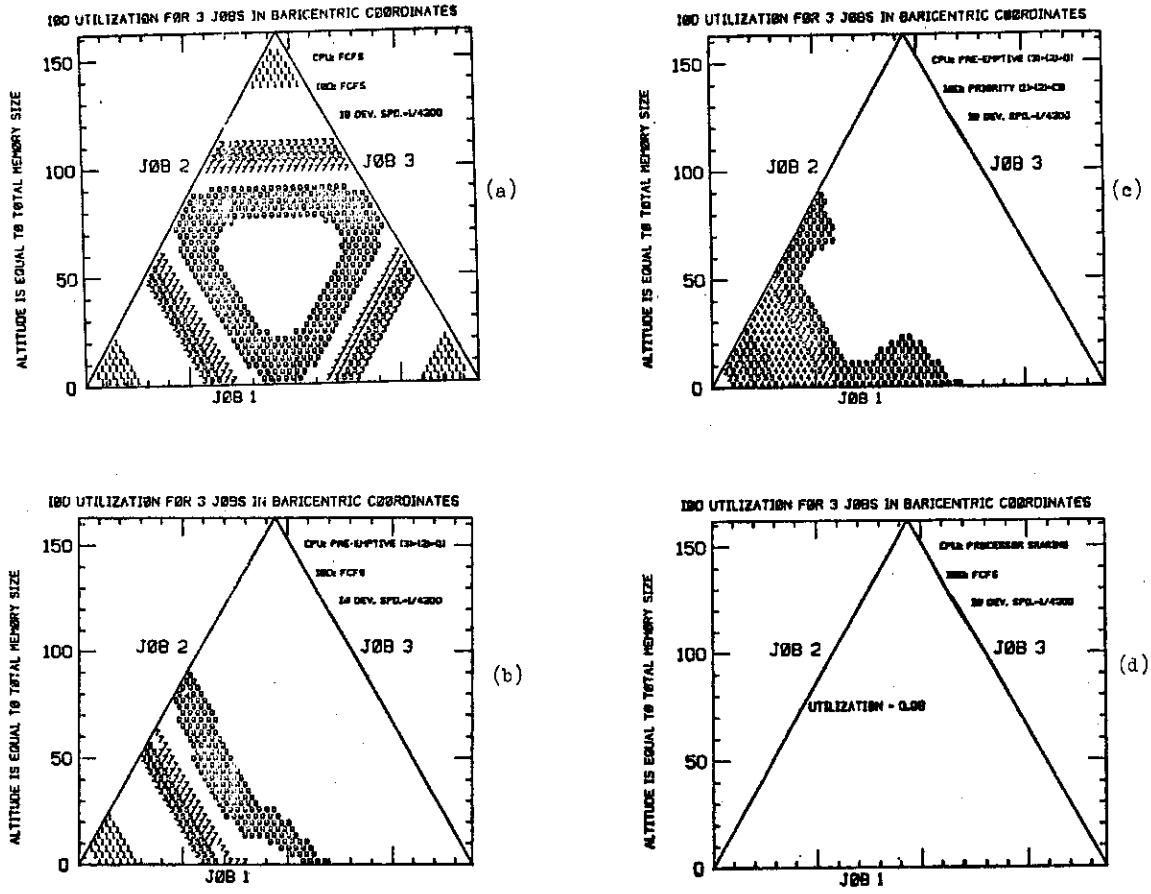


Fig. 4.15 IOD utilization with four types of scheduling
IOD speed=1/4200

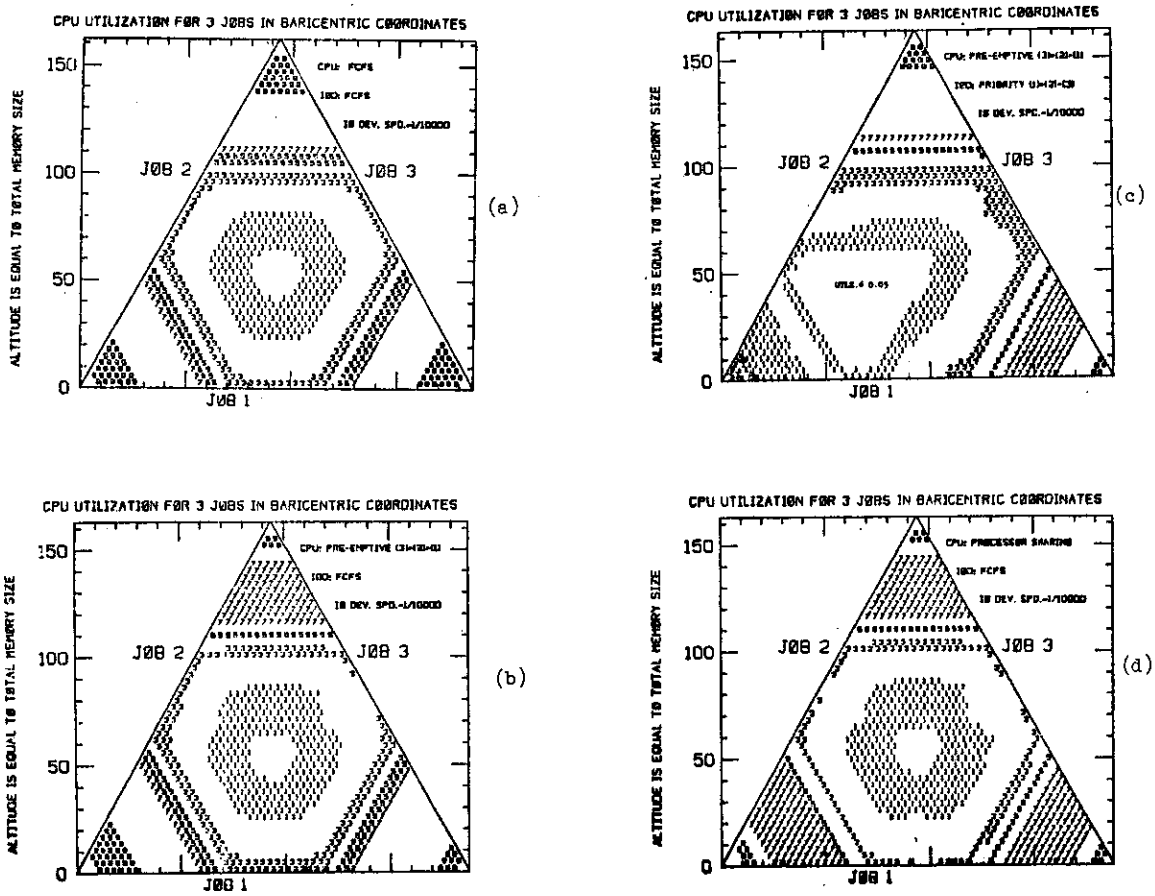


Fig. 4.16 CPU utilization with four types of scheduling
IOD speed=1/10000

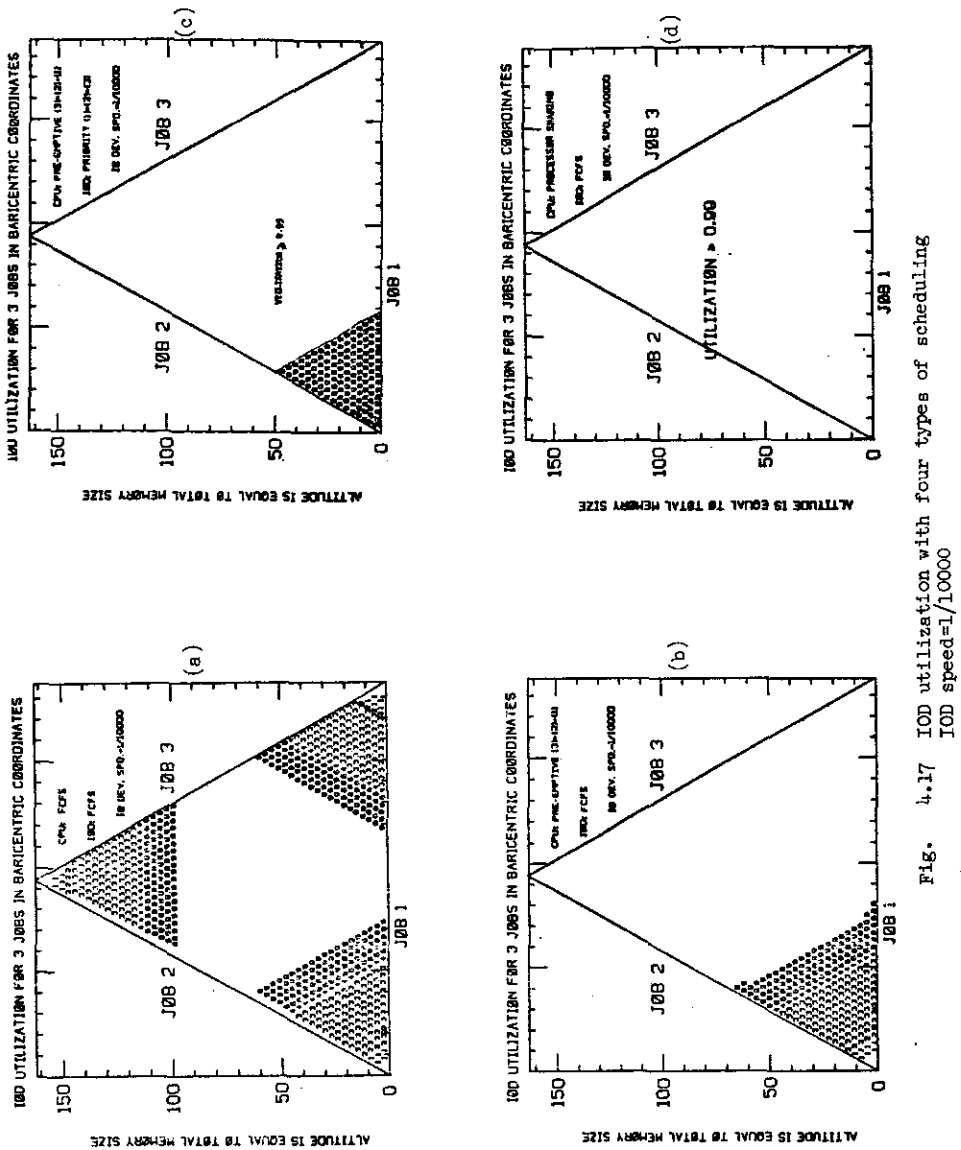


Fig. 4.17 IOD utilization with four types of scheduling
IOD speed=1/10000

of job 3 for very large memory space m and the sharply and unevenly decreasing service rate of the other two jobs for very small values of m.

Processor sharing is the last type of scheduling which we shall consider here. In this scheme, the CPU is shared among all jobs present in the center. We, therefore, expect that the flow of the jobs in CPU should become smoother so that no job can temporarily block the execution of any other job.

In order to simplify the visual comparison of the effect of scheduling on the device utilization in the model, we have arranged the CPU and IOD utilization plots for all four types of scheduling in Figures 4.14 to 4.17. In Figures 4.16 and 4.17, a slower IO device is used.

When processor sharing scheduling is used at the CPU, the IOD utilization is high for all regions, as we can see in Figures 4.15d and 4.17d. The CPU utilization pattern in this case is symmetric with respect to the center of the triangle (Figures 4.14d and 4.16d). When compared to Type I scheduling, we get about 10% to 20% less CPU utilization with Type IV scheduling in the corresponding regions.

4.3.6 Effect of Device Scheduling on Queueing Times

Earlier we discussed the effect of memory allocation policies on the queueing time of the jobs at the CPU. In that discussion, we used Type I scheduling and showed that the queueing time curves follow the pattern of CPU utilization curves with respect to different allocations, and we used Figure 4.9 to demonstrate this point. In the same figure, we have also shown the average job queueing time at the CPU for two jobs and Type II scheduling (dotted curve). The essence of the effect of priority scheduling of the queueing times can be seen in this plot. We recall that

with Type II scheduling, job 2 has higher preemptive priority than job 1 at the CPU. Referring to the figure we can see that as the CPU service time of job 2 decreases, the total average queueing time at the CPU decreases accordingly. This curve goes even lower than the minimum average queueing time obtained from Type I scheduling with symmetric allocation.

As the CPU service time of the higher priority job increases, the right half of the figure, the average queueing time under Type II scheduling, approaches the curve of Type I scheduling.

Thus, when the total performance of the system is considered, we have seen that when a higher priority is given to the short jobs the average queueing times decrease significantly at the cost of moderate degradation of CPU efficiency. As we can see in Table 4.5, for N=2 jobs, this degradation is fairly small compared with CPU utilization under Type I scheduling.

In Figure 4.18, the reciprocal of the harmonic average dilation time at the CPU and IOD for N=3, and Type II and III scheduling, is shown. As before, the high values for D_s^{-1}, s=CPU or IOD, indicate the low dilation times. We note that when we go from Type II to Type III scheduling, the major change at the CPU is basically the enlargement of the areas with lower dilation times.

When we use Type III scheduling, the interpretation of the waiting times at the IOD becomes rather interesting. For instance, when job 3, which has the highest preemptive priority at the CPU, has the longest service time in that center, it becomes the only job which does not experience any queueing delay at the IOD, in spite of the fact that it has the least priority in this center (Figure 4.18d, around lower left vertex). This is because this job blocks the other jobs at the CPU and when it

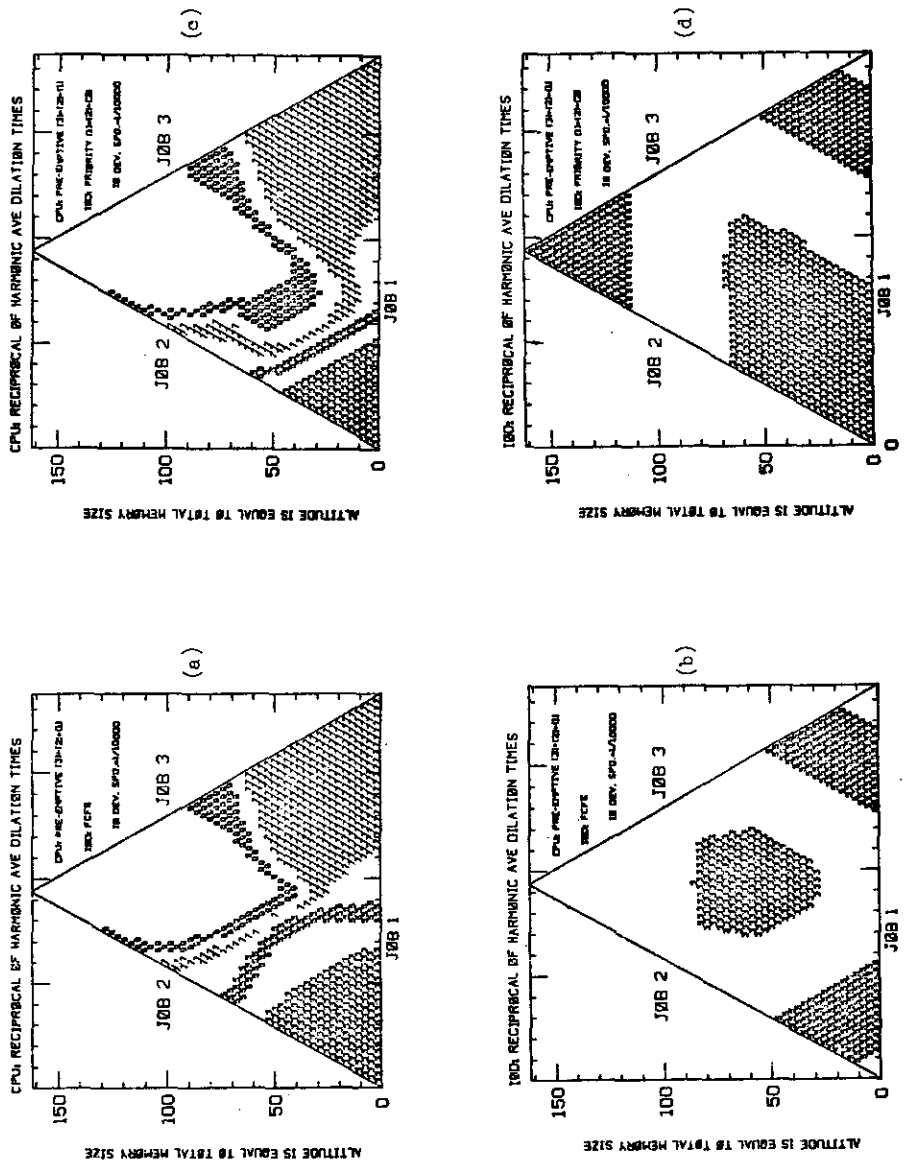


Fig. 4.18 Reciprocal of average harmonic dilation times in CPU and IOD with type II and III scheduling

leaves that center it will most probably find an empty IOD.

In the following two sections, we shall investigate the effect of the speed and the service mechanism at the IOD on the performance of the system.

m_1	m_2	CPU utilization		IOD utilization		Average CPU queueing time	
		I	II	I	II	I	II
113	50	0.75	0.65	0.41	0.98	14136	384
112	51	0.71	0.61	0.46	0.98	11657	361
111	52	0.67	0.57	0.51	0.98	9505	340
110	53	0.63	0.54	0.56	0.98	7710	318
109	54	0.61	0.51	0.59	0.98	6742	316
108	55	0.58	0.50	0.61	0.98	6265	348
107	56	0.58	0.49	0.62	0.98	5959	383
106	57	0.57	0.49	0.63	0.97	5690	418
105	58	0.54	0.46	0.67	0.97	4784	425
104	59	0.51	0.43	0.71	0.97	4011	436
103	60	0.46	0.40	0.75	0.97	3202	464
102	61	0.46	0.40	0.76	0.96	3013	524
101	62	0.44	0.39	0.78	0.96	2679	587
100	63	0.41	0.37	0.81	0.95	2222	611
99	64	0.40	0.37	0.82	0.95	2075	676
98	65	0.39	0.37	0.83	0.94	1905	744
97	66	0.39	0.37	0.84	0.94	1784	832
96	67	0.39	0.37	0.85	0.93	1760	934
95	68	0.39	0.37	0.85	0.93	1672	997
94	69	0.39	0.38	0.85	0.92	1654	1109
93	70	0.39	0.38	0.86	0.91	1621	1203

TABLE 4.5 CPU, IOD Utilizstions and Average Queueing Time, Type I and Type II Schedulings.

4.37 Effect of IO Device Speed on the Performance of the Model

For the numerical evaluation, we have considered two different IOD speeds. There is a slow IO device with the average speed of 1/10000 and a relstively faster device with an average speed of 1/4200. The service times at the IOD are, therefore, exponentially distributed with the rate equal to the average speed of the IO unit used. The speed of the faster device is taken to be almost equal to the fault rate of the WATFIV program model at m=M/2. In Table 4.6, the IOD and CPU utilization for N=2 and two different IO device speeds are shown (also Figures 4.4 and 4.5).

Memory allocation		CPU utilization		IOD utilization	
m_1	m_2	1/4200	1/10000	1/4200	1/10000
30	133	0.94	0.87	0.10	0.23
35	128	0.94	0.87	0.10	0.23
40	123	0.94	0.87	0.10	0.23
45	118	0.94	0.87	0.10	0.23
50	113	0.92	0.81	0.15	0.32
55	108	0.81	0.61	0.33	0.60
60	103	0.76	0.51	0.44	0.71
65	98	0.69	0.41	0.57	0.83
70	93	0.69	0.40	0.61	0.85
75	88	0.67	0.38	0.66	0.88
80	83	0.67	0.38	0.66	0.88

TABLE 4.6 CPU and IOD Utilization for Two Different IO Device Speeds and Type I Scheduling.

Since both centers have FCFS service policy, the flow of IO requests arriving at the IOD is determined mainly by the job which has the longer CPU service time. In this table, we can see that as IO requests become

more frequent the speed of the paging device becomes more crucial in the efficiency of the CPU. This point can be seen by comparing the first four entries of the table with the last three entries. The CPU utilization improvement for the first four entries is only about 7% using a device which is 2.4 times faster. In the last three entries, this improvement is about 29%.

As we reduce the speed of the IO device, the requests stay longer in this center and the utilization increases correspondingly. For a low rate of IO requests, from the first four entries in Table 4.6 we get 13% increase, and for the higher rate of IO requests we have 22% increase from the last three entries in the same table. Of course, as we mentioned before, the IO request traffic is related to the allocation policy in CPU.

4.3.8 Paging Drum Model

So far we have assumed that the service time of jobs at the IOD is exponentially distributed. From our earlier discussion about the paging drum, one would question the validity of this assumption when a paging drum is used. We will use a more realistic drum model to see how sensitive the results of our analysis are to the detailed behavior of the IO device.

In order to have a reasonable basis for the comparison, we define the rotational period so that the average service time at the paging drum is equal to the IOD service time in the previous assumptions. The average service time at the paging drum is equal to the average rotational latency plus the transfer time of one page. Therefore, if s is the number of sectors and T is the rotational period, the average service time is equal to T/2 + T/s. We can find T by equating this quantity to the speed of our, say, slower disk.

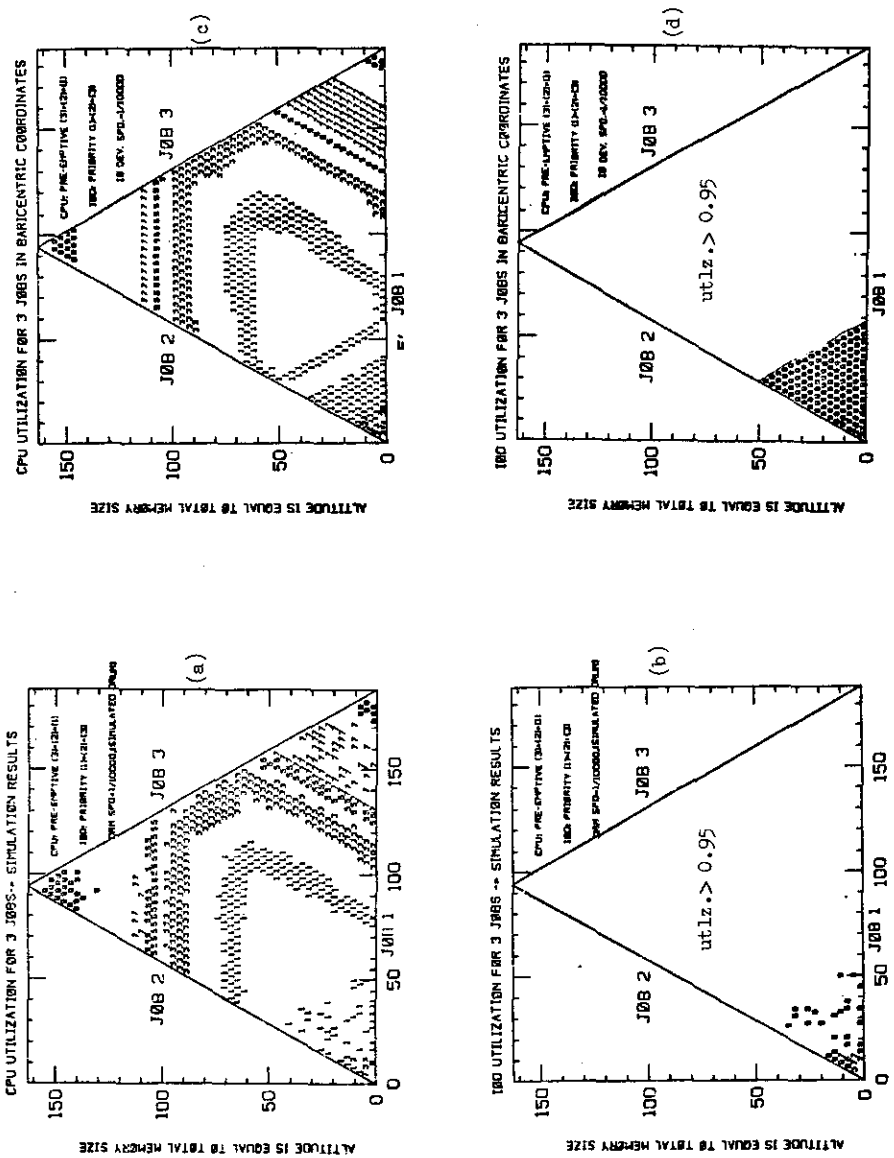


Fig. 4.19 Comparison of the results from simulation with drum model with the results of analysis with exponential IOD service time assumption.

We use a simulation to find the device utilization of the model for N=3 and Type III scheduling. When an IO request arrives at the paging drum, the random position of the Read/Write head and the sector number of the requested page are found. Then the IOD service time is computed as the sum of the rotational latency to bring the head to the beginning of the sector and the transfer time of one sector.

In Figure 4.19, we can see the results obtained from the simulation. Figures 4.19a and 4.19b give the CPU and IOD utilization using a paging drum. On the same page, Figures 4.19c and 4.19d give the same quantities obtained from the analysis of the model using the exponential service time assumptions. Inspecting these figures, we notice the similarity of the results obtained from the two approaches. The CPU and IOD utilization are basically very close together in the corresponding areas.

In the next section, we validate the earlier results by using a more realistic program behavior model.

4.3.9 Simulation of the Model Using Actual Program Traces

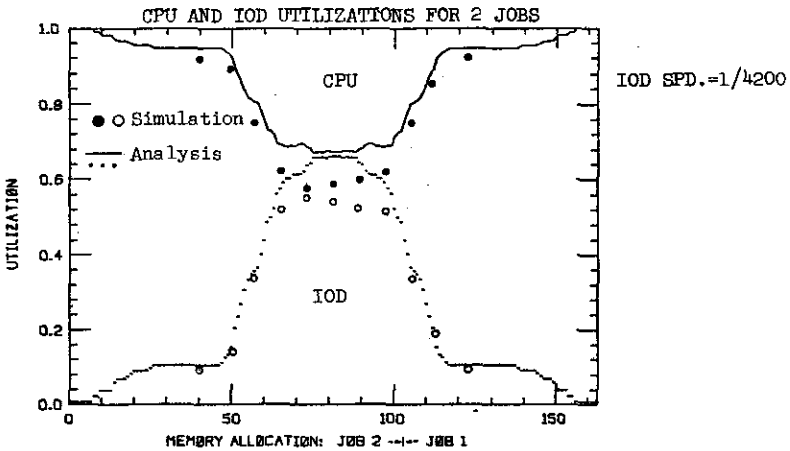
The service times of a job at the CPU is essentially the time between two successive IO requests (here, page faults). In our analysis, so far, we have assumed that the interfault intervals are exponentially distributed with a mean which is obtained by measuring the average interfault periods in an actual program. Earlier we saw that the actual interfault distribution of the programs have generally a higher coefficient of variation than one. To see how the analytical results compare with the results obtained when more realistic program behavior is assumed, we perform a trace driven simulation of the model. The simulation is driven by the actual page reference trace of the WATFIV program. In this case, the occurrence

of a page fault is precisely determined by using a local LRU replacement policy for each active program. Since the LRU rule is used, it is more convenient to work with the LRU distance trace of the programs. The assumption for service distribution at the IOD is still an exponential distribution as before.

The result of simulation for N=2 and Type I and II scheduling are shown in Tables 4.7a and 4.7b respectively. In the same tables, the CPU and IOD utilizations with exponential assumption on CPU service times are shown. These points are plotted in Figures 4.20 and 4.21.

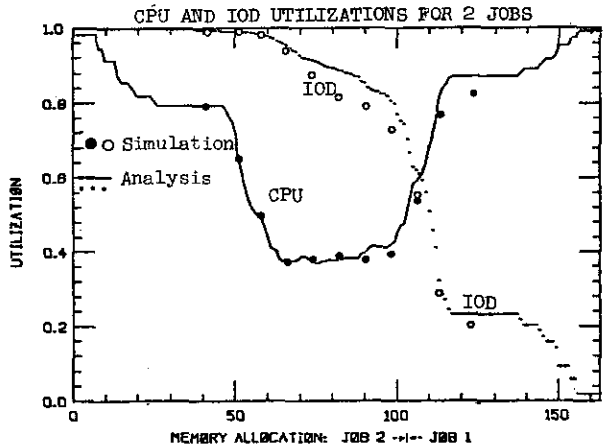
For both types of scheduling, the results of simulation are fairly consistent with these predicted by the model. The maximum discrepancy occurs at symmetric allocation with Type I scheduling which is about 9% for CPU utilization and 14% for IOD utilization.

In using the actual program traces in the simulation, one hard problem was to use the identical copies of one program while keeping the behavior of each program fairly independent from each other. After some experiments, it was decided to consider the trace of the program as a cyclic string. One job was started executing from the beginning of the string and the other job was started from some point far away. Since the trace of the program (WATFIV here) is very long, we could get fairly independent behavior from the jobs in the simulation run.



IOD SPD.=1/4200

Fig. 4.20 Comparison of trace driven simulation with analytical results. Type I scheduling.



IOD SPD.=1/10000

Fig. 4.21 Comparison of trace driven simulation with analytical results. Type II scheduling.

m_1	m_2	CPU Utilization		IOD Utilization	
		program (WATFIV)	exp. model	program (WATFIV)	exp. model
123	40	0.92	0.94	0.09	0.10
113	50	0.89	0.94	0.14	0.15
106	57	0.75	0.80	0.34	0.36
98	65	0.61	0.69	0.50	0.57
90	73	0.57	0.68	0.54	0.64
82	81	0.58	0.67	0.52	0.66
74	89	0.59	0.68	0.52	0.64
66	97	0.61	0.68	0.50	0.59
63	100	0.64	0.71	0.46	0.52
58	105	0.75	0.80	0.33	0.37
51	122	0.85	0.89	0.19	0.20
41	122	0.92	0.94	0.09	0.10

TABLE 4.7a Disk Speed = 1/4200, Type I Scheduling

m_1	m_2	CPU Utilization		IOD Utilization	
		program (WATFIV)	exp. model	program (WATFIV)	exp. model
122	41	0.79	0.79	0.99	0.99
112	51	0.65	0.65	0.98	0.98
105	58	0.50	0.49	0.98	0.97
97	66	0.38	0.37	0.94	0.94
89	74	0.38	0.37	0.87	0.91
81	82	0.39	0.37	0.80	0.88
73	90	0.37	0.39	0.78	0.85
65	98	0.39	0.42	0.73	0.79
57	106	0.54	0.59	0.55	0.61
50	113	0.77	0.81	0.29	0.32
40	123	0.82	0.87	0.20	0.23

TABLE 4.7b Disk Speed = 1/10000.
Type II Scheduling [IOD: FCFS, CPU: (2) > (1)]

4.4 CONCLUSION

In this chapter, we studied a model for the interaction of a multiprogrammed CPU and its paging device. A closed cyclic queue was used to investigate the effect of memory allocation policies, scheduling disciplines, and the characteristics of IO devices on the performance and the flow of the jobs in the system. We have allowed each job to have its own independent service time at the CPU, which is determined by the size of the common memory which is assigned to it. The paging behavior of each program is obtained by the measurements on the paging statistics of an actual program. A trace driven simulation has been used to validate the result of the analysis when the exact paging behavior of a program is used.

Most of the results we have obtained in this chapter, rather than being strictly conclusive, demonstrate basically the fundamental relationship between the decision policies and the performance measures, and the tradeoffs involved. For instance, when short jobs are given higher CPU priority, the queueing time decreases significantly at the cost of slightly lower CPU utilization compared with FCFS policy. The extent of many such tradeoffs have been explored in this chapter by considering a large number of possible memory allocation schemes and execution priority assignments. In this regard, one of the major problems is the presentation of the results in a readable format. We were able to obtain fairly descriptive illustrations by using the baricentric coordinates for the system with up to three jobs.

Some of the major results which can be summarized are as follows. We have seen that in this model when the page fault rate of the programs are not the same, the CPU utilization increases. We obtained the minimum

CPU utilization with the symmetric allocation of the memory among active jobs. Scheduling can have significant effect on the resource utilization and average waiting time at each center. When the jobs with shorter CPU times are given higher priority in that center, the contention at the IO device can decrease CPU utilization and throughput. We can alleviate this problem either by increasing the availability of IO devices or reverse the priority of the jobs at the IOD. Processor sharing scheduling gives slightly lower CPU utilization than FCFS policy. However, the IO device is fully utilized with the latter scheduling at the CPU. The system completion rate increases if we allocate a larger share of the memory to the jobs which have a shorter total CPU service requirement.

The simulation of the model showed that when we use fairly skewed paging behavior the results are comparable to those predicted by the queueing analysis. Therefore, we expect that our results should be applicable in more realistic environments.

4.5 APPENDIX (Generalized Wald's equation)

Let R be a random number with p.d.f. H(.) and assume $E[R] < \infty$. Let U_i, i=1,2,... be i.i.d. random numbers which specify the time between successive events in a renewal process. Assume the U_i's are exponentially distributed with the rate α. Let N(t) be a stochastic process which counts the number of renewals from time 0 to t.

Since the U_i's are exponentially distributed, N(t) is defined by a Poisson process, where:

$$Pr\{N(t) = n\} = e^{-\alpha t} \frac{(\alpha t)^n}{n!} .$$

We want to find the expected number of renewals during the random time R.

We have:

$$Pr\{N(R) = n\} = \int_0^\infty e^{-\alpha t} \frac{(\alpha t)^n}{n!} dH(t)$$

Then

$$E[N(R)] = \sum_{n=0}^{\infty} n \, Pr\{N(R) = n\} = \sum_{n=0}^{\infty} n \int_0^\infty e^{-\alpha t} \frac{(\alpha t)^n}{n!} dH(t)$$

$$E[N(R)] = \int_0^\infty e^{-\alpha t} dH(t) \sum_{n=0}^{\infty} n \frac{(\alpha t)^n}{n!} =$$

$$= \int_0^\infty e^{-\alpha t} dH(t) \, \alpha t \sum_{n=0}^{\infty} \frac{(\alpha t)^n}{n!}$$

$$= \int_0^\infty e^{-\alpha t} dH(t) \, \alpha t \, e^{\alpha t} =$$

$$= \alpha \int_0^\infty t dH(t) = \alpha \cdot E[R] = \frac{E[R]}{E[U]} \quad \blacksquare$$

4.6 BIBLIOGRAPHY

1. Baskett, F., Chandy, J.M., Muntz, R.R., "Open, closed and mixed networks of queue with different classes of customers," JACM 22, 2 (April 1975).
2. Chamberlin, D.D., Fuller, S.H., Liu, L.Y., "An analysis of page allocation strategies for multiprogramming systems with virtual memory," IBM J. of Res. Dev., (September 1973).
3. Coffman, E.G., Ryan, T.A., "A study of storage partitioning using a mathematical model of locality," CACM 15, 3 (March 1972).
4. Denning, P.J., Spirn, J.R., "Dynamic storage partitioning," 4th Symposium on Operating Systems Principle, ACM (1973).
5. Fuller, S.H., Baskett, F., "An analysis of drum storage units," JACM 22, 1 (January 1975).
6. Gordon, W.J., Newell, G.F., "Closed queueing systems with exponential servers," Oper. Res. 15 (1967).
7. Koenigsberg, E., "Cyclic queues," Oper. Res. Quart. 9,1 (1958).
8. MacEwen, G.H., "A preliminary study of disk driven process scheduling," Tech. Rep. No. 33, Dept. of Computer and Information Science, Queen's University, Kingston, Ontario, Canada.
9. Oden, P.H., Shedler, G.S., "A model of memory contention in a paging machine," CACM 15, 8 (August 1972).
10. Sekino, A., "Throughput analysis of multiporgramming virtual memory computer system," IBM Res. Reprt RC-4092 (October 1972).
11. Shedler, G.S., Tung, C., "Locality of page reference strings," SIAM J. on Computing, 1,3 (September 1972).
12. Sherman, Baskett, Broown

13. Smith, A., "Performance analysis of computer components," Ph.D. thesis, Computer Science Dept., Stanford University (August 1974).

14. Stonebraker, M., "Optimal memory allocation in a multiprogrammed, paged environment," Dept. of Electrical Engineering and Computer Science, University of California, Berkeley.

15. Strauss, J.C., "An analytic model of the HASP execution task monitor," CACM 17, 12 (December 1974).

CHAPTER 5

SUMMARY AND FURTHER WORK

In this chapter, we summarize the major results obtained in this work and suggest some areas for further study. For more detail discussion, we refer our readers to the conclusion section of each foregoing chapter.

In Chapter 1, the objective was to get an insight into the dynamic page reference behavior of computer programs. In this regard, we considered the generated working set size string and the LRU stack distance string. We considered each string as a realization of a stochastic process and used the tools from the analysis of time series to investigate the properties of the underlying structure. For instance, from the power spectrum of the observed LRU stack distances, we concluded that the sequence mainly consists of random fluctuations. This approach could be a basis for the comparison of the effect of system parameters on each string. We examined the effect of page size and window size variations on the generated working set size sequence. Time series analysis can also be used to validate the models for different aspects of program behavior. We demonstrated the problems with a working set size model in capturing the serial correlation of the sampled working set sizes, by comparing the autocorrelation coefficients of the actual observations with those computed from the model.

Time series analysis have shown to be valuable in studying the computer performance problems. Because of the multiplicity of parameters and complexity of modern computers, other techniques like multidimensional and cluster analyses seem to be also appropriate for the study of different aspects related to the performance of these systems.

In Chapter 2, we considered the problem of the performance of paging algorithms. We presented new results on the performance of several algorithms. We established the result that inexpensive and practical algorithms, like CLOCK and Modified Working Set, give performances which are close to the performance of more elaborate algorithms, like LRU and Working Set, respectively. We found the expected fault rate of MWS for the independent reference model. A harder problem is to find the expected fault rate of CLOCK for the same model, and show that it is close to the fault rate of the LRU algorithm.

In Chapter 3, we proposed the A$\emptyset$ Inversion Model and obtained the major result that the independent reference model can be used effectively to predict the actual fault rate of programs under several algorithms and with different memory sizes. The technique is also very promising in the evaluation of filing systems and evaluation of memory hierarchies with unequal Read/Write operation costs.

In Chapter 4, we considered the problem of the interaction of page scheduling and device scheduling in a multiprogramming system. We showed that when the main memory is partitioned among active jobs, we obtain better CPU utilization by allowing at least one program to execute longer in CPU. For two identical programs, this amounts to dividing the main memory asymmetrically between them. We showed that by proper execution priority assignment in CPU and IOD, we can tune the system to obtain high device utilization and low job waiting times. As an extension of the model, we might consider multiple processing units with shared IO devices. We can explore the performance of several possible memory allocation policies and scheduling disciplines in the system.