MAC TR-148

PROGRAM RESTRUCTURING FOR VIRTUAL MEMORY SYSTEMS

Jerry William Johnson

March 1975

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2095 which was monitored by ONR Contract No. N00014-70-A-0362-0006.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

This empty page was substituted for a blank page in the original document.

PROGRAM RESTRUCTURING FOR VIRTUAL MEMORY SYSTEMS

by

Jerry William Johnson

ABSTRACT

The problem area addressed in this report* is program restructuring, a method of reordering the relocatable sectors (subroutine and data modules) of a program in its address space to increase the locality of the program's reference behavior, thereby reducing the number of page fetches required for its execution in a virtual memory system.

Theoretical upper and lower (optimum) bounds are derived for the paging performance of programs over all partitions of relocatable sectors into pages.

Program restructuring techniques are developed which use intersector reference models based on sector working sets and sector stack distances. These intersector reference models identify the local reference behavior, and clustering procedures are developed that use this local reference behavior to rearrange sectors into pages such that significant improvement in paging performance is obtained.

Results of measurements of paging performance obtained in the computer laboratory are discussed. The relationship between the paging performance of a program restructured by the practical restructuring algorithms and the theoretical bounds on paging performance are compared.

---

*This Technical Report reproduces a thesis of the same title submitted to the Department of Electrical Engineering, M.I.T., on June 15, 1974, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

ACKNOWLEDGEMENT

I especially express my appreciation to my thesis supervisor, Professor Stuart E. Madnick, for the substantial time and effort he spent supervising the thesis and in particular for his enthusiasm throughout the course of the research.

I also wish to thank Professor J. D. Bruce and Professor V. R. Pratt for their helpful comments which greatly improved the presentation of the work and for their encouragement throughout the course of the research.

Appreciation is extended to IBM's Cambridge Scientific Center for making the CP-CMS computer system available for conducting the experimental part of this research. I also wish to single out Don Hatfield and Coyt Tillman of IBM for their helpful assistance and many editorial comments.

I thank the members of the Programming Technology Division of Project MAC for making the Dynamic Modeling System available for composing and reproducing this report on-line. I also thank Albert Vezza for his encouragement and many helpful suggestions during the research period, Stewart Galley for his unifying editorial comments and Susan Pitkin for performing all the on-line editing that transpired between the thesis and this report.

I especially thank my wife, Janet N. Johnson, for her patience and understanding throughout my years of graduate study at M.I.T.

The author is grateful to Project MAC and IBM for their financial support.

Work reported herein was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency, Department of Defense, under office of Naval Research Contract Nonr-4102(01).

TABLE OF CONTENTS

SECTION	PAGE
ABSTRACT	i
ACKNOWLEDGEMENT	ii
TABLE OF CONTENTS	iii
LIST OF FIGURES	vii
LIST OF TABLES	ix
CHAPTER 1 INTRODUCTION	1
1.1 Introduction	1
1.2 Motivation	1
1.3 The Nature of Program Restructuring	3
1.4 Importance of Program Restructuring	7
1.4.1 Comeau's Results	8
1.4.2 Results of Hatfield and Gerald	8
1.4.3 Program Design Considerations	10
1.4.4 Related Performance Benefits	12
1.5 Related Research and the Need for Further Research	12
1.5.1 Intersector Reference Models	16
1.5.2 Reordering Procedures	17
1.5.3 Sector Ordering Evaluators	19
1.5.4 Performance Bounds	20

1.6 Summary of Goals 22

CHAPTER 2 FORMALIZATION OF VIRTUAL MEMORY SYSTEMS 24

2.1 Introduction 24

2.2 Major Parameters of a Two-Level Virtual Memory System 24

2.2.1 Configuration 25

2.2.2 Program Behavior 28

2.2.3 Automatic Management Algorithm 30

2.3 The Virtual Storage Model 32

2.4 Performance Measures 32

2.4.1 Effective Capacity 33

2.4.2 Effective Cost 33

2.4.3 Effective Access Time 33

2.4.4 Page Trace Simulation 36

2.5 Page Fetch Function Performance Model 38

2.5.1 Replacement Algorithm Considerations 39

2.5.2 Program STructure Considerations 44

2.6 Sector Fetch Function Performance Model 50

CHAPTER 3 PAGING PERFORMANCE BOUNDS 54

3.1 Introduction 54

3.2 Lower Bounds 56

3.3 Upper Bounds 71

3.4 Simple Example of Computing Bounds 75

3.5 Extensions to Lower Bounds 80

3.6 Bound for Working Set Management 106

3.6.1 Lower Bounds for Working Set Management 110

3.6.2 Upper Bounds for Working Set Management 119

CHAPTER 4 INTERSECTOR REFERENCE MODELS 122

4.1 Introduction 122

4.2 Intersector Reference Models 123

4.2.1 The HG Intersector Model 124

4.2.2 Working Set Intersector Reference Models 126

4.2.3 LRU Stack Intersector Reference Model 136

CHAPTER 5 CLUSTERING PROCEDURES 143

5.1 Introduction 143

5.2 Clustering Procedures 143

5.3 Nearest Neighbor Methods 144

5.4 Hatfield and Gerald Method 150

5.5 Sector Interchange Procedure 151

5.6 Intercluster Bonding Method 157

CHAPTER 6 EXPERIMENTAL RESULTS 167

6.1 Introduction 167

6.2 Restructuring Phase 1 174

6.2.1 Constrained Procedures 175

6.2.2 Unconstrained Procedures 180

6.2.3 Theoretical Bounds 184

6.3 Restructuring Phase 2 191

6.4 Restructuring Phase 3 194

6.5 Effects of Input Data 197

CHAPTER 7 DISCUSSION AND CONCLUSION 203

7.1 Introduction 203

7.2 Summary 203

7.3 Further Work 204

REFERENCES 206

LIST OF FIGURES

FIGURE		PAGE
1	(a) Two Level Hierarchical System	27
	(b) Virtual or Composite Memory System	27
	(c) Representative Parameters for Several Virtual Memory Systems	27
2	Logical Address Structure	29
3	Lower Bound on FFp Given by Theorem 1	65
4	The Allowable Values of FFp as a Function of |Mp|	76
5	Parachor Curve of FFs(|Ms|,ST,Fd,Ro)	134
6	Parachor Curve Illustrating Values for D	142
7	Results for Phase 1	176
8	Results for Phase 1	177
9	Results for Phase 1	179
10	Results for Phase 1	181
11	Results for Phase 1	183
12	Results for Phase 1	185
13	Results for Phase 1	189
14	Results for Phase 1	190
15	Results for Phase 2	192
16	Results for Phase 2	195
17	Results for Phase 3	196

18 Results for Phase 1 199

19 Results for Phase 1 200

20 Results for Phase 2 202

21 Results for Phase 3 202

LIST OF TABLES

TABLE		PAGE
1	Major Parameters of Two-Level Hierarchical Virtual Memory Systems	26
2	Example of Page Trace Simulation to Determine FFp	37
3	Parameters for Curves of Figure 12	186
4	Parameters for Curves of Figure 15	193

This empty page was substituted for a blank page in the original document.

CHAPTER 1

1.1 Introduction

In this chapter, the problem of restructuring programs to improve their paging performance in virtual memory systems is presented.

1.2 Motivation

As the use of multiprogramming and virtual memory techniques has become more widespread, the performance of paged virtual memory hierarchies has become more important. The fact that paged virtual memory systems can be made to perform efficiently at all depends primarily on an inherent property of program behavior known as "program locality" [D1,D2,D3,D4]. Program locality arises from empirical observations that actual programs cluster their memory references so that, during any interval of time, only a subset of the information available is actually referenced. If a program is favoring a subset of its information at some particular time, we should like very much to have this subset in primary memory. As a result, much of the research efforts made to optimize the performance of programs in virtual memory systems were spent devising strategies for page management algorithms that could maximize the probability of finding in primary memory the information

needed by the CPU at the time it is referenced, thereby minimizing the number of page fetches. Several studies [B1,B2,D2] have shown that this probability strongly depends on the reference patterns of the program being executed, that is, on how distributed in the virtual address space are the information items successively referenced by the processor. Generally, the higher the degree of locality of a program, the higher the performance of the virtual memory system with respect to that program. However, several comparisons of page replacement algorithms have been reported [B1,M1,C1], often realizing as much as 30 to 40 percent improvement from one algorithm to another for certain programs. In particular, an algorithm has been found [B1,M1] that gives the minimum number of page fetches for a program. Even though the minimum replacement algorithm is practically unrealizable, as it requires a knowledge of the future page references of the program every time a page fetch occurs, the algorithm is important because one can use it as a theoretical bound against which the performance of any other paging algorithm can be compared.

In all the studies of developing page management algorithms to increase the performance of virtual memory systems, the program's page reference pattern and hence its locality is considered as a non-modifiable input to the system. In contrast to the exploitation of the existing locality of programs by paging algorithms, relatively little attention has been paid to another important method of obtaining better performance from virtual memory systems. This method is to increase the

degree of locality of the program to be executed. Even less research has been focused on developing bounds on the performance improvement due to optimum program locality.

In this report, we propose to focus most of our research efforts in the study of program restructuring [C2,H1,D2], a method of rearranging the relocatable sectors (subroutine and data modules) of a program, to increase the locality of the program's reference behavior and thereby reduce the number of page fetches required for execution in a virtual memory system. The essential idea behind program restructuring to bring about this localization in its reference behavior is to take sectors of the program that are used closely together in time and cluster them closely together in the virtual address space.

1.3 The Nature of Program Restructuring

The nature of program restructuring methods that have been proposed so far can be classified along several dimensions. With respect to the extent of the programmer's involvement, restructuring can be manual or automatic, depending on whether rearrangement decisions are made by man or computer. With respect to the level at which restructuring is applied, we can make a distinction between external and internal reordering. In external reordering, the sectors which are rearranged in virtual memory are relocatable sectors of instructions

and/or data. Internal restructuring consists of reordering parts of relocatable sectors with respect to each other or simply deciding where to insert page breaks in the code [K1,Y1]. External restructuring is faster and cheaper since it never requires reprogramming. With respect to the type of information on which a restructuring procedure is based, there are static methods, which only make use of the knowledge of the static properties of the program, and dynamic methods, which are based on data, collected during execution, representing the dynamic behavior of the program.

Algorithms for automatic restructuring can be applied at compilation time if they are static: typical examples are those methods which construct a graph model of the program to be restructured, whose sectors are represented by vertices (whose weight is the size of the sector) and arcs represent the transitions (data or control references), and then cluster vertices according to connectivity considerations or to the cyclic structure of the graph [B3,L1,R1,Y2]. We are interested in automatic, external program restructuring methods based on the program's dynamic behavior and in subsequent discussions we will simply call this program restructuring.

In order to provide more insight into the character of program restructuring which we will study, we make the following general assumptions. A program consists of a finite set of relocatable data and procedure sectors. These sectors are opaque, since we are concerned with the interactions among the sectors and we are not concerned with what

goes on inside each sector. The average size of a relocatable sector is small with respect to the size of a page (between one-tenth and one-half page size).

Informally, the basic approach to program restructuring is to run the program with a set of "typical" input data, record the sector reference behavior, formulate an intersector reference model based on the recorded information, and then apply a program restructuring procedure which uses the model of intersector reference behavior to reorder or partition the sectors into logical pages such that the intersector references among sectors in different pages is minimized.

The aim of program restructuring is to increase the locality of the program's address reference pattern by reordering the relocatable sectors in virtual memory such that sectors that are needed within a relatively short time of one another are found in the same logical page or in logical pages that would otherwise tend to be in primary memory at the same time. The act of restructuring will tend to create a situation in which there are either very strong or very weak affinity bonds between logical pages. The resultant goal of program restructuring is to minimize the page fetches required by a program during its execution in a virtual memory system. This is a very difficult goal to achieve because the number of page fetches is a function of primary memory allocated to the program, the page size, the fetch and replacement policies, the sector reference behavior, and the selected ordering of sectors into

logical pages.

In order to pose more formally the nature of the restructuring problem for any program modeled by a set of relocatable sectors of specified size and a measured sector trace describing the sector reference behavior, we need the following definitions.

A program is regarded as a directed graph G of m nodes, of size $S_i > 0$, $i = 1,\ldots,m$. The nodes represent relocatable sectors. Let N be the page size, such that $0 < S_i < N$ for all i. Let $C = (c_{ij})$, $i,j = 1,\ldots,m$ be a weighted connectivity matrix describing the edges of G. The edges of G represent the intersector reference behavior of the program. With edge (i,j) is associated a cost $c_{ij} \geq 0$ of traversing that edge. How to choose the best intersector reference model C from the measured sector trace is an important research problem. However, c_{ij} might represent the probability that sector i references sector j, or c_{ij} might be the total number of times sector i makes a data reference or a transfer of control to sector j, or ideally c_{ij} would represent the number of page fetches which would occur due to sector i referencing sector j in a given virtual memory system unless i and j were grouped into the same page.

Let n be the number of logical pages of the restructured program. An n-way restructuring of G is a set of nonempty, pairwise disjoint subsets (pages) of G, $p_1,\ldots,p_n$ such that

$U_{i=1}^{n} Pi = G$ and $|p_i| \leq N$ for all i, where $|p_i|$ stands for the size of subset p_i, and equals the sum of the sizes of all the sectors of Pi. The cost definition for the restructured G is the summation of C_{ij} over all i and j such that i and j are in different subsets (pages). The cost is thus the sum of all external costs in the partition of G. A restructuring of G is optimal if it achieves minimum external cost or equivalently maximum internal cost, because the total cost of all edges is constant.

We can now point to two distinct and difficult problems associated with program restructuring. One is, given G and C, how to find an optimum restructuring of G, and the other is how to model the intersector reference behavior C such that an optimum solution to the restructuring problem formulated on C gives the minimum number of page fetches for a virtual memory system.

1.4 Importance of Program Restructuring

The potential of program restructuring for improving the performance of programs running in a virtual memory system can be best illustrated by citing some reported results.

1.4.1 Comeau's Results

The first published results of program restructuring to increase the performance of programs in a virtual memory system was in 1967 by L. W. Comeau [C2]. Comeau reports that the ordering of relocatable sectors of code over virtual pages can have a profound effect on paging performance. In particular, he found that the number of page fetches during an assembly could be decreased by a factor of five by changing the ordering of the monitor modules at load time. Four orderings of the monitor modules were compared under the same primary memory constraints and the same paging algorithms. The alphabetical ordering produced 6500 page fetches, the random order gave 4200 fetches, and order based on knowledge of the page size and functions of the modules resulted in 2400 fetches, and an ordering based on the knowledge of the functions of the modules, page size and a detailed history of intermodule activity generated while the program was in execution produced 1200 fetches.

A subsequent experiment by Tsaco, Comeau and Margolin [T1], performed on an IBM/360 Model 40 in a CP/40 environment, shows that paging activity is reduced much more by a good load sequence of operating system subroutines than by replacement algorithms.

1.4.2 Results of Hatfield and Gerald

In 1971 Hatfield and Gerald [H1] reported that improvements in paging performance, on the IBM/360 Model 67, in the range of two-to-one to ten-to-one can occur by using experimental techniques, based on information compiled from sector reference traces, to restructure the relocatable sectors of compilers, editors, and assemblers. This is a significant reduction in the number of page fetches experienced by existing, frequently executed programs, and how close this is to the optimum reduction is currently unknown.

Also, they present an excellent discussion supported by many detailed measurements, which shows that the sector reference behavior of most programs they examined (especially the system programs: compilers, assemblers, editors, etc.) proved to be remarkably insensitive to the input data in rather large domains. This is very important because there is no merit in tracing a program, massaging the traced data, reloading sectors, and measuring changes in paging rates if the improvement only holds for the particular set of input data used when it was being traced. Fortunately, the relative number of intersector references of many commonly used programs is rather insensitive to input data. However, it is certainly still true, especially for particular application programs, that the uniformity of intersector references over a range of input data should be established before sector reordering on the basis of intersector behavior is attempted.

In addition, they reported that program restructuring to increase the locality in program reference patterns can have a much more profound effect on paging performance in a virtual memory system than page replacement algorithms.

1.4.3 Program Design Considerations

Another technique of increasing the degree of locality of programs, but certainly not the easiest to accomplish, consists of teaching the programmers how to design more local programs [B4,B5,G1,M1], making them aware of the important language translator considerations, providing them with unambiguous feedback about the paging performance of their programs and showing them how the system penalizes those programs which exhibit a poor degree of locality. The typical attitude of virtual memory system designers may be expressed by Denning [D2] when he states, "it is not known whether programmers can be properly educated, inculcated with the 'right' rules of thumb, so that they habitually produce programs with "good" locality." Unfortunately, the freedom of the programmers from the need to worry about physical memory space and its management in a virtual memory system is a major obstacle to their education in the art of locality.

Therefore, especially for frequently executed programs such as operating systems, assemblers, compilers, editors, production programs,

etc., we can see the appeal and the potential rewards of the program restructuring approach, that is, to design the program without excessively caring about its locality, and then to rearrange its relocatable code and data sectors in the virtual address space so as to make its reference pattern more local.

1.4.4 Related Performance Benefits

If we can reduce the number of page fetches required by program restructuring, we will get improved performance in several areas:

1. Reduced time spent paging.
2. Less supervisory overhead spent in main memory and paging management.
3. Better throughput on the average, because a program will interfere with others less.
4. Better paging operation when it is needed, because there will be less contention for the paging device.

1.5 Related Research and the Need for Further Research

The only comprehensive research in the area of automatic program restructuring was reported by Hatfield and Gerald [H1]. The essence of their work can be interpreted in the following context. A program consisting of m relocatable sectors occupying n logical pages of virtual memory was run with a typical set of input data and sufficient information was recorded during the run to produce a complete sector trace. A complete sector trace is the time sequence of all sector references (instruction and data references) during program execution.

A "nearness matrix" C for modeling intersector behavior was constructed from the sector trace. The nearness matrix is an mxm matrix, whose entry $C_{ij}(1 \leq i \leq m, 1 \leq j \leq m)$ is the number of times sector j followed sector i in the sector trace or equivalently the number of times sector i referenced sector j during the execution of the program. This matrix is equivalent to a directed graph G of m nodes where the arc from node i (corresponding to sector i) to node j has C_{ij} as its weight.

No computationally feasible procedure was found to produce and prove an optimum restructuring of G, based on C, into pages, i.e. one that minimized the summation of C_{ij} over all i and j such that sector i and sector j are grouped into different pages. Instead heuristic approaches were used to restructure G. One method used essentially the largest values of the eigenvectors of C as a basis for grouping sectors together. Another heuristic approach which gave slightly better results was a procedure which attempted to cluster sectors into pages, under the constraint that the size of each cluster be no greater than the page size, such that the square of the interconnecting weighted arc distances between pages were minimized.

The latter heuristic approach is quite similar to the procedure reported by Charney [C3] which partitions a network of interconnecting components into groups of components such that the total number of interconnecting wires between groups tends to be minimized.

As Hatfield and Gerald pointed out, a disadvantage of program restructuring formulated on the nearness matrix C is that the nearness matrix contains global information about sector interaction, whereas paging depends on local reference patterns. For example, consider two sector reference traces S_1 and S_2. Assume that sectors i and j are referenced exactly k times in both traces. Let $S_1 = \alpha_1 (ij)^k \alpha^k_2$ and $S_2 = \alpha_1 (ij\alpha_2)^k$ where α_1 and α_2 represent long sector reference strings. The value C_{ij} is k in both cases and C_{ji} is larger in S_1. Therefore, the probability that the clustering algorithm will group i and j together is greater for S_1 than S_2. However, the cost of not grouping them together is greater for S_2, since the number of page faults due to the references j immediately following those to i will be at most 1 for S_1 for all real memory sizes greater than one and can be k for S_2 for certain α_2's. In other words, even an optimum solution of the restructuring problem formulated on the nearness matrix may not give the minimum number of page faults.

Hatfield and Gerald realized that there are many cases where the nearness matrix alone does not have all the information needed for producing a good sector ordering and that the ordering obtained by the restructuring algorithm from the available information is based on heuristics. Accordingly they supplemented the automatic sector reordering phase with a hand finishing phase of additional sector reordering based on complex human interpretation of the program's use of virtual memory over the course of its execution as displayed via an

interactive graphics package. Even though the reordering phase based on human decisions provided additional improvements in paging performance, it can be quite time consuming, and the results are somewhat dependent on the imagination and insight possesed by the programmer making the decisions. Furthermore, the absence of any knowledge about the maximum possible improvement makes it difficult to determine a suitable stopping point based on some cost-performance criteria.

In order to determine if a new ordering is actually better or worse than an old ordering, they simulated the paging performance of each ordering over a range of primary memory sizes and page replacement policies. Evaluation of sector orderings by simulation can be an expensive process if many sector orderings are compared.

Based on the current state of research into the problem of program restructuring as discussed above, we can identify several areas of potentially rewarding research. We will assume that a program is modeled by a set of relocatable sectors of specified size and a sector trace describing the sector reference behavior.

1.5.1 Intersector Reference Models

We need a model of intersector reference behavior C, defined over the sector trace, that incorporates more of the local reference behavior of the program upon which paging actually depends than that captured by the nearness matrix. For example, the probability that a reference from sector i to sector j will cause a page fault is related to such local information as the time elapsed since the last reference to sector j and the number of distinct sectors referenced since the last reference to sector j in the sector trace. If the time is short since sector j was last referred to and little virtual memory space was used during that time, it is probable that sector j is still in primary memory and a new reference will not cause a page fetch. If the time and space traversed between references to j are large, it is probable that a page fetch will occur unless j is grouped into the same page as the referencing sector or some recently referenced sector. We propose to formulate and investigate two approaches which seem to have potential for identifying and quantifying local sector reference behavior which can be used to weight C_{ij} entries. These approaches are based on sector working sets and sector stack distances defined over the sector trace.

1.5.2 Reordering Procedures

Another area concerns finding better procedures for restructuring or grouping the m relocatable sectors of a program into n logical pages such that the reordered program achieves or tends to achieve the minimum external cost formulated on an intersector reference model C. A strictly exhaustive procedure for finding the minimum cost grouping is often out of the question. To see this, consider the simple problem of dividing m sectors into pages containing g sectors each. The total number of groupings is as follows:

$$\text{Groupings} = \frac{m!}{(g!)^{m/g}\,(m/g)!}$$

For most values of m and g, this expression yields a very large number; for example, if m = 40 and g = 4, it is greater than 10^{25}. Formally, the problem could be solved as an integer linear programming problem, with a large number of constraint equations necessary to express the uniformity of the partition [J1]. However, since it seems likely that any direct approach to finding an optimal solution will require an inordinate amount of computation, the quest for better heuristic methods appears to be the best approach. The first and foremost consideration in developing heuristics for combinatorial problems of this type is finding

a procedure that is powerful and yet sufficiently fast to be practical. A process whose running time grows exponentially with the number of sectors is not likely to be practical.

1.5.3 Sector Ordering Evaluators

A computationally inexpensive evaluator of sector orderings is needed so that a new ordering can be estimated as better or worse than an old ordering without simulating paging performance for a primary memory size and page replacement algorithm.

One theoretical approach recently reported by Sekino [S4] may be applied, given a sector ordering into pages and the probabilities of sector i referencing sector j for all i and j, to compute the page fetch probability. However, a major drawback of this approach is that after the probabilities of going from one system state to another are computed (where a system state is determined by the r pages of an n page program in primary memory, the page being referenced, and the state of the replacement algorithm), then, even in its simplest formulation, the solution of $r*\binom{n}{r}$ simultaneous equations are required (a solution computationally infeasible for values of n and r usually encountered in real programs).

Another approach relies on the ability to construct a matrix model describing the intersector reference behavior from the sector trace, given additional knowledge about the size of available primary memory and the paging policies, such that the cost of a sector ordering (i.e. the cost of the interpage arcs cut) produced by a reordering algorithm, is proportional to the number of page fetches expected for

that ordering. How successful is this approach or any other computationally inexpensive approach is an open research question, but the existence of this problem and the potential expense of any solution points out, in part, the immense value of the next research topic.

1.5.4 Performance Bounds

The tremendously large number of sector orderings, and the difficulty and expense involved both in choosing a relatively good ordering and in evaluating a new ordering as better or worse than an old ordering illustrate the vital need to have theoretical bounds on the optimum improvement in the paging performance of virtual memory systems through program restructuring.

If bounds on the minimum number of page fetches which could occur during execution of a program for any reordering of relocatable sectors into logical pages were known, they could be used: to determine whether or not a given program should be considered for restructuring based on its current paging performance; to evaluate the results of a restructuring procedure, whether automatic, manual or both, for a given program; and to recognize when a good program structure is found.

Automatic restructuring procedures based on heuristics appear to be the only computationally feasible approach. It is unlikely that any

one procedure will provide near optimum solutions for all programs. One attractive methodology for program restructuring when bounds on the optimum performance are known is to have a set of automatic restructuring procedures available which can be successively applied to a particular program until a reasonably good solution is obtained. In the case when no reasonably good solution is found automatically, a decision to consider manual restructuring and its extent can be made based on the potential for additional improvement versus its expected cost.

The theoretical work reported in the literature to date in developing bounds on the paging performance in virtual memory systems that can result from program restructuring is nil. We will present a formal approach to this problem and some preliminary results in the next two sections of this report.

It is our objective to develop upper and lower bounds on the number of page fetches which can occur over all reorderings of sectors into logical pages of a program, for any program modeled by: a set of relocatable sectors of specified size, a sector trace describing the intersector behavior, any two-level virtual memory system modeled by its page size, primary memory size available to the program, and page replacement and fetch policies.

1.6 Summary of Goals

The goals of this thesis are as follows:

1. Formalize and analyze the effect of the structural ordering of a program's relocatable sectors upon its paging performance in virtual memory systems.

2. Develop theoretical bounds on the optimum improvement in the paging performance of programs in virtual memory systems which can result from restructuring the relocatable sectors of programs.

3. Develop theoretical bounds on how "bad" the paging performance of programs can get if the "worst" ordering of relocatable sectors is chosen.

4. Formalize new models of program reference behavior, such as intersector reference models based on sector working sets and sector stack distances, and analyze their effect on reordering procedures for improving the paging performance

of programs.

5. Design and develop practical algorithms for restructuring programs to improve their paging performance in virtual memory systems.

6. Perform measurements to compare the relationship between the improvements in paging performance produced by these practical algorithms and the optimum improvement specified by the theoretical bounds.

CHAPTER 2

FORMALIZATION OF VIRTUAL MEMORY SYSTEMS

2.1 Introduction

In this section a formalization of the fundamental characteristics of two-level virtual memory systems is presented and certain performance measures are derived. The purpose of this chapter is to develop the terminology and the framework necessary to view this research in its proper perspective.

2.2 Major Parameters of a Two-Level Virtual Memory System

Figure 1 and Table 1 present the major parameters of a two-level virtual memory system. These parameters can be grouped into three categories: (1) Configuration, (2) Automatic Management Algorithms, and (3) Program Behavior.

2.2.1 Configuration

Virtual memory is assumed in this thesis to be implemented by paging on a two-level hierarchical physical memory system consisting of primary memory, Mp, and secondary memory, Ss. (Note that we have chosen the notation Ss for secondary memory, i.e., secondary storage, because the notation Ms would lead to notational conflicts later in this report). Each storage device is partitioned into physical blocks called pages. A page is the basic unit of information transferred between Mp and Ss. The page size (usually 4,096 or 2,048 bytes) is denoted by N. Each memory device is further characterized by its random access time T_i, transfer rate B_i, cost/byte C_i, and capacity in pages $|M_i|$. We assume that $Tp < Ts$, $Bp > Bs$, $Cp > Cs$ and $|Mp| < |Ss|$.

Configuration

1. M_p is the primary store
2. S_s is the secondary store
3. $|M_i|$ is the size in pages of the i-th store
4. B_i is the transfer rate of the i-th store
5. C_i is the cost/unit of the i-th store
6. T_i is the average access time of the i-th store
7. N_i is the number of bytes in a page (page size)

Memory Management Algorithm

1. F is the fetch algorithm
2. R is the replacement algorithm

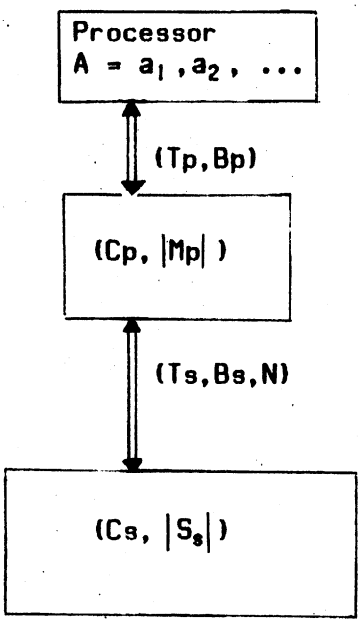
Program Behavior

1. A is the logical address trace

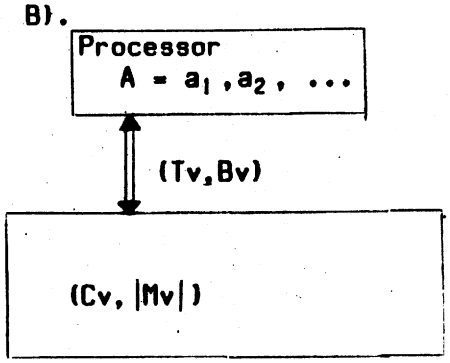
Table 1

Major Parameters of Two-Level
Hierarchical Virtual Memory Systems

A).



B).



C).

	IBM/360-67	IBM/370-165
Mp	Core	Cache
|Mp|	192 pages	16K bytes
Cp	$1.53/byte	8.80/byte
Tp	375 ns	160 ns
Bp	21Mb/s	100Mb/s
Ms	Disk	Main Store
$\|S_s\|$	2048 pages	512K bytes
Cs	$0.04/byte	$0.50/byte
Ts	8.6 ms	1.44μs
Bs	1.2Mb/s	16Mb/s
N	4096 bytes	32 bytes
Tv	805 ns	230 ns
Cv	$0.18/byte	$0.77/byte
|Mv|	2048 pages	512K bytes

Figure 1.

A). Two Level Storage Hierarchy System. B). Virtual or Composite Memory System. C). Representatative Parameters for Several Virtual Memory Systems.

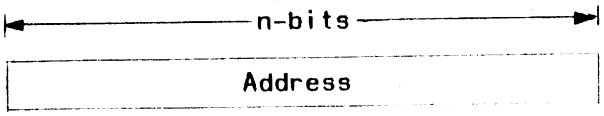
2.2.2 Program Behavior

The processor, under program control, generates a sequential sequence of references to the storage system. The processor references are in the form of logical address references or virtual memory references which serve to uniquely identify each unit of stored information independent of its location in Mp or Ss. The time sequence of logical address references is called an address trace, A and is defined as:

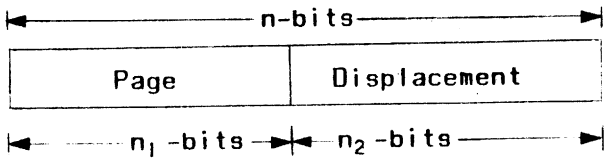
$A = a^1, a^2, \ldots, a^L$.

Each logical address, a^i, may be separated into a logical page reference and an offset within that logical page. This separation process is pictorially illustrated in Figure 2 where the set of 2**n possible addresses are partitioned into $2^{**}n_1$ pages of $2^{**}n_2 = N$ logical addresses each. The time sequence of logical page references is called a page trace, P and is defined as:

$P = p^1, p^2, \ldots, p^L$ where p^i = integer (a^i/N).



a)Logical Address



$(n = n_1 + n_2)$

b) Logical Address Partitioned into

Page Address and Displacement

Figure 2

Logical Address Structure

Information movement between Mp and Ss is accomplished by transferring pages between Mp and Ss. We can analyze interlevel movement for address trace A by considering the corresponding page trace P.

One method of constructing a representation or model of a complex activity such as program behavior is to first analyze a particular characterization and then gradually introduce additional detail. In the case of program behavior, it is convenient to begin by considering only the address trace and the corresponding page trace. Later, we will consider the effect of the program's structure on its behavior.

2.2.3 Automatic Management Algorithm

Since a processor can service only that portion of a program which resides within primary memory, which is relatively small in size, the operating system must exercise a special algorithm, called a paging algorithm, to keep the "most active" pages of a program in primary memory. This is accomplished by transferring pages of the program back and forth between primary and secondary memories. The goal of a paging algorithm is to maximize the number of times logical information is in the primary memory when being referenced.

The paging algorithm must consist of two basic policies. The Fetch policy, F, decides when and which information should be moved up from Ss to Mp. The Replacement policy, R, decides when and which pages should be transferred down from Mp to Ss.

Definitions

1. $Q = \{a,b,\ldots\}$ is a finite set of logical pages
2. $P = p^1, p^2, \ldots p^L$ is a page trace with $p^t \in Q$.
3. $M_p^t \subseteq Q$ is the contents of Mp at time t.
4. $F = f^1, f^2, \ldots f^L$ is a finite time sequence of L sets, $f^t \subseteq Q$, $1 \leq t \leq L$.
5. $R = r^1, r^2, \ldots, r^L = (\phi)$ is a finite time sequence of L sets, $r^t \subseteq Q$, $1 \leq t \leq L$.
6. $M_p^t = (M_p^{t-1} - r^t) \cup f^t$, $1 \leq t \leq L$.
7. F and R are valid if $f^t \cap M_p^{t-1} = \phi$, $r^t \subseteq M_p^{t-1}$ and $p^t \in M_p^t$, $1 \leq t \leq L$.

The F and R policies are defined to denote a particular realization of a paging algorithm for a given trace P. For a page trace and initial primary memory state M_p^0, a F-policy and a R-policy together determine the time sequence of primary memory states that will occur as the virtual memory system processes the trace. We will consider only valid F and R policies. That is, none of the pages fetched at time t, f^t, may be in primary memory at time t-1; the set

of pages removed at time t, r^t, must be in primary memory at t-1; and the page reference at time t, p^t, must be in primary memory at time t.

2.3 The Virtual Storage Model

A two-level hierarchical virtual storage system, V, is composed of all the parameters described above:

V = f(<configuration>,<program behavior>,<algorithms>)

V = f(<|Mp|,Tp,Cp,Bp,|Ss|,Ts,Cs,Bs,N>,<A>,<F, R>)

The rationale for two-level hierarchical virtual memory systems as shown in Figure 1 is to couple expensive low capacity fast memories, Mp, with inexpensive large capacity slower memories, Ss, such that the composite or virtual memory system approaches the speed of the expensive memory and the capacity and cost/unit of storage of the inexpensive memory.

2.4 Performance Measures

The rationale for a virtual memory system, V, immediately suggests three measures of its effective performance. These three measures are its effective capacity |Mv|, effective cost/unit, Cv, and effective access time, Tv.

2.4.1 Effective Capacity

The effective capacity |Mv| = |Ss| is achieved through the paging algorithm of the virtual memory system and the constraint that all logical pages initially reside in Ss.

2.4.2 Effective Cost

The effective cost Cv is defined as follows:

$$Cv = \frac{Cp|Mp|+Cs|Ss|}{|Mp|+|Ss|}$$

The effective cost Cv is seen to approach the cost Cs under the usual condition that the size of secondary memory is much larger than the size of primary memory.

2.4.3 Effective Access Time

For simplicity in developing techniques for analyzing and providing insight into the much more difficult problem of the effective

access time, Tv, we will first consider a demand fetch policy, F_d . Later, our considerations will focus on other fetch policies.

Assume that, at time t, the processor generates a logical address reference a^t, which refers to page p. At that point in time, the page p may reside in Mp or Ss. Under a demand fetch policy Fd, if p is in Mp, the reference proceeds and no page movement occurs. Otherwise, if p is in Ss, a page fault or page fetch occurs and the page is automatically transferred to Mp and the reference proceeds. If Mp were already full, the removal policy, R, must be employed to remove some page in Mp to provide space for the new page request.

Formally, a demand page fetch policy Fd, for a virtual memory system V is defined as follows:

Recall that

1. $P = p^1, p^2, \ldots, p^L$ is the page trace determined from A and N.
2. $F_d = f_d^1, f_d^2, \ldots, f_d^L$ is a valid fetch policy.
3. $R = r^1, r^2, \ldots, r^L$ is a valid removal policy.
4. $M_p^t = (M_p^{t-1} \cup f_d^t) - r^t$.

Definition of F_d

1. If $p^t \in M_p^{t-1}$, then $f_d^t = r^t = \phi$.

2. If $p^t \notin M_p^{t-1}$ and $|M_p^{t-1}| < |Mp|$,
   then $f_d^t = \{p^t\}$ and $r^t = \phi$.
3. If $p^t \notin M_p^{t-1}$ and $|M_p^{t-1}| = |Mp|$,
   then $f_d^t = \{p^t\}$ and $r^t = \{a\}$
   where $a \in M_p^{t-1}$ and a is selected by
   the removal algorithm.

Under demand paging, the primary memory Mp simply fills as required by 1 and 2, while the first |Mp| pages are referenced. Subsequently, referenced pages are swapped between Mp and Ss as required by 1 and 3.

Let FFp, the number of page fetches from Ss during the processing of a page trace P, be defined as the page fetch function and its value given by:

$$FFp = \Sigma_{t=1}^{L} |f^t|.$$

By analogy to the page fetch function, the number of references satisfied by Mp is called the page success function, SFp, and it can be computed as

$$SFp = |P| - FFp.$$

The effective access time, Tv, of a virtual memory system V, is defined as follows:

$$Tv = \frac{FFp}{|P|} Ts + \left(1 - \left(\frac{FFp}{|P|}\right)\right) Tp$$

The value of the effective access time Tv, is seen to approach the fast access time Tp, of primary memory as the value of the fetch frequency function, FFp/|P|, is reduced toward zero or equivalently, for a given page trace P, as the value of the page fetch function FFp approaches zero. Therefore, we see that the value of FFp is a crucial measure of the performance of a program in a virtual memory system. In general, we wish to minimize the page fetch function in order to minimize the effective access time Tv.

2.4.4 Page Trace Simulation

One method to determine the value of the page fetch function FFp, for a given virtual memory system V is to compute the resultant page trace P, from the address trace A and the page size N, then simulate the paging algorithms, F and R, and record the contents of Mp at each step of the page trace. Table 2 illustrates this step-by-step simulation, assuming demand paging and LRU (Least Recently Used) removal. The contents of Mp are shown ordered to reflect the LRU ordering: the top page is the page most recently fetched into Mp; the bottom page is the page least recently used by the program and is the

Virtual memory system V = f (<|Mp|, Tp,Cp,Bp,|Ss|,Ts,Cs,Bs,N>, <A>, <F,R>) with parameters

$A = a^1, a^2, \ldots, a^{12}$

P = a,b,a,b,c,c,b,a,a,b,b,a, where p^i = integer (a^i/N).

However, we have used lower case letters to represent logical page addresses instead of page numbers because it simplifies the presentation.

|P| = 12

Q = {a b c} and $|Q| = 3 \leq |Ss|$

|Mp| = 2

F = demand fetch, F_d

R = LRU replacement, R_{LRU}

Simulation:

Time	1	2	3	4	5	6	7	8	9	10	11	12
Page Trace,P	a	b	a	b	c	c	b	a	a	b	b	a
Fetch, F	a	b	0	0	c	0	0	a	0	0	0	0
Remove R_{LRU}	0	0	0	0	a	0	0	c	0	0	0	0
M_p^t contents	a	b	a	b	c	c	b	a	a	b	b	a
after time t		a	b	a	b	b	c	b	b	a	a	b

RESULTS:

$FFp = \Sigma_{j=1}^{12} |f_d^t| = 4$

$\frac{FFp}{|P|} = 4/12$

$Tv = \frac{TS}{3} + \frac{2Tp}{3}$

Table 2

Example of Page Trace Simulation to Determine FFp

page selected for removal when necessary.

2.5 Page Fetch Function Performance Model

From the above discussion, we observe that several parameters of a virtual memory system V=f(<|Mp|,Tp,Cp,Bp,|Ss|,Ts,Cs,Bs,N>, <A>, <F,R>) influence the value of the page fetch function, FFp. These parameters are the page size N, the program's storage reference pattern A, and the removal policy R, the fetch policy F and the size of primary memory |Mp|. Therefore, we define

$$FFp = FFp(|Mp|,N,A,F,R).$$

The significance of all these parameters on the page fetch function measure will be considered and investigated. Special emphasis will be focused on analyzing and understanding the relationship between the program's structure and the logical address trace.

We will not elaborate in great detail, but it should be pointed out that, for hierarchically-structured virtual memory systems of more than two levels, say K levels, and demand paging (those studied by Madnick [M3]), we can derive the effective page trace and thus the page fetch function for paging to the i-th level from level i-1 (level 1 is primary memory). To illustrate this, note that the resultant fetch

policy at level i-1, $F_{i-1} = f^1_{i-1}, f^2_{i-1}, \ldots f^L_{i-1}$, is essentially the page trace P_i for level i. There is an easy compression of F_{i-1} to omit the values of $f^t_{i-1} = \phi$ and a minor relabeling required to adjust for the difference in page size used by M_i and M_{i-1} of $P^t_i = f^t_{i-1}\ (N_{i-1} - 1/N_i)$. This procedure is applicable for all levels $1 \leq i \leq k$, and the goal of a k-level memory system is to minimize $\Sigma^{k-1}_{i=1} FFp_i * Tp_{i+1}$.

2.5.1 Replacement Algorithm Considerations

Even though we will be primarily concerned with the effect of a program's structure on the value of the page fetch function, FFp, we need to consider some important effects of the page removal algorithm on FFp. Many removal algorithms have been proposed and studied in the past, such as First-In-First-Out (FIFO), Least Recently Used (LRU), and Belady's [B1] Optimum algorithm (O). We will define these removal algorithms under demand fetch to illustrate how particular algorithms may be specified in our general model of removal policies, and to establish exactly what these algorithms mean, since they will be referred to frequently in the remainder of the thesis. Furthermore, we have chosen to discuss this particular subset of removal algorithms because they will enable us to present several important and well known properties of removal algorithms which will eventually be needed in our research. Let:

1. $P = p^1, p^2, \ldots, p^L$ be a page trace computed from A and N.
2. $|Mp|$ = number of page frames in primary memory, Mp.
3. Mp^t = the set of pages in Mp at time t.
4. $Fd = f_d^1, f_d^2, \ldots, f_d^L$ be a demand fetch policy as previously defined. Recall that the definition of Fd specifies all the mechanics of paging except the page to be selected for replacement.

The LRU removal policy, R_{LRU}, is defined for demand fetch, Fd, as $R_{LRU} = r_{LRU}^1, r_{LRU}^2, \ldots, r_{LRU}^L$ where
$r_{LRU}^t = \phi$ if $f_d^t = \phi$ or $|Mp^{t-1}| < |Mp|$; otherwise,
r_{LRU}^t = a, where a is the page in Mp which was least recently referenced.

The optimum removal policy, Ro, is defined for demand fetch, Fd, as $Ro = r_0^1, r_0^2, \ldots, r_0^L$ where $r_0^t = \phi$ if $f_d^t = \phi$ or $|Mp^{t-1}| < |Mp|$; otherwise, r_0^t = a, where a is the page in Mp^{t-1} with the longest future time to next reference in the page trace, P, from p^t. If $a \in Mp^{t-1}$ is never referenced again, then its time of next reference is assumed to be ∞. If a page must be removed at time t, and several pages have the same longest future time to next reference (i.e., all equal to ∞) then remove any one of the pages.

Under demand fetch, the First-In-First-Out replacement policy, R_{FIFO} is defined as

$R_{FIFO} = r^1_{FIFO}, r^2_{FIFO}, \ldots, r^L_{FIFO}$ where

$r^t_{FIFO} = \phi$ if $f^t_d = \phi$ or $|Mp^{t-1}| < |Mp|$; otherwise,

$r^t_{FIFO} = a$ where a is the page in Mp^{t-1} which has been in Mp^{t-1} longer than any other page in Mp^{t-1}.

We now present several well known properties of these replacement algorithms.

Lemma 1.

For a given page trace, P, primary memory size of |Mp| page frames, and demand fetch policy, Fd, then the number of page fetches using any valid removal policy Ra is greater than or equal to the number of page fetches using the optimum replacement policy, Ro. The proof of this Lemma can be found using various techniques in [A1,M1] and is not repeated here.

Inclusion Property:

Under demand fetch, Fd, any replacement policy is said to satisfy the inclusion property if for all page traces, P,

a. $Mp^t(1) \subset Mp^t(2) \subset \ldots \subset Mp^t(n)$, where $Mp^t(j)$ is the contents of primary memory Mp at time t if the size of Mp is j page frames (i.e., $|Mp| = j$), $1 \leq j \leq n$.

b. At any time t after Mp has become filled, there is a strict

replacement ordering referred to as the "replacement stack," RS, RS = rs(1),rs(2),...,rs(n), where rs(j) = $Mp^t(j)-Mp^t(j-1)$ for j = 1,2,...,n, and rs(n) is the page to be removed next.

The general class of demand-fetch replacement algorithms which satisfy the inclusion property are referred to as "stack algorithms" in the literature. The class of stack algorithms, as noted by Denning [D1], "contains all the reasonable algorithms."

Lemma 2.

The number of page fetches required by any stack algorithm for any page trace is a monotonic function of primary memory size, |Mp|, in page frames. To see this, note that if there is a fetch at time t for a primary memory of a given size, there must also be one at time t for every primary memory of smaller size. The proof of this Lemma can be found in [D1,M1].

Lemma 3.

Demand fetch with LRU removal and demand fetch with Optimum replacement are stack algorithms. The proof of this Lemma can be found in [M1].

We will refer to the above well-known properties several times in the rest of this thesis. At this point in time, we can immediately conclude that, for any |Mp| and A,

a. $FFp(|Mp|,N,A,Fd,Ro) \leq FFp(|Mp|,N,A,Fd,Ra)$ from Lemma 1, when Fd, Ro are demand fetch and optimum removal policies and Fd, Ra are demand fetch and any removal policies.

b. $FFp(|Mp|,N,A,Fd,R_{LRU}) \leq FFp(|Mp'|,N,A,Fd,R_{LRU})$ and $FFp(|Mp|,N,A,Fd,Ro) \leq FFp(|Mp'|,N,A,Fd,Ro)$ from Lemmas 2 and 3 where $|Mp| \geq |Mp'|$.

Due to its simplicity, the FIFO replacement algorithm was used in many of the early paging systems. In recent times it has been discovered that FIFO has certain disturbing pecularities, such as the possibility that the number of page fetches will double for a memory size increase of one page frame [A1,M1]. Hence, FIFO is not a stack algorithm, and we cannot claim that, for any A and $|Mp|$, $FFp(|Mp|,N,A,Fd,R_{FIFO}) \leq FFp(|Mp'|,N,A,Fd,R_{FIFO})$, where $|Mp|>|Mp'|$.
Thus, we observe that the inclusion property of stack algorithms is an important property.

Various forms of the LRU replacement algorithm frequently occur in contemporary virtual memory systems. Empirically, LRU replacement has been found to closely approximate the paging performance obtained by the optimum algorithm for many actual programs. The optimum policy is not physically realizable since it requires future knowledge about reference behavior, but it can be used as a theoretical basis for performance comparison with practical algorithms. However, the value of the page

fetch function,

$FFp(|Mp|,N,A,Fd,Ro) = \Sigma_{i=1}^{L} |f_d^i|$ is physically realizable [B6] since it does not require future knowledge.

For any page trace $P = p^1, p^2, \ldots, p^L$ and primary memory size $|Mp|$, Belady has given a one-pass procedure which will compute the value that $|f_d^t|$ would take on under optimum removal for any $1 \leq t \leq L$ without any knowledge of the page trace after t (i.e., $p^{t+1}, p^{t+2}, \ldots, p^L$). In particular, this procedure determines whether $|f_d^t| = 1$ or $|f_d^t| = \phi$, but it does not specify of what page f_d^t consists.

2.5.2 Program Structure Considerations

In this section, we will extend the page fetch function performance model to account for the program's structure.

The programs we consider are defined to consist of a set of m relocatable sectors of specified sizes. The structure of a program is specified by a particular load ordering sequence of its sectors in its virtual address space. This ordering is called a <u>sector ordering</u> SO, and is defined as

$SO = S_1, S_2, \ldots, S_m$

where S_1 denotes the first, S_2 the second, and S_m the last sector loaded in the virtual address space. Thus a program can have m! distinct structures, one for each possible sector ordering, SO. However, once a sector ordering is chosen, it does not change during the execution of the program. Let $|S_j|$ be the size of the jth sector and let $L|S_j|$ be the load address of S_j in the virtual address space of the program. If the sectors are loaded contiguously in virtual memory, then $L|S_j| = \Sigma_{i=1}^{j-1} |S_i|$. In any event, we assume that the structure of a program is completely specified by its sector ordering SO, which is further defined to include the size and load addresses of all its sectors. Therefore the sector ordering SO of a program specifies the load sequence, $S_1, S_2, \ldots, S_m$, and the values of $|S_j|$ and $L|S_j|$ for all $1 \leq j \leq m$.

We have previously modeled the program behavior by its logical address trace $A = a^1, a^2, \ldots, a^L$ and have shown that the address trace A and the page size N are sufficient to determine the page trace $P = p^1, p^2, \ldots, p^L$. However, the address trace and hence the page trace depends on the particular sector ordering chosen for the program. For example, if a^t, the logical address referenced at time t, is within sector j, then the value of a^t depends on where S_j is in the sector ordering SO.

In order to study the effect of a program's structure on its paging performance, we will model a program's behavior by its sector trace. The sector trace ST of a program is defined to be the time sequence of sector references and is given by

$$ST = S^1, S^2, \ldots, S^L$$

where S^t denotes the sector referenced at time t.

Given the logical address trace A corresponding to a specific sector ordering SO, the sector trace ST can be easily computed from the load addresses of the sectors. Then this sector trace can be used to compute the page trace resulting from any program restructuring specified by a new sector ordering if the sectors do not cross page boundaries.

In particular, given a program modeled by its sector trace ST and its sector ordering SO, the page referenced at time t, p^t, is given by

$$p^t = \text{integer } (L|S^t| / N),$$

where S^t is the sector referenced at time t in the sector trace ST, $L|S^t|$ is the load address of sector S^t given by the sector ordering SO, and N is the page size. We are assuming at this point that individual sectors do not cross page boundaries.

As long as this is true, we can define the restructuring of a program as a partition of the relocatable sectors into logical pages. In particular, let,

1. $Q = \{S_1, S_2, \ldots, Sm\}$ be the set of relocatable sectors making up a program.
2. n = the number of logical pages of size N of the restructured program.

Then an n-way restructuring of P is defined as a partition $\Pi = (\Pi_1, \Pi_2, \ldots, \Pi n)$ where Π has the following properties:

a. $\cup_{i=1}^{n} \Pi i = Q$, $\Pi i \cap \Pi j = \phi$ for all $i \neq j$.

b. $\sum_{S_k \in \Pi i} |Sk| \leq N$ for all Πi, $1 \leq i \leq n$.

Thus, we see that a partition, Π, specifies the set of relocatable sectors grouped into each logical page. We will assume that the set of sectors in π_1 are loaded one after another into logical page 1, then the set of sectors in π_2 are loaded one after another into logical page 2, etc., until all the sectors are loaded in the logical address span of the program. If $\sum_{S_k \in \Pi i} |Sk| < N$, then there will be a hole or a non-referenced area in the top of page i.

Therefore, given any partition, Π, of the relocatable sectors into logical pages and any sector trace, we can compute the page trace

immediately. For example, let S^t be the sector referenced at time t in the sector trace and let $S^t \epsilon \Pi$ j, then the page, p^t, referenced at time t is j.

From the above discussion, we observe that -- given any two-level virtual memory system V, with page size N, with primary memory size of |Mp| page frames, with any valid page fetch algorithm Fa, and with any valid page removal algorithm Ra-- we have the value of the page fetch function FFp. This FFp is for a program whose structure is modeled by any partition, Πa, and whose reference behavior is modeled by a sector trace ST. FFp can be uniquely defined in terms of the following parameters:

FFp=FFp(|Mp|,N, Πa,ST,Fa,Ra).

For a particular virtual memory system, V, the values of |Mp|, N, Fa,Ra are fixed, and a given reference behavior fixes the value of ST. Under these conditions, the value of FFp becomes a function of the different partitions of relocatable sectors into pages. However, as pointed out in Chapter 1, the number of different partitions becomes astronomical for many typical programs. For example, phase 1 of the AED compiler has 10^{75} different partitions. For such programs it is impossible from any practical point-of-view to determine the best program structure (the Π that minimizes FFp) for a given reference behavior by trying out all partitions.

From our discussion in Chapter 1, we know that for a given sector trace, a partition Π which groups sectors into pages such that the number of intersector references between pages of the partition is minimized may not minimize FFp. In fact, we presented a quite plausible sector trace where such a Π would indeed be a very bad partition. One major goal of this thesis is to find some way of computing the minimum value of FFp over all partitions.

If upper and lower bounds on the value of FFp over all partitions can be found, then a particular program structure could be evaluated as good or bad. Furthermore, those bounds would provide a means of evaluating the ability of practical clustering procedures to produce a good program structure.

The practical drawback of the model developed for the page fetch function, FFp, is that sectors are not allowed to cross page boundaries. Even though this may not be a serious drawback, we will eventually try to extend the model of FFp to take into account the case when sectors may cross page boundaries.

2.6 Sector Fetch Function Performance Model

We will now define a measure on the information transfer between the two levels of a virtual memory system which is independent of the sector ordering. In the next section, we will employ this measure to find theoretical upper and lower bounds on the value of the page fetch function over all sector partitions.

If we assume that the basic unit of information transfer between the two levels of a virtual memory system V' is a sector instead of a page, we can formulate a measure on the interlevel movement of information during the execution of a program which is independent of its sector ordering.

Let FFs, the number of sector fetches which occur in a virtual memory system V' during the processing of a sector trace ST, be defined as the sector fetch function. The processing of a sector trace in V' is called sectoring and can be interpreted similarly to the processing of a page trace or paging in V as previously discussed.

Since the virtual memory system, V', for sectoring is to be modified slightly from the virtual memory system, V, used in our discussion of paging, we need to define the notion of sectoring more precisely.

The parameters of a demand sectored virtual memory system, V', are defined as follows:

1. |Ms| is called the size of the primary memory, Ms.
   |Ms| is the number of sector frames in the primary memory. The size of these sector frames, say in bytes, need not be the same. Instead we assume that the size of a sector frame in bytes is exactly equal to the size in bytes of the sector it contains. Thus, the size in bytes of any sector frame and of Ms can vary with time if the sector sizes are different, but the important fact is that the number of sector frames in Ms is fixed and equal to |Ms|. In contrast, we should point out that the size, |Mp|, of the primary memory, Mp, for a paged virtual memory system, V, was defined to be the number of page frames of fixed size N in the primary memory Mp.

2. $ST = S^1, S^2, \ldots, S^L$ is a sector trace of a program.
3. $Fd = f_d^1, f_d^2, \ldots, f_d^L$ is the demand sector fetch policy of V'.
4. $R = r^1, r^2, \ldots, r^L$ is the sector removal policy of V'.

Let Ms^t denote the set of sectors in primary memory at time t and $|Ms^t|$ denote the cardinality of this set.

Now, demand sectoring and the value of the sector fetch function, FFs, is defined as follows:

a. If $S^t \in Ms^{t-1}$, then $f_d^t = r^t = \phi$ and $Ms^t = Ms^{t-1}$.

b. If $S^t \notin Ms^{t-1}$ and $|Ms^{t-1}| < |Ms|$, then $f_d^t = \{S^t\}$, $r^t = \phi$ and $Ms^t = Ms^{t-1} + \{S^t\}$

c. If $S^t \notin Ms^{t-1}$ and $|Ms^{t-1}| = |Ms|$, then $f_d^t = \{S^t\}$, $r^t = \{S\}$ and $Ms^t = Ms^{t-1} + \{S^t\} - \{S\}$ where $S \in Ms^{t-1}$, and S is selected in accordance with the removal algorithm.

d. $FFs = \Sigma_{i=1}^{L} |f_d^i|$.

The value of the sector fetch function FFs, for any sector trace, ST, can be uniquely determined by simulating algorithm Fd and R for a primary memory of size |Ms| at each step of the sector trace. Therefore, we define

FFs = FFs(|Ms|,ST,Fd,R).

It should be clear that the value of FFs will be the same for any sector ordering, since the sector trace is independent of the sector ordering. It should also be clear, from the definition of |Ms| and parts a. and b. of the definition of demand sectoring, that the value of FFs for a given sector trace is independent of the sector sizes. We do

not need to be concerned with the implementation problems associated with the variable sector frame sizes of V', since we will be using the sector fetch function only as an analytic tool, and since we can determine the value of FFs through simulation without even knowing the sector sizes.

In the next Chapter, the sector fetch function, FFs, will be utilized to provide upper and lower bounds for the page fetch function, FFp.

This empty page was substituted for a blank page in the original document.

CHAPTER 3

PAGING PERFORMANCE BOUNDS

3.1 Introduction

In this chapter, we will investigate the effect of a program's structure on its paging performance in a virtual memory system. We will begin by presenting theoretical upper and lower bounds on the value of the page fetch function, FFp(|Mp|,N, Πa,ST,F,R), over all partitions, Πa, of relocatable sectors into logical pages for fixed values of the other parameters.

Recall that the value of the page fetch function, FFp(|Mp|,N, Π,ST,F,R), is the number of page fetches a program would experience in a two-level virtual memory system, V, with primary memory size of |Mp| page frames of size N, using the page fetch and removal algorithms, F and R, respectively, for a given sector trace, ST, and program structure, Π. We would like to present a uniform method that would bound the value of the page fetch function, FFp, over all partitions, Πa, of relocatable sectors into logical pages for "any" fixed values of the remaining parameters. The merit of such a uniform bounding method would be two-fold. First, it would be applicable to any two-level

virtual memory system, V, that is, any values of |Mp|, N, F, and R. Second, it would be applicable for any program behavior characterized by a sector trace.

In contrast to a uniform approach, a second approach would be to bound the value of FFp over all partitions when certain or all of the remaining parameters are constrained. For example, we could assume that |Mp| = 1, F = demand fetch, R = FIFO replacement and ST = any fixed sector trace, and then derive bounds on FFp over all Π_a. Clearly, the disadvantage of the second approach is that it would have quite limited applications. However, one advantage of the second approach is that the additional knowledge gained by fixing certain parameters of the virtual memory system could permit the utilization of bounding methods which would result in tighter bounds. We will investigate both approaches in this chapter. We have the conviction that a uniform approach over all virtual memory system parameters and all sector traces is vital for general applicability. However, given a uniform bounding method, it would certainly be worthwhile to investigate the possibility of obtaining tighter bounds when feasible constraints on certain parameters of the virtual memory system are specified.

We begin by imposing constraints upon the structure of the program, that is, on the partitions, Π , of relocatable sectors into pages, and then gradually remove these constraints.

3.2 Lower Bounds

Let us constrain the structure of a program such that each logical page contains at most k sectors. In particular, let:

1. Program = $\{S_1, S_2, \ldots, S_m\}$ be a finite set of m relocatable sectors such that $|S_i| \leq N$ for $0 \leq i \leq m$; that is, the sector size in bytes is less than the page size, N, in bytes; otherwise, the sector size may vary.
2. $\Pi a = \{\Pi_1, \Pi_2, \ldots, \Pi_n\}$ be any partition of the m relocatable sectors into n logical pages where the number of sectors $|\Pi_j|$ in page j satisfies the constraint $1 \leq |\Pi_j| \leq k$.
3. Recall from our definition of Π that $\sum_{S_j \epsilon \Pi_i} |S_j| \leq N$ must always be true.

Thus, we are currently concerned with all the partitions, Πa, which restructure a program such that each logical page has k or fewer sectors. The sector sizes may vary, but the sum of the sector sizes grouped into a page must not exceed the page size. With this rather flexible constraint on the allowable partitions, we can find a lower bound for the value of the page fetch function, FFp, over all such partitions for a given sector trace and any virtual memory system. We present this lower bound in Theorem 1.

Theorem 1

Given any two-level virtual memory system V, with page size N, primary memory size |Mp|, and any valid page replacement algorithm Ra, any valid page fetch algorithm Fa, and any sector trace STa, then, for any partition Πa, of relocatable sectors into logical pages of the program where each page contains at most k sectors, the minimum number of page fetches given by the page fetch function model, FFp, has a lower bound given by:

$k*FFp(|Mp|,N,\Pi a,STa,Fa,Ra) \geq FFs(|Ms| = |Mp|*k, ST = STa, Fd, Ro)$

where the value of the sector fetch function, FFs, is the number of sector fetches which occur in a two-level virtual memory system V', with primary memory size |Ms| = |Mp| *k, the same sector trace STa, demand fetch Fd, and optimum replacement Ro.

Corollary 1a

The size of Mp in bytes is equal to the size of Ms in bytes if each page is completely filled with exactly k sectors of the same size.

Proof of Theorem 1

Notation and properties

Let $STa = x^1, x^2, \ldots, x^L$ where x^t is the sector referenced at time t. For virtual memory system V and FFp let:

1. $\Pi a = \{ \Pi_1, \Pi_2, \ldots, \Pi_n \}$ be any partition of sectors into the n logical pages of the program where each page contains at most k sectors. (This interpretation of a partition will be useful later in this thesis.)
2. $P = p^1, p^2, \ldots, p^L$ be the resultant page trace computed uniquely from ST and Πa, such that if $x^t \in \Pi j$, then $p^t = j$.
3. M_p^t be the set of pages in Mp at time t and $M_p^0 = \phi$.
4. $F_a = f_a^1, f_a^2, \ldots f_a^L$ be any fetch policy where $f_a^t \cap M_p^{t-1} = \phi$ and $|f_a^t|$ = the number of pages in f_a^t and $x^t \in [M_p^{t-1} \cup f_a^t]$.
5. $R_a = r_a^1, r_a^2, \ldots r_a^L$ be any removal policy where $r_a^t \subseteq M_p^{t-1}$ and $x^t \notin r_a^t$.
6. $M_p^t = (|M_p^{t-1} \cup f_a^t|) - r_a^t$

Given the above notation and properties, we will first prove:

Lemma 4.

For each Fa and Ra there exists a demand fetch and removal policy, Fd and Rd, for the FFs model such that

$k * FFp(|Mp|, N, \Pi a, STa, Fa, Ra) \geq FFs(|Ms| = |Mp| * k, ST=STa, Fd, Rd)$.

Proof:

For the FFs model, Fd and Rd will be constructed by forming a sequence of valid replacement and fetch policies

$(F_1, R_1), (F_2, R_2), \ldots, (Fh, Rh)$ where:

1. $F_1 = f_1^1, f_1^2, \ldots, f_1^L$ and $f_1^t = \cup f_a^t$ = the set of sectors making up the set of pages in f_a^t, for $1 \leq t \leq L$, where $|\cup f_a^t|$ = the number of sectors in the set.
2. Similarly $R_1 = r_1^1, r_1^2, \ldots, r_1^L$ and $r_1^t = \cup r_a^t$, for $1 \leq t \leq L$.
3. $F_h = F_d = f_d^1, f_d^2, \ldots, f_d^L$ and $R_h = R_d = r_d^1, r_d^2, \ldots, r_d^L$, for $1 \leq t \leq L$ where
   $f_d^t = r_d^t = \phi$ if $x^t \in M_d^{t-1}$; $f_d^t = x^t$ and
   $r_d^t = \phi$ if $x^t \notin M_d^{t-1}$ and $|M_d^{t-1}| < |Ms|$;
   $f_d^t = x^t$ and $r_d^t = $ be M_d^{t-1} if $x^t \notin M_d^{t-1}$ and $|M_d^{t-1}| = |Ms|$; and
   $M_d^t = (M_d^{t-1} \cup f_d^t) - r_d^t$ to satisfy demand sectoring.

For reasons of expediency, the proof of Lemma 4 will be divided into two parts, Lemmas 4a and 4b.

Lemma 4a:

If $|Ms| \geq ||Mp||$, then for $(F_1, R_1) = (Fa, Ra)$, there exists a valid sequence of sector replacement and fetch policies $(F_1, R_1), (F_2, R_2), \ldots, (Fh, Rh)$ such that $(Fh, Rh) = (Fd, Rd)$ and $\Sigma_{t=1}^{L} |f_1^t| \geq \Sigma_{t=1}^{L} |f_d^t|$; where $||Mp||$ denotes the maximum number of sectors that could ever be found in Mp. (Note that, in Lemma 4, $||Mp|| = |Mp| * k$,)

A proof similar to Lemma 4a has been givey by [M1] for pure paging systems. However, we need the following proof to make our extensions easier to understand.

Proof of Lemma 4a.

The procedure for constructing F_j and R_j from their immediate predecessors F_{j-1} and R_{j-1} in the FF_s model for $1 \leq j \leq h$ is:

STEP 1.

Choose the smallest t such that f_{j-1}^t and/or r_{j-1}^t do not satisfy demand sectoring.

STEP 2.

Let z' be the sector $\{x^t\}$ referenced at time t in the FFs model.

CASE 1.

Now suppose that f_{j-1}^t does not satisfy demand sectoring.

1a.

If $t < L$ and $z' \in f_{j-1}^t$, then set $f_j^t = \{z'\}$, and $f_j^{t+1} = f_{j-1}^{t+1} \cup (f_{j-1}^t - \{z'\})$. This construction insures that f_j^{t+1} contains the sectors already fetched by the FF_p model but not fetched by the FF_s model (i.e. deferred sector fetches).

1b.

If $t = L$ and $z' \in f^t_{j-1}$, then set $f^t_j = \{z'\}$.

1c.

If $t < L$ and $z' \notin f^t_{j-1}$, then set $f^t_j = \phi$, and $f^{t+1}_j = f^{t+1}_{j-1} \cup f^t_{j-1}$. Note that this allows the reference $x^t = z'$ to proceed because sector $z' \in M^t_{j-1}$. $z' \in M^t_{j-1}$, since $z' \in Mp^t$ and $|Mj| = |Ms|$ for all $1 \leq j \leq h$, and since $|Ms| \geq ||Mp||$. The last fact, $|Ms| \geq ||Mp||$, allows M_{j-1} to hold $||Mp||$ sectors; therefore we can always keep a sector in M_{j-1} until the corresponding page is removed from Mp as shown in CASE 2 below.

1d.

If $t = L$ and $z' \notin f^t_{j-1}$, then set $f^t_j = \phi$. The reference proceeds due to the same argument as given in 1c.

In all subcases of CASE 1 note that F_j is valid since $f^t_j \notin M^{t-1}_j$ for $1 \leq t \leq L$, that F_j satisfies demand sectoring at least up through time t, and that $\Sigma^t_{j-1} |f^t_j| \leq \Sigma^t_{j-1} |f^t_{j-1}|$.

CASE 2

Now suppose that r^t_{j-1} does not satisfy demand sectoring.

2a.

If $t < L$, and $f^t_j = \{z'\}$ and $|M^{t-1}_{j-1}| = |Ms|$, set $r^t_j = \{b'\}$ for some $b' \in r^t_{j-1}$ and $r^{t+1}_j = r^{t+1}_{j-1} \cup (r^t_{j-1} - b')$. Note that since $|M^{t-1}_{j-1}| = |Ms|$ and $f^t_{j-1} \neq \phi$, then $r^t_{j-1} \neq \phi$ and the

above operations are always defined. Also, note that r_j^{t+1} is constrained here and in all subcases to contain only the sectors already removed in pages by the FF_p model but not yet removed by the FF_s model; therefore, a sector will not be removed from FF_s until the corresponding page is removed from FF_p. This constraint is enforceable since the memory size of FFs at each step j, $|Mj| = |Ms|$, satisfies the relation $|Mj| \geq ||Mp||$ for $1 \leq j \leq h$.

2b.

If $t = L$ and $f_j^L = \{z'\}$ and $|M_{j-1}^{t-1}| = |Ms|$, then

$r_j^L = \{b'\} \subseteq r_{j-1}^L$.

2c.

If $t < L$, and $f_j^t = \phi$ or $|M_{j-1}^{t-1}| < |Ms|$, then set $r_j^t = \phi$

and $r_j^{t+1} = r_{j-1}^{t+1} \cup r_{j-1}^t$.

2d.

If $t = L$, and $f_j^L = \phi$ or $|M_{j-1}^{t-1}| < |Ms|$, then $r_j^L = \phi$.

In all subcases of CASE 2, note that R_j is valid since $r_j^t \subseteq M_j^{t-1}$ for $1 \leq t \leq L$, and that R_j satisfies demand sectoring at least up through time t.

A final comment: if it ever occurs that $z' \in r_{j-1}^t$ and $z' \in f_{j-1}^t$, then simply remove z' from both. This only reduces the value of $|f_j^t|$, and it takes care of the case when a page is fetched into and replaced from Mp without having all of its sectors referenced. The above procedure, after being applied at most h times, must terminate

with a valid replacement and fetch policy pair (R_d, F_d) such that:

$$\Sigma_{i=1}^{L} |f_i^t| \geq \Sigma_{i=1}^{L} |f_d^t|.$$

Hence, Lemma 4a is proved.

Choosing |Ms| = |Mp| *k satisfies Lemma 4a and we immediately get

$\Sigma_{i=1}^{L} |f_i^t| \geq \Sigma_{i=1}^{L} |f_d^t|$ = FFs(|Ms|=|Mp| *k,ST,Fd,Rd).

Lemma 4b.

$\Sigma_{i=1}^{L} |f_i^t| \leq$ k*FFp(|Mp|,N, Πa ,STa,Fa,Ra).

Proof:

$\Sigma_{i=1}^{L} |f_i^t| = \Sigma_{i=1}^{L} |Uf_a^t| = \Sigma_{i=1}^{L} |f_a^t| \, |Uf_a^t| \, / \, |f_a^t|$.

But $|Uf_a^t| \, / \, |f_a^t| \leq k$, since $|Uf_a^t|$ is the number of sectors in f_a^t and $|f_a^t|$ is the number of pages in f_a^t. Hence,

$\Sigma_{i=1}^{L} |f_i^t| \leq k * \Sigma_{i=1}^{L} |f_a^t|$ = k* FFp(|Mp|,N,Πa,STa,Fa,Ra).

Lemma 4b is proved.

From Lemmas 4a and 4b, we immediately get

k*FFp(|Mp|,N, Πa ,STa,Fa,Ra) $\geq$ FFs(|Ms| = |Mp| *k,STa,Fd,Rd),

and Lemma 4 is proved.

From Lemma 1 of Chapter 2 , we know that

FFs(|Ms| = |Mp| *k,ST,Fd,Rd) $\geq$ FFs(|Ms| = |Mp| *k,ST,Fd,Ro).

From Lemma 1 and Lemma 4 we immediately get

$k*FF_p(|Mp|,N,\Pi a,STa,Fa,Ra) \geq FF_s(|Ms| = |Mp|*k,STa,Fd,Ro)$

and Theorem 1 is proved.

Proof of Corollary 1a.

The size of Mp in bytes is |Mp|*N, and the size of Ms in bytes is (|Mp|*k) frames* (N/k) bytes/frame = |Mp|*N.

Now, a few comments about Theorem 1. For any given program behavior characterized by a sector trace, Theorem 1 provides a method of computing a lower bound on the inprovement in paging performance over all sector partitions into logical pages, when pages are constrained to have k or fewer sectors. The lower bound given by Theorem 1 is valid for any virtual memory system. Another beneficial property of Theorem 1 is that the lower bound is specified in terms of a stack algorithm. We know that Ro is a stack algorithm from Lemma 3. Furthermore, it is well known that, for all stack algorithms, the number of page fetches required to process a page trace can be computed for all primary memory sizes from one simulation run. For a general discussion of the procedure, the interested reader should see [M1], and for a particular discussion of a simulation procedure for the optimum replacement algorithm which requires only one pass through the page trace, reference is made to [B5]. We implemented the latter method for the sector fetch function, FFs, and from one simulation run through any sector trace we were able to plot FFs(|Ms| = |Mp| * k,ST,Fd,Ro)/k as a function of |Mp|.

Figure 3 conveys the general shape of this bound.

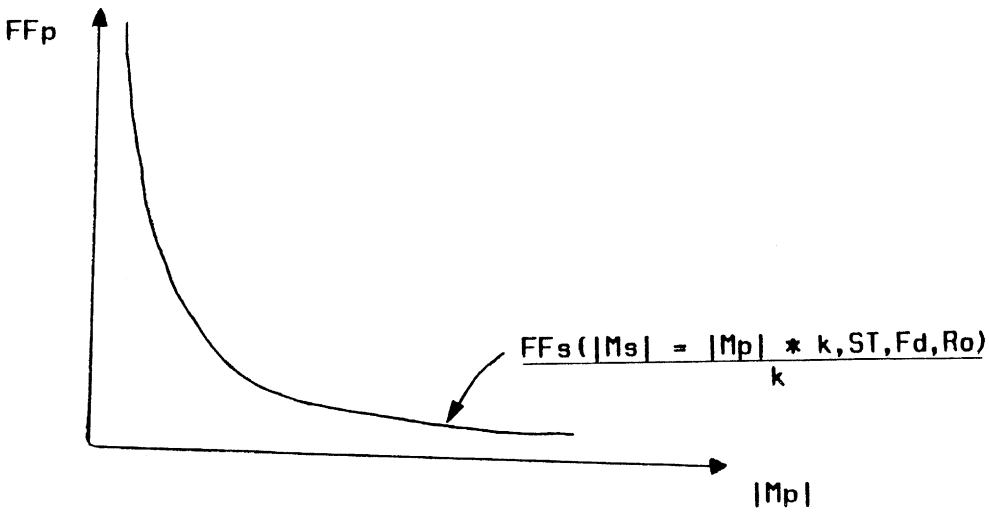


FIGURE 3.

Lower Bound on FFp Given by Theorem 1

The utility of such a curve as shown in Figure 3 is as follows. Theorem 1 states that the number of page fetches given by the page fetch function FFp(|Mp|,N, Πa,ST,Fa,Ra) for the same sector trace cannot be reduced below the curve shown in Figure 3 by any reordering of sectors into logical pages regardless of the paging algorithms employed.

Given that we have a procedure for lower bounding the effects of a program's structure on its paging performance in any virtual memory system, an interesting question is, just how tight is this bound for popular virtual memory systems? If Fa is constrained to be demand fetch and Ra is constrained to be LRU, FIFO or Optimum replacement, then we could prove, by example, that the lower bound on FFp given by Theorem 1 can be the greatest lower bound for certain sector traces and only a lower bound for others. We will show that it can be the greatest lower bound in a following example later in this thesis.

We will present and discuss empirical results in Chapter 6 which illustrate that the bound given by Theorem 1 is indeed rather tight for real programs running in a paged virtual memory system using demand fetch and LRU replacement. We will not discuss particular empirical results in this chapter because we want to relate the results to intersector reference models, to clustering procedures and to theoretical bounds at the same time. Intersector Reference models will be developed in Chapter 4 and clustering procedures in Chapter 5, and in Chapter 6 we show the results of applying these methods to restructure real programs such that

the resulting number of page fetches is quite close to the theoretical bound developed in this chapter for most memory sizes and popular paging algorithms.

Now consider restricting the fetch and replacement policies of FFp to be demand fetch and LRU replacement. Under this restriction, can we replace the optimal sector replacement policy, Ro, of the sector fetch function, FFs, by some less efficient policy such as LRU and hence produce a tighter lower bound on FFp over all partitions? This line of logic led to the following question: is it true that $k*FFp(|Mp|,N, \Pi a,STa,Fd,R_{LRU}) \geq FFs(|Ms| = |Mp| * k,STa,Fd,R_{LRU})$?

It seems intuitive that the above conjecture would be true even for the case where each logical page contained exactly k sectors. Here, the sectored memory could contain exactly the same number of sectors as the paged memory could contain. Futhermore, at most k sector fetches would be required to bring into Ms the same information brought into Mp by one page fault. One might expect that, for programs having a good structure, i.e., all pages contain sectors that are used together, each page fetch should produce k sector fetches. Hence, we have divided the value of FFs by k in the conjecture. In spite of its intuitive appeal, we can prove that the conjecture is not true for all program behavior. In order to validate this claim, we present the following Theorem.

Therem 2

For any two-level virtual memory system V, with page size N, primary memory size |Mp|, demand fetch Fd, and LRU replacement R_{LRU}, then there exists a sector trace ST, and a partition Π of relocatable sectors into logical pages where each page contains k sectors, such that $k*FFp(|Mp|,N,\Pi,ST,Fd,R_{LRU}) < FFs(|Ms| = |Mp| *k,ST,Fd,R_{LRU})$, where the value of the sector fetch function FFs is the number of sector fetches which occur in a two-level virtual memory V', with primary memory size |Ms| = |Mp| * k, using demand fetch Fd, LRU replacement R_{LRU}, and the same sector trace ST.

Proof

Consider the virtual memory system with the parameters:

|Mp| = 3 pages

k = 3 or each page of size N contains three sectors.

|Ms| = |Mp| *k = 9 sector frames

F = demand or Fd

R = LRU or R_{LRU}

Program = {abcdefghijkl}, a set of 12 relocatable sectors of size N/3.

ST = $(adgjklhiefbc)^2$.

|ST| = 24

****

Consider Π = {abc, def, ghi, jkl} where A = abc, B = def, etc. Then for ST = adgjklhiefbc adgjklhiefbc

P = ABC DDD CCB BAA ABC DDD CCB BAA

Fd = ABC D00 000 0A0 000 D00 000 0A0

R_{LRU} = 000 A00 000 0D0 000 A00 000 0D0

M_p^t = ABC DDD CCB BAA ABC DDD CCB BAA

AB CCC DDC C0B BAB CCC DDC C0B

A BBB BBD DCC CCA BBB BBD DCC

same

$FFp = \Sigma_{i=1}^{24} |f_p^i| = 7$ page fetches

****

Now, we compute the number of sector fetches for the same sector trace.

ST = adgjk lhief bcadg jklhi efbc

Fd = adgjk lhief bcadg jklhi efbc

R_{LRU} = 00000 0000a dgjkl hiefb cadg

Ms = adgjk lhief bcadg jklhi efbc

adgj klhie fbcad gjklh iefb

adg jklhi efbca dgjkl hief

ad gjklh iefbc adgjk lhie

a dgjkl hiefb cadgj klhi

adgjk lhief bcadg jklh

adgj klhie fbcad gjkl

adg jklhi efbca dgjk

ad gjklh iefbc adgj

same

FFs = $\Sigma_{i=1}^{24} |f_s^i| = 24$ sector faults.

$\therefore$ FFp = 7 < FFs/k = 24/3 = 8 QED.

****

It is interesting to observe that, if the above sector trace, ST = $(adgjklhiefbc)^2$, consisting of two cycles through the same sector reference pattern, were generalized to a sector trace ST = $(adgiklhiefbc)^n$, consisting of n cycles, then FFp = 3+2n and FFs = 12n. Hence, FFp is approximately a factor of 2 less than (FFs)/k for large n. These last two values of FFp and FFs are easily verified by observing that the paging and sectoring simulations of every cycle after the first are respectively the same.

In our empirical studies of the paging behavior of real programs, we found instances where

$k*FFp(|Mp|, N, \Pi, ST, Fd, R_{LRU}) < FFs(|Ms| = |Mp| *k, ST, Fd, R_{LRU})$.

These instances occurred for memory sizes |Mp| in the region of low paging rates under good program structures, i.e., under partitions which produced low values for FFp.

We point out in passing that other similar attempts to bound FFp for certain memory constraints failed. For example, $k*FFp(|Mp|,N, \Pi ,ST,Fd,R_{FIFO})$ is not lower bounded by $FFs(|Ms| = |Mp| *k,ST,Fd,R_{FIFO})$.

The interested reader may verify this by going through the simulation in the proof of Theorem 2 with R_{FIFO} and ST = (a def bc ghi jkl de), while keeping everything else the same.

3.3 Upper Bounds

How large can the value of the page fetch function become by choosing the "worst" program structure, that is, the program structure which results from the partition, Π, that maximizes the value of FFp?

Theorem 3

Given any two-level virtual memory system V, with page size N, primary memory size |Mp|, demand fetch Fd, LRU replacement R_{LRU}, and any sector trace STa, then for any partition, Πa, of the relocatable sectors into logical pages of the program, the maximum number of page fetches given by the page fetch function FFp is upper bounded by $FFp(|Mp|,N, \Pi a,STa,Fd,R_{LRU}) \leq FFs(|Ms| = |Mp|,ST = STa,Fd,R_{LRU})$, where the value of the sector fetch function, FFs, is the number of sector fetches which occur in a two-level virtual memory system V', with primary memory size |Ms| = |Mp|, demand fetch Fd, and LRU replacement

R_{LRU} , using the same sector trace ST = STa.

Proof: Let:

ST = $x^1, x^2, \ldots, x^L$ be any sector trace.

$\Pi = \{ \Pi_1, \Pi_2, \ldots, \Pi_n \}$ be any partition of sectors into pages.

$P = p^1, p^2, \ldots, p^L$ be the resultant page trace computed from Π, and ST.

M_p^t = contents of memory of FFp model at time t.

M_s^t = contents of memory of FFs model at time t.

Fp = $f_p^1, f_p^2, \ldots, f_p^L = F_d$ of FFp.

Rp = $r_p^1, r_p^2, \ldots, r_p^L = R_{LRU}$ of FFp.

Fs = $f_s^1, f_s^2, \ldots, f_s^L = F_d$ of FFs.

Rs = $r_s^1, r_s^2, \ldots, r_s^L = R_{LRU}$ of FFs.

Suppose, at time t in the FFp model, that $p^t = z$, the page containing the set of sectors Π_z is referenced. Then, at time t in the FFs model, $x^t = z'$ is the sector referenced, where sector $z' \in \Pi_z$.

CASE 1.

Suppose $p^t \in Mp^{t-1}$. Then $f_p^t = \phi$.

If $x^t \in Ms^{t-1}$, then $f_s^t = \phi$, and $|f_p^t| = |f_s^t|$.

If $x^t \notin Ms^{t-1}$, then $f_s^t = \{b'\} \notin Ms^{t-1}$, and

$|f_p^t| < |f_s^t|$.

CASE 2.

Suppose $p^t \notin Mp^{t-1}$. Then $f_p^t = \{z\}$, and $r_p^t = \{b\} \subseteq Mp^{t-1}$ under LRU.

If $x^t \notin Ms^{t-1}$, then $f_s^t = \{z'\}$, $r_s^t = \{b'\} \subseteq Mp^{t-1}$ under LRU, and $|f_p^t| = |f_s^t|$.

If $x^t \in Ms^{t-1}$, then $f_s^t = \phi$, and $|f_p^t| > |f_s^t|$. This condition causes a problem.

We will prove that $p^t \notin Mp^{t-1}$ and $x^t \in Ms^{t-1}$ can never occur together.

Assume $x^t \in Ms^{t-1}$. Let $t' < t$, be the largest time, t', such that $x^{t'} = x^t$, then $p^t \in Mp^{t'}$. Since $p^t \notin Mp^{t-1}$, then there occurred at least $|Mp|$ distinct page references to Mp in the interval $(t-1-t', t-1)$ none of which were p^t. Therefore, these were at least $|Ms|=|Mp|$ distinct sector references to Ms in the interval $(t-1-t', t-1)$ none of which were x^t and $x^t \in Ms^{t-1}$ but this contradicts $Rs = R_{LRU}$. Thus, $x^t \notin Ms^{t-1}$ if $p^t \notin Mp^{t-1}$.

Hence, $\Sigma_{i-1}^L |f_p^t| \leq \Sigma_{i-1}^L |f_s^t|$ and the Theorem is proved.

Corollary 3a

$$FFs(k* |Mp|, ST, Fd, Ro)/k \leq FFp(|Mp|, N, Ha, STa, Fd, R_{LRU}) \leq FFs(|Mp|, ST, Fd, R_d)$$

Proof:

Follows immediately from Theorems 1, 3.

Theorem 3 provides an upper bound on the value of the page fetch function, FFp, over all partitions, Πa, of the relocatable sectors into logical pages for virtual memory systems which employ the popular demand fetch and LRU removal algorithms. Under what conditions will the upper bound given by Theorem 3 be the least upper bound or even a tight upper bound?

Let the interval of time between a fetch of any page and the subsequent removal of that page be called a page lifetime. Now, consider a partition, Πc, of sectors into logical pages, such that, during a lifetime, of any page, only one of the sectors of that page is referenced. However, let this one sector be referenced any number of times in a given page lifetime, and let the particular sector which is referenced vary from lifetime to lifetime. We will say that such a partition satisfies the page lifetime constraint.

For any partitions which satisfy the page lifetime constraint, it is obvious that Theorem 3 is the least upper bound. This implies that the extent to which partitions exist which group sectors together which are not used close together in time is the extent to which Theorem 3 will produce a tight bound.

Since LRU is also a stack algorithm, the values for the upper bound given by Theorem 3 can be computed for all memory sizes by one simulation of the sectoring activity for $FFs(|Ms| = |Mp|, ST, Fd, R_{LRU})$.

Therefore, by applying Theorems 1 and 3 a graph similar in form to that shown in Figure 4 can be obtained. The gap between the two curves represents the range of values of the page fetch function, FFp, over all partitions when demand page fetch and LRU page replacement policies are employed. For a particular program structure, the value of FFp in relation to the upper and lower bounds can be used to evaluate the potential of program restructuring.

In Chapter 6, we will present empirical results which show that the bounds given by Theorem 3 are quite reasonable for several actual programs. This implies that real programs can have sector arrangements which result in a lot of page fetches. In fact we found in our studies of real programs that the actual value of the page fetch function can vary by a factor of tens for two different orderings of sectors into the logical pages. All of these results for real programs are given in Chapter 6. However, we will now present an example which will show the logistics of applying Theorems 1 and 3.

3.4 Simple Example of Computing Bounds

We have chosen a very simple, compressed sector trace of a rather small program so that (a) we can illustrate the actual computation of the upper and lower bounds and (b) we can easily obtain the best and worst sector partitions. Note that this example does not represent any of the

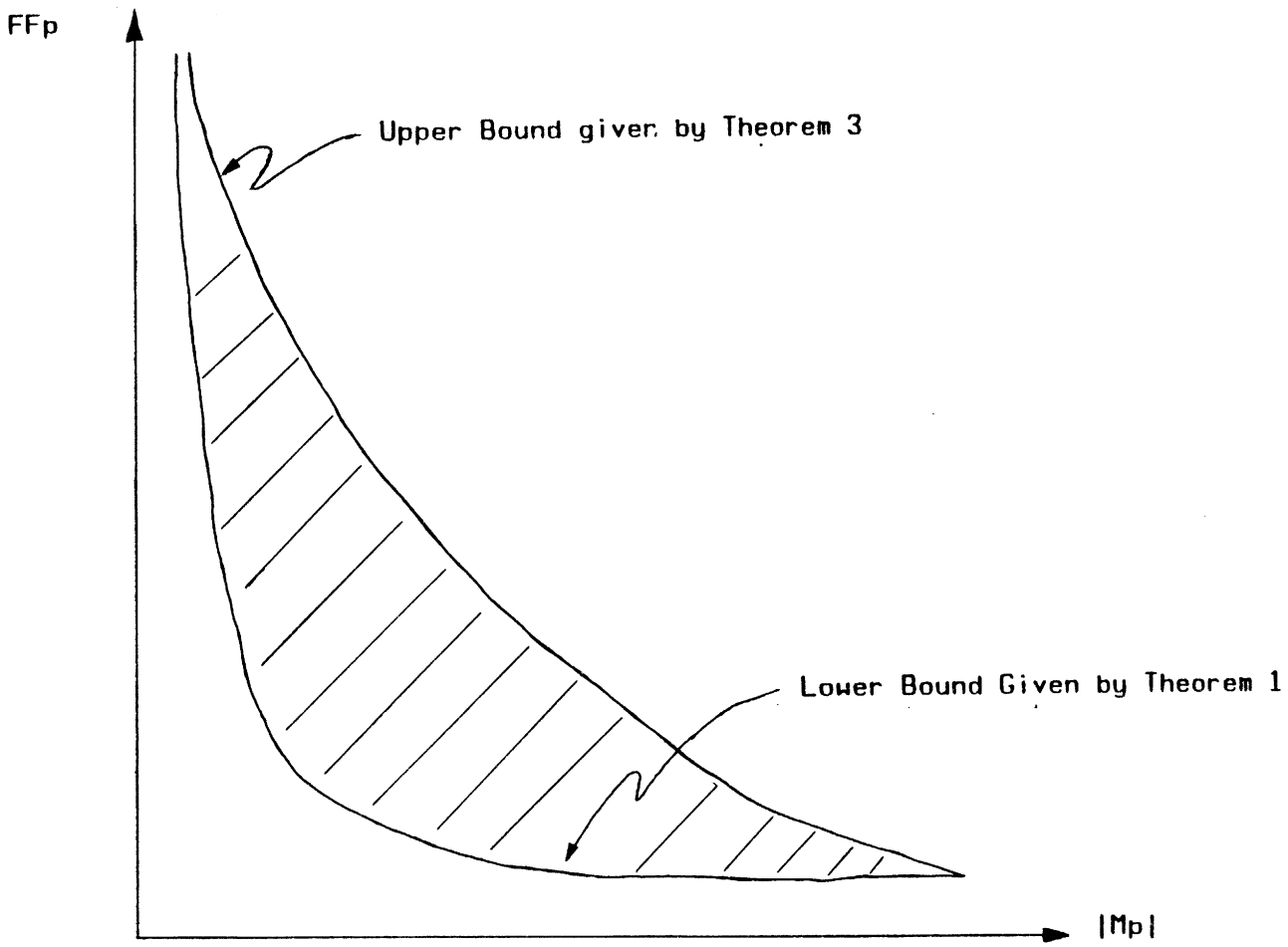


FIGURE 4.

The Allowable Values of FFp as a Function of |Mp|

real programs we tested, since in those cases, the minimum number of references in any sector trace was over 1/2 million. Even though this example does not represent an actual program, it does indicate that, even when 2/3 of this program can fit into primary memory, there is a wide variation in its paging behavior over sector partitions. It also illustrates that there are simple sector traces where the bounds given by Theorems 1 and 3 are simultaneously the greatest lower bound and the smallest upper bound, respectively.

Example of Results:

Consider a virtual memory system with parameters:

|Mp| = 2.

k=3 sectors per page.

F = demand or F_d.

R = LRU, or R_{LRU} .

Program = {abcdefghi}, a set of 9 relocatable sectors of size N/3.

ST = aehae hbdgb dgaeh bficf ibeha dgadg.

|ST| = 30.

****

Applying Theorem 1, we compute FFs(|Ms|=6,ST,Fd,Ro):

ST = aehae hbdgb dgaeh bficf ibeha dgadg

F_d = aeh00 0bdg0 00000 0fic0 0000a dg000

R_0 = 00000 00000 00000 0dga0 0000c fi000

M_s^t = aehae hbdgb dgaeh bficf ibeha dgadg

aeha ehbdg bdgae hbfic figeh adgad

aeh aehbd gbdga ehbfi cfibe hadga

aehh hhbdg aehbb bcfib ehhhh

aee eehbd gaehh hhcfi beeee

aa aaehb dgaee eehcf ibbbb

Theoretical minimum = 12/3 = 4 page fetches.

****

Applying Theorem 3, we compute FFs(|Ms|=2,ST,Fd,R_{LRU}):

ST = aehae hbdgb dgaeh bficf ibeha dgadg

F_d = aehae hbdgb dgaeh bficf ibeha dgadg

R_{LRU} = 00aeh aehbd gbdga ehbfi cfibe hadga

M_s^t = aehae hbdgb dgaeh bficf ibeha dgadg

aeha ehbdg bdgae hbfic fibeh adgad

Theoretical maximum = 30 page fetches.

****

There are: $\frac{9!}{(3!)^{9/3}\ (9/3)!}$ = 280 distinct ways of reordering the 9 relocatable sectors into 3 pages.

****

Consider Π_1 = (abc def ghi) where page A = abc , etc.

Now we compute FFp(|Mp| = 2, Π_1 ,ST,Fd,R_{LRU}).

For ST = (aehae hbdgb dgaeh bficf ibeha dgadg), we get

P = ABCAB CABCA BCABC ABCAB CABCA BCABC

F_d = ABCAB CABCA BCABC ABCAB CABCA BCABC

R_{LRU} = 00ABC ABCAB CABCA BCABC ABCAB CABCA

M_p^t = ABCAB CABCA BCABC ABCAB CABCA BCABC
ABCA BCABC ABCAB CABCA BCABC ABCAB

FFp = $\Sigma_{t=1}^{L}$ $|f_d^t|$ = 30 page fetches for Π_1 = theoretical maximum.

****

Consider Π_2 = (dag beh cfi), where page A = dag , etc.

Now we compute FFp(|Mp| = 2, Π_2 ,ST,Fd,R_{LRU}).

For ST = (aehae hbdgb dgaeh bficf ibeha dgadg), we get

P = ABBAB BBAAB AAABB BCCCC CBBBA AAAAA

F = AB000 00000 00000 0C000 0000A 00000

R = 00000 00000 00000 0A000 0000C 00000

M_p^t = ABBAB BBAAB AAABB BCCCC CBBBA AAAAA
AABA AABBA BBBAA ABBBB BCCCB BBBBB

FFp = $\Sigma_{i=1}^{30} |f_d^i| = 4$ page fetches for Π_2 = theoretical minimum.

****

In the above example, the theoretical minimum value of $FF_p = 4$ from Theorem 1 and the theoretical maximum value of $FF_p = 30$ from Theorem 3 were found to be the greatest lower bound and the smallest upper bound respectively over all partitions, Π .

3.5 Extensions to Lower Bounds

In section 3.2, lower bounds were derived for the case where each page contained at most k sectors. In this section, we would like to relax this constraint.

What were the problems associated with the constraint that pages of a partition must contain at most k sectors? There are no problems when the sectors are all the same size. However, when the sizes of the sectors vary considerably, it becomes more complex to determine the best k. For example, if one chooses k to be, the maximum number of sectors which could fit into any page, then the set of all partitions are allowable, but the value of

$$\frac{FFs(|Ms|=|Mp| \ *k, ST, Fd, Ro)}{k}$$

might not produce a bound which is as tight as we can produce. This is due to two reasons. First, since |Ms|=|Mp| * k, the size of Ms might be larger than necessary to always hold the sectors present in pages of Mp. Note that some pages of Mp might hold fewer than k sectors and that FFs is a monotonically decreasing function of |Ms|. Second, perhaps we can reduce the divisor k when some pages must contain fewer than k sectors.

On the other hand, if one chooses k to be some value less than the maximum number of sectors which could fit into a page, then some of the partitions are not considered.

We will now consider all partitions of relocatable sectors into pages. The only constraint is as before,

$\sum_{S_j \in \Pi_i} |S_j| \leq N$ for all i, which simply

states that the size of any block of the partition in bytes must be less than the page size, N, in bytes. Note that this set of all partitions is the same as the set of partitions when k is chosen equal to the maximum number of sectors which could physically fit into a page. However, we will find tighter bounds.

Consider a program which consists of m relocatable sectors of various sizes. We define the "sector size vector", SS, to be a sequence of sizes of these m sectors, $SS = |S_1|, |S_2|, \ldots, |S_m|$, such that $|S_i| \leq |S_j|$ for all $i \leq j$, $1 \leq j, i \leq m$, where |Si| is the size of Si in

bytes. Recall that:

|Mp| is the number of page frames in the paged memory, Mp.

N is the page frame size in bytes.

Now we define a function, f_1, in terms of |Mp|, N, and SS:

f_1 (|Mp|,N,SS) = the maximum number of sectors of sizes in SS which can be packed into a set of |Mp| page frames of size N bytes each, when sectors are not allowed to cross page boundaries.

Example.

Let:

$|S_1| = |S_2| = |S_3| = 1000$ bytes; $|S_4| = 2000$ bytes;

$|S_5| = |S_6| = 3000$ bytes.

N = 4000 bytes

then,

$f_1(1,N,SS) = 3$

$f_1(2,N,SS) = 5$

$f_1(3,N,SS) = 6$

Since the computation of f_1 can become a complex combinatorial problem in itself, we will give an easy method of computing an upper bound for f_1.

The function f_1^v is defined in terms of |Mp|,N,SS as follows.

f_1^v (|Mp|,N,SS) = W if and only if

$\Sigma_{i=1}^{w} |Si| \geq |Mp|*N$ and $\Sigma_{i=1}^{w-1} |Si| < |Mp|*N$.

It should be clear that f_1 (|Mp|,N,SS) $\leq f_1^v$ (|Mp|,N,SS) for all |Mp|,N,SS. For the above example,

f_1^v (1,N,S) = 4

f_1^v (2,N,S) = 5

f_1^v (3,N,S) = 6.

Let us interpret a particular form of f_1; that is, if |Mp| = 2, then f_1 (2,N,SS) is by definition the maximum number of sectors which can be packed into 2 page frames of N bytes each.

We can use f_1 (|Mp|,N,SS), f_1 (2,N,SS) and the sector fetch function, FFs, to lower bound the page fetch function, FFp, as follows:

Theorem 4.

Given any two-level virtual memory system V, with page size N, primary memory size |Mp|, any valid page replacement algorithm Ra, demand page fetch Fd, and any sector trace STa, then for any partition Πa of the relocatable sectors into the logical pages of the program, the minimum number of page fetches given by the page fetch function FFp is lower bounded by

$$FFp(|Mp|,N,\Pi a,STa,Fd,Ra) \geq \frac{(FFs(|Ms| - f_1(|Mp|,N,SS),ST,Fd,Ro))}{f_1(2,N,SS)/2} - \Delta,$$

where $\Delta = \frac{2f_1(1,N,SS) - f_1(2,N,SS)}{f_1(2,N,SS)}$, and

where the value of the sector fetch function FFs is the number of sector fetches which occur in a two-level virtual memory system V', with primary memory size $|Ms| = f_1(|Mp|,N,SS)$, using demand fetch Fd, optimum replacement algorithm Ro, and the same sector trace ST = STa. The function f_1 is as previously defined, and SS is the sector size vector.

Corollary 4a

$$FFp(|Mp|,N,\Pi a,STa,Fd,Ra) \geq \frac{(FFs(|Ms| = f_1(|Mp|,N,SS),ST,Fd,Ro))-1}{f_1(2,N,SS)/2}$$

Corollary 4b

$$FFp(|Mp|,N,\Pi a,STa,Fd,Ra) \geq \frac{(FFs(|Ms| = W_1,ST,Fd,Ro)-1,}{W_2/2}$$

where W_1 equals either $f_1(|Mp|,N,SS)$ or $f_1^u(|Mp|,N,SS)$, and W_2 independently of W_1 equals either $f_1(2,N,SS)$ or $f_1^u(2,N,SS)$. Corollary 4b says that we can lower bound FFp in terms of the easily computed function f_1^u.

Corollary 4c

$$\frac{FFs(|Ms| = |Mp| * k,\ \Pi a,STa,Fd,Ro)}{k} \leq \frac{FFs(|Ms| = f_1(|Mp|,N,SS),ST,Fd,Ro)}{f_1(2,N,SS)/2}$$

Corollary 4c states that the bounds given by Theorem 4 may be tighter than the bounds given by Theorem 1 where k is the maximum number of sectors which can physically fit into a page.

Proof of Theorem 4

Notation and properties

Let $ST_a = x^1, x^2, \ldots, x^L$ where x^t is the sector referenced at time t. For virtual memory system V and FFp, let:

1. $\Pi_a = \{\Pi_1, \Pi_2, \ldots, \Pi_n\}$ be any partition of sectors into the n logical pages of the program where each page contains any number of sectors such that $\sum_{S_j \in \Pi i} |S_j| \leq N$ for $1 \leq i \leq n$.
2. $P = (p^1, p^2, \ldots, p^L)$ be the resultant page trace computed uniquely from ST and Πa, such that, if $x^t \in \Pi j$, then $p^t = j$.
3. $Fa = f_a^1, f_a^2, \ldots, f_a^L$ be the demand fetch policy, where $f_a^t \in \Pi a$ and $f_a^t \notin Mp^{t-1}$ and $|f_a^t| = 1$ or 0, the number of pages in f_a^t. Note that we have chosen to denote Fd for FFp by Fa to avoid notational conflict with the Fd for FFs.
4. $Ra = r_a^1, r_a^2, \ldots, r_a^L$ be any removal policy where $r_a^t \in \Pi a$ and $r_a^t \leq Mp^{t-1}$ and $|r_a^t| = 1$ or 0, the number of pages in r_a^t.
5. M_p^t be the set of pages in Mp at time t and $M_p^0 = \emptyset$.
6. $M_p^t = (Mp^{t-1} \cup f_a^t) - r_a^t$.

First we prove Lemma 5.

Lemma 5:

There exists valid demand fetch and removal policies, Fd and Rd, for the FFs model such that

$$FFp(|Mp|,N,\Pi a,STa,Fd,Ra) \geq \frac{FFs(|Ms| = f_1(|Mp|,N,SS),ST,Fd,Rd)}{f_1(2,N,SS)/2} - \Delta ,$$

where $\Delta = \frac{2f_1(1,N,SS)-f_1(2,N,SS)}{f_1(2,N,SS)}$

Proof:

For the FFs model, Fd and Rd will be constructed by forming a sequence of valid replacement and fetch policies $(F_1,R_1),(F_2,R_2),\ldots,(Fh,Rh)$, where:

1. $F_1 = f_1^1, f_1^2, \ldots, f_1^L$ and $f_1^t = g(f_s^t)$ = the set of sectors making up the page in f_s^t, for $1 \leq t \leq L$.
2. Similarly $R_1 = r_1^1, r_1^2, \ldots, r_1^L$ and $r_1^t = g(r_s^t)$, for $1 \leq t \leq L$.
3. $Fh = Fd = f_d^1, f_d^2, \ldots, f_d^L$ and $R_h = R_d = r_d^1, r_d^2, \ldots, r_d^L$, for $1 \leq t \leq L$ where
   $f_d^t = r_d^t = 0$ if $x^t \in M_d^{t-1}$; $f_d^t = x^t$ and
   $r_d^t = 0$ if $x^t \notin M_d^{t-1}$ and $|M_d^{t-1}| < |Ms|$;
   $f_d^t = x^t$ and $r_d^t = \{b\} \subset M_d^{t-1}$ if
   $x^t \notin M_d^{t-1}$ and $|M_d^{t-1}| = |Ms|$; and
   $M_d^t = (M_d^{t-1} \cup f_d^t) - r_d^t$ to satisfy demand sectoring.

Since $|Ms| = f_1(|Mp|, N, SS) \geq ||Mp||$, Lemma 4a says that the above construction exists such that

$$\Sigma_{i=1}^{L} |f_1^i| \geq \Sigma_{i=1}^{L} |f_d^i|$$

Therefore, we have Fact 1:

Fact 1.

$$\Sigma_{i=1}^{L} |f_1^i| \geq \Sigma_{i=1}^{L} |f_d^i| = FFs(|Ms| = f_1(|Mp|, N, SS), ST, Fd, Rd).$$

Now, let's prove Fact 2.

Fact 2.

$$\Sigma_{i=1}^{L} |f_1^i| \leq ((f_1(2, N, SS) FFp(|Mp|, N, \Pi a, STa, Fd, Ra) + f_1(2, N, SS) * \Delta))/2$$

Proof.

$$\Sigma_{i=1}^{L} |f_1^i| = \Sigma_{i=1}^{L} |g(f_a^i)| - \Sigma_{i=1}^{L} |f_a^i| |g(f_a^i)|$$

since $|f_a^i| = 1$ iff $|g(f_a^i)| > 0$ and $|f_a^i| = 0$ iff $|g(f_a^i)| = 0$.

Note that $|g(f_a^i)|$ is the number of sectors in the page specified by f_a^i.

Also, note that $\Sigma_{i=1}^{L} |f_a^i| = FFp(|Mp|, N, \Pi a, STa, Fd, Ra)$.

Now let's compress $Fa = f_a^1, f_a^2, \ldots, f_a^L$ to get

$F_a' = f_a'^1, f_a'^2, \ldots, f_a'^{L'}$ by taking out all the $f_a^i = 0$.

Clearly $\Sigma_{t=1}^{L} |f_a^t| = \Sigma_{t=1}^{L'} |f_a''^t|$ and

$\Sigma_{t=1}^{L} |f_a^t||g(f_a^t)| = \Sigma_{t=1}^{L'} |f_a''^t||g(f_a''^t)| = \Sigma_{t=1}^{L} |f_1^t|$.

Furthermore, note that, under the definition of demand fetch, no two successive page fetches can be to the same page. This is obvious, since under demand fetch a page is fetched and is kept in primary memory until it has to be removed to make room for another page.

Therefore no two successive values $g(f_a''^t)$ and $g(f_a''^{t+1})$ can be the same.

Now, the sum $\Sigma_{t=1}^{L'} |f_a''^t||g(f_a''^t)|$ is clearly maximized if, for all odd t, $|g(f_a''^t)|$ is equal to the maximum number of sectors which can fit in a page, and if, for all even t, $|g(f_a''^t)|$ is equal to the next maximum number of sectors which can fit in a page. Thus,

$$\Sigma_{t=1}^{L} |f_1^t| = \Sigma_{t=1}^{L'} |f_a''^t||g(f_a''^t)| \leq \Sigma_{t=1}^{L'} |f_a''^t| \frac{|f_1(2,N,SS)}{2} \text{ for even } L', \text{ and}$$

$$\Sigma_{t=1}^{L} |f_1^t| = \Sigma_{t=1}^{L'} |f_a''^t||g(f_a''^t)| \leq \Sigma_{t=1}^{L'-1} |f_a''^t| \frac{|f_1(2,N,SS)}{2} + |f_a''^{L'}| f_1(1,N,SS)$$

for odd L'.

Note that $f_1(1,N,SS)=f_1(1,N,SS)+\frac{f_1(2,N,SS)}{2}-\frac{f_1(2,N,SS)}{2}$, and thus

$$\Sigma_{l=1}^{L} |f_l^t| \leq \frac{f_1(2,N,SS)}{2} \Sigma_{l=1}^{L'} |f_0^{t'}| + f_1(1,N,SS) - \frac{f_1(2,N,SS)}{2}$$

for all L'.

Hence, $\Sigma_{l=1}^{L} |f_l^t|$

$\leq (f_1(2,N,SS)\ FFp(|Mp|,N,\Pi a,STa,Fd,Ra)+(2f_1(1,N,SS)-f_1(2,N,SS))/2$

and Fact 2 is proved.

From Fact 1 and Fact 2, we have

$$FFp(|Mp|,N,\Pi a,STa,Fd,Ra) \geq \frac{FFs(|Ms| - f_1(|Mp|,N,SS),ST,Fd,Rd) - \Delta}{f_1(2,N,SS)/2}$$

This proves Lemma 5.

Now, from Lemma 1, we know that,

$$FFs(|Ms| - f_1(|Mp|,N,SS),ST,Fd,Rd) \geq FFs(|Ms| - f_1(|Mp|,N,SS),ST,Fd,Ro)$$

Therefore, Theorem 4 follows immediately. QED.

Proof of Corollary 4a:

It follows immediately from the fact that

$$0 \leq \frac{2f_1(1,N,SS)-f_1(2,N,SS)}{f_1(2,N,SS)} \leq 1.$$

Proof of corollary 4b:

Corollary 4b follows directly from Theorem 4 and Lemmas 2, 3. Lemmas 2, 3 give,

$FFs(|Ms|=f_1(|Mp|,N,SS),ST,Fd,Ro) \geq FFs(|Ms|=W_1,ST,Fd,Ro)$

since $W_1 \geq f_1(|Mp|,N,SS)$. The divisor goes through since $W_2 \geq f_1(2,N,SS)$.

Proof of corollary 4c:

Corollary 4c follows from Lemmas 2, 3 since $|Mp| * k \geq f_1(|Mp|,N,SS)$, and $k \geq f_1(2,N,SS)/2$.

To compute the lower bound of Theorem 4, simply make one sector simulation run through the sector trace and record the number of sector fetches for each possible sector memory size. Then for a particular value of $|Mp|$, use $f_1(|Mp|,N,SS)$ to select the proper value of FFs and divide by $f_1(2,N,SS)$ to get the bound.

If the objective is to lower bound FFp over all partitions, then Theorem 4 may give tighter bounds than Theorem 1 if the range of sector sizes is large. For this is the case when $f_1(|Mp|,N,SS) \leq k * |Mp|$. Furthermore, $f_1(|Mp|,N,SS)$ can become substantially less than $k * |Mp|$ for large values of $|Mp|$. The term, $f_1(2,N,SS)/2$, in the lower bound is the average value of k for the two pages having the largest number of sectors. We cannot extend this average over all pages, since every other page fetch could be to the page containing the largest number of sectors,

while every intervening fetch could be to the page having the second largest number of sectors. Even if all pages are fetched, if the above behavior occurs sufficiently often in the execution of a program, then we still cannot average over all pages.

Is there any way to compensate for the case when some sectors are much larger than others? For ease in the following discussion, let the average vaue of k for the two pages having the largest number of sectors, $f_1(2,N,SS)/2$, be denoted by k', and let the average size of these sectors be denoted by N/k'. In order to illustrate some typical values one may encounter, we point out that for the real programs we investigated, the values of k' were on the order of 3 to 6, and, hence, N/k' was 1/3 to 1/6 of a page for a page size of 4096 bytes. Now let's assume that we are given a particular program, Q, and we compute the value of N/k' and find that there are several sectors whose sizes are considerably larger than N/k'. Now consider what happens if we break up these large sectors into as many subsectors as we can without increasing the value of k'. This new program with the large sectors replaced by the smaller subsectors is called Q^*. Given Q^*, it is still quite easy to compute a sector trace over Q^* from the address trace. We call such a sector trace ST^*. Using this sector trace, ST^*, and the program, Q^*, we can apply Theorem 4 to compute the lower bound on the page fetch function, FFp, over all partitions, Πa^*, of sectors of Q^* into logical pages. We present two important observations on this lower bound:

A). This lower bound is valid over all partitions of sectors of Q' into pages. Therefore, the lower bound is certainly true for all the partitions over Q^* that are constrained to comply with Q. That is, if a page in a partition of Q^* contained one subsector of a sector, then it would have to contain all the subsectors of that sector. This restriction on the set of all partitions over Q^* simply produces the set of partitions which result when reprogramming is not allowed. Let Πar^* denote any such restricted partitions of Q^*.

B). This lower bound using ST^* and Πar^* over Q^* is probably much larger for most real programs than the lower bound computed by Theorem 4 using ST and Πa over Q. The rationale for this is simply that it will take several subsector fetches to bring into the sectored memory the same information that could be fetched by one large-sector fetch.

Observation B need not necessarily be true; that is, the lower bound which results from breaking up the large sectors could theoretically be smaller than the lower bound computed by not breaking up the large sectors. However, this presents no practical problems. Since both methods will produce valid lower bounds, we simply compute both and use the greater lower bound. In our analysis of real programs, we found that the lower bound computed from breaking up the large sectors was substantially larger than the lower bound computed when the large sectors were not divided.

We will now formalize the notions of Q^* and ST^* and define the relationship between Q and Q^* and between ST and ST^*. Then, Theorem 5 is presented, which states that the page fetch function, FFp, is lower bounded in terms of the sectoring behavior given by ST^*.

Let, $Q = Q_1 \cup Q_2$ = {set of m relocatable sectors of any program}
where $Q_1 = \{S_1, S_2, \ldots, S_k\}$, $Q_2 = \{S_{k+1}, S_{k+2}, \ldots, S_m\}$
such that $f_1(2,N,SS)/2 = k/2$ and $|Si| \leq |Sj|$ for all $Si \in Q_1$ and $Sj \in Q_2$.
Let, $SS = |r_1|, |r_2|, \ldots, |r_k|, |r_{k+1}|, \ldots, |r_m|$ be the sector size vector of Q; that is, $r_i \in Q$ and $|r_i| \leq |r_j|$ for $i \leq j$ and $|r_m| \leq N$, the page size.
Note that $|r_k|$ is the size of the largest sector in Q_1.
Furthermore, note that the above construction is always possible.

Now, we break up the large sectors of Q into subsectors. Let

$Si = \{Si_1^*\}$ for $1 \leq i \leq k$ and

$Si = \{Si_1^*, Si_2^*, \ldots, Si_{l_i}^*\}$ for $k < i \leq m$ such that

$|r_k| \leq |Sij^*|$.

This last constraint is sufficient to guarantee that $(f_1(2,N,SS))/2 = k/2$ does not change because of the small subsectors.
In practice, one could choose $|Sij^*| = |r_k|$ for $1 \leq j < l_i$ and $|r_k| \leq |Sij^*| < 2|r_k|$ for $j = l_i$.

Now define, $Q^* = Q_1^* \cup Q_2^*$ = {set of m' relocatable sectors of the same program}

where $Q_1^* = \{S_{11}^*, S_{22}^*, \ldots S_{kk}^*\}$ and

$Q_2^* = \{S_{k+1,1}^*, S_{k+1,2}^*, \ldots, S_{k+1,|_{k+1}}^*, \ldots, S_{m',1}^*, S_{m',2}^*, \ldots, S_{m',|_{m'}}^*\}$.

Let, $SS^* = |r_1^*|, |r_2^*|, \ldots, |r_{m'}^*|$ be the sector size vector of Q^*, $|r_i^*| \leq |r_j^*|$ for $i \leq j$.

Note that $(f_1(2,N,SS))/2 = (f_1(2,N,SS^*))/2$.

Given any address trace, A, and the sector ordering of the programs Q and Q^* for that address trace, we can easily compute:

$ST = S^1, S^2, \ldots, S^L$ for Q and $ST^* = S^{*1}, S^{*2}, \ldots, S^{*L}$ for Q^*, where

$S^t \in Q$ and $S^{*t} \in Q^*$.

Note that, if $S^{*t} = Sij^*$ then $S^t = Si$ for $1 \leq t \leq L$.

Thus, we can also compute ST from ST^*.

Theorem 5 is presented in terms of the above definitions of Q^* and ST^*.

Theorem 5.

Given any two-level virtual memory system V, with page size N, primary memory size |Mp|, any valid page replacement algorithm Ra, demand page fetch Fd, and any sector trace STa, then for any partition,

Πa, of the relocatable sectors into the logical pages of the program, Q, the minimum number of page fetches given by the page fetch function, FFp, is lower bounded by

$$FFp(|Mp|,N,\Pi a,STa,Fd,Ra) \geq \frac{FFs(|Ms| = f_1(|Mp|,N,SS^*),ST = STa^*,Fd,Ro)}{f_1(2,N,SS)/2} - \Delta,$$

where $\Delta = \dfrac{2f_1(1,N,SS) - f_1(2,N,SS)}{f_1(2,N,SS)}$,

and where the value of the sector fetch function FFs is the number of sector fetches which occur in a two-level virtual memory system V', with primary memory size $|Ms| = f_1(|Mp|,N,SS^*)$, using demand fetch Fd, optimum replacement algorithm Ro, and sector trace $ST = STa^*$. The function f_1 is previously defined, SS is the sector size vector of Q, and SS^* is the sector size vector of Q^*.

Proof:

Let Q, ST, Q^* and ST^* be exactly as defined immediately before Theorem 5 was stated.

Let $\Pi a^* = \{ \Pi_1^*, \Pi_2^*, \ldots, \Pi_n^* \}$ be any partition of the relocatable sectors of Q^* into logical pages, where page $k = \Pi_k^*$ for $1 \leq k \leq n$ and $\sum_{S_{ij}^* \in \Pi_k^*} |Sij^*| \leq N$.

Applying Theorem 4 to Q^* gives by simple substitution,

$$FFp(|Mp|,N,\Pi a^*,ST^*,Fd,Ra) \geq \frac{FFs(|Ms| = f_1(|Mp|,N,SS^*),ST^*,Fd,Ro)}{f_1(2,N,SS^*)/2} - \Delta,$$

and since $f_1(2,N,SS^*)/2 = f_1(2,N,SS)/2$ we get

$$FFp(|Mp|,N,\Pi a^*,ST^*,Fd,Ra) \geq \frac{FFs(|Ms| = f_1(|Mp|,N,SS^*),ST^*,Fd,Ro)}{f_1(2,N,SS)/2} - \Delta.$$

Let $\Pi a = \{\Pi_1, \Pi_2, \ldots, \Pi n\}$ be any partition of relocatable sectors of Q into logical pages such that $\sum_{S_i \epsilon \Pi_k} |Si| \leq N$ and $Si \epsilon Q$,

where page $k = \Pi k$ for $1 \leq k \leq n$.

Given any Πa, then we construct Πar^* as follows:

$\Pi ar^* = \{\Pi r_1^*, \Pi r_2^*, \ldots, \Pi r_n^*\}$ such that,

for all $Si \epsilon \Pi k$, $Sij^* \epsilon \Pi rk^*$

for $1 \leq k \leq l_i$ and page $k = \Pi rk^*$ for $1 \leq k \leq n$.

Now,

$$FFp(|Mp|, N, \Pi ar^*, ST^*, Fd, Ra) \geq \frac{FFs(|Ms| = f_1(|Mp|, N, SS^*), ST^*, Fd, Ro)}{f_1(2, N, SS)/2} - \Delta,$$

since the set of all Πar^* is a subset of the set of all Πa^*.

Now we prove that

$FFp(|Mp|, N, \Pi ar^*, ST^*, Fd, Ra) = FFp(|Mp|, N, \Pi a, ST, Fd, Ra)$

We need to show that the page trace

$P^* = p^{*1}, p^{*2}, \ldots, p^{*L}$, computed from Πar^* and ST^*, is the same as the page trace $P = p^1, p^2, \ldots, p^l$, computed from Πa and ST.

Let $ST^* = S^{*1}, S^{*2}, \ldots, S^{*L}$ and $ST = S^1, S^2, \ldots, S^L$.

Let the sector referenced in ST^* at time t be S^{*t} for $1 \leq t \leq L$,

Then $S^{*t} = Sij^*$ for some $1 \leq i \leq m'$ and $1 \leq j \leq l_i$,

and $Sij^* \in \Pi rk^*$ for some $1 \leq k \leq n$. Hence, $p^{*t} = k$.

Given $S^{*t} = Sij^*$, then $S^t = Si$, and, given $Sij^* \in \Pi rk^*$, then $Si \in \Pi k$. Hence, $p^t = k$, and we have $p^{*t} = p^t$ for $1 \leq t \leq L$.

Therefore,

$FFp(|Mp|,N, \Pi ar^*,ST^*,Fd,Ra) = FFp(|Mp|,N, \Pi a,ST,Fd,Ra)$ and

$$FFp(|Mp|,N\ \Pi a,ST,Fd,Ra) \geq \frac{FFs(|Ms| = f_1(|Mp|,N,SS^*),ST^*,Fd,Ra)}{f_1(2,N,SS)/2} - \Delta,$$

QED.

The following simple example is given to illustrate that Theorem 5 can produce a tighter bound than Theorem 4. This example is made as simple as possible such that the mechanics of applying Theorem 5 can be presented.

Example:

Let $Q = \{S_1, S_2, \ldots, S_{12}\}$ where $|Si| = 1000$ bytes for $1 \leq i \leq 8$, and $|Si| = 4000$ bytes for $8 < i \leq 12$ and $N = 4000$ bytes. Now let's divide Si for $8 < i \leq 12$ into four parts, each being 1000 bytes long; i.e., Si becomes $\{Si_1, Si_2, Si_3, Si_4\}$ where $|Sij| = 1000$ bytes for $1 \leq j \leq 4$. Thus,

$Q^* = \{S_1, S_2, S_3, \ldots, S_8, S_{91}, S_{92}, S_{93}, S_{94}, \ldots, S_{12,1}, S_{12,2}, S_{12,3}, S_{12,4}\}$.

Let ST' = $S_1, S_2, S_3, \ldots, S_8, S_{91}, S_{92}, S_{93}, S_{94}, \ldots, S_{12,1}, S_{12,2}, S_{12,3}, S_{12,4}$.
This represents the compressed reference
behavior of one pass through Q' where every
unit of Q' is touched. It is reasonable to assume that such sector
behavior could represent one pass through a small loop of a much larger
real program.

Now, ST=$S_1, S_2, S_3, \ldots, S_8, S_9, S_9, S_9, S_9, \ldots, S_{12}, S_{12}, S_{12}, S_{12}$.
Evaluating FFp(|Mp|,N=4000, Πa,ST,Fd,Ra), gives 6 page fetches when
$\Pi_1 = \{S_1, S_2, S_3, S_4\}$, $\Pi_2 = \{S_5, S_6, S_7, S_8\}$ and
$\Pi i = \{Si+6\}$ for $2 < i \leq 6$, and |Mp| and Ra take on any values. It
should be clear that this partition minimizes FFp.

Theorem 4 gives a lower bound for FFp of

$$\frac{FFs(|Ms| = f_1(|Mp|,N,SS),ST,Fd,Ro)}{f_1(2,N,SS)/2} - \Delta = (12/4)-0 = 3,$$

for all values of |Ms|. Note that $f_1(2,N,SS) = 8$ and
$f_1(1,N,SS) = 4$, hence $\Delta = 0$. Theorem 5 gives a lower bound for FFp
of $\frac{FFs(|Ms| = f_1(|Mp|,N,SS^*),ST^*,Fd,Ro)}{f_1(2,N,SS)/2} - \Delta = (24/4)-0 = 6,$

for all values of |Ms|. Thus, Theorem 5 gives the greater lower bound,
and it is a factor of 2 better than the bound given by Theorem 4.

Now we extend Theorems 4 and 5 to include the cases where sectors can be any size, and we let the sectors cross page boundaries.

We now present Theorem 6 which lower bounds FFp over all sector orderings SO into the n-page logical address space. The sectors can be any size and may cross page boundaries. This model corresponds to the case where sectors are clustered together into groups and then these groups are packed into the virtual address space.

Since sectors may cross page boundaries, one may not be able to determine the page trace from the sector trace ST. We define SOT to be the sector trace consisting of ordered pairs of elements:

SOT = $(S^1,O^1),(S^2,O^2),\ldots,(S^L,O^L)$ where S^t is the sector referenced at time t and O^t is the offset in S^t referenced at time t. Given a sector trace SOT and a sector ordering SO as defined in Chapter 2, the page trace follows immediately.

Note that SOT* is exactly the same as ST* except that the elements of SOT* are simply ordered pairs. Also note that the construction of Q* is not affected by allowing sectors to cross page boundaries.

Theorem 6.

Given any two-level virtual memory system V, with page size N, primary memory size |Mp|, any valid page replacement algorithm Ra, demand page fetch Fd, and any sector trace SOTa, then for any sector

ordering SOa, of the relocatable sectors into the logical address space of the program Q, the minimum number of page fetches given by the page fetch function FFp, is lower bounded by

A.

$$FFp(|Mp|,N,SOa,SOTa,Fd,Ra) \geq \frac{FFs(|Ms| = f_1^v(|Mp|,N,SS),ST = SOTa,Fd,Ro)}{f_1^v(2,N,SS)/2} - \Delta$$

and by

B.

$$FFp(|Mp|,N,SOa,SOTa,Fd,Ra) \geq \frac{FFs(|Ms| = f_1^v(|Mp|,N,SS'),ST = SOTa^*,Fd,Ro)}{f_1^v(2,N,SS)/2} - \Delta$$

where $\Delta = \frac{2f_1^v(1,N,SS) - f_1^v(2,N,SS)}{f_1^v(2,N,SS)}$,

and where the value of the sector fetch function FFs is the number of sector fetches which occur in a two-level virtual memory system V', with primary size |Ms|, using demand fetch Fd and optimum replacement Ro, and sector trace ST = SOTa in part A and ST = SOTa* in part B.

Proof of Theorem 6:

Let $SOT_a = (S^1,O^1),(S^2,O^2),\ldots,(S^L,O^L)$, where S^t is the sector referenced at time t and O^t is the offset. For virtual memory system V and FFp, let:

1. SOa be any sector ordering of the relocatable sectors in the n pages of the address space of program Q.
2. $P = p^1, p^2, \ldots, p^L$ be the resultant page trace computed uniquely from SOTa and SOa, such that $p^t = (L(S^t)+O^t)/N$.
3. $Fa = f_a^1, f_a^2, \ldots, f_a^L$ be the demand fetch policy, where $f_a^t = |p^t|$ or $\emptyset$; $f_a^t \cap Mp^{t-1} = \emptyset$. Note that we have chosen to denote Fd of the FFp model by Fa to avoid notational conflict with the Fd of the FFs model.
4. $Ra = r_a^1, r_a^2, \ldots, r_a^L$ be any removal policy under demand fetch, where $r_a^t \subseteq Mp^{t-1}$ and $|r_a^t| = 1$ or $\emptyset$.
5. $Mp^t = (Mp^{t-1} - r_a^t) \cup f_a^t$ and $Mp^0 = \emptyset$.

First we prove the following lemma.

Lemma 6:

There exists a valid demand fetch and removal policy, Fd and Rd, for the FFs model such that

$$FFp(|Mp|,N,SOa,SOTa,Fa,Ra) \geq \frac{FFs(|Ms| = f_1^v(|Mp|,N,SS),SOTa,Fd,Rd)}{f_1^v(2,N,S)/2} - \Delta,$$

where $\Delta = \frac{2f_1^v(1,N,S) - f_1^v(2,N,SS)}{f_1^v(2,N,SS)}$.

For the FFs model, Fd and Rd will be constructed by forming a valid sequence of replacement and fetch policies

$(F_1, R_1), (F_2, R_2), \ldots, (F_h, R_h)$, where:

1. $F_l = f_l^1, f_l^2, \ldots, f_l^L$, and $f_l^t = g(f_s^t)$ = the set of sectors having any of their parts in f_s^t for $1 \leq t \leq L$.
2. Similarly, $R_l = r_l^1, r_l^2, \ldots, r_l^L$, and $r_l^t = g(r_s^t)$ = the set of sectors having any of their parts in r_s^t for $1 \leq t \leq L$.
3. $Fh = fd = f_d^1, f_d^2, \ldots, f_d^L$, and
4. $Rh = Rd = r_d^1, r_d^2, \ldots, r_d^L$, for $1 \leq t \leq L$, where
   $f_d^t = r_d^t = \emptyset$, if $x^t \in Md^{t-1}$; $f_d^t = x^t$ and
   $r_d^t = \emptyset$, if $x^t \notin Md^{t-1}$ and $|Md^{t-1}| < |Ms|$;
   $f_d^t = x^t$ and $r_d^t = (b) \subset Md^{t-1}$, if $x^t \notin Md^{t-1}$ and $|Md^{t-1}| = |Ms|$; and $Md^t = (Md^{t-1} - r_d^t) \cup f_d^t$ to satisfy demand sectoring.

Lemma 4a is still true for this case when sectors may cross page boundaries. The proof of Lemma 4a when sectors are allowed to cross page boundaries is <u>exactly</u> the same as before except that we add the following to the proof. (Recall that z' is the sector referenced at time t.)

If it ever occurs that $z' \in f_{j-1}^t$ and $z' \in Mp^{t-1}$, then simply remove z' from f_{j-1}^t. This only reduces the value of $|f_j^t|$ and it keeps sector z' from being added to the deferral sector list when z' is in the sectored memory.

Since $|Ms| = f_1^u(|Mp|,N,SS) \geq ||Mp||$, Lemma 4a says that the above construction exists such that

$$\Sigma_{i=1}^{L} |f_i^t| \geq \Sigma_{i=1}^{L} |f_d^t| - FFs(|Ms| - f_1^u(|Mp|,N,SS),SOT,Fd,Rd)$$

Fact 3

$$\Sigma_{i=1}^{L} |f_i^t| \leq \frac{(f_1^u(2,N,SS)FFp(|Mp|,N,SO,SOT,Fd,Rd))}{2} + \frac{f_1^u(2,N,SS) * \Delta}{2}$$

The proof of Fact 3 is exactly the same as Fact 2 of theorem 4 except that $|g(f_0^t)|$ becomes the number of sectors having any of their parts in f_0^t.

Hence Lemma 6 is true. Lemma 6 and Lemma 1 prove part A of the Theorem.

Proof of part B.

Given any address trace A and any Q, construct Q*, SOT, and SOT* exactly as in Theorem 5, except denote the elements of SOT and SOT* as ordered pairs.

The proof of part B is almost exactly the same as the proof of Theorem 5. We point out the exceptions below.

Instead of applying Theorem 4 to Q^* as in Theorem 5, we apply part A of Theorem 6 to Q^* and use the fact that $f^u_1(2,N,SS) = f^u_1(2,N,SS^*)$ to get

$$FFp(|Mp|,N,SOa^*,SOT^*,Fd,Ra) \geq \frac{FFs(|Ms|=f^u_1(|Mp|,N,SS^*),SOT^*,Fd,Ro)}{f^u_1(2,N,SS)/2} - \Delta$$

In the proof of Theorem 5, we restricted the set of Πa such that subsectors could not be in different pages. Here we restrict the set SO^*_A of all SOa^* to get the subset SO^*_{AR}. Let $x \in SO^*_A$, then $x \in SO^*_{AR}$ if the subsectors of each sector in x occur together as a subsequence of $SOar^*$, and if the subsectors of each sector are ordered in the subsequence as they occur in the sector. We are simply restricting the set of all SOa^* such that we get the set of all SOa when the common subsectors of each subsequence of each $SOar^*$ are concatenated together.

Since the above result, $FFp \geq FFs$, is true for all SOa^*, it must be true for any constrained subset of SOa^*. In particular it must be true for all $SOar^*$. Thus

$$FFp(|Mp|,N,SOar^*,SOT^*,Fd,Ra) \geq \frac{FFs(|Ms| = f^u_1(|Mp|,N,SS^*),SOT^*,Fd,Ro)}{f^u_1(2,N,SS)/2} - \Delta$$

Now we need to show, as in Theorem 5, that the page trace P^* computed from $SOar^*$ and SOT^* is the same as the page trace P computed from SOT and SOa. This is obvious from the construction of $SOar^*$ and SOT^*. That is, P^{*t} computed from (S^{*t}, O^{*t})

and SOar* must be the same as P^t computed from (S^t, O^t) and SOa. Thus, FFp(|Mp|,N,SOa,SOT,Fd,Ra) = FFp(|Mp|,N,SOar*,SOT*,Fd,Ra) and the proof of B follows immediately. QED.

3.6 Bounds for Working Set Management

Theorems 1-5 give upper and lower bounds on the number of page fetches required to execute a program in any fixed primary memory size. However, there are paging algorithms which exploit the important program property of locality by attempting to dynamically allocate various amounts of primary memory space to a program as it executes. Recall that, intuitively, locality means that during a given interval of execution a program addresses only a subset of total addressable space. However, for different intervals, the size of this subset may vary. From this notion of locality comes that of "working sets", and a theory of primary memory based on this notion has been proposed and extensively investigated in [D1,D2,D3]. Therefore, we will extend our definition of the page fetch function, FFp, to include working set memory management.

In order to incorporate the page working set concept into the methodology we adopted in Chapter 2 for presenting paging algorithms, recall the following definitions. Assume that:

$Q = \{A,B,..\}$ is a finite set of logical pages.

$P = p^1, p^2, ..., p^L$ is a page trace with $p^t \in Q$.

$Mp^t \subseteq Q$ is the contents of Mp at time t.

$F = f^1, f^2, ..., f^L$ is the page fetch policy.

$R = r^1, r^2, ..., r^L$ is the page replacement policy.

A paging algorithm based on the page working set principle is defined as follows.

a. $Wp(0,T) = \phi$

b. $Mp^t = Wp(t,T)$ and $|Mp^t| = wp(t,T)$, $0 \leq t \leq L$

c. $f^t = \phi$ if $p^t \in Wp(t-1,T) = Mp^{t-1}$, $1 \leq t \leq L$

d. $f^t = p^t$ if $p^t \notin Wp(t-1,T) = Mp^{t-1}$, $1 \leq t \leq L$

e. $r^t = Wp(t,T) - Wp(t-1,T)$; note that $|r^t| \leq 1$, $1 \leq t \leq L$

Thus, we see that under a page working set strategy, the contents of primary memory at time t, Mp^t, is simply the working set, $Wp(t,T)$, and that the amount of primary memory allocated to a program expands and contracts as the working set size $wp(t,T)$ expands and contracts. A page reference at time t, p^t, causes a page fetch into primary memory if and only if p is not in the working set at time t-1. Note also that a page is removed from primary memory at time t if and only if it is in the working set at time t-1 and it is no longer in the working set at time t.

From the above discussion, we observe that the number of page fetches required by a program during its execution using the page working set memory management technique is uniquely determined from the page trace, P, and the working set parameter, T. Therefore, the definition of the page fetch function, FFp, under page working set memory management can be expressed as a function of the following parameters:

$FFp = FFp(|M_p^t| = wp(t,T), N, \Pi a, STa, Wp(t,T))$.

The parameters in this definition of FFp for working sets are identical to those previously presented for the page fetch function, FFp, except for two instances. The first parameter, which denotes the primary memory size, is equated to wp(t,T) to illustrate that the size of Mp varies with the size of working set. The other instance is strictly notational, i.e., we have replaced the fetch and replacement parameters, F and R, with Wp(t,T) to illustrate that the F and R policies are those defined for working set memory management. We could have used Fw and Rw, but we think that Wp(t,T) is simply clearer.

We can also extend the definition of the sector fetch function, FFs, such that it denotes the number of sector fetches which occur in a virtual memory system during the processing of a sector trace under sector working set memory management.

Consider a program whose behavior is modeled by a sector trace, ST. Then the sector working set at time t, Ws(t,T), is defined to be the distinct set of sectors referenced in the sector trace, ST, during the time interval (t-T,T). The number of sectors in the sector working set at time t is defined to be the sector working set size and is denoted by ws(t,T). The maximum value of the sector working set size for a given sector trace is denoted ws(t,T)max. Note that ws(t,T)max $\leq$ T. Let:

a. Program = {a,b,..}, a finite set of relocatable sectors.

b. ST = $S^1, S^2, \ldots, S^L$, a sector trace with $S^t \in$ Program.

c. $M_s^t \subseteq$ Program, the set of sectors in primary memory

at time t.

d. $F = f^1, f^2, \ldots, f^L$, the sector fetch policy.

e. $R = r^1, r^2, \ldots, r^L$, the sector replacement policy.

Then the sector behavior of a program using sector working set memory management is defined as:

a. $Ws(0,T) = \phi$

b. $Ms^t = Ws(t,T)$ and $|Ms^t| = ws(t,T)$, $0 \leq t \leq L$

c. $f^t = \phi$ if $S^t \in Ws(t-1,T) = Ms^{t-1}$, $1 \leq t \leq L$

d. $f^t = S^t$ if $S^t \notin Ws(t-1,T) = Ms^{t-1}$, $1 \leq t \leq L$.

e. $r^t = Ws(t,T) - Ws(t-1,T)$, $0 \leq t \leq L$.

Thus, the contents of primary memory at time t is the sector working set at time t, $Ws(t,T)$, and a sector reference at time t causes a sector fetch if and only if $S^t \notin Ws(t-1,T)$. Note that the set of sectors that are generated by the sector working set strategy to be in primary memory at time t is $Ws(t,T)$, no matter what the sizes of the individual sectors are.

The sector fetch function, FFs, for the sector working set strategy becomes,

$$FFs = FFs(|Ms^t| = ws(t,T), ST, Ws(t,T)).$$

We observe, as before, that the value of the sector fetch function, FFs, which is the number of sector fetches required to process a sector trace, is uniquely determined by the ST and the $Ws(t,T)$ parameters.

The notion of characterizing the local behavior of a program in terms of its sector working set has two potential applications. The first is to utilize the time varing sector working set to identify the sectors which should be clustered together in order to minimize page faults. This application turns out to be very useful and is discussed in full detail in Chapter 4. The second is to find upper and lower bounds on the paging behavior, FFp, of programs using the page working set strategy in terms of the sector behavior, FFs, using sector working set memory management. This approach proves successful for the upper bounds but fails for the lower bounds. Even though the approach fails to produce lower bounds, Part A of the following theorem points out an interesting relationship that can exist between the number of page fetches and the number of sector fetches for programs using working set memory management.

3.6.1 Lower Bounds for Working Set Management

Recall that ws(t,T)max is defined to be the maximum value of the sector working set size for a given sector trace.

Theorem 7

Given any two-level virtual memory system V, with page size N, primary memory size $|Mp^1| = wp(t,T)$, using paged working set memory management Wp(t,T), and sector trace STa, then for any partition, Πa, of

the relocatable sectors into logical pages of the program where each page contains k or fewer sectors, the minimum number of page fetches, given by $FFp(|Mp^1| = wp(t,T), N, \Pi a, STa, Wp(t,T))$,

A. is not lower bounded by

$$\frac{FFs(|Ms^1| = ws(t,k_1,T), ST = STa, Ws(t,k_1,T))}{k * k_2} \text{ and}$$

B. is not lower bounded by $\frac{FFs(|Ms| = k * T, ST-STa, Fd, R_{LRU})}{k}$ but

C. is lower bounded by $\frac{FFs(|Ms| = k * ws(t,T)max, ST = STa, Fd, Ro)}{k}$,

where the value of the sector fetch function, FFs, is the number of sector fetches which occur in a two-level virtual memory system V', with primary memory size |Ms|, with the same sector trace ST-STa, using sector working set management in Part A, using demand fetch, LRU replacement in Part B and using demand fetch, optimum replacement in Part C. The value of k_1 and k_2 are any arbitrarily large integers greater than 1. (The value of f_1 is as previously defined, and SS is the sector size vector.) The value of ws(t,T)max is the maximum value of ws(t,T) over ST.

Part A of the above theorem states that there are sector traces such that the number of sector fetches required to process the sector trace is arbitrarily larger than the number of page fetches required to

process the corresponding page trace under a good sector ordering. Moreover, it states that this is even true when the window size of the sector working set is made arbitrarily large and the resulting number of sector fetches divided by an arbitrarily large constant. We claim that this is counter-intuitive, because a) if the sector working set window size were simply kT, then the sector working set could contain the same number of sectors as those contained in a page working set of size T; and b) dividing FFs by k alone would account for the fact that as many as k sector fetches are required to bring a page of information into primary memory.

Proof of Part A:

We need to show that there exists a set of parameters such that

$$FFp(|Mp^t| = wp(t,T), N, \Pi, ST, W_p(t,T)) < \frac{FFs(|Ms^t| = ws(t,k_1 T), ST, Ws(t,k_1 T))}{k * k_2}$$

Let:

$T = 2$

k_1, k_2 = any fixed arbitrarily large integers.

$m > k_1 T$

$n > k * k_2$

$k = 2$

Program = (abxy), a set of 4 relocatable sectors

each of size S, where S = N/k.

$ST = ((ax)^m (by)^m)^n$ be the sector trace.

$\Pi = \{(ab), (xy)\}$ where page A = {a,b} and page X = {x,y}.

$P = ((AX)^m (AX)^m)^n = (AX)^{2mn}$ is the page trace.

$Wp(\emptyset,T) = Ws(\emptyset,k_1 T) = \emptyset$.

a. Now, it is clear from the definition of $Wp(t,T)$ and $P = (AX)^{2mn}$ that

$FFp(|Mp^t| = wp(t,T), N, \Pi, ST, W_p(t,T)) = 2$ for all m and n.

b. Now, to evaluate FFs.

$ST = ((ax)^m (by)^m)^n$ implies $FFs(|Ms^t| = ws(t,k_1 T), ST, Ws(t,k_1 T)) = 4n$.

Proof:

Part 1.

Consider the substring reference pattern $(ax)^m$. Observe that the first reference to this substring occurs at times $t = 1+4mi$ for $i = \emptyset,1,\ldots,n-1$. $Ws(\emptyset,k_1 T) = \emptyset$ by definition.

$ws(t,k_1 T) = (b,y)$ for $t = 1+4mi$ $i = 1,2,\ldots,n-1$.

This is true because for each of these times, t, the last 2m references were to b or y. Since $2m > k_1 T$, only b and y can be in $Ws(t,k_1 T)$; and since $k_1 T \geq 2$, both b and y must be in $Ws(t,k_1 T)$.

Hence, for each of the n occurrences of the substring $(ax)^m$ in the sector trace, exactly two sector fetches are required to bring a and x into the working set, where they stay while processing the remaining references in the substring, since $k_1 T \geq 2$.

Part 2.

Consider the substring reference pattern $(by)^m$. The first reference to this substring occurs at times $t = 1+m(4i-2)$ for $i = 1,2,..,n$.

The $Ws(t,k_1 T) = (a,x)$ for $t = 1+m(4i-2)$, $i = 1,2,...,n$, since at each of these times, t, the last 2m references were to a or x. Since $m > k_1 T$, only a and x can be in $Ws(t,k_1 T)$; and since $k_1 T \geq 2$, both a and x must be in $Ws(t,k_1 T)$.

Thus, for each of the n occurrences of the substring $(by)^m$ in the sector trace, two sector fetches are required to bring b and y into $Ws(t,k_1 T)$, and moreover only two are required since $k_1 T \geq 2$.

Therefore, $FFs(|Ms^t| = ws(t,k_1 T),ST,Ws(t,k_1 T)) = 4n$.

Now,

$FFs/(k*k_2) = 4n/(k*k_2) > (4k*k_2)/k*k_2 = 4 > FFp = 2$

and this proves part A of Theorem 7.

What causes this strange behavior in the number of sector fetches? Is it true for only strange and rare sector traces or could it be expected to occur in many common sector traces? We claim that this behavior could occur in many sector traces. In order to provide some insight into this claim, consider the sector trace $ST = \alpha_1\ \alpha_2\ \alpha_3$, where $\alpha_2 = ((ax)^m (by)^m)^n$ and α_1, α_3 represent any long sector reference strings. The proof of part A shows that the ratio

(FFs/FFp) > k_2 for the substring α_2, where k_2 can be made arbitrarily large by choosing n sufficiently large. Therefore, the ratio (FFs/FFp) can still be made arbitrarily large for fixed α_1 and α_3 by simply making n sufficiently large. A generality of this brief argument says that, when a sector trace has any substring consisting of tight embedded loops, the number of sector fetches may become much larger than the corresponding number of page fetches. One explanation of this phenomenon is as follows: tight inner loops (i.e., $(bx)^m$)) drown out the benefit gained by making the sector window size large (i.e., the value of Ws(t,T) becomes (bx) if m > T), while the outer loop causes the sectors in the inner loops to be fetched over and over. In contrast, the paged working set having a small window size, relative to m, is able to contain all the sectors in the embedded loops (i.e., (ax), (by)) throughout consecutive cycles of the outer loop, if at least one sector from each inner loop is grouped into the same page.

From the above discussion, we observe that the page working set can contain more of the most recently referenced sectors than the sector working set, even when the latter has an arbitrarily large window size. We can eliminate this condition by redefining the sector working set as follows. Recall that the sector working set, Ws(t,T), has been defined to contain the set of distinct sectors referenced in the last T references. If we modified the definition of Ws(t,T) such that it contains the set of T most recently referenced sectors, and if we choose T to be k times the page working set window size, then the page working

set could never contain more of the most recently referenced sectors than those contained by this sector working set. However, this new definition of the sector working set is equivalent to demand fetch, LRU replacement in a memory of fixed size equal to k times the page working set window size. Thus, a plausible conjecture is that the number of page fetches under a page working set strategy could be lower bounded by the number of sector fetches under demand fetch, LRU replacement in a memory size as described above. However, Part B of Theorem 7 states that this conjecture is not true.

Proof of Part B.

We have to show that there exists a set of parameters such that $FFp(|Mp^t| = wp(t,T), N, \Pi a, STa, Wp(t,T)) < \frac{FFs(|Ms| = k*T, ST = STa, Fd, R_{LRU})}{k}$.

Let:

Program = {a,b,c,d,e,f,g,h}, a set of
8 relocatable sectors of size N/2.

k = 2

N = twice the sector size.

T = 3.

ST = (acd bef bgh acd aef b) be the sector trace.

|ST| = 16.

Πa = {{a,b}, {c,d}, {e,f}, {g,h}}, where page
A = {a,b}, page B = {c,d}, etc.

P = (A BB ACC ADD ABB ACC A) be the resulting page trace.

$Wp(0,T) = M_s^0 = 0.$

$|Ms| = k*T = 6.$

Simulation of paging behavior to get FFp gives:

```
P        = ABB ACC ADD ABB ACC A
Fw       = AB0 0C0 0D0 0B0 0C0 0
Wp(t,T)  = ABB ACC ADD ABB ACC A
             AA BAA CAA DAA BAA C
                 B   C   D   B
                 ↑contents of Wp(t,T) immediately before 6th
                  reference.
```

Thus, $FFp = \Sigma_{i=1}^{16} |f_w^t| = 6$ page fetches.

Simulation of sector behavior gives;

```
ST = acd bef bgh acd aef b
F  = acd bef 0gh acd 0ef b
M  = acd bef bgh acd aef b
       ac dbe fbg hac dae f
        a cdb efb gha cda e
          acd def bgh hcd a
           ac cde fbg ghc d
            a acd efb bgh c
```

Results:

$FFs = \Sigma_{i=1}^{16} |f_d^i| = 14$ sector fetches.

$FFp = 6 < FFs/k = 14/2 = 7$, QED.

However, if we change the LRU replacement algorithm of Part B to the optimum replacement algorithm, then the value of FFp under page working set management can be lower bounded. This lower bound is given by Part C of Theorem 7.

Proof of Part C.

Note that $|Mp^t| = wp(t,T) \leq ws(t,T)max \leq T$.

a.

$FFp(|M_p^t| = wp(t,T), N, \Pi a, ST_a, Wp(t,T))$

$\geq FFp'(|Mp'| = w_s(t,T)_{max}, N, \Pi a, STa, Fd, R_{LRU})$,

since $Mp^t = Wp(t,T) \subseteq Mp^{"t}$, by definition of $Wp(t,T)$ and the definition of $Mp^{"t}$ under demand fetch, LRU replacement; that is, $Mp^{"t}$ always contains the set consisting of the $|Mp'| = ws(t,T)max$ most recently referenced pages, while Mp^t contains the set consisting of the $wp(t,T)$ most recently referenced pages.

b.

$FFp'(|Mp^{"t}| = w_s(t,T)_{max}, N, \Pi a, ST_a, F_d, R_{LRU})$

$$\geq \frac{FFs(|M_s| = k * w_s(t,T)_{max}, ST = ST_a, F_d, R_0)}{k}$$

by Theorem 1, and this proves part C of Theorem 7

Corollary to Theorem 7, Part C.

$FF_p(|M_p^t| = w_p(t,T), N, \Pi a, ST_a, W_p(t,T))$

$$\geq \frac{FF_s(|M_s| = f_1(w_s(t,T)_{max}, N, SS), ST = ST_a', F_d, R_a)}{f_1(2,N,SS)/2} - \Delta,$$

where Δ and f_1 are defined as in Theorem 5.

Proof.

We know from the proof of Part C that

$FF_p(|Mp^t| = w_p(t,T), N, \Pi a, ST_a, Wp(t,T))$

$\geq FF_p'(|Mp'| = w_s(t,T)_{max}, N, \Pi a, ST_a, Fd, R_{LRU})$,

and applying Theorem 5 to FFp' proves the corollary immediately.

3.6.2 Upper Bounds for Working Set Management

An upper bound on the number of page fetches for virtual memory systems using the page working set strategy is given in Theorem 8.

Theorem 8

Given any two-level virtual memory system V, with page size N, primary memory size $|Mp^t| = wp(t,T)$, using page working set memory management Wp(t,T), and any sector trace STa, then for any partition, Πa, of the relocatable sectors into logical pages, where each page contains k or fewer sectors, the maximum number of page fetches given by the page fetch function, FFp, is upper bounded by

$FFp(|Mp^t| = w_p(t,T), N, \Pi a, STa, Wp(t,T)) \leq FFs(|Ms^t| = w_s(t,T), ST, Ws(t,T))$,

where the value of the sector fetch function FFs is the number of sector fetches which occur in a two-level virtual memory system V', with primary memory size $|Ms^t| = ws(t,T)$, the same sector trace ST = STa, using sector working set management Ws(t,T).

Proof:

Let:

$Q = \{S_1, S_2, \ldots, Sm\}$ = {set of relocatable sectors of the program}.

$\Pi a = \{\Pi_1, \Pi_2, \ldots, \Pi n\}$ be any partition of Q such that $|\Pi j| \leq k$,

$1 \leq j \leq n$.

$ST = x^1, x^2, \ldots, x^L$ be any sector trace, where $x^t \in Q$, $1 \leq t \leq L$.

$P = p^1, p^2, \ldots, p^L$ be the page trace, where $p^t = j$ if $x^t \in \Pi j$.

$Mp^t = Wp(t,T)$ be the set of pages in memory of FFp at time t.

$Ms^t = Ws(t,T)$ be the set of sectors in memory of FFs at time t.

$Fp = f_p^1, f_p^2, \ldots, f_p^L$ = demand fetch policy of FFp.

$Rp = r_p^1, r_p^2, \ldots, r_p^L$ = working set replacement policy of FFp.

$Fs = f_s^1, f_s^2, \ldots, f_s^L$ = demand fetch policy of FFs.

$Rs = r_s^1, r_s^2, \ldots, r_s^L$ = working set replacement policy of FFs.

Suppose at time t, in the FFp model, that $p^t = j$, the page j containing the set of Πj sectors, is referenced. Then at time t, in the FFs model, $x^t = a$ is the sector referenced, where $a \in \Pi j$. We need to show that $\Sigma_{i=1}^{L} |f_p^t| \leq \Sigma_{i=1}^{L} |f_s^t|$.

Case 1. Suppose $p^t \in Mp^{t-1} = Wp(t-1,T)$; then $f_p^t = \phi$.

a. If $x^t \in Ms^{t-1} = Ws(t-1,T)$, then $f_s^t = \emptyset$ and $|f_p^t| = |f_s^t|$.

b. If $x^t \notin Ms^{t-1} = Ws(t-1,T)$, then $f_s^t = |a| \notin Ws(t-1,T)$ and $|f_p^t| < |f_s^t|$.

Case 2. Suppose $p^t \notin Mp^{t-1} = Wp(t-1,T)$; then $f_p^t = |j|$.

a. If $x^t \notin Ms^{t-1} = Ws(t-1,T)$, then $f_s^t = |a| \notin Ws(t-1,T)$ and $|f_p^t| = |f_s^t|$.

b. If $x^t \in Ms^{t-1} = Ws(t-1,T)$, then $f_s^t = \emptyset$ and $|f_p^t| > |f_s^t|$. This condition illustrates the only way that page fetches can exceed sector fetches. However, if we show that $p^t \notin Wp(t-1,T) \Rightarrow x^t \notin Ws(t-1,T)$, then case 2b can never occur. Let $p^t \notin Wp(t-1,T)$, and assume $x^t \in Ws(t-1,T)$. Since $x^t \in Ws(t-1,T)$, there exists a time t' in the interval (t-1-T,t-1) such that $x^t = x^{t'}$. Let $p^{t'} = k$ be the page referenced at time t' in the page trace P. We know that $x^{t'} \in \Pi k$, since sectors are not allowed to cross page boundaries. We also know that $p^{t'} \in Wp(t-1,T)$ because the window size is T for both the page working set Wp and the sector working set Ws. But this contradicts the assumption; therefore $x^t \notin Ws(t-1,T)$.

Hence, $\Sigma_{i=1}^{L} |f_p^t| \leq \Sigma_{i=1}^{L} |f_s^t|$ and the theorem is proved.

CHAPTER 4

INTERSECTOR REFERENCE MODELS

4.1 Introduction

In the previous chapter, we presented upper and lower bounds on the number of page fetches which could occur in a virtual memory system, for a given program reference behavior, over any restructuring of the relocatable sectors into logical pages of the program. The next phase is to develop and present practical techniques for restructuring a program to achieve good locality of reference for the program in virtual memory systems. The task of program reorganization for virtual memory systems will be separated into two logical parts. The first part is to develop automatic techniques for identifying the dynamic intersector reference behavior of programs executing in virtual memory systems. The second part is to provide clustering procedures which utilize the intersector reference behavior to rearrange the relocatable sectors of a program into its logical pages such that good locality of reference exists in the page trace of the restructured program. The basic idea of the second part is to assign the most strongly related sectors to common pages.

In this chapter, we address the problem of intersector reference models. In the next chapter, automatic clustering procedures are presented, and finally, in Chapter 6, the results of applying these

methods to real programs are investigated and compared with the theoretical bounds.

4.2 Intersector Reference Models

It is known that a program's page reference patterns have a strong effect on paging performance in virtual memory systems. It is also known [H1] that the sector reference behavior of many common programs, such as compilers, assemblers, editors, etc., proves to be remarkably insensitive to the input data in rather large domains. For example, the studies of Hatfield and Gerald [H1] revealed that the groups of sectors which were used frequently together in the assembly of one program turned out to be essentially the same as the groups of sectors which were used frequently together in the assembly of another program. The basic difference between assemblies was that the groups of sectors which were used together for short input programs were simply used together more often for long input programs. Supported by these empirical observations of Hatfield and Gerald, we decided to characterize the reference behavior of a program by its sector trace and to base our practical restructuring methods on this reference behavior. We will elaborate on the soundness of this decision in Chapter 6 when we compare the paging performance of real programs over program structures derived from different sector traces.

Another important reason for basing restructuring methods on a sector trace is that the results of the last chapter may be used to compare the paging behavior of a restructured program with the theoretical best and worst paging behavior for that sector trace.

Given a sector trace, our objective is to specify the strength of the intersector references such that a clustering procedure that groups the strongly connected sectors together into logical pages produces a program structure that tends to minimize the number of page fetches. We begin by presenting Hatfield and Gerald's [HG] intersector reference model for defining the strength of connection between sectors.

4.2.1 The HG Intersector Reference Model

The HG intersector reference model consists of a symmetric matrix, H, showing the strength of connection between the sectors of the program to be reorganized. Let:

$Q = \{S_1, S_2, \ldots, Sm\}$ be the program of m relocatable sectors;

$ST = S^1, S^2, \ldots, S^L$ be a sector trace of the program.

Then

$H = [Hij]$ for $i, j = 1,2,\ldots,m$, where $Hij = \Sigma_{t=1}^{L} k(i,j,t)$,

where k(i,j,t) = 1 if S^t=i and S^{t+1} = j, or S^t =j and S^{t+1} =i;
and k(i,j,t) = 0 otherwise.

Thus, the value of Hij is simply the number of times that sector i referenced sector j plus the number of times that sector j referenced sector i in the sector trace.

Using this intersector reference model, Hatfield and Gerald were able to find improvements in the number of page fetches on the order of two-to-one to ten-to-one by clustering sectors with large Hij values into the same page. This is the same as clustering sectors into pages such that the value of Hij is small for i and j in different pages.

Even though these results are quite impressive, the values of Hij in the HG intersector reference model do not contain any information about the length of the time interval between successive references of sector i to sector j. Hence, the strength of connection, Hij, between sector i and j is the same for large time intervals and short time intervals. However, paging may depend quite heavily on the length of these time intervals. For example, assume that sector i references sector j 100 times (Hij = 100) in a sector trace of 200,000 references. Now let's consider two different plausible examples of how these references could occur. First, these references could occur with short time intervals between them such that all 100 references occur within 500 successive references of the sector trace. Second, these references could occur with some long time intervals between them such that 10 of these references could be found in each 20,000 successive references of the

sector trace. Even though the strength of connection is the same for these two examples, the tendency for a reference from sector i to sector j to cause a page fetch when they are not in the same page can be considerably larger in the second example.

Furthermore, the tendency of a reference from sector i to sector j to cause a page fetch is related to such local information as the time elapsed since the last reference to sector j and the number of distinct sectors referenced since the last reference to sector j in the sector trace. If the time is short since sector j was last referenced, and little virtual memory space was used during that time, it is probable that sector j is still in primary memory and a new reference will not cause a page fetch. If the time and space traversed between references to j is large, it is likely that a page fetch will occur unless j is grouped into the same page as the referencing sector or some recently referenced sector. We will now present two intersector reference models which have potential for identifying and incorporating local sector reference behavior into the strength of connection between sectors.

4.2.2 Working Set Intersector Reference Models

The sector working set, Ws(t,T), will be used to define the strength of connection between sectors for a given sector trace.

Let:

$Q = (S_1, S_2, \ldots S_m)$ be a program of m-relocatable sectors.

$ST = S^1, S^2, \ldots S^L$ be a sector trace of the program where $S^t \in Q$.

$P = P^1, P^2, \ldots P^L$ be the resulting page trace of the program where P^t is the page referenced at time t.

If $S^t = Sj$ is the sector referenced at time t, then we define $P^t = Psj$ to denote the page referenced at time t. Psj is to be interpreted as the page containing sector j. We have adopted this notation to make the following discussion easier to understand.

Recall that the sector working set, Ws(t,T), is defined to be the set of distinct sectors referenced in the time interval t-T to t of the sector trace. Similarly, the page working set, Wp(t,T), is the set of distinct pages referenced in the time interval t-T to t of the page trace.

FACT 1.

Let $S^t = Sj \in$ and let $Sj \notin Ws(t-1,T)$. Then $P^t = Psj \in Wp(t-1,T)$ iff $Sj \in Psi$ for some $Si \in Ws(T-1,T)$.

The proof of Fact 1 follows immediately from the definition of Wp(t-1,T), which is the set of distinct pages in the sequence $Ps^{t-1-T}, Ps^{t-T}, \ldots, Ps^{t-1}$, and the definition of Ws(t-1,T), which is the set of distinct sectors in the sequence

$S^{t-1-T}, S^{t-T}, \ldots, S^{t-1}$.

Fact 1 states that, when sector j is referenced at time t and sector j is not in the sector working set, then the page referenced at time t will be in the page working set if sector j is grouped into a page with any one of the sectors in the sector working set. Furthermore, it states that the page referenced at time t will not be in the page working set if sector j is not grouped into a page with one of the sectors in the sector working set.

FACT 2.

Let $S^t = Sj$ and let $Sj \in Ws(t-1,T)$. Then $P^t = Psj \in Wp(t-1,T)$. Fact 2 also follows immediately from the definition of $Ws(t,T)$ and $Wp(t,T)$.

Fact 2 states that, when sector j is referenced at time t and sector j is in the sector working set, then the page referenced at time t will be in the page working set.

FACT 3.

We want the entry $Wij + Wji$ in the intersector reference model to be the number of page fetches which will go away if sector i and sector j are grouped into the same page.

Using the above three facts as a basis, we present the procedure for constructing the intersector reference model, $W = [Wij]$, for $i,j = 1,2,\ldots,m$. At each instant of time, t, for $1 \leq t \leq L$, do the following.

Step 1. If $S^t = Sj$ and $Sj \notin Ws(t-1,T)$, then increment Wij by 1 for all $Si \in Ws(t-1,T)$.

Step 2. If $S^t = Sj$ and $Sj \notin Ws(t-1,T)$, then increment Wjj by 1.

Step 3. If $S^t = Sj$ and $Sj \in Ws(t-1,T)$, then no increment is required.

Simply stated, the above procedure works as follows. If sector j is not in the sector working set when it is referenced, then increment its connectivity strength with all the sectors in the sector working set. Moreover, if sector j is in the sector working set when it is referenced, then do not change the strength of connection between sector j and the other sectors.

We observe that the value of the intersector strength becomes

$Wij = \Sigma_{l=1}^{L} k(i,j,t)$,

where $k(i,j,t) =$ 1 if $S^t = Sj \notin Ws(t-1,T)$ and $Si \in Ws(t-1,T)$,
1 if $S^t = Sj \notin Ws(t-1,T)$ and $i = j$,
0 otherwise.

Note that Wij + Wji is the number of page fetches which will go away if sectors i and j are grouped together in the same page. The sum of the diagonal elements of the intersector reference model, $\Sigma_{j=1}^{m} Wjj$, is

the number of sector fetches which occurred for the sector working set. This will also be the number of page fetches for the page working set if no sectors are combined together in pages. The number of page fetches after combining only sectors i and j will be $\Sigma_{j=1}^{m} Wjj - Wij - Wji$.

FACT 4.

If exactly two sectors are grouped into each of the n logical pages, then the number of times a page is referenced and not found in the page working set is given by

$$\Sigma_{j=1}^{m} Wjj - \sum_{\substack{i,\, j \in Pk \\ i \neq j}} Wij + Wji \quad \text{for } 1 \leq k \leq n.$$

Fact 4 follows directly from the construction of Wij, since Wij + Wji is the number of page fetches which are eliminated by grouping i and j together in the same page, and since grouping i and j together does not affect the value of $W_{kl} + W_{lk}$ for grouping any other two sectors k and l together in a different page.

Unfortunately, we cannot extend Fact 4 to handle the case when more than two sectors are allowed to be grouped into a page. This occurs because the matrix, W, does not contain enough information to determine the number of page fetches which will be eliminated by grouping three or more sectors into a page. For example, Wjj is the number of fetches of sector j. Wij and Wkj are the number of times that sector i and sector k, respectively, were in the working set when a fetch of j was made. The

problem is that sectors i and k both may have been in the sector working set at the time that a reference to j caused a fetch. Let the number of sector fetches of sector j, which will be resolved by grouping sectors i, k, and j together into a page, be denoted by Rikj.

Then,

$MAX[Wij, Wkj] \leq Rikj \leq Wij + Wkj.$

We should point out at this time that the above relations can be utilized in a clustering procedure. Suppose sectors i, j, and k are grouped together into a page. Then the unresolved sector fetches of i, j, and k, denoted by U'ijk, is the number of page fetches of this page which will occur if no other sector is grouped with i, j, and k.

But

$U'ijk \leq Wii + Wjj + Wkk - MIN[Rikj] - MIN[Rijk] - MIN[Rjki].$

Note, also, from Fact 4, that

$U'ij = Wii + Wjj - Wij - Wji$, for the case of two sectors in a page.

Therefore, a clustering procedure could dynamically determine a lower bound on the number of page fetches which could be resolved by adding another sector to a page.

Since the value of Wij depends on the window size T of the sector working set Ws(t,T), we need to elaborate on how one selects a "good value" for T.

For real programs, we measured the improvement in paging performance for restructured programs as a function of T. That is, we computed the intersector reference model W for various values of T, and for each W we restructured the program and computed its paging performance. The detailed results of these experiments are presented in Chapter 6. However, the significant characteristics of these results are as follows. For a given program, the best improvements in paging performance, as a function of T, occur for a rather large bandwidth of T values. For example, values of $1000 \leq T \leq 5000$ produced essentially the same and the best improvement in paging performance of certain programs. For all programs tested, the bandwidth of T values that resulted in the best improvement in paging performance was several thousand instructions; however, the location of this bandwidth of T values in the set of all T values varied from program to program. A serendipitous observation of the correlation between the bandwidth of good T values and the "knee" of the parachor curve of the sector fetch function, FFs(|Ms|,ST,Fd,Ro), produced an interesting empirical result.

The parachor curve is a graph of FFs(|Ms|,ST,Fd,Ro) versus the amount of primary memory |Ms| available for execution. A typical parachor curve for FFs is shown in Figure 5. The value of FFs is a monotonically decreasing function of |Ms|. For most observed programs, there is a threshold region at which,

a) if the amount of primary memory is decreased further, the number of sector fetches increases very rapidly, and,

b) if the amount of primary memory is increased further, the number of sector fetches decreases very slowly.

This threshold region is depicted in Figure 5 and is called the knee of the parachor curve. The values of |Ms| in the knee of the parachor indicate how many sectors are required to be in the primary memory to maintain a "reasonable" level of performance.

Let the average sector working set size be denoted by $w_s(T)$ and be defined as,

$w_s(T) = (1/L) \cdot \Sigma_{t=1}^{L} ws(t,T)$

Now we present a method which identifies values of the window size T for use in the construction of the intersector reference model W.

Experimental Result:

For all the programs we tested, the bandwidth of T values which resulted in the best improvement in paging performance corresponds to those values of T for which the average sector working set size $w_s(T)$ was equal to

a) some value of |Ms| in the knee of the parachor curve of FFs(|Ms|,ST,Fd,Ro), or to

b) some value of |Ms| slightly smaller than those values of |Ms| found in the knee of the parachor curve.

This experimental result was particularly handy in our research, since we had already computed the parachor curve of FFs(|Ms|,ST,Fd,Ro)

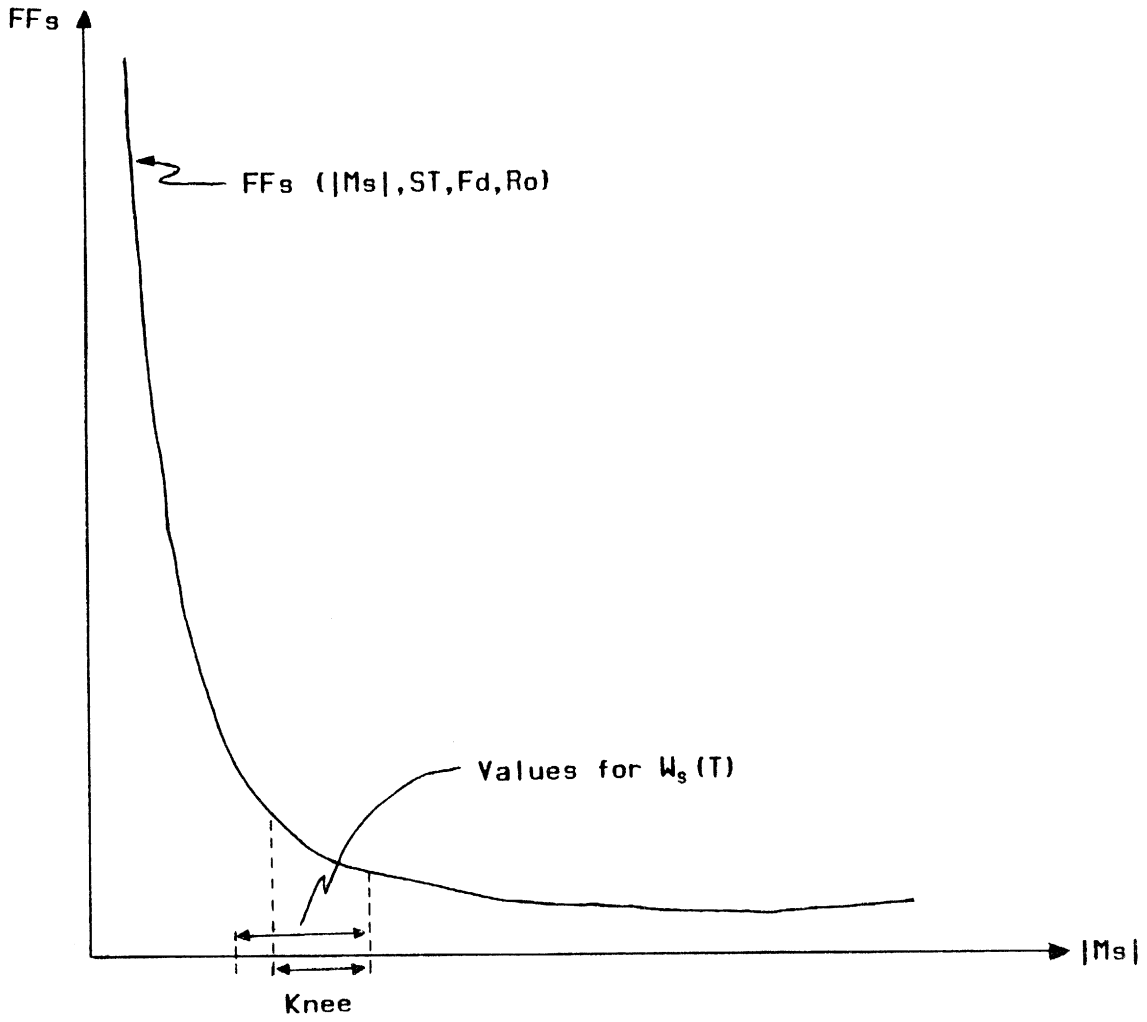


FIGURE 5.

Parachor Curve of FFs (|Ms|,ST,Fd,Ro)

for use in establishing the lower bounds.

If the window size, T, is very small, for example T=1, then the value of Wij is much larger than the number of page fetches resolved by grouping i and j together for most memory sizes. On the other hand, if the value of T is very large, for example 25,000, then the value of Wij is much smaller than the number of page fetches resolved by grouping i and j together for most memory sizes. However, if T is such that the average working set size is in the knee of the parachor curve, then the value of wij represents the intersector activity when the program has just enough space to execute efficiently. This corresponds to the intersector activity that we want to represent in the intersector reference model, W.

In addition to the above intersector reference model based on the sector working set, we decided to investigate the potential of the following model. Let the intersector reference, W', be a m x m matrix defined as follows:

$W'ij = \Sigma_{t=1}^{L} k(i,j,t)$ for $i,j = 1,2,\ldots,m$,

where $k(i,j,t) = 1$ if $S = Sj \in Ws(t-1,T)$ and $Si \in Ws(t-1,T)$;

0 otherwise.

The value of W'ij is the number of times that sector j was referenced when sector j and sector i were both in the sector working set. Therefore, if the value of W'ij is large, then Si and Sj were in the sector working set together many times. Note that W'jj is the number

of references to sector j which will not cause a page fetch. In contrast, Wjj of the previous model is the number of references to sector j which will cause a page fetch unless Sj is grouped with some Si. However, W'ij does measure the tendency for sectors i and j to be found in the sector working set together. Clustering procedures which group sectors into pages with large W'ij values will tend to reduce the size of the page working set and hence increase the locality of the restructured program.

We conclude with a few comments about the intersector reference models based on the sector working set, Ws(t,T). The HG intersector reference model, H, is a special case of the intersector reference model, W. They are the same when W is computed from a sector working set with window size, T, equal to one. The notion of using sector working sets to define the strength of connection between blocks has been investigated concurrently but independently of this work by Masuda [M6] and Ferrari [F1]. Masuda's use of block working sets is quite different from this work, while Ferrari's is similar in some aspects.

4.2.3 LRU Stack Intersector Reference Model

The "LRU sector stack" will be used to define the strength of connection between sectors for a given sector trace.

Consider demand fetch, LRU replacement on a sector trace, $ST = S^1, S^2, \ldots, S^t, \ldots, S^L$, over a set of m-relocatable sectors. From Chapter 2, we know that LRU satisfies the inclusion property, i.e., $Ms^t(1) \subseteq Ms^t(2) \subseteq \ldots \subseteq Ms^t(m^t) = Ms^t(m^t+1) = Ms^t(m^t+2) = \ldots$ where $Ms^t(j)$ is the contents of the sector memory Ms at time t when the size of Ms is j sector frames (i.e., $|Ms^t| = j$), and m^t is the number of distinct sectors referenced in the sequence $S^1, S^2, \ldots, S^t$.

Because of the inclusion property, the primary memory contents Ms^t at any time t and for all capacities can be represented in the following terse and useful way. We order the distinct set of sectors in the sequence $S^1, S^2, \ldots, S^t$ into a list called the LRU sector stack which is defined as $SS^t = SS^t(1), SS^t(2), \ldots, SS^t(m^t)$ where $SS^t(i) = Ms^t(i) - Ms^t(i-1)$. Note that

$$Ms^t(i) = \begin{cases} \{SS^t(1), SS^t(2), \ldots SS^t(i)\} & \text{for } i \leq m^t; \\ \{SS^t(1), SS^t(2), \ldots SS^t(m^t)\} & \text{for } i > m^t. \end{cases}$$

The LRU sector stack has no entries at time $t = 0$. The top of the stack is defined as $SS^t(1)$, while the bottom of the stack is defined as $SS^t(m^t)$.

The LRU sector stack, just after sector reference S^t at time t, is simply the list of the set of m^t sectors of the program ordered according to recency of usage; i.e., $SS^t(k)$ is the kth most recently used sector relative to S^t.

The position of sector j in the stack just before sector reference S^t, at time t, is defined as the sector stack distance and is denoted by $\Delta^t j$. Furthermore, $\Delta^t j = \infty$ if Sj has not been referenced. Thus,

$$\Delta^t j = \begin{cases} k \text{ if } SS^t(k) = Sj. \quad 1 \leq k \leq m^t \\ \infty \text{ otherwise} \end{cases}$$

From the definition of stack distances, we observe that $S^t = Sj$ will cause a sector fetch under demand fetch, LRU replacement unless $\Delta^t j \leq |Ms|$ where |Ms| is the number of sector frames in the sectored primary memory.

Now, two facts are presented which relate the sector stack distances at time t with the parameters of a paged virtual memory system using demand fetch and LRU replacement on the page trace $P = p^1, p^2, \ldots, p^t, \ldots, p^L$ in a primary memory of |Mp| page frames. The page, p^t, referenced at time t must contain the sector S^t, referenced at time t.

FACT 1.

Let $S^t = Sj$, and let $\Delta^t j > |Mp|$. Then $p^t \in Mp^t$ if Sj is grouped into the same page with some Si where $\Delta^t i \leq |Mp|$.

Proof.

Note that $\Delta^t j > |Mp|$ states that the sector stack distance at time t to sector j is greater than the number of page frames in Mp. Suppose Sj is grouped with some Si, where $\Delta^t i \leq |Mp|$. Then the sector,

Si, is among the |Mp| most recently referenced sectors. Therefore, the page containing Si must be among the |Mp| most recently referenced pages, since we are assuming that the sectors are smaller than pages.

FACT 2.

Let $S^t = Sj$, and let $\Delta^t j \leq |Mp|$. Then $p^t \in Mp^t$.

Fact 2 follows from the argument applied to Si in Fact 1. We can use Facts 1 and 2 as a basis for defining the strength of connection between sectors. Fact 2 states that, if $S^t = Sj$ and $\Delta^t j \leq |Mp|$, then Sj will not cause a page fetch; hence, for such references, the strength of connection between Sj and the other sectors need not be incremented. However, if $\Delta^t j > |Mp|$, then Sj will not cause a page fetch when it is grouped with any sector Si with $\Delta^t i \leq |Mp|$. For the latter case, the strength of connection between Sj and all Si with $\Delta^t i \leq |Mp|$ will be incremented by 1.

Now, we define the intersector reference model based on the LRU sector stack distance as a m x m matrix, U, where

$$U_{ij} = \sum_{t=1}^{L} V(i,j,t) \text{ and}$$

$$V(i,j,t) = \begin{cases} 1 \text{ if } S^t = Sj \text{ and } \Delta^t j > 0 \text{ and } \Delta^t i \leq 0; \\ 1 \text{ if } S^t = Sj \text{ and } \Delta^t j > 0 \text{ and } i = j; \\ 0 \text{ otherwise.} \end{cases}$$

If the value of D is one, then the intersector reference model, U, is the same as the intersector reference model of Hatfield and Gerald. However, we got the best results (fewest page fetches after restructuring) with values of D equal to the number of sectors, |Ms|, corresponding to the high side of the knee of the parachor curve FFs(|Ms|,ST,Fd,Ro). Figure 6 shows the typical shape of FFs as a function of |Ms| and the range of the values of D which gave excellent results for all real programs we investigated.

One explanation which provides some insight into why the values of D corresponding to the knee region of FFs produce reasonable values for the strengths of connection between sectors is as follows.

If D is very small, say 1, then the strength of connection between two sectors, Uij, is proportional to the number of page fetches only when the paged primary memory has one page frame. However, most large programs will not execute efficiently when allocated one page frame. If the value of |Mp| for efficient execution is much larger than D=1, then the strength of connection Uij for some i and j may not even be loosely proportional to the number of page fetches resolved when they are grouped together. For very small values of D, Uij may be excessively larger than the number of page fetches which are resolved by grouping i and j together; for very large values of D, Uij may be excessively smaller than the number of page fetches resolved when i and j are placed together. Values of D in the region of the knee of the curve represent

the intersector activity when the program has just enough space to execute efficiently. This is the intersector activity that we want the intersector reference model to measure.

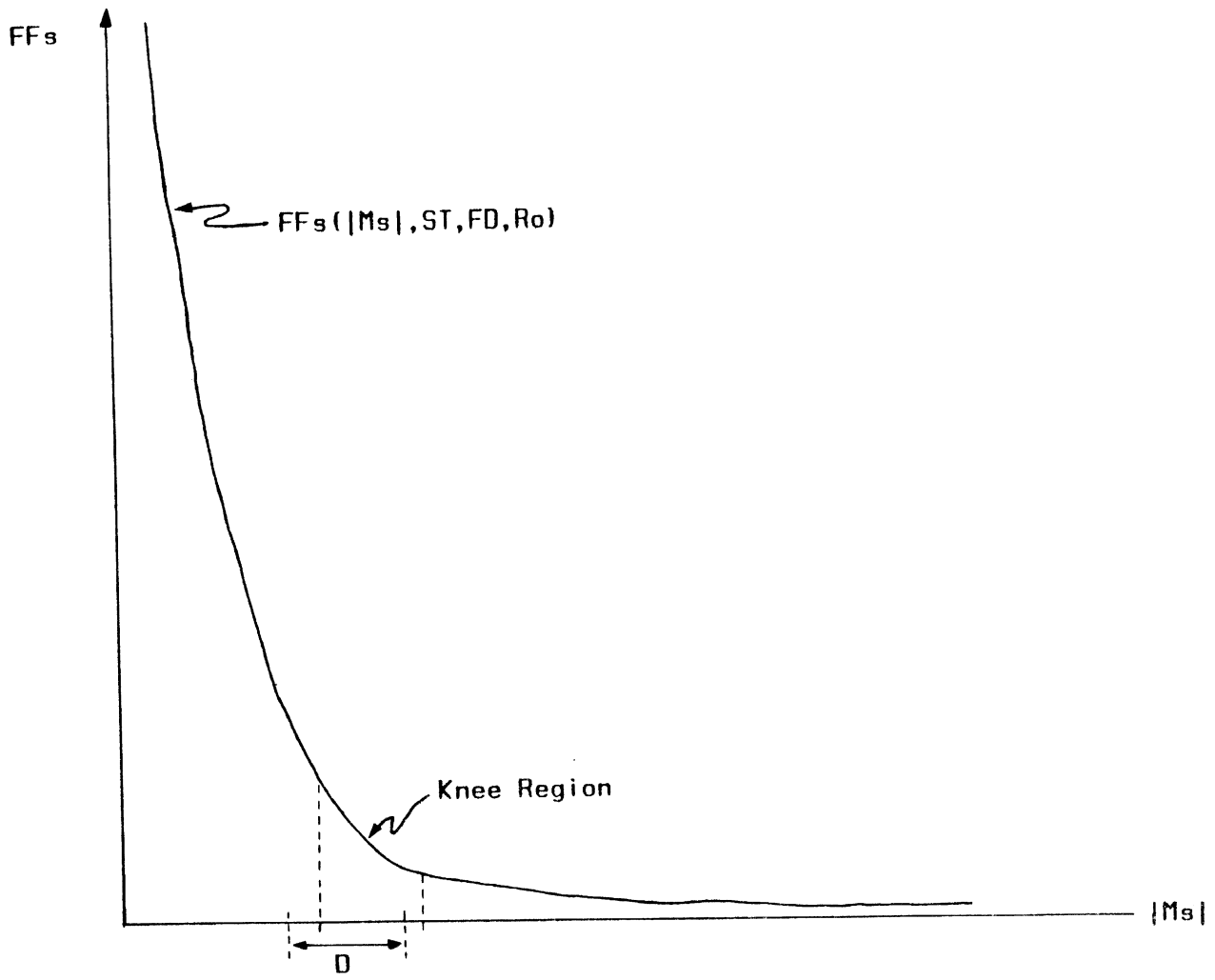


FIGURE 6.

Parachor Curve Illustrating Values For D

This empty page was substituted for a blank page in the original document.

CHAPTER 5

CLUSTERING PROCEDURES

5.1 Introduction

The purpose of this chapter is to present the automatic clustering methods which were used in conjunction with the intersector reference models to restructure programs. The experimental results which show the effect of these clustering techniques on the paging performance of restructured programs are presented in Chapter 6.

5.2 Clustering Procedures

The clustering methods presented in this chapter may be applied to any of the intersector reference matrix models of Chapter 4. Hence, we will denote any of these intersector reference models with the generic C = [Cij]. In those cases where a particular intersector reference model is needed, the notation of Chapter 4 will be used.

We know of no efficient procedure to produce and prove the optimal partition of sectors into pages to maximize the sum of the intersector

connections Cij within all pages. Several clustering procedures based on heuristic approaches are presented in this chapter which have the following significant properties. First, they are completely automatic; that is, these procedures are not based on manual or "eyeball" reorderings. Second, all these procedures produced restructured programs which showed substantial improvements in their paging performance. Third, these clustering procedures are quite fast.

The technique of the following clustering procedures is to take an intersector reference model of intersector bond strengths and cluster relocatable sectors into pages such that the sum of the sector bonds within pages tends to be maximized.

5.3 Nearest Neighbor Methods

In this section, we present several hierarchical methods which cluster the nearest two clusters under a specified bond strength definition one after another.

Given any two clusters of relocatable sectors, Gx and Gy, the intercluster bond is denoted by B(x,y). Several intercluster bond definitions are given below; then a clustering procedure is defined over the intercluster bonds.

In the following definitions, the intersector reference matrix, C = [Cij], is assumed to be symmetric. If the intersector reference matrix is not symmetric, then each occurrence of Cij should be replaced with (Cij + Cji)/2. The notation |Gx| denotes the size of cluster Gx in bytes, and N denotes the page size in bytes.

A. Constrained Nearest Neighbor Bond

The Constrained Nearest Neighbor bond, CNN, between any two clusters Gx and Gy is defined as

$$B(x,y) = \text{Max } \{Cij : i \epsilon Gx, j \epsilon Gy\} \text{ when } |Gx| + |Gy| \leq N.$$

undefined when $|Gx| + |Gy| > N$.

B. Constrained Farthest Neighbor bond

The Constrained Farthest Neighbor Bond, CFN, between any two clusters, Gx and Gy, is defined as

$$B(x,y) = \min \{Cij: i \epsilon Gx, j \epsilon Gy\} \text{ when } |Gx| + |Gy| \leq N;$$

undefined when $|Gx| + |Gy| > N$.

C. Constrained Average Neighbor Bond

The Constrained Average Neighbor bond, CAN, between any two clusters, Gx and Gy, is defined as

$$B(x,y) = (1/n_{xy}) \Sigma_{i \in Gx} \Sigma_{j \in Gy} \; Cij \text{ when } |Gx| + |Gy| \leq N;$$

undefined when $|Gx| + |Gy| > N$.

Here n_{xy} is the number of $Cij > 0$ with $i \in Gx$, $j \in Gy$. Note that n_{xy} is the number of arcs between Gx and Gy, and it is not the sum of the values on these arcs.

D. Constrained Average Neighbor Weighted Bond

The Constrained Average Neighbor Weighted bond, CANW, between any two clusters, Gx and Gy, is defined as

$$B(x,y) = n_{xy} * (1/n_{xy}) \Sigma_{i \in Gx} \Sigma_{j \in Gy} \; Cij \text{ when } |Gx| + |Gy| \leq N;$$

undefined when $|Gx|+|Gy| > N$.

Hence,

$$B(x,y) = \Sigma_{i \in Gx} \Sigma_{j \in Gy} \; Cij \text{ when } |Gx| + |Gy| \leq N.$$

A clustering procedure is now defined for use with any one of the above definitions of B(x,y).

First, choose any one of the above definitions of B(x,y). Second, partition the m relocatable sectors of a program into exactly m clusters, where each cluster contains one sector. Then, at each step in the clustering process, the nearest two clusters are combined to form a new cluster. The nearest two clusters are defined to be the two

clusters Gx and Gy which have the largest value of B(x,y). When the sum of the size of the two clusters becomes larger than the page size in the clustering process, these two clusters are not considered to be connected; that is, their bond strength is undefined. The process comes to an end when new clusters cease to appear.

When the above clustering procedure is applied to the Constrained Nearest Neighbor bond definition of B(x,y), it will be referred to as the CNN procedure; when applied to the CAN definition of B(x,y), it will be referred to as the CAN procedure, etc.

All of these clustering methods are computationally fast, easy to implement, and they tend to group the sectors with the strongest intersector strengths, Cij, into the same page. Hence, they tend to minimize the interaction of sectors clustered into different pages.

The CNN, CFN, and CAN procedures are variations of clustering procedures which are widely used in the field of multivariate analysis. The Constrained Average Neighbor Weighted bond, CANW, procedure was developed in this research. In fact, we experimented with several weighted versions of the CNN, CFN and CAN procedures. However, the CANW procedure consistently produced program structures which required fewer page fetches than the program structures produced by the CNN, CFN, and CAN procedures or by any of the other weighted versions we examined. One explanation for the success of the CANW procedure is that at each

step it combines the two clusters which have the most total intersector connections between them.

In the above Constrained Neighbor bond definitions, CNN, CFN, CAN, and CANW, the constraint $|Gx| + |Gy| \leq N$ insures that the size of a cluster never exceeds the page size. However, natural clusters of sectors may in reality be larger or smaller than a page size. It is of course conceivable to make clusters covering several pages without any consideration of page sizes and to assign each of them to several contiguous pages. In order to evaluate the merits of allowing clusters to become any natural size, we experimented with

a) the Unconstrained Nearest Neighbor bond, UNN,

b) the Unconstrained Farthest Neighbor bond, UFN,

c) the Unconstrained Average Neighbor bond, UAN, and

d) the Unconstrained Average Neighbor Weighted bond, UANW,

where UNN, UFN, UAN, and UANW are defined to be exactly the same as CNN, CFN, CAN, and CANW, respectively, with the exception that the constraint $|Gx| + |Gy| \leq N$ is not present in the unconstrained cases. That is, in the unconstrained cases, clusters may be combined independently of their sizes.

The clustering procedure for the constrained clusters had to be modified slightly in order to be applicable for the unconstrained clusters. The clustering procedure for the unconstrained clusters is as follows.

Choose any one of the unconstrained definitions of B(x,y). Partition the m relocatable sectors of a program into exactly m clusters, where each cluster contains one sector. Then, at each step in the clustering process, the nearest two clusters (i.e., the two with the largest value of B(x,y)) are combined. Now we will define what we mean by combine.

Let the two clusters which are to be combined at any step of the clustering process be denoted by

$Gx = Sx_1, Sx_2, \ldots, Sx_i$ and

$Gy = Sy_1, Sy_2, \ldots, Sy_j$,

where the cluster Gx is defined to be the ordered list of i sectors, and the cluster Gy is defined to be the ordered list of j sectors. The combination of the clusters Gx + Gy is defined to be the ordered list of i + j sectors

$Gx + Gy = Sx_1, Sx_2, \ldots, Sx_i, Sy_1, Sy_2, \ldots, Sy_j$.

Since each cluster starts out with one sector, the above definition of combining two clusters insures that the relative order in which sectors are clustered is preserved. This is important in the unconstrained case, because the clustering procedure ends when all the clusters which are connected are grouped into one giant cluster, which could be the whole program.

Note that the order of the sectors in the constrained clusters is not important, because a constrained cluster will always fit into a page.

5.4 Hatfield and Gerald Method

The Hatfield and Gerald clustering procedure can be applied to any intersector reference matrix model, C = [Cij]. The HG clustering procedure is defined in detail in [H1] and is briefly summarized below. Let

E = [Eij], i,j = 1,2,...,m (m is the number of sectors),

where

Eij = -Cij when i≠j

$\Sigma_{j=1}^{m}$ Cij + 2m when i = j.

The inverse matrix of E is calculated, then a row in the inverse is chosen, and a set of sectors in that row are clustered into a page, and the process is iterated until all sectors are assigned.

We thank Don Hatfield for providing a copy of his restructuring program for use in our restructuring experiments.

5.5 Sector Interchange Procedure

The sector interchange procedure, SIP, is developed in this section. The SIP begins with the set of m relocatable sectors of a program partitioned into n blocks. That is, assume that a partition, Π, of the set of sectors, $\{S_1, S_2, \ldots, Sm\}$, making up a program is given. Let Π be denoted by
$\Pi = \{\Pi_1, \Pi_2, \ldots, \Pi n\}$ where $|\Pi j|$ is the number of sectors in the j-th block of Π.

The blocks, Πx, of Π may represent the logical pages of a program, where the sum of the sizes of the sectors making up a block of Π is less than the page size, or the blocks of Π may represent natural clusters of sectors, where the sum of the sizes of the sectors making up a block may be greater than a page size.

The basic strategy of the sector interchange procedure, SIP, is to reassign sectors to blocks of Π by exchanging two sectors of different blocks when the exchange provides a positive contribution to the sum of the sector connections within blocks. In order to be more precise, we need to define a few terms. Let

C = [Cij] be a symmetric intersector reference matrix for $i, j \leq m$, and

P = $\{S_1, S_2, \ldots, Sm\}$ denote the set of sectors making up a program.

Definitions:

The complement of Πx is denoted $\neg\Pi x$ and

$\neg\Pi x = \{\Pi j \in \Pi : \Pi j \neq \Pi x\}$

Let $Si \in \Pi x$; then the intrablock bond of sector i, Si, with block Πx is defined as

$B(i,\Pi x) = \Sigma_{j \in \Pi x} Cij$

Let $Si \in \Pi x$ and $Si \notin \Pi y$; then the interblock bond of sector i with block Πy is defined as

$B(i,\Pi y) = \Sigma_{j \in \Pi y} Cij$

Let $Si \in \Pi x$; then the interblock bond of sector i with all other blocks is defined as

$B(i,\neg\Pi x) = \Sigma_{j \in \neg\Pi x} Cij$

The quality of the bond for the ith sector is defined as

$q_n(i) = B(i,\Pi x) - B(i,\neg\Pi x)$, where $Si \in \Pi x$.

The quality of a sector partition Π is defined as

$Q_n = \Sigma_{Si \in P} q_n(i)$

The goal of the sector interchange procedure, SIP, is to maximize the quality Q_n by interchanging sectors between blocks of the partition. We now present an efficient method to find an optimal assignment of sectors to blocks under the constraint that each interchange consists of exchanging a sector of one block with a sector of another block.

Lemma 6

Let Si ϵ Πx and Sj ϵ Πy. If Si and Sj are interchanged, the net gain in the quality Q_n, denoted by $\Delta Q_n (i,j)$, is given by

$$\Delta Q_n (i,j) = 4[B(j,\Pi x) - B(j,\Pi y) + B(i,\Pi y) - B(i,\Pi x) - 2Cij].$$

Proof:

Let Si ϵ Πx, Sj ϵ Πy and Πx, Πy ϵ Π. Now, interchange sectors Si and Sj which produces the new partition Π'.

$\Delta Q_n (i,j) = Q_n - Q_n' = \Sigma_{Sk\epsilon P} q_n(k) - \Sigma_{Sk\epsilon P} q_n'(k) = \Sigma_{Sk\epsilon P} q_n(k) - q_n'(k)$.

Let $\Delta q(k) = q_n(k) - q_n'(k)$.

Now we consider 5 cases.

Case 1. $\Delta q(k) = 2(Ckj - Cki)$ for all $k\epsilon \Pi x$, $k \neq i$.

Case 2. $\Delta q(k) = 2(Cki - Ckj)$ for all $k\epsilon \Pi y$, $k \neq j$.

Case 3. $\Delta q(k) = 0$ for all $k\epsilon \neg(\Pi y + \Pi x)$

Case 4. $\Delta q(i) = B(j,\Pi x) - B(j,\Pi y) - B(j,\Pi x + \Pi y) - 2Cij$
$- B(i,\Pi x) + B(i,\Pi y) + B(i,\Pi x + \Pi y)$

Case 5. $\Delta q(j) = B(j,\Pi x) - B(j,\Pi y) + B(j,\Pi x + \Pi y) - 2Cij$
$- B(i,\Pi x) + B(i,\Pi y) - B(i,\Pi x + \Pi y)$.

Now,

$$\Delta Q_{\pi}(i,j) = \sum_{\substack{k\in\pi x \\ k\neq i}} q(k) + \sum_{\substack{k\in\pi y \\ k\neq j}} \Delta q(k) + \Delta q(i) + \Delta q(j)$$

$$= 2[B(j,\Pi x) - B(i,\Pi x) - Cij]$$

$$+ 2[B(i,\Pi y) - B(j,\Pi y) - Cij] + \Delta q(i) + \Delta q(j)$$

$$\Delta Q_{\pi}(i,j) = 4[B(j,\Pi x) - B(j,\Pi y) + B(i,\Pi y) - B(i,\Pi x) - 2Cij].$$

QED.

Now we present a Lemma which permits us to quickly select the Sj and Si for exchange.

Lemma 7:

If $\Delta Q_\pi (i,j)$ is positive, then $q_\pi (i)+q_\pi (j)$ is negative.

Proof:

$$q_\pi(i) = B(i,\Pi x) - B(i,\neg\Pi x)$$

$$= B(i,\Pi x) - B(i,\Pi y) - B(i,\neg(\Pi x+\Pi y))$$

similarly,

$$q_\pi (j) = B(j,\Pi y) - B(j,\Pi x) - B(j,\neg(\Pi x+\Pi y))$$

From Lemma 6,

$\Delta Q_\pi (i,j) = 4[B(j,\Pi x) - B(j,\Pi y) + B(i,\Pi y) - B(i,\Pi x) - 2Cij]$. Hence,

$\Delta Q_\pi (i,j) = -4[q_\pi (i) + q_\pi (j) + B(i,\neg(\Pi x + \Pi y)) + B(j,\neg(\Pi x + \Pi y)) + 2Cij]$.

But $B(i,\neg(\Pi x + \Pi y)) + B(j,\neg(\Pi x + \Pi y)) + 2Cij \geq 0$, and

$\Delta Q_\pi (i,j) > 0$. Thus $q_\pi (i) + q_\pi (j) < 0$.

QED.

FACT 1:

The maximum value of $\Delta Q_\pi (i,j) = -4(q_\pi (i) + q_\pi (j))$.

This fact follows directly from the proof of Lemma 7.

FACT 2:

If $\Delta Q_\pi (i,j) > 0$, then (i,j) must be an element of the Interchange set, I_π, where

$I_\pi = \{(i,j) : q_\pi (i)+q_\pi (j) < 0,\ i,j \in P\}$.

This fact follows immediately from Lemma 7.

Now we iteratively define the sector interchange procedure, SIP. We assume that an initial partition, Π^0, and an intersector reference matrix, C, are given.

The operations performed in the kth pass are these:

a. Compute the set $I_{\Pi^{K-1}}$

b. Select a pair (i,j) such that

$\Delta Q_{\Pi^{K-1}}(i,j) \geq \Delta Q_{\Pi^{K-1}}(u,v)$ for

all $(u,v) \in I_{\Pi^{K-1}}$

c. If $\Delta Q_{\Pi^{K-1}}(i,j) > 0$, then interchange sectors i and j of Π^{k-1} to get Π^k, and go to the (k + 1)th pass.

If $\Delta Q_{\Pi^{K-1}}(i,j) \leq 0$, then stop with Π^{k-1}.

The SIP has to terminate at some pass k, since Cij is finite. If it terminates on the kth step, then Π^{k-1} is optimum in the sense that no pairwise interchange can increase the value of $Q_{\Pi^{K-1}}$. This is obvious, since $I_{\Pi^{K-1}}$ contains all the possible candidates (i,j) that could possibly make $\Delta Q_{\Pi^{K-1}}$ positive, and since at termination $\Delta Q_{\Pi^{K-1}}(u,v) < 0$ for all $(u,v) \in I_{\Pi^{K-1}}$.

In each pass of the previous algorithm, by keeping the list of sectors in the set $I_{\Pi^{K-1}}$ sorted and using Fact 1, the algorithm can be made much more efficient.

The sector interchange procedure, SIP, is particularly useful when one has a partition, Π, where the blocks of Π represent natural clusters of sectors. Another application of SIP is in the evaluation of breaking up huge sectors into smaller parts by reprogramming.

An ongoing research project between the author and Don Hatfield of IBM is to evaluate the potential benefit of reprogramming and then restructuring a very large data base system. The rationale for reprogramming is to divide the very large sectors (over 10 pages long) into relocatable subsectors and then restructure the new program. Theorem 1 can be used to predict the theoretical best paging performance if the large data base program were broken up into exactly k sectors per page. Then, given an intersector reference matrix and a partition, Π, of k sectors per block, the sector interchange procedure, SIP, can be used to restructure the program.

5.6 Intercluster Bonding Method

The purpose of the intercluster bonding method is to identify natural clusters of dense sector interactions. This task is accomplished by permuting the rows and columns of an intersector reference matrix model in such a way as to group the numerically larger matrix elements together.

The definition of the intercluster bond measure is given first, then we illustrate the capability of this measure to cluster the larger matrix elements together, and then we present a fast approximate method of permuting the rows and columns of a given matrix such that the intercluster bond measure tends to be maximized.

Given a symmetric intersector reference matrix C = [Cij] for i, j = 1,2,...,m which represents the intersector activity between the m relocatable sectors of a program, we define the intercluster bond measure, ICB, as

$$ICB(C) = \sum_{i=1}^{m} \sum_{j=1}^{m} Cij(C_{i-1,j} + C_{i+1,j} + C_{i,j-1} + C_{i,j+1})$$

where $C_{0,j} = C_{m+1,j} = C_{i,0} = C_{i,m+1} = 0$ by definition and $Cij \geq 0$.
We point out that the bond strength between two nearest-neighbor elements of C is their product.

The intercluster bond measure, ICB, is defined so that a matrix C that has dense clusters of numerically large elements will have a large ICB when compared with the same matrix whose columns and rows are permuted such that numerically large elements are more uniformly distributed over the array cells. In order to illustrate the sensitivity of ICB(C) to the degree of clumpiness of the large values of Cij, we present the following two simple examples. Example 1 shows the same matrix with 5 different row and column permutations. Matrix C_5

which has the largest intercluster bond measure contains two noninteracting clusters. One cluster consists of the sectors a and c, while the other cluster consists of the sectors b and d. The fact that matrix C_1 could be reordered to produce two noninteracting clusters is not readily apparent even for this simple example. Example 2 shows a slightly more complicated matrix. Matrix C_4 of example 2 is characterized by a block checkerboard form, where the blocks of sectors along the main diagonal represent the primary sector clusters and the off-diagonal blocks indicate the intercluster interactions. Matrix C_5 which has the largest intersector bond measure of Example 2 has the same set of primary clusters as Matrix C_4 but it differs from C_4 in that the clusters which interact the most are ordered adjacent to each other. The intercluster bond measure, ICB, tends to be maximum when the most strongly intraconnected sectors are clustered together and the most strongly interconnected clusters are clustered together. We call ICB the intercluster bond measure because it tends to cluster the intercluster connections as well as cluster sectors.

In our experimental studies, sector orderings which produced the largest values for the intercluster bond measure provided as good as or better improvements in the paging performance than any other program restructuring method tested.

Example 1:

	a	b	c	d
a	10	0	10	0
b	0	8	0	8
c	10	0	10	0
d	0	8	0	8

C_1 matrix

$ICB(C_1) = 0$

	a	b	c	d
a	10	0	10	0
b	0	8	0	8
d	0	8	0	8
c	10	0	10	0

C_2 matrix

$ICB(C_2) = 256$

	a	b	c	d
a	10	0	10	0
c	10	0	10	0
b	0	8	0	8
d	0	8	0	8

C_3 matrix

$ICB(C_3) = 656$

	a	b	d	c
a	10	0	0	10
c	10	0	0	10
b	0	8	8	0
d	0	8	8	0

C_4 matrix

$ICB(C_4) = 912$

	a	c	b	d
a	10	10	0	0
c	10	10	0	0
b	0	0	8	8
d	0	0	8	8

C_5 matrix

$ICB(C_5) = 1312$

Example 2:

	a	b	c	d	e	f	g	h
a	10	10	0	0	4	4	0	1
b	10	10	0	0	4	4	0	0
c	0	0	8	8	0	0	1	1
d	0	0	8	8	0	0	1	1
e	4	4	0	0	10	10	0	0
g	0	1	0	1	0	7	0	7
f	4	0	4	0	10	0	10	0
h	1	1	1	0	0	7	0	7

C_1

$ICB(C_1) = 1548$

	a	e	c	h	b	f	d	g
a	10	4	0	1	10	4	0	0
e	4	10	0	0	4	10	0	0
c	0	0	8	1	0	0	8	1
h	0	0	1	7	0	0	1	7
b	10	4	0	0	10	4	0	0
f	4	10	0	0	4	10	0	0
d	0	0	8	1	0	0	8	1
g	0	0	1	7	0	0	1	7

C_2

$ICB(C_2) = 1560$

	a	b	c	d	e	g	h	f
a	10	10	0	0	4	0	1	4
b	10	10	0	0	4	0	0	4
c	0	0	8	8	0	1	1	0
d	0	0	8	8	0	1	1	0
e	4	4	0	0	10	0	0	10
g	0	0	1	1	0	7	7	0
h	1	0	1	1	0	7	7	0
f	4	4	0	0	10	0	0	10

C_3

$ICB(C_3) = 1864$

	a	b	c	d	e	f	g	h
a	10	10	0	0	4	4	0	1
b	10	10	0	0	4	4	0	0
c	0	0	8	8	0	0	1	1
d	0	0	8	8	0	0	1	1
e	4	4	0	0	10	10	0	0
f	4	4	0	0	10	10	0	0
g	0	0	1	1	0	0	7	7
h	1	0	1	1	0	0	7	7

C_4

$ICB(C_4) = 2776$

	a	b	e	f	c	d	g	h
a	10	10	4	4	0	0	0	1
b	10	10	4	4	0	0	0	0
e	4	4	10	10	0	0	0	0
f	4	4	10	10	0	0	0	0
c	0	0	0	0	8	8	1	1
d	0	0	0	0	8	8	1	1
g	0	0	0	0	1	1	7	7
h	1	0	0	0	1	1	7	7

C_5

$ICB(C_5) = 3536$

Note that the definition of ICB may be decomposed into the two parts as follows:

$ICB(C) = ICB(C_R) + ICB(Cc)$, where

$ICB(C_R) = \Sigma_{i=1}^{m} \Sigma_{j=1}^{m} Cij(C_{i-1,j} + C_{i+1,j})$

$ICB(Cc) = \Sigma_{j=1}^{m} \Sigma_{i=1}^{m} Cij(C_{i,j-1} + C_{i,j+1})$

The value of $ICB(C_R)$ is the sum of the row bonds and the value of $ICB(Cc)$ is the sum of the column bonds.

Property 1:

The values of the row bonds, $\Sigma_{j=1}^{m}\ Cij(C_{i-1,j} + C_{i+1,j})$ are not affected by any permutation of the m columns of C.

Proof:

Let $\lambda = \{\lambda(1), \lambda(2), \ldots \lambda(m)\}$ denote any permutation of the m columns of C producing the new matrix $D = [Dij] = [C_{i,\lambda(j)}]$.

Then, for any $1 \leq i \leq n$,

$$\Sigma_{j=1}^{m}\ Cij(C_{i-1,j} + C_{i+1,j}) = \Sigma_{j=1}^{m}\ C_{i,\lambda(j)}\ (C_{i-1,\lambda(j)} + C_{i+1,\lambda(j)}).$$

This is clearly true, since i is fixed over the summation of all j. Thus, for every term in the summation on the left, $Cij(C_{i-1,j} + C_{i+1,j})$, there must be a value k, $1 \leq k \leq m$, such that

$$Cij(C_{i-1,j} + C_{i+1,j}) = C_{i,\lambda(k)}\ (C_{i-1,\lambda(k)} + C_{i+1,\lambda(k)}).$$

Property 2:

The values of the column bonds, $\Sigma_{i=1}^{m}\ Cij(C_{i,j-1} + C_{i,j+1})$ are not affected by any permutation of the m rows of C.

Proof is the same as that of property 1.

Property 3:

$ICB(C_R) = ICB(Cc)$ for symmetric matrices C.

Proof:

$$\begin{aligned} ICB(C_R) &= \Sigma_{i=1}^{m} \Sigma_{j=1}^{m} Cij(C_{i-1,j} + C_{i+1,j}) \\ &= \Sigma_{i=1}^{m} \Sigma_{j=1}^{m} Cji(C_{j,i-1} + C_{j,i+1}) \\ &= \Sigma_{j=1}^{m} \Sigma_{i=1}^{m} Cij\ (C_{i,j-1} + C_{i,j+1}) \\ &= ICB(Cc). \end{aligned}$$

Property 4:

The contribution to ICB(C) from any row is only affected by the two adjacent rows. The contribution to ICB(C) from any column is only affected by the two adjacent columns.

Property 4 is obvious, since the contribution to ICB(C) from any row i is $\Sigma_{j=1}^{m} Cij(C_{i-1,j} + C_{i+1,j})$ and from any column j is $\Sigma_{i=1}^{m} Cij(C_{i,j-1} + C_{i,j+1})$.

From properties 1 and ,2 the maximization of ICB(C) over all column and row permutations reduces to two separate optimizations. One is for the rows, $ICB(C_R)$, and the other for the columns, ICB(Cc).

From properties 1, 2, and 3, we know that the row permutation which maximizes $ICB(C_R)$, is the same as the column permutation that maximizes ICB(Cc). Thus, all we need to do is find a row permutation

that maximizes $ICB(C_R)$, then reorder the rows and columns of C according to this permutation to maximize ICB(C).

The problem can be stated formally as follows:

Let $\lambda = [\lambda(1), \lambda(2), \ldots, \lambda(m)]$ denote a permutation of m columns of C producing the new matrix $D = [Dij] = [C_{i,\lambda(j)}]$.

Maximization of the summed column bonds ICB(Cc) is given by,

Max over λ of $\Sigma_{j=1}^{m} \Sigma_{i=1}^{m} Dij\ [D_{i,j-1} + D_{i,j+1}]$,

where λ ranges over all m! possible permutations.

This may be transformed into a quadratic assignment problem for which optimal and suboptimal algorithms have been published [G3]. These suboptimal algorithms were not used, since they are too time comsuming for large m, i.e., they require operations which rise with the fifth power of the matrix size.

Now we define a suboptimal method which exploits the nearest-neighbor feature [M5] of property 4. This method is much faster than the optimal methods and is believed to produce near optimal orderings. The intercluster bond method is as follows:

A. First compute and save the set of intercolumn bonds for all pairs (i, j) of columns, i.e.,

$\Sigma_{k=1}^{m} C_{ki} * C_{kj}$ for all $1 \leq i, j \leq m$, $i \neq j$.

B. Pick one of the columns arbitrarily, put it into a list, and set

k=1.

C. For each of the remaining m-k columns, compute the contribution to the intercluster bond measure for each of the k+1 possible positions to the left and to the right of each of the k columns already placed in the list. Place the column that gives the largest incremental contribution to the intercluster bond measure in its best location in the list.

D. If k=m, stop; otherwise, increment k by 1 and repeat step C.

When the above procedure terminates, simply order the rows and columns of C in accordance with the list of columns.

Property 5:

The time for the execution of the clustering process in step C grows as m^3.

To see this, note that

$$\Sigma_{k=1}^{m} (k+1)\ (m-k) = m^3/6 + m^2/2 - (2m/3).$$

The intercluster bond method will cluster the sectors into disjoint groups if this is possible.

CHAPTER 6

EXPERIMENTAL RESULTS

6.1 Introduction

The purpose of this chapter is to report on an experimental study of the paging performance of programs. The objective of this study is to evaluate the practical restructuring methods developed in Chapters 4 and 5. The evaluation consists of two basic parts. First, the paging performance produced by the different restructuring methods are related and contrasted with one another. Second, the improvements in paging performance produced by the practical restructuring methods are compared with the theoretical best and worst improvements as given by the bounds in Chapter 3.

We have performed experiments, using the IBM System/360 Model 67 at the Cambridge Scientific Center, on compilers, assemblers and a large data base program. The results of a specific example will be presented in detail. We have chosen as an example the restructuring of a highly modular compiler [A3]. This example is selected because we have experimental results for all of our restructuring methods applied to this compiler. The author and Don Hatfield of IBM plan to publish the results of using some of these methods to restructure a "large data

base" system as soon as our results are completed.

This compiler has 4 phases. Phase 0 is a very small root phase which simply has Phase 1 read in, and, when Phase 1 is over, has Phase 2 read in, and, when Phase 2 is over, has Phase 3 read in. Each of the phases is a separate overlay in the sense that they do not share any address space. Therefore, we may think of Phases 1, 2, and 3 as three separate programs. There are between 70 and 100 relocatable sectors per phase. For each compilation, we computed three distinct sector traces. One trace was for Phase 1, one for Phase 2, and one for Phase 3. In particular, from the time that a phase was loaded into the address space until its subsequent removal, a full instruction trace of all references to the relocatable modules of that phase was recorded. This instruction trace and the load address of all the relocatable sectors (modules) are sufficient to compute the sector trace.

In order to compare the effectiveness of the different arrangements of sectors into the virtual address space, LRU and OPT paging simulators were developed for a single user paging against himself. Input to the simulator was a sequence of page requests generated from the full instruction trace and a new ordering of sectors into the address space. A modified version of the one pass OPT algorithm by Palermo and Belady [B6] was used as the OPT paging simulator.

When sectors have been assigned to pages, one problem remains. What to do about page boundaries? Holes in pages can occur if sectors do not fit evenly into pages. For most real programs, we have two alternatives. First, we do not allow sectors to cross page boundaries, which may cause empty space within the pages. Second, we pack sectors one after another into the virtual address space, leaving no holes but allowing the sectors to cross page boundaries. Hatfield [H1] has reported on the relative success of the latter approach.

For our experiments, we packed sectors one after another into the virtual address space, leaving no holes between the sectors. That is, given a partition Π of the sectors in blocks, we placed the blocks of the partitions into the virtual address space one after another. The unconstrained average neighbor weighted bond, UANW, procedure was used to automatically order the clusters for insertion into the address space, unless the clustering procedure produced ordered clusters.

The next few sections report on the results of the restructuring experiments performed on the different phases of the compiler. The basic structure of these experiments on each phase is as follows.

A. A full instruction trace is recorded and mapped into a sector trace.

B. An intersector reference matrix model is constructed from the sector trace.

C. A clustering procedure, based on a particular intersector reference matrix, is used to partition the relocatable sectors into blocks.

D. The resulting ordered blocks of the partition are inserted into the address space one after another.

E. The paging performance of the restructured program is simulated using LRU replacement (sometimes OPT replacement is used). We chose LRU replacement because so many contemporary virtual memory systems use some form of this algorithm.

F. The theoretical upper and lower bounds on the paging performance are computed by applying the methods of Chapter 3 to the sector trace of step A and compared with the performance found in step E.

In order to identify the parameters of the page fetch function, $FFp(|Mp|,N,\Pi a,STa,Fd, R_{LRU})$, which are associated with each curve in the following graphs, these conventions are presented.

1. $|Mp|$, the size of the primary memory in pages, is used as the horizontal axis of the graphs. In addition to the values of $|Mp|$, the horizontal axis is tagged with the memory size in K bytes (K=1024).

2. N, the page size in these experiments, is 4096 bytes.

3. A partition Π of relocatable sectors into clusters is denoted by Πx or Πy for ease in interpreting the results in the following

figures. Πx is used to denote a "bad" partition, i.e., one which tends to maximize or produce a relatively large value of FFp. Πy denotes a "good" partition, i.e., one which tends to minimize the value of FFp.

A particular value of Πy is denoted by specifying the intersector reference matrix and the clustering procedure which produced it. For example,

$$\Pi y(W, T=2500, CNN)$$

is defined to denote the value of Πy which is computed from the working set matrix, W, with a window size of T=2500, using the constrained nearest neighbor procedure, CNN.

The intersector reference matrix models used to specify a particular Πy will be identified in terms of the following symbols:

W = outside working set matrix model

W' = inside working set matrix model

T = window size of working set model

U = LRU sector stack matrix model

D = sector stack distance

H = Hatfield and Gerald matrix model

The clustering procedures used to specify a particular value of Πy will be one of the following:

CNN = constrained nearest neighbor

CFN = constrained farthest neighbor

CAN = constrained average neighbor

CANW = constrained average neighbor weighted

UNN = unconstrained nearest neighbor

UFN = unconstrained farthest neighbor

UAN = unconstrained average neighbor

UANW = unconstrained average neighbor weighted

HG = Hatfield and Gerald method

SIP = sector interchange procedure

ICB = intercluster bond method

As another example,

$\Pi y(U,D=20,ICB)$

represents the partition named Πy when it is computed from U, with D=20, using the ICB procedure.

In the presentation of these experimental results, We chose to denote the program structure in terms of Π instead of the sector ordering SO, because the clustering procedure is clearer when stated in terms of Π. However, the reader should be aware that the blocks of the partition are allowed to cross page boundaries in order to eliminate holes in the address space.

4. A particular value of SOTa will be denoted by SOT_1, SOT_2, and SOT_3 for the three phases 1, 2, and 3 respectively. Furthermore, SOTia, SOTib, etc., will represent the sector trace of the ith phase from input program a, b, etc, when the distinction is important. For

example, SOT_2 a denotes the sector trace of Phase 2 from input program a. Note that all of the sector traces in the simulations are ordered pairs (S,O) where S is the sector and O is the offset referenced. This is necessary because we are allowing sectors to cross page boundaries.
5. The fetch and replacement algorithms are denoted as before, i.e., Fd, R_{LRU}, Ro, etc.

In order to find a Πx that tends to maximize the value of FFp, we investigated random sector orderings, sector orderings based on sector sizes, lexical orderings (i.e., alphabetical on some character in the sector name), and sector orderings produced by the following procedure, called BAD. Take the list L of m sectors, ordered according to their position in the address space under a good program structure, and do the following to produce a partition Πx of the m relocatable sectors into n logical pages.

1. Take the first n sectors of L and put each of them into one of n separate lists.
2. Take the next n sectors of L and put each of them into one of the above n separate lists.
3. Repeat 2 until there are no more sectors in L. Then,
4. the collection of the n separate lists becomes Πx.

It turned out that all of the above methods of generating Πx usually produced a Πx that caused the value of FFp to be very large.

6.2 Restructuring Phase 1

Throughout this section we use the same sector trace, SOT_1. In section 6.5 we compare the results of program restructuring over several sector traces. Our results support the claim of Hatfield and Gerald, "many commonly used programs are rather insensitive to input data."

However, we did attempt to choose a program for tracing that contained most of the features of the language and that was relatively long. That is, this program was not trivial. The sector trace of this program contained 7,521,205 references. Moreover, $|SOT_1|$=2,001,027, $|SOT_3|$=3,859,636 and $|SOT_2|$=1,660,542.

The value of Πx is fixed for Figures 7-14 and represents the program structure B_1 which occurs when the sectors are arranged in the address space according to their size. Even though the structure produced by the BAD procedure resulted in slightly more page fetches for most memory sizes, we selected Πx based on the sector lengths (calld B_1) because this represents a plausible method of loading sectors used by some operating systems. The choice of Πx is used as a basis for illustrating the actual improvement in the paging performance which can occur for real programs which are restructured according to some Πy.

6.2.1 Constrained Procedures

The curves of Figures 7 and 8 and the lower curves, labeled C, D, and E, of Figure 9 show the ratio of the page fetch functions $FFp(|Mp|,N,\Pi x,SOT_1,Fd,R_{LRU})$ and $FFp(|Mp|,N,\Pi y,SOT_1,Fd,R_{LRU})$ as a function of primary memory size $|Mp|$ in pages and as a function of Πx and Πy where Πy is constrained. Πy is constrained when the blocks of Πy correspond to the clusters produced by any clustering procedure and the size of these clusters is constrained to be less than or equal to the page size.

These figures reveal that the orderings of the relocatable sectors into primary memory can have substantial influence on the paging performance of virtual memory systems. Moreover, they illustrate that substantial improvements in paging performance occur over a relatively wide range of primary memory sizes.

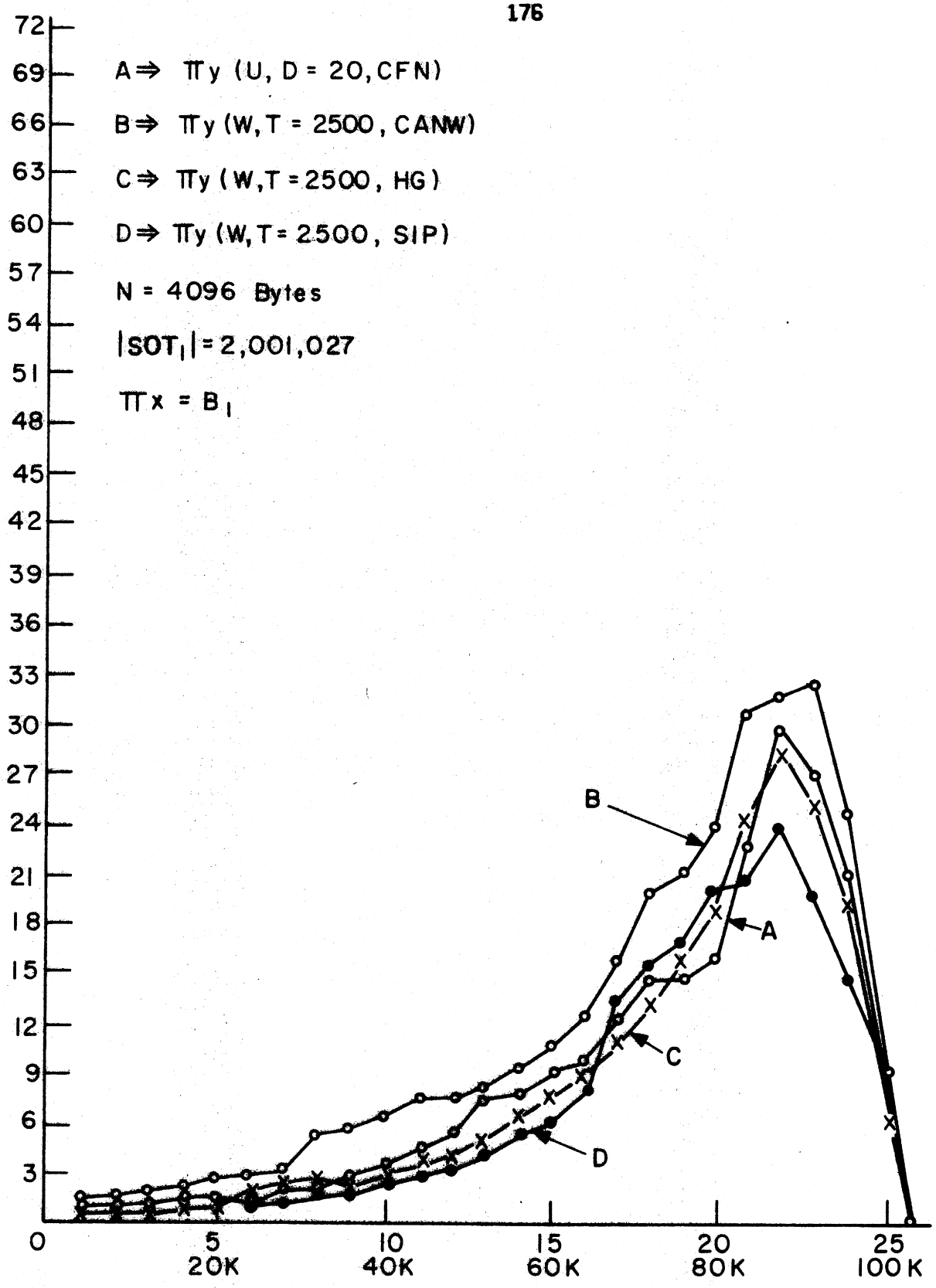


FIGURE 7 FFp (|Mp|, N, TTx, SOT_1, Fd, R_{LRU}) / FFp (|Mp|, N, TTy, SOT_1, Fd, R_{LRU}) vs |Mp| FOR PHASE I OF AED COMPILER

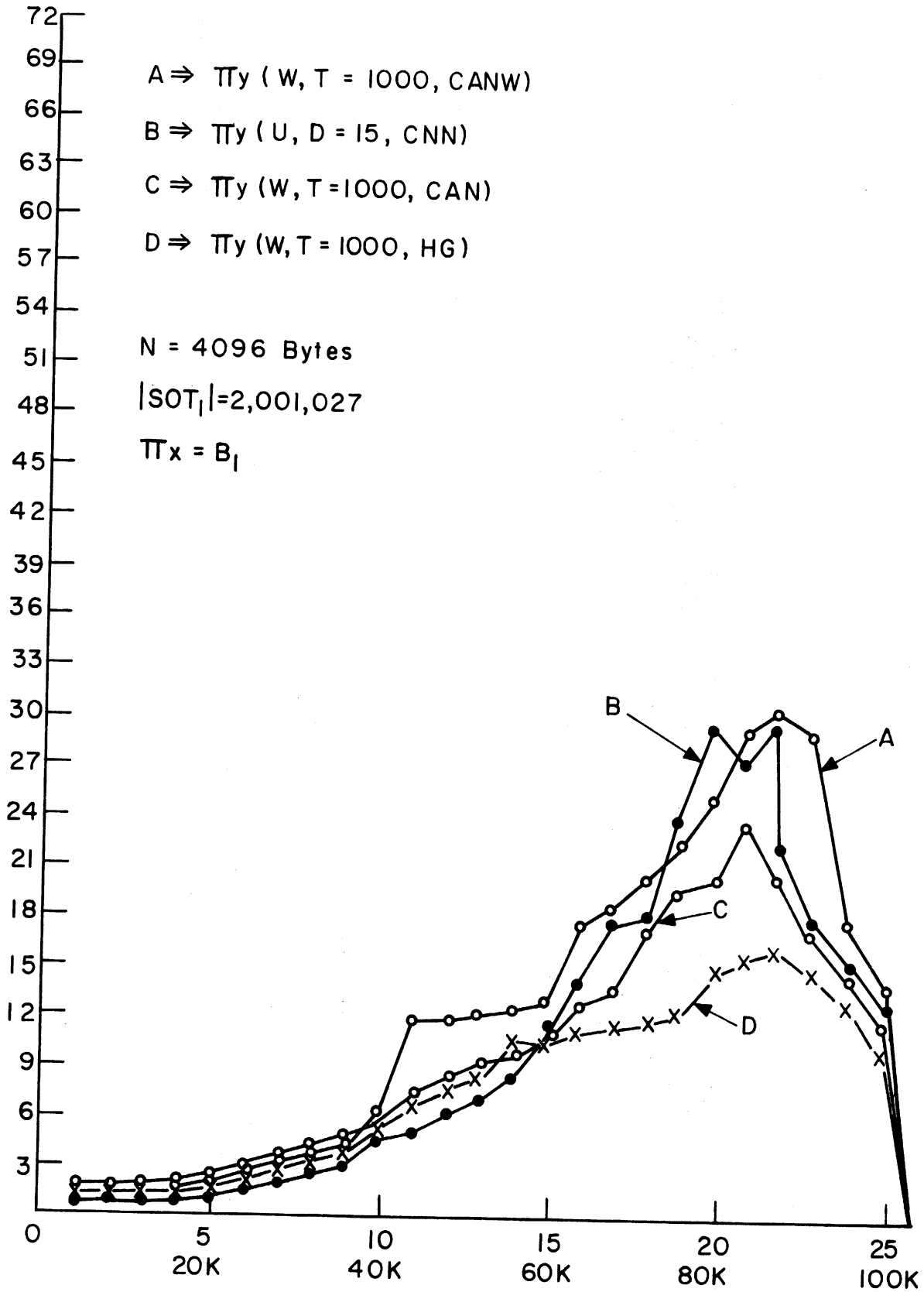


FIGURE 8 FFp (|Mp|, N, TTx, SOT_1, Fd, R_{LRU}) / FFp (|Mp|, N, TTy, SOT_1, Fd, R_{LRU}) vs |Mp| FOR PHASE I OF AED COMPILER

The degree of improvement in paging performance shown in these figures (i.e., 7-8) is significantly larger than any previously published improvements obtained by restructuring. One rationale for this is that the intersector reference matrix models based on the working set and the LRU stack distances capture the intersector activity upon which paging depends. That is, the value, Cij, of the entry in the intersector reference matrices used in these experiments may have a strong tendency to be proportional to the number of page fetches which will go away if sector j is grouped with sector i. In particular, note the improvement in paging performance depicted by curves E, D, and C of Figure 9, which use the HG clustering technique on the sector working set intersector reference matrix. This improvement is about twice as much as that reported by Hatfield and Gerald [H1] when the same clustering procedure is applied to the HG intersector reference model. Recall that the HG intersector reference model is the same as the sector working set model when T=1.

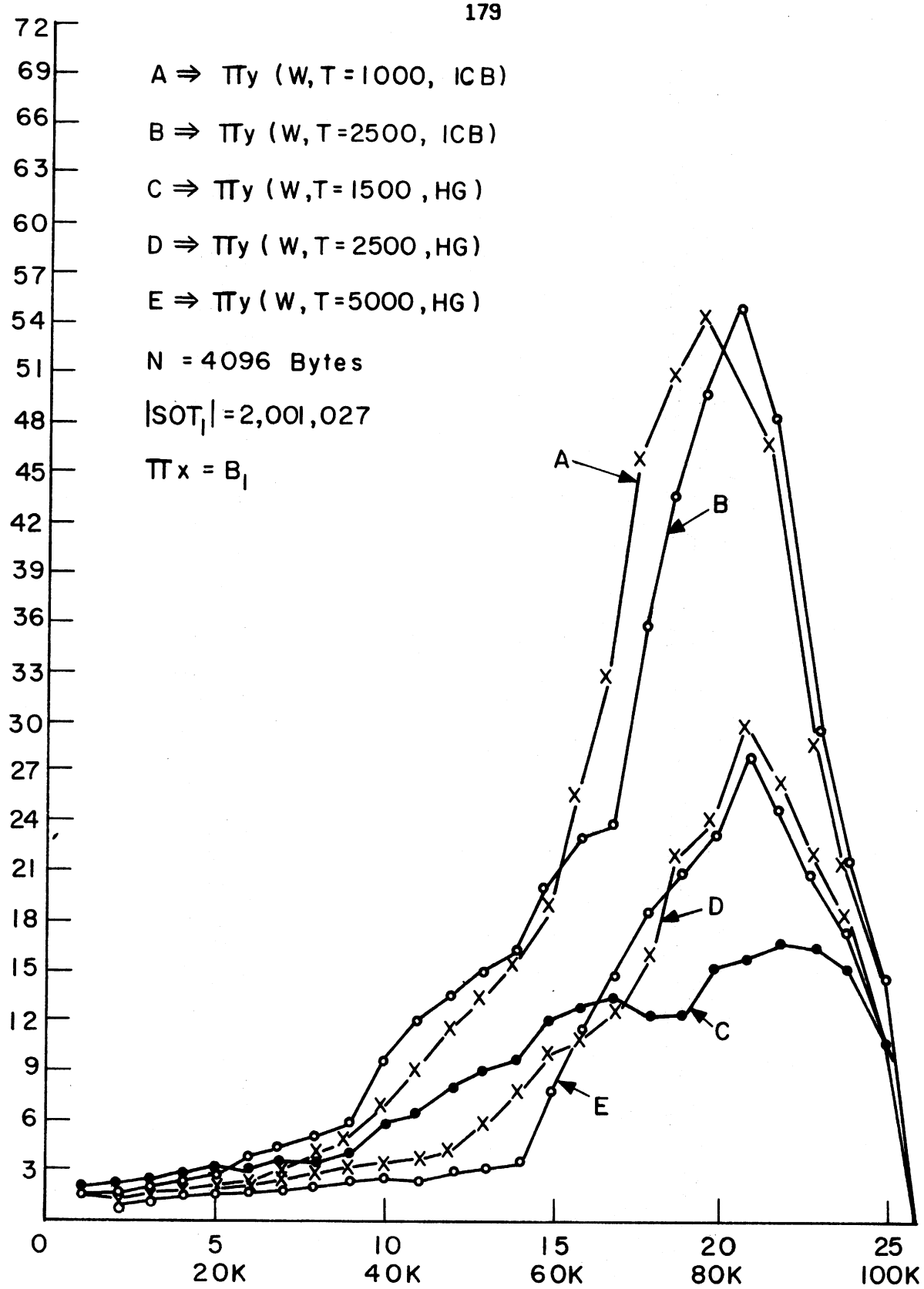


FIGURE 9 FFp (|Mp|, N, TTx, SOT_1, Fd, R_{LRU}) / FFp (|Mp|, N, TTy, SOT_1, Fd, R_{LRU}) vs |Mp| FOR PHASE I OF AED COMPILER

6.2.2 Unconstrained Procedures

The unconstrained clustering procedures presented in Chapter 5 cluster the relocatable sectors into natural clusters without any constraint on the sum of the sector sizes making up a cluster. To date, no work has been reported in the literature which incorporates this rather simple idea into clustering procedures.

The curves identified by labels A and B of Figure 9 show the improvement in paging performance which occurred when natural clusters were formed. These natural clusters were produced by the intercluster bond method, ICB, using the sector working set intersector reference model. These curves illustrate that natural clusters can provide significantly better improvements in the paging performance than the improvement provided by the constrained clustering techniques.

The curves of Figure 10 (except curve D) show the improvement in paging performance for several unconstrained clustering techniques. The curve labelled D in Figure 10 shows the improvement in paging performance provided by the existing compiler structure.

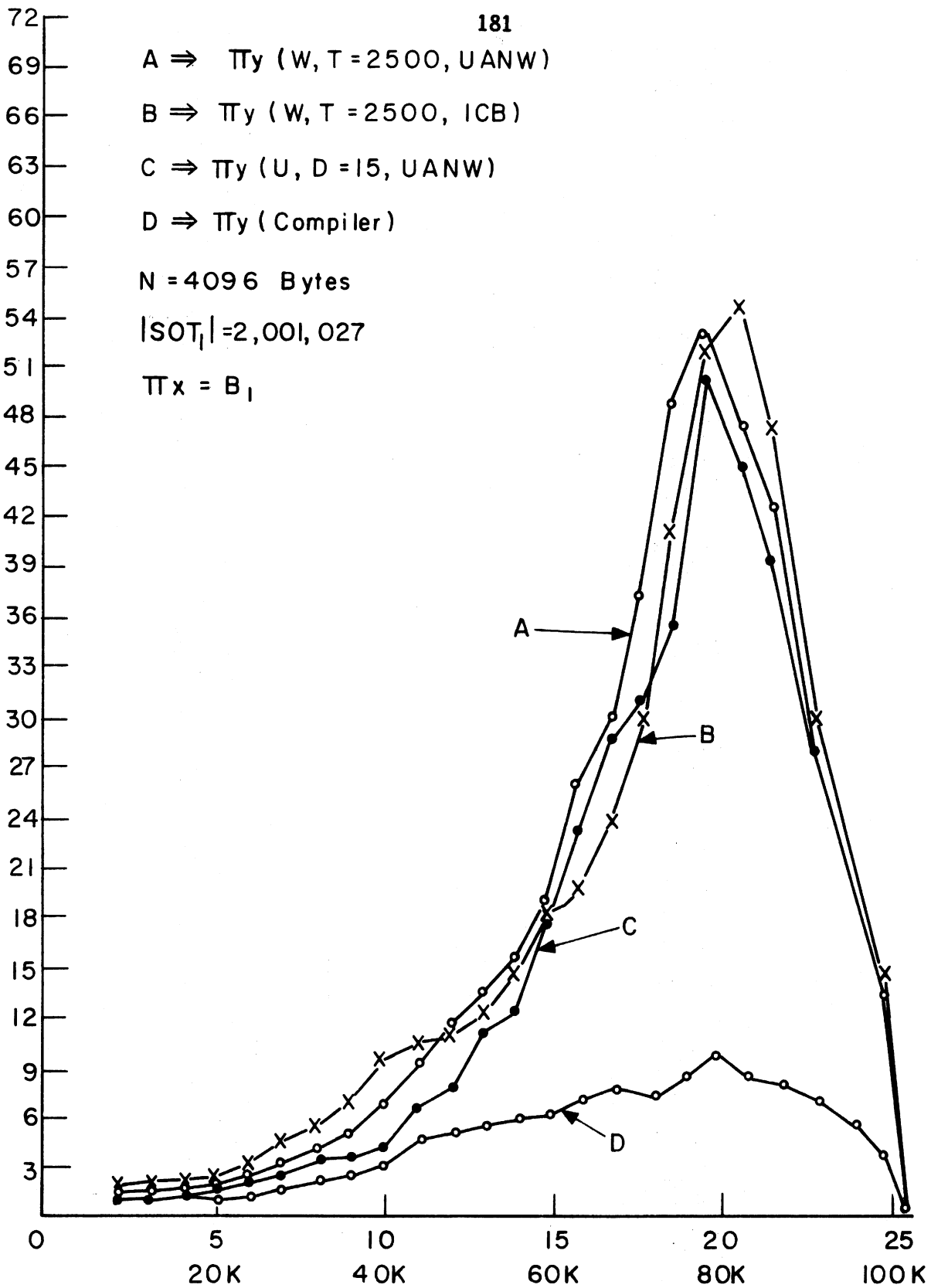


FIGURE 10 FFp (|Mp|, N, TTx, SOT_1, Fd, R_{LRU})/FFp (|Mp|, N, TTy, SOT, Fd, R_{LRU}) vs |Mp| FOR PHASE 1 OF AED COMPILER

Recall that all these improvements are relative to the program structure Πx formed by arranging the sectors into the address space in order of their sizes. Curve D shows that the existing compiler structure is substantially better than that provided by Πx and significantly worse than any of the unconstrained techniques.

Figure 11 shows the effects of the unconstrained average neighbor weighted bond procedure UANW on the paging performance as a function of T for the working set intersector reference model W. The significant characteristics of the curves shown in Figure 11 is that the improvements in paging performance are relatively the same over a broad range of T values.

Note the tendency of the curves in Figure 11 to peek in the center region of the primary memory sizes. This tendency is due primarily to the following two "principles" pushing a curve together from both sides. The first principle is that for small values of $|Mp|$, one clustering method "cannot win" over another method. The second principle is that for large values of $|Mp|$, one clustering approach "cannot lose" over another approach. However, in the middle range of the values of $|Mp|$, there may be enough primary memory available to contain most of the sectors referenced close together in time when they are clustered together into groups. Note that in this region there can be two levels of clustering for good structures. The first level is that sectors are clustered together by the clustering procedure. The second level is

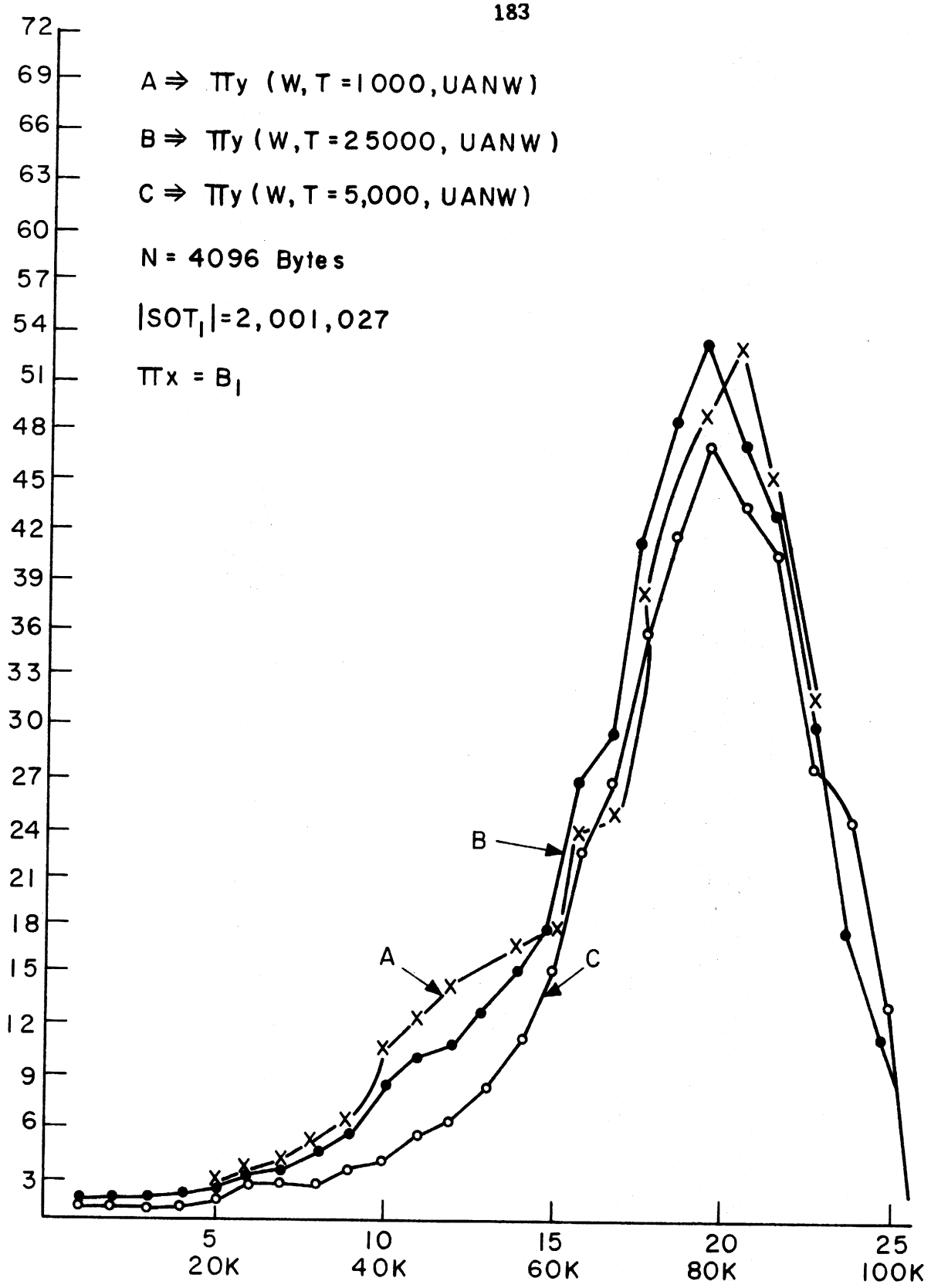


FIGURE 11 FFp (|Mp|, N, TTx, SOT_1, Fd, R_{LRU})/FFp (|Mp|, N, TTy, SOT_1, Fd, R_{LRU}) vs |Ms| FOR PHASE I OF AED COMPILER

that clusters are clustered together by the paging mechanism.

6.2.3 Theoretical Bounds

In Figure 12 the performance for the best program structure, i.e., the one produced by Πy(W,T=2500,UANW), is compared with the theoretical best performance given by Theorem 8. Observe that Table 3 precisely defines the parameters for the curves shown in Figure 12. Curve B shows the ratio of the page fetches experienced by the program under the structure produced by Πy(W,T=2500,UANW) to the theoretical lower bound on the page fetches. That is, curve B depicts $FFp(|Mp|,N,\Pi y,SOT_1,Fd,R_{LRU})$/the Lower Bound. This ratio can never be less than one and would be equal to one when the theoretical best performance occurred for a given program structure. Figure 12 shows several significant characteristics. The performance produced by the structure Πy(W,T=2500,UANW) is relatively close to the lower bound for large regions of primary memory size. Furthermore, it is close to the lower bound in the primary memory regions of low paging rates. This latter fact can be seen by observing the curves in Figure 13. Curve D of Figure 13 shows the number of page fetches for the structure Πy(W,T=2500,UANW), and curve A shows the theoretical lower bound for the number of page fetches over all Πy.

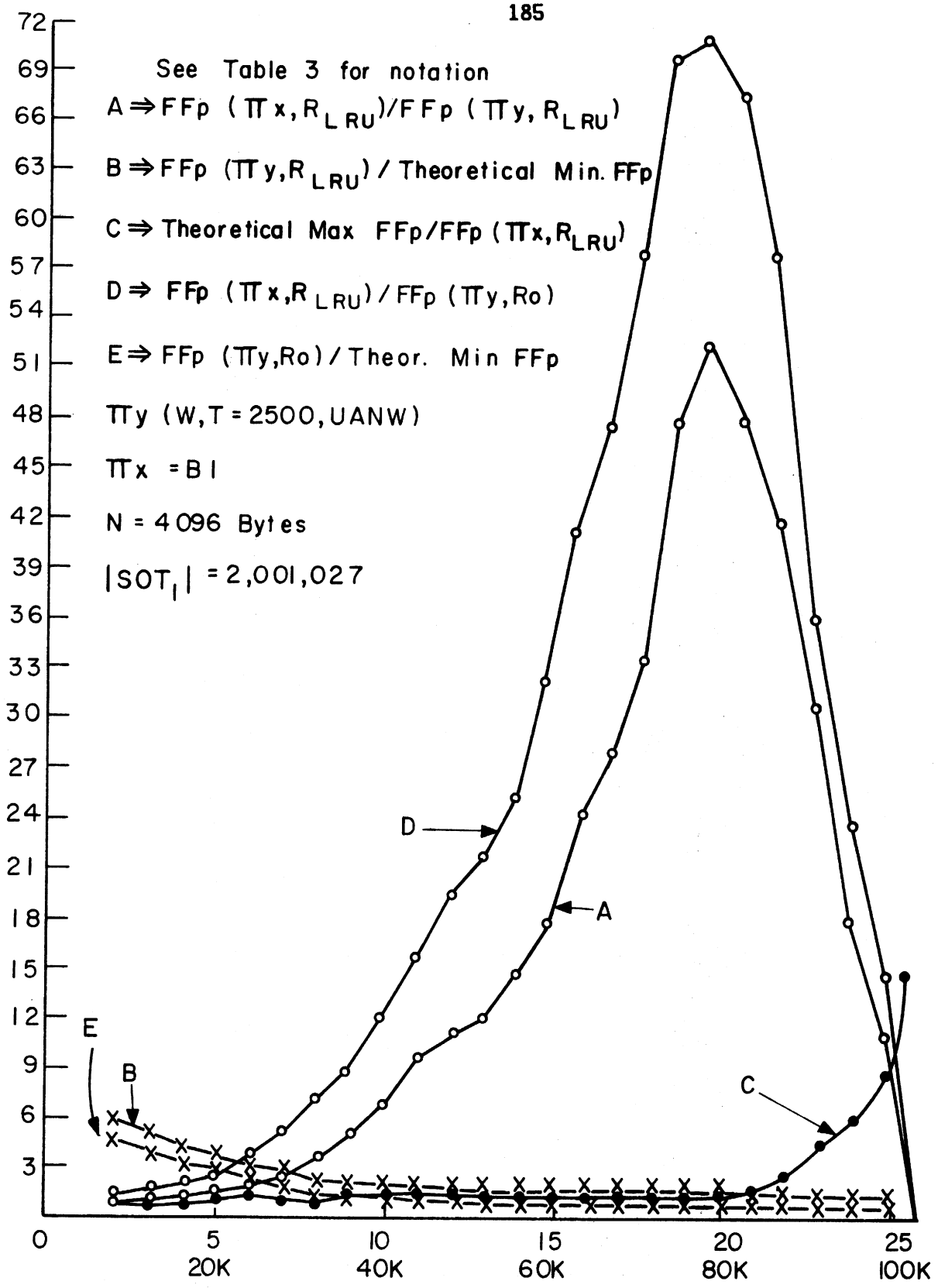


FIGURE 12 Comparison of Actual and Theoretical Ratios of FFp FOR PHASE I OF AED COMPILER

Graph A is:

$$FFp(|Mp|,N,\Pi x,SOT_1,Fd,R_{LRU})/FFp(|Mp|,N,\Pi y,SOT_1,Fd,R_{LRU})$$

Graph B is:

$$FFp(|Mp|,N,\Pi y,SOT_1,Fd,R_{LRU})/\frac{FFs(|Ms| = f_1(|Mp|,N,SS^*),SOT_1^*,Fd,Ro)}{f_1(2,N,SS)/2} - \Delta$$

Graph C is:

$$FFs(|Ms| = |Mp|,SOT_1,Fd,R_{LRU})/FFp(|Mp|,N,\Pi x,SOT_1,Fd,R_{LRU})$$

Graph D is:

$$FFp(|Mp|,N,\Pi x,SOT_1,Fd,R_{LRU})/FFp(|Mp|,N,\Pi y,SOT_1,Fd,Ro)$$

Graph E is:

$$FFp(|Mp|,N,\Pi y,SOT_1,Fd,Ro)/\frac{FFs(|Ms| = f_1(|Mp|,N,SS^*),SOT_1^*,Fd,Ro)}{f_1(2,N,SS)/2} - \Delta$$

where Πx = B1, Πy(W,T = 2500,UANW), N = 4096 Bytes

$|SOT_1|$ = 2,001,027

Note that $\frac{FFs(|Ms| = f_1(|Mp|,N,SS^*),SOT_1^*,Fd,Ro)}{f_1(2,N,SS)/2} - \Delta$
shown in B and E above is the lower bound of FFp given in Theorem 6.

Note that $FFs(|Ms| = |Mp|,SOT_1,Fd,R_{LRU})$

shown in C above is the upper bound of FFp given by Theorem 3.

Table 3
Parameters for Curves in Figure 12

Curve B of Figure 12 indicates that the lower bound may be loose for very small values of |Mp| or that the structure Πy(W,T=2500,UANW) does not cluster sectors very well for small values of |Mp|. The conjecture is that the lower bound may be loose for very small values of |Mp| since this phenomenon is observed in all of our experiments. This is not a serious practical drawback, because even for the lower bound the paging activity is prohibitively large for very small |Mp|. Since the lower bound is valid over all replacement algorithms, we compared the ratio of the performance of the good structure Πy using OPT replacement to the lower bound. This ratio is curve E of Figure 12.

Curve C of Figure 12 illustrates the ratio of the theoretical upper bound given by Theorem 3 to the bad performance. The bad performance is the number of page fetches produced with the structure Πx.

The upper bound is relatively close to the "worst" performance resulting from the structure Πx for most values of |Mp|. For large values of |Mp| the upper bound is not very tight. The upper bound will be tight as long as the sectors which are clustered into a page are never used together when that page is in Mp. However, as the size of Mp increases, it becomes more and more difficult for this condition to be satisfied. Hence, the upper bound grows very rapidly for values of |Mp| approaching the length of the program. However, for values of |Mp| in the region where the program would probably be run, the upper bound is reasonable.

Figure 13 shows the number of page fetches given by:

A. the lower bound.

B. the upper bound.

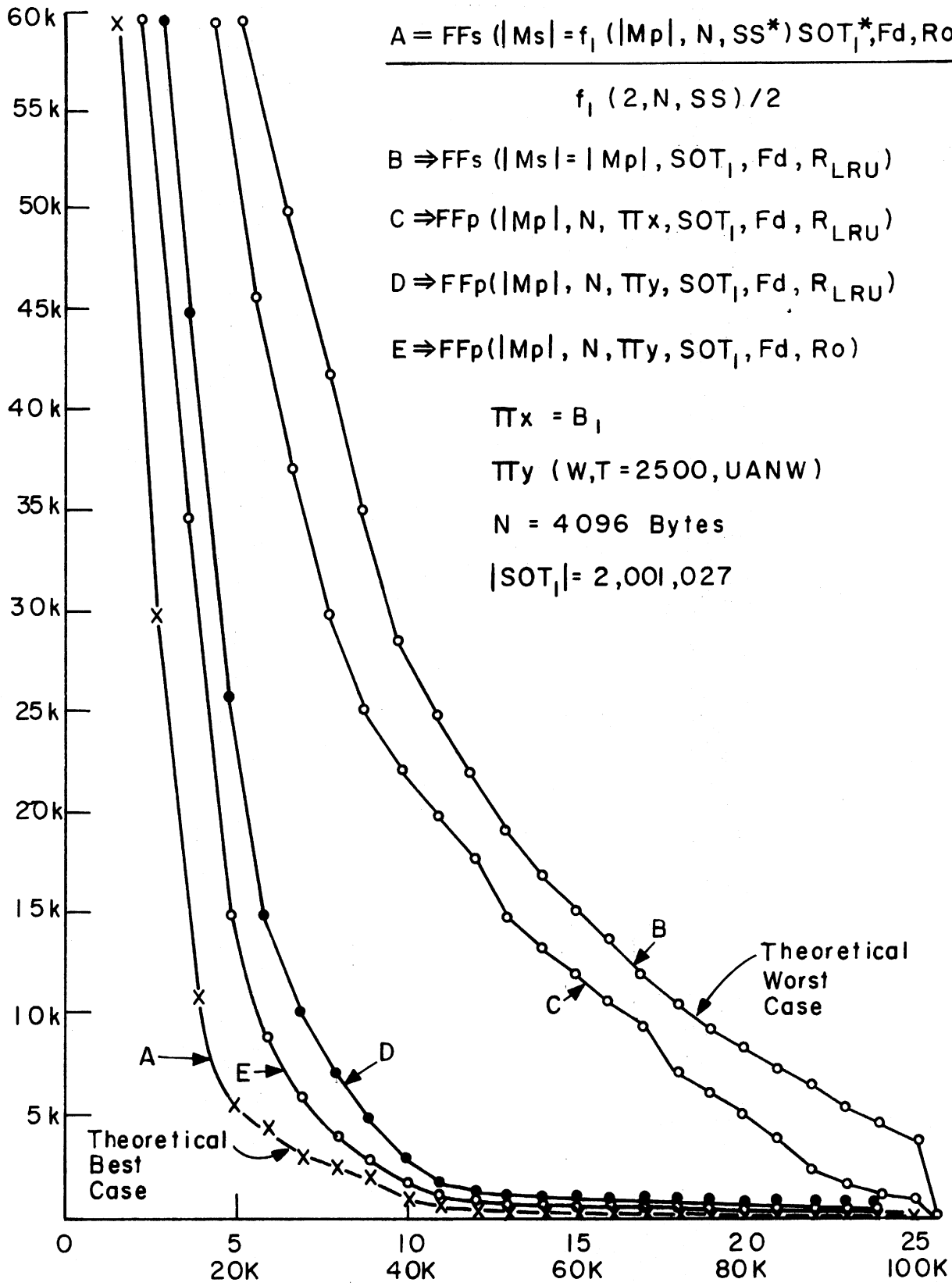
C. the bad structure, Πx.

D. the good structure, $\Pi y(W,T=2500,UANW)$ under LRU.

E. the good structure, $\Pi y(W,T=2500,UANW)$ under OPT.

Figure 14 is simply the values for curves A, C, and D of Figure 13 shown at a much larger scale.

In summary, Figures 9-14 show that the paging performance may vary by a factor of 12 to 30 for large regions of primary memory size $|Mp|$. This occurs when the unconstrained clustering procedures are used in conjunction with the sector working set and the LRU stack intersector reference matrices; that is, for $\Pi y(W,T=2500,UANW)$, $\Pi y(W,T=2500,ICB)$ and $\Pi y(U,D=15,UANW)$. The use of clustering procedures which cluster sectors into natural clusters can produce program structures which require significantly fewer page fetches than required by program structures based on constrained clustering procedures.



$$A = \frac{FFs\,(|Ms| = f_1\,(|Mp|, N, SS^*)\,SOT_1^*, Fd, Ro) - \Delta}{f_1\,(2, N, SS)/2}$$

$B \Rightarrow FFs\,(|Ms| = |Mp|, SOT_1, Fd, R_{LRU})$

$C \Rightarrow FFp\,(|Mp|, N, \pi x, SOT_1, Fd, R_{LRU})$

$D \Rightarrow FFp\,(|Mp|, N, \pi y, SOT_1, Fd, R_{LRU})$

$E \Rightarrow FFp\,(|Mp|, N, \pi y, SOT_1, Fd, Ro)$

$\pi x = B_1$

$\pi y\ (W,T = 2500, UANW)$

$N = 4096$ Bytes

$|SOT_1| = 2{,}001{,}027$

FIGURE 13 Total Page Fetches vs |Mp|
FOR PHASE I OF AED COMPILER

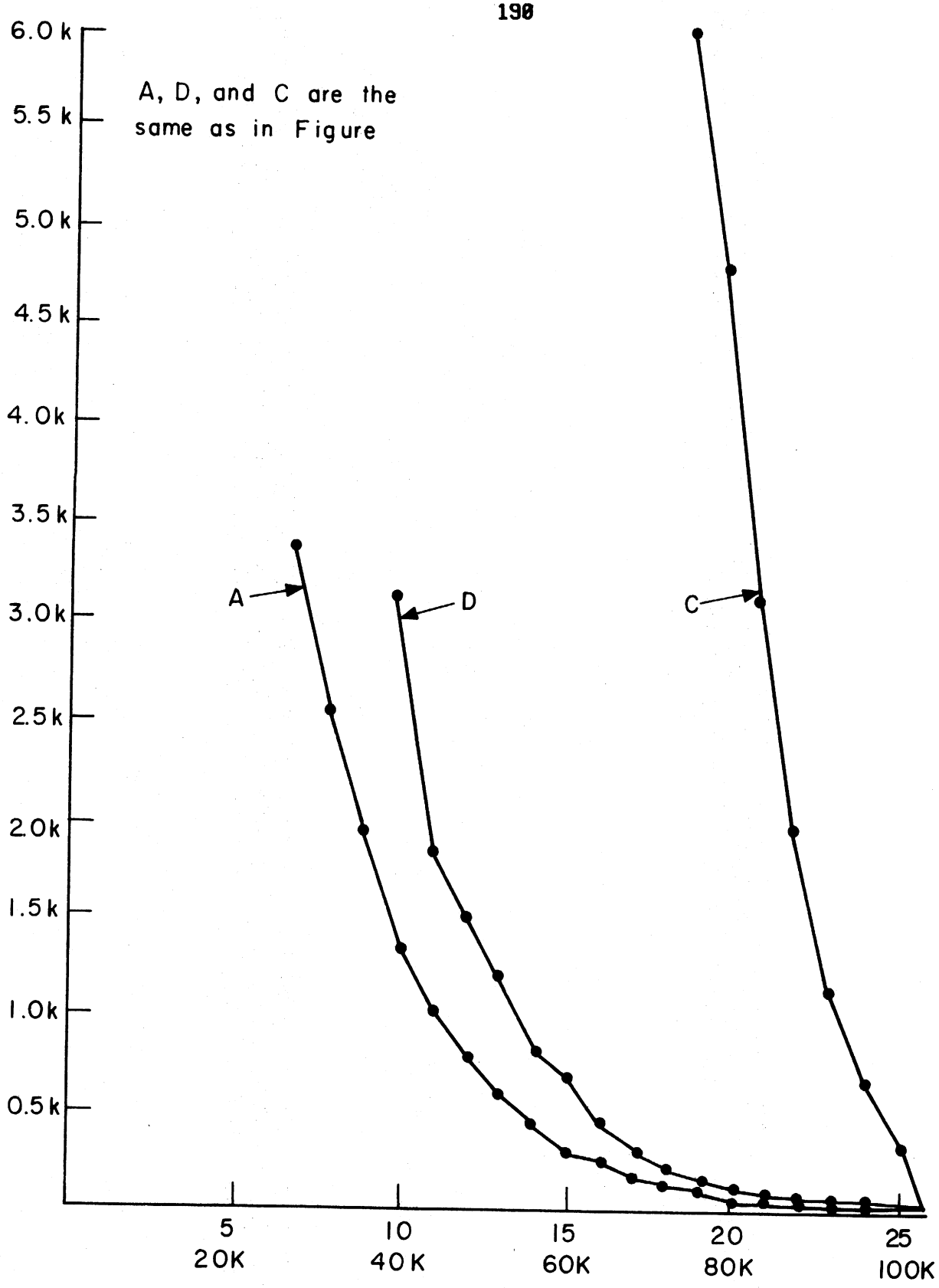


FIGURE 14 Enlarged Scale for Curves A, C, and D of Figure FOR PHASE I OF AED COMPILER

6.3 Restructuring Phase 2

Figure 15 shows the results of restructuring Phase 2 over sector trace SOT_2, where $|SOT_2|$=1,660,542. Table 4 precisely defines the curves of Figure 15. The bad order $\Pi x = B_2$ for Phase 2 is computed by the procedure BAD, which is compared to the order produced by Πy(W,T=2500,UANW). The curves of Figure 15 may be interpreted similarly to those of Figure 12 of Phase 1. The variation in the paging performance of Phase 2 as a function of program structure is not as large as that of Phase 1. However, the largest improvement in the paging performance of Phase 2 occurs when approximately one half of Phase 2 can fit into primary memory.

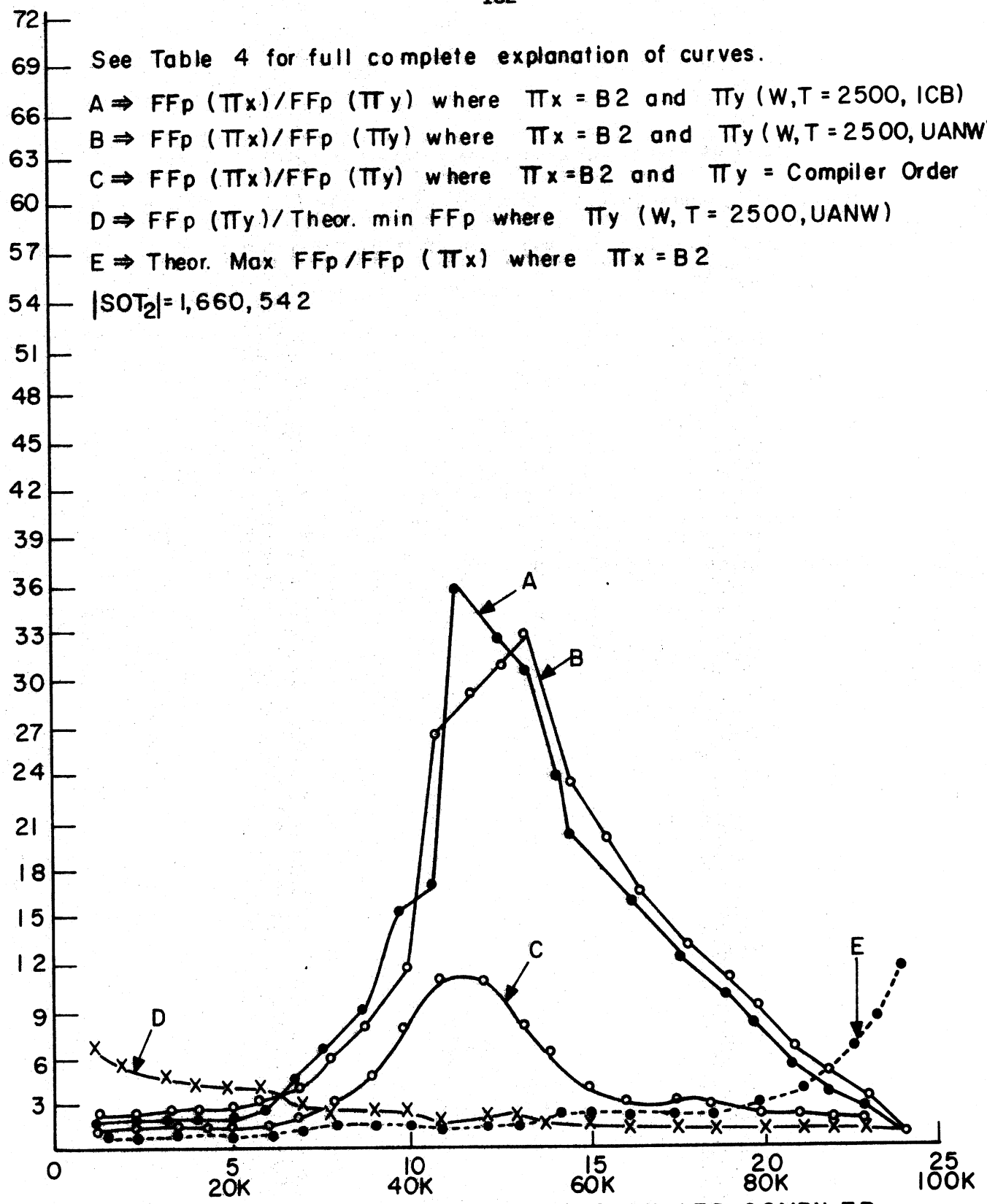


FIGURE 15 Page Fetch Ratios PHASE 2 OF AED COMPILER

Graph A is:

$FFp(|Mp|, N, \Pi x, SOT_2, Fd, R_{LRU}) / FFp(|Mp|, N, \Pi y, SOT_2, Fd, R_{LRU})$

Πx = B2 and Πy(W, T = 2500, ICB)

Graph B is:

$FFp(|Mp|, N, \Pi x, SOT_2, Fd, R_{LRU}) / FFp(|Mp|, N, \Pi y, SOT_2, Fd, R_{LRU})$

Πx = B2 and Πy(W, T = 2500, UANW)

Graph C is:

$FFp(|Mp|, N, \Pi x, SOT_2, Fd, R_{LRU}) / FFp(|Mp|, N, \Pi y, SOT_2, Fd, R_{LRU})$

Πx = B2 and Πy = Compiler Order

Graph D is:

$$FFp(|Mp|, N, \Pi y, SOT_2, Fd, R_{LRU}) / \frac{FFs(|Ms| = f_1(|Mp|, N, SS^*), SOT_2^*, Fd, Ro)}{f_1(2, N, SS)/2} - \Delta$$

Πy(W, T = 2500, UANW)

Graph E is:

$FFs(|Ms| = |Mp|, SOT_2, Fd, R_{LRU}) / FFp(|Mp|, N, \Pi x, SOT_2, Fd, R_{LRU})$

Πx = B2

Table 4

Parameters for Curves in Figure 15

6.4 Restructuring Phase 3

Phase 3 is restructured from a sector trace SOT_3 which contained 3,859,636 references. The program structure Πx is a random ordering of sectors into the virtual address space. Program structures

$\Pi y(W,T=2500,ICB)$, $\Pi y(W,T=2500,UANW)$ and

$\Pi y(U,D=20,ICB)$

produced substantial improvements in the paging performance over $\Pi x = B_3$. These improvements are illustrated in curves A, B, and C of Figure 16. These curves have the highest peaks of any improvements over sector orderings that we found. Curve D of Figure 16 shows the ratio of the paging performance obtained from Πx to the performance of the existing compiler ordering. The theoretical lower and upper bounds are presented in Figure 17 in the same manner as for Phase 1 and 2.

Now we present a few general comments about Phase 1, 2, and 3. All three phases indicate that significant variations in paging performance can occur for different arrangements of the relocatable sectors in virtual memory. The unconstrained clustering procedures, ICB and UANW, produced the best program performance over all memory sizes for all three phases. The constrained procedures are not shown for Phases 2 and 3 since they produced the same relative improvement in these phases as in Phase 1. The theoretical lower bounds are relatively good indicators of the best paging performance of all three phases for all but the smallest primary memory sizes.

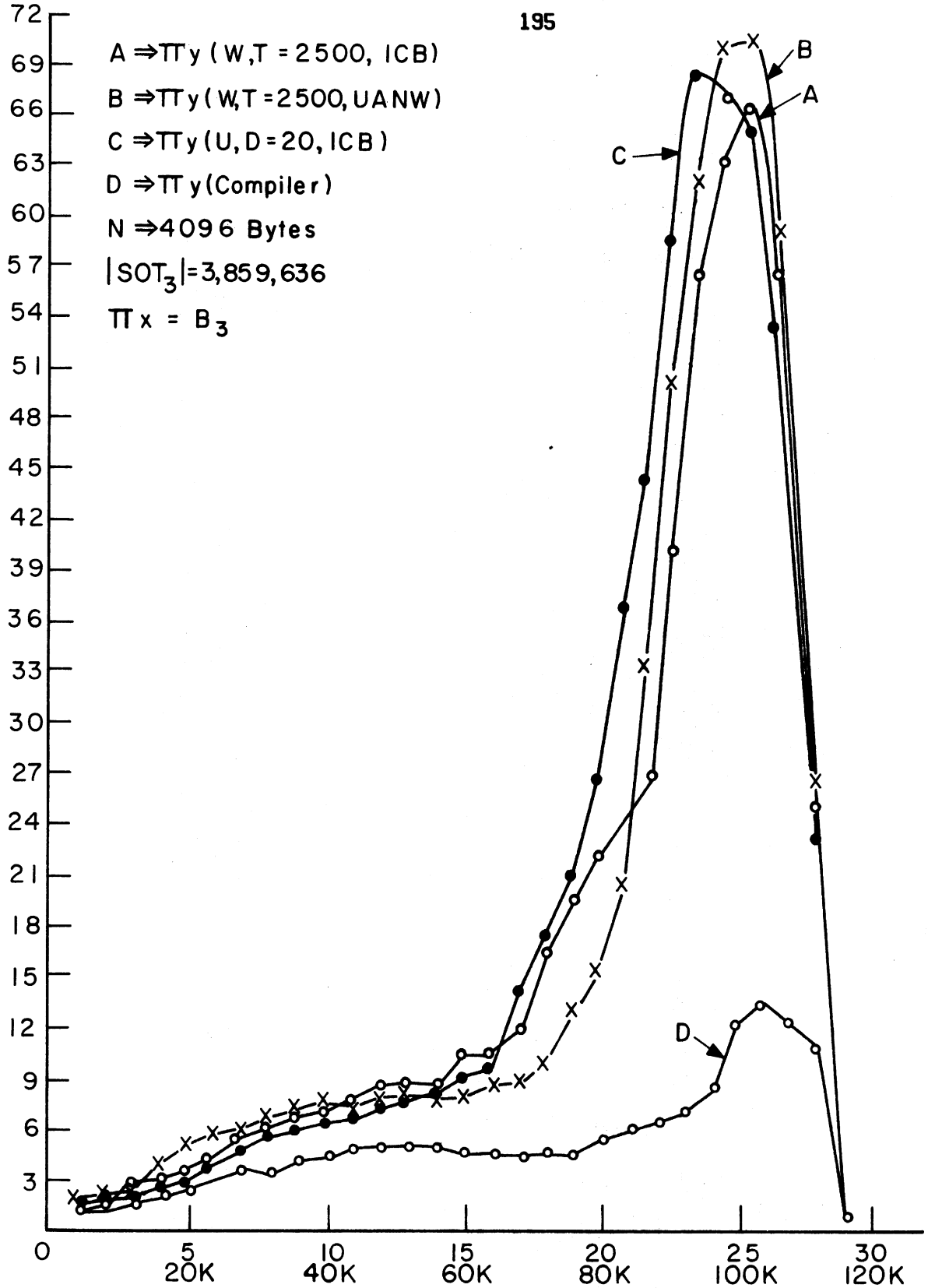


FIGURE 16 FFp (|Mp|, N, TTx, SOT_3, Fd, R_{LRU}) / FFp (|Mp| N, TTy, SOT_3, Fd, R_{LRU}) vs |Mp| FOR PHASE 3 OF AED COMPILER

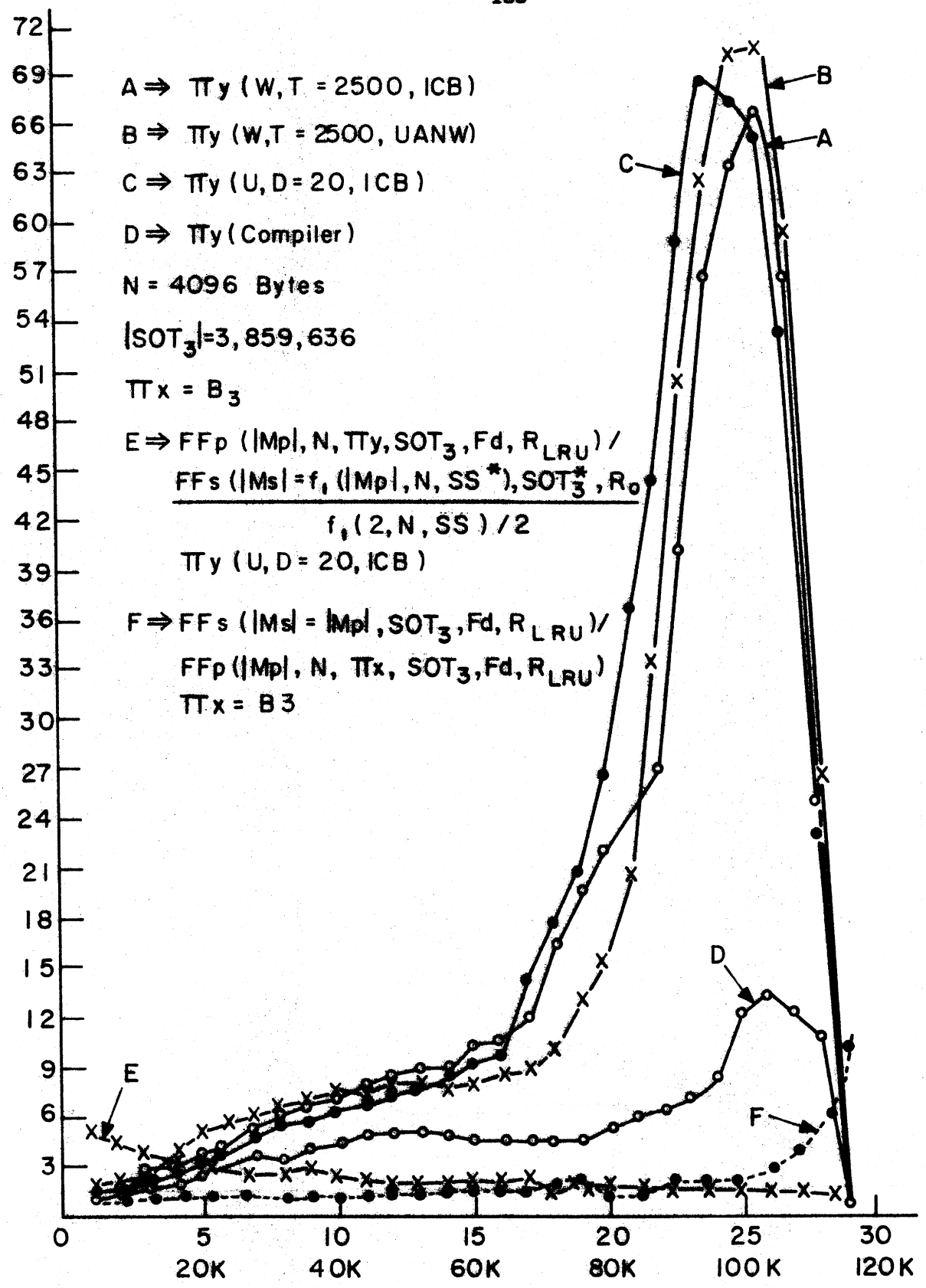


FIGURE 17 FFp (|Mp|, N, TTx, SOT_1, Fd, R_{LRU}) / FFp (|Mp|, N, TTy, SOT_1, Fd, R_{LRU}) vs |Mp| FOR PHASE 3 OF AED COMPILER

6.5 Effects of Input Data

In order to establish the effect that the input program to be compiled has on the paging performance, we conducted the following experiments:

Experiment 1:

A. We took the above sector trace SOT_1 and computed the program structure $\Pi y(W,T=2500,UANW)$.

B. We measured a second program trace $SOT_1 a$ which corresponds to a completely different program and restructured the compiler to get $\Pi ya(W,T=2500,UANW)$ based on $SOT_1 a$.

C. A third sector trace $SOT_1 b$ was measured, and, based on this sector trace, the program structure $\Pi yb(W,T=2500,UANW)$ was computed.

All three of the program structures, Πy, Πya and Πyb should tend to minimize the page fetches for the traces SOT_1, $SOT_1 a$, and $SOT_1 b$ respectively. However, will the structures specified by Πya or by Πyb tend to minimize the page fetches for SOT_1?

Figure 18 contains all the information shown in Figure 13 for Phase 1. That is, it shows the value of the page fetch function FFp for SOT_1 and Πy as curve D, and it shows the other curves of Figure 13 for visual comparison. Curve F in Figure 18 represents the value of FFp as a function of the same reference behavior SOT_1 and Πya. Curve G illustrates the value of FFp as a function of the same reference behavior SOT_1 and Πyb.

Therefore, the curves D, F, and G represent the paging performances of Phase 1 of the compiler for a single sector trace and three different partitions of sectors into clusters. The results of this experiment reveal that a good program structure generated from one sector trace is a good program structure for other sector traces.

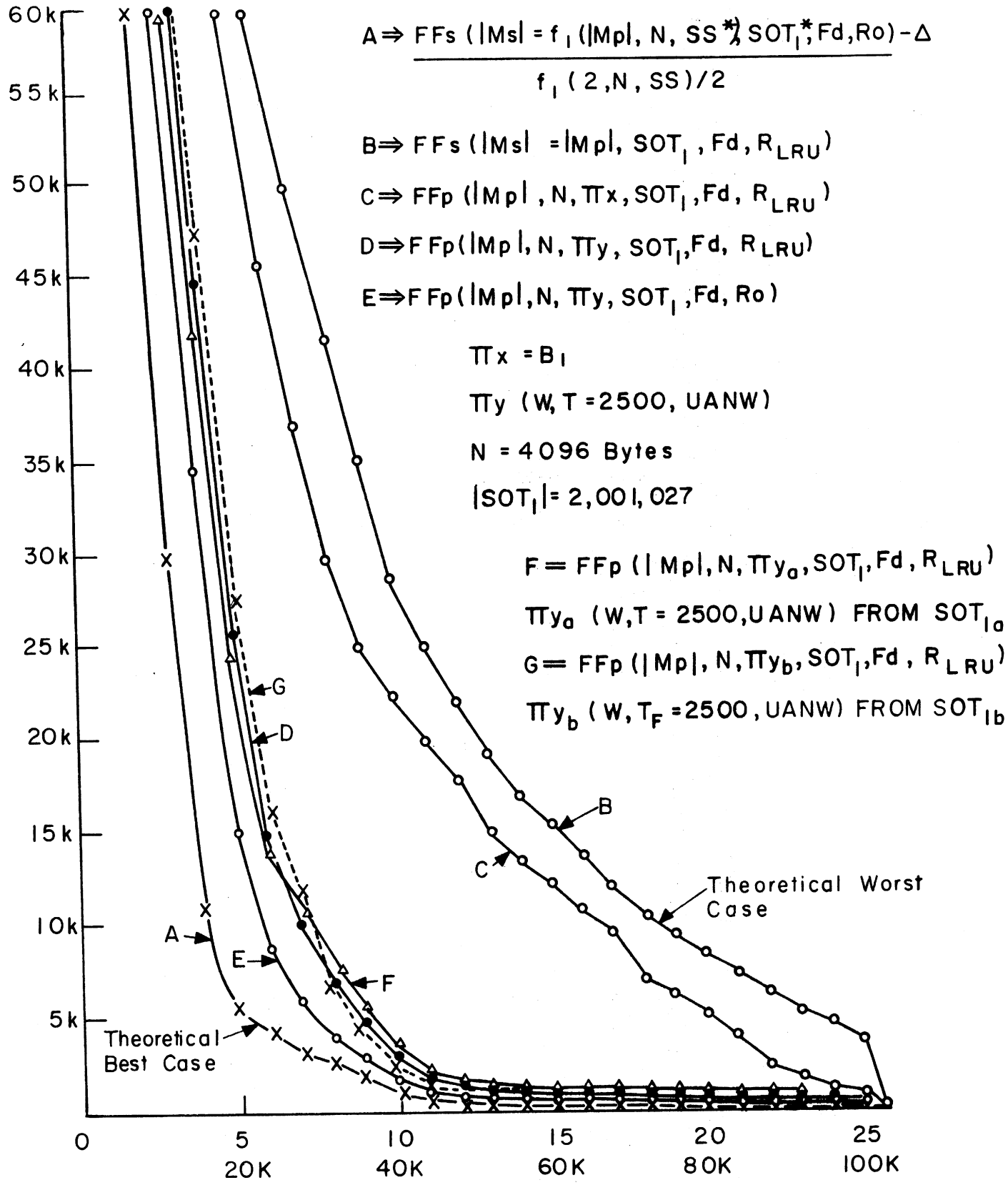
Experiment 2:

Now we give another experiment. For Πy, Πya and Πyb from the above experiment, we use the BAD procedure on each Π to get Πx, Πxa, and Πxb respectively. Then, using the same sector trace SOT_1, the following ratios are computed and plotted in Figure 19.

A. $FFp(..,\Pi x,SOT_1,..)/FFp(..,\Pi y,SOT_1,..)$

B. $FFp(..,\Pi xa,SOT_1,..)/FFp(..,\Pi ya,SOT_1,..)$

C. $Ffp(..,\Pi xb,SOT_1,..)/FFp(..,\Pi yb,SOT_1,..)$



$$A \Rightarrow \frac{FFs\,(|Ms| = f_1(|Mp|, N, SS^{*}, SOT_1^{*}, Fd, Ro) - \Delta}{f_1(2, N, SS)/2}$$

$B \Rightarrow FFs\,(|Ms| = |Mp|, SOT_1, Fd, R_{LRU})$

$C \Rightarrow FFp\,(|Mp|, N, \Pi x, SOT_1, Fd, R_{LRU})$

$D \Rightarrow FFp\,(|Mp|, N, \Pi y, SOT_1, Fd, R_{LRU})$

$E \Rightarrow FFp\,(|Mp|, N, \Pi y, SOT_1, Fd, Ro)$

$\Pi x = B_1$

$\Pi y\,(W, T = 2500, UANW)$

$N = 4096$ Bytes

$|SOT_1| = 2{,}001{,}027$

$F = FFp\,(|Mp|, N, \Pi y_a, SOT_1, Fd, R_{LRU})$

$\Pi y_a\,(W, T = 2500, UANW)$ FROM SOT_{1a}

$G = FFp\,(|Mp|, N, \Pi y_b, SOT_1, Fd, R_{LRU})$

$\Pi y_b\,(W, T_F = 2500, UANW)$ FROM SOT_{1b}

FIGURE 18 Total Page Fetches vs |Mp| FOR PHASE I OF AED COMPILER

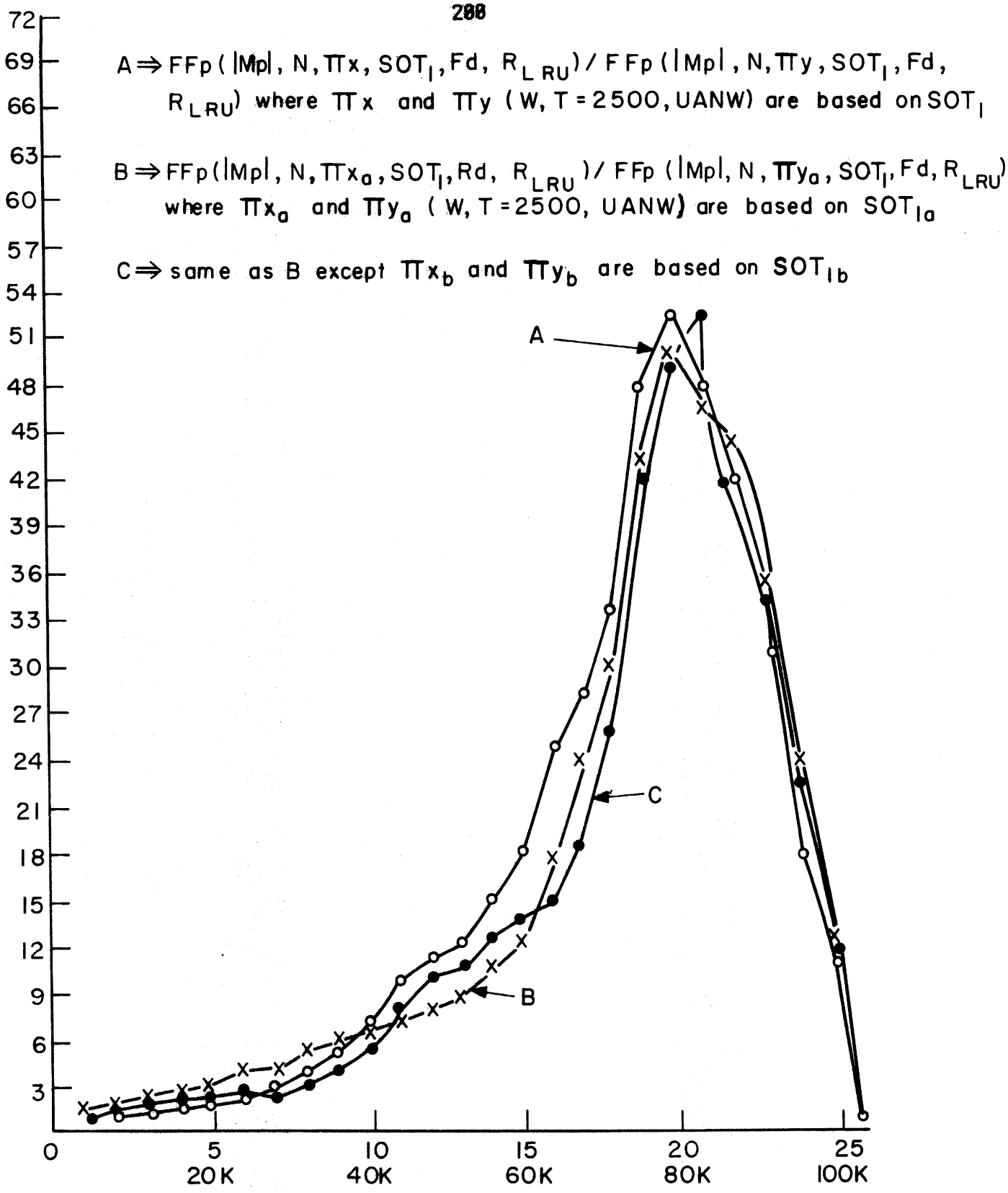


FIGURE 19 Comparison of Page Fetch Ratios for Different Program Structures FOR PHASE I OF AED COMPILER

These ratios are the improvements in paging performance over the same sector trace for three pairs of program structures. Each pair consists of a BAD structure and a good structure. Furthermore, each pair is constructed from a different sector trace. However, the possible improvement in paging performance for each pair is nearly the same.

Experiments 3 and 4:

Experiments 3 and 4 for Phase 2 and 3 respectively are quite similar to Experiment 1 for Phase 1. The only difference is that, in 3 and 4, the ratios of $FFp(..,\Pi y,SOT_2,..)/FFp(..,\Pi ya,SOT_2,..)$ and of $FFp(..,\Pi y,SOT_2,..)/FFp(..,\Pi yb,SOT_2,..)$ are plotted as shown in Figures 20 and 21 instead of the magnitude of these values of FFp shown in Figure 18. In Figure 18 it is difficult to distinguish between the three curves because of the scale problems. Figures 20 and 21 do away with the scale problems but do not show the relationship of these values to the overall picture as is done in Figure 18. From Figures 20 and 21 we observe that a good program structure computed from one sector trace turns out to be a good program structure for another sector trace.

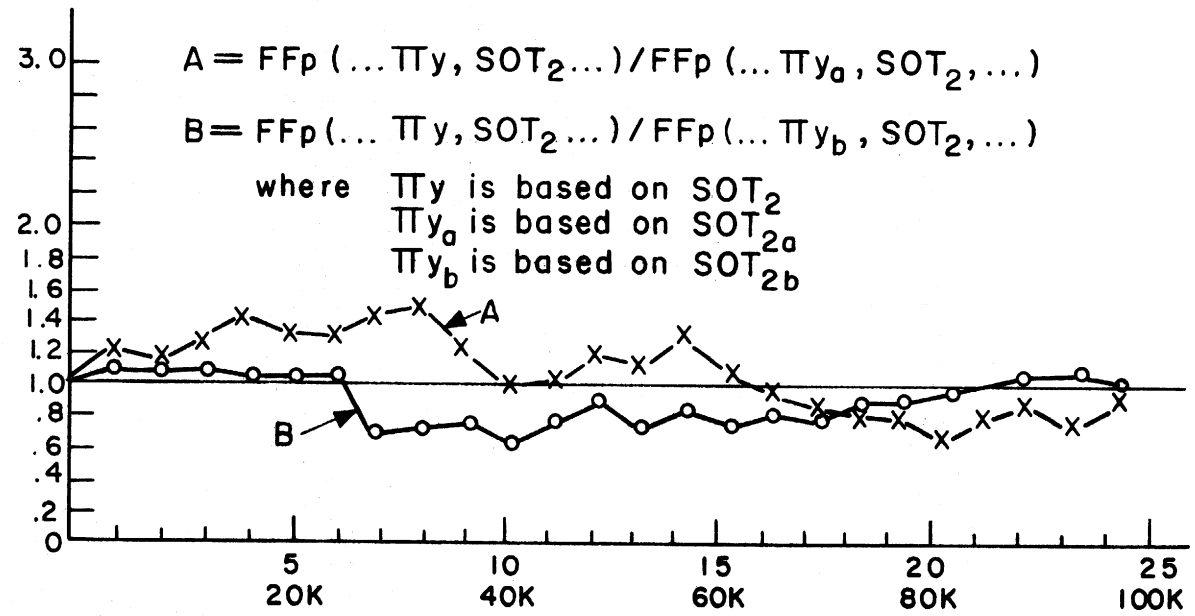


FIGURE 20 Ratios of Page Fetches For TT Based on Different Sector Traces FOR PHASE 2 OF AED COMPILER

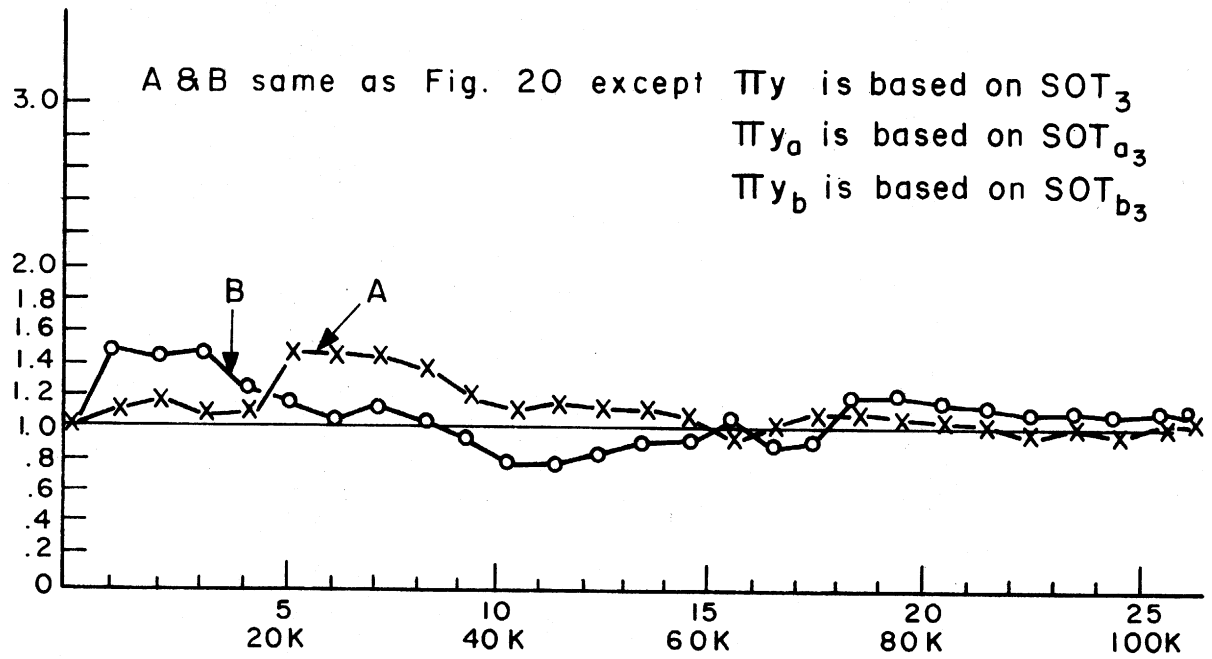


FIGURE 21 Ratios of Page Fetches For TT Based on Different Sector Traces FOR PHASE 3 OF AED COMPILER

CHAPTER 7

DISCUSSION AND CONCLUSION

7.1 Introduction

This report has presented theoretical and experimental results which show that program restructuring has a significant effect on the paging performance of virtual memory systems.

7.2 Summary

The problem of restructuring programs to improve their paging performance in virtual memory systems was presented in Chapter 1.

In Chapter 2 we formalized the notion of the page fetch function and the sector fetch function. The page fetch function models the paging behavior, and the sector fetch function models the sectoring behavior.

In Chapter 3 the sector fetch function was used to produce upper and lower theoretical bounds in the page fetch function over all

reorderings of the relocatable sectors into the address space.

Intersector reference models based on sector working sets and LRU stack distances were developed in Chapter 4. In Chapter 5 several clustering methods were developed which used the intersector reference models to produce a restructured program.

In Chapter 6 the effect of program restructuring on the paging performance of real programs was investigated empirically and theoretically. In particular, we showed that improvements in paging performance of factors of 20 to 40 is not uncommon for relatively large regions of primary memory size.

7.3 Further Work

The research reported in this report provides a basis for additional investigation in several areas of program restructuring.

The work described in this report addresses a problem that is as hard as the seemly intractable problems studied by Cook [C5] and Karp [K6]. Recent work by several people has revealed fast algorithms for near optimal solutions to some of these problems. The clustering techniques described in Chapter 5 have been shown of value for particular but not trivial examples that occur in practice. It would be

of considerable interest to know to what extent these techniques can be relied on over all possible sector traces. Can our techniques be shown to yield solutions that come within a factor of two of our lower bounds? If not, are there algorithms that do come near our lower bounds? Alternatively,can our lower bounds be improved?

We did not investigate the problem of sector duplication in this thesis. We claim that the results of Chapter 3 can be applied in a straightforward manner to produce lower bounds on the paging performance when sector duplication is allowed. Another related problem is how to incorporate sector duplication into the intersector reference models and into the clustering procedures.

Another area is the problem of deciding when it is best for sectors to cross page boundaries and when it is best to have holes in pages.

An ongoing research project between the author and Don Hatfield of IBM is to use the theoretical results of Chapter 3 to evaluate the potential benefit of reprogramming and then restructuring a very large data base system. This large data base system has sectors which are over 10 pages long. For example, Theorem 1 can be used to predict the theoretical best paging performance if the large data base system is broken up into k sectors per page. Thus, the problem is to determine the k that provides the best theoretical improvement and then use the magnitude of this improvement as a basis for deciding whether or not reprogramming is advisable.

REFERENCES

A1 Aho, A. V., P. J. Denning, and J. D. Ullman, "Principles of Optimal Page Replacement", Jour. ACM, Vol. 18, No. 1, Jan. 1971, pp. 80-93.

A2 Arora, S. R., and A. Gallo, "Optimal Sizing, Loading and Re-loading in a Multi-Level Memory Hierarchy System", AFIPS Conf. Proc., Vol. 38, 1971, pp. 337-344.

B1 Belady, L. A., "A Study of Replacement Algorithms for a Virtual-Storage Computer", IBM Systems Jour., Vol. 5, No. 2, 1966, pp. 78-101.

B2 Brawn, B. S., and F. G. Gustavson, "Program Behavior in a Paging Environment", AFIPS Conf. Proc., Vol 33, Part 2, 1968, pp. 1019-1032.

B3 Baer, J., and R. Caughey, "Segmentation and Optimization of Programs from Cyclic Structure Analysis", AFIPS Conf. Proc., Vol. 40, 1972, pp. 23-36.

B4 Baer, J., and G. R. Sager, "Measurement and Improvement of Program Behavior Under Paging Systems", in Statistical Computer Performance Evaluation, ed. by W. Freiberger

(proceedings of a conference held at Brown University, Nov. 1971), Academic Press, New York, N.Y., 1972, pp 241-246.

B5 Brawn, B. S., F. G. Gustavson, and E. S. Mankin, "Sorting in a Paging Environment", Comm. ACM, Vol. 13, No. 8, Aug. 1970, pp.483-494.

B6 Belady, L. A., and F. P. Palermo, "On-line Measurement of Paging Behavior by the Multivalued MIN Algorithm", IBM Jour. Res. Develop., Vol. 18, No. 1, Jan. 1974, pp. 2-19.

C1 Coffman, E. G., and L. C. Varian, "Further Experimental Data on the Behavior of Programs in a Paging Environment", Comm. ACM, Vol. 11, No. 7, July 1968, pp. 471-474.

C2 Comeau, L. W., "A Study of the Effect of User Program Optimization in a Paging System", ACM Symp. on Operating System Principles, Gatlinburg, Tenn., 1967.

C3 Charney, H. R. and D. L. Plato, "Efficient Partitioning of Components"Proc. SHARE/ACM/IEEE Design Automation Workshop, Washington, D. C., July 1968, paper no. 16.

C4 Corbato, F. J., "A Paging Experiment With the Multics System", In Honor of Philip M. Morse, edited by H. Feshbach and K. U.

Ingard, MIT Press, Cambridge, Mass., 1969, pp. 217-228.

C5 Cook, S.A., "The Complexity of Theorem-Proving Procedures", Proc. of Third Annual ACM Symp. on Theory of Computing, 1971, pp. 151-158.

D1 Denning, P. J., "The Working-set Model for Program Behavior", Comm. ACM, Vol. 11, No. 5, May 1968, pp. 323-333.

D2 Denning, P. J., "Virtual Memory", Computing Surveys, Vol. 2, No. 3, Sept. 1970, pp. 153-190.

D3 Denning, P. J., "On Modeling Program Behavior", AFIPS Conf. Proc., Vol. 40, 1972, pp. 937-944.

F1 Ferrari, D., "A Tool for Automatic Program Restructuring," Proc. ACM Ann. Conf., Aug. 1973, pp. 228-231.

G1 Guertin, R. L., "Programming in a Paging Environment", Datamation Vol. 18, No. 2, Feb. 1972, pp. 48-55.

G2 Gilmore, P. C., and R. E. Gomory,"The Theory and Computation of Knapsack Functions", Operations Res., Vol. 14, 1966, pp. 1045-1074.

H1 Hatfield, D. J. and J. Gerald, "Program Restructuring for Virtual Memory", IBM Systems Jour., Vol. 10, No. 3, 1971, pp. 168-192.

H2 Hatfield, D. J., "Experiments on Page Size, Program Access Patterns and Virtual Memory Performance", IBM Jour. Res. Develop., Vol. 16, No. 1, January 1972, pp. 58-66.

I1 Informatics, Inc., "Experiments in Automatic Paging", Report RADC-TR-71-231, Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, New York, Nov. 1971.

J1 Jensen, P. A., "Optimum Network Partitioning", Operations Res., Vol. 19, 1971, pp.916-932.

J2 Jarvis, R. A., and E. A. Edward, "Clustering Using a Similarity Measure Based on Shared Near Neighbors", IEEE Trans. on Computers, Vol. C-22, No. 11, November 1973, pp. 1025-1034.

K1 Kernighan, B. W., "Optimal Sequential Partitions of Graphs",Jour. ACM, Vol. 18, No. 1, Jan. 1971, pp. 34-40.

K2 King, W. F., III, " Analysis of Demand Paging Algorithms", Proc. IFIP Congress, TA-3, August 1971, pp. 485-490.

K3 Kuehner, C. J. and B. Randell, "Demand Paging in Perspective", AFIPS Conf. Proc., Vol. 33, Part 2, 1968, pp. 1011-1018.

K4 Kernighan, B. W., "Some Graph Partitioning Problems Related to Program Segmentation", Ph.D. Thesis, Princeton Univ., Princeton, N. J., Jan. 1969, 177 pp..

K5 Kernighan, B.W., and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs", The Bell System Technical Journal, Vol. 49, No. 2, Feb. 1970, pp. 291-308.

K6 Karp, R. M., "Reducibilities Among Combinatorial Problems", Complexity of Computer Computations, edited by R. E. Miller and J. W. Thatcher, Plenum Press, 1972, pp. 85-103.

L1 Lowe, T.C., "Automatic Segmentation of Cyclic Program Structures Based on Connectivity and Processor Timing", Comm. ACM, Vol. 13, No. 1, Jan. 1970, pp. 3-6.

L2 Lewis, P. A. W. and P. C. Yue, "Statistical Analysis of Program Reference Patterns in a Paging Environment", Proc. IEEE International Computer Society Conference, Sept. 1971, pp. 133-134.

L3 Lew, A., "On Optimal Pagination of Programs", University of Hawaii Information Sciences Report, Honolulu, Hawaii, 1970.

L4 Luccio, F., and M. Sami, "On The Decomposition of Networks in Minimally Interconnected Subnetworks", IEEE Trans. on Computers, Vol. Ct-16,pp. 184-188, May, 1969.

L5 Lukes, J. A., "Combinatorial Solutions to Partitioning Problems", STAN-CS-72-293, Stanford University, May 1972, 130 pp.

L6 Ling, R.F., "On the Theory and Construction of K-Clusters,"

M1 Mattson, R. L., J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies", IBM Systems Jour., Vol. 9, No. 2, 1970, pp. 78-117.

M2 McKellar, A. C., and E. G. Coffman, "Organizing Matrices and Matrix Operations for Paged Memory Systems", Comm. ACM, Vol. 12, No. 3, March 1969, pp. 153-164.

M3 Madnick, S. E., "Storage Hierarchy Systems", MIT Project MAC Report MAC-TR-107, Massachusetts Institute of Technology, Cambridge, Mass., April 1973, 155 pp.

M4 Madnick, S. E. and J. J. Donovan, "Operating Systems", McGraw-Hill, New York, 1974.

M5 McCormick, J, W. T., Jr., et al., "Problem Decomposition and Data Reorganization by a Clustering Technique", Operations Res., Vol. 20, 1972, pp. 993-1009.

M6 Masuda, T., et al., "Optimization of Program Organization in Virtual Storage Systems by Cluster Analysis", unpublished working paper, 1974.

M7 Miyamoto, I., "Data Reference Characteristics of Database Application Program", Nippon Electric Company, Limited, Fuchu, Tokyo, unpublished working paper.

P1 Pratt, V. R., "An N LOG N Algorithm to Distribute N Records Optimally in a Sequential Access File", Complexity of Computer Computations, edited by R. E. Miller and J. W. Thatcher, Plenum Press, 1972, pp. 111-118.

R1 Ramamoorthy, C. V., "The Analytic Design of a Dynamic Look Ahead and Program Segmenting System for Multiprogrammed Computers", Proc. ACM National Conf., 1966, pp. 229-240.

S1 Saltzer, J. H., "A Simple Linear Model of Demand Paging Performance", MIT Project MAC Report in progress.

S2 Spirn, J. R., and P. J. Denning, "Experiments with Program Locality", AFIPS Conf. Proc., Vol. 41, Part 1, 1972, pp. 611-622.

S3 Smith, J. L., "Multiprogramming Under a Page on Demand Strategy", Comm. ACM, Vol. 10, No. 10, Oct. 1967, pp. 636-646.

T1 Tsao, R. F., L. W. Comeau, and B. H. Margolin, "A Multi Factor Paging Experiment 1: The Experiment and the Conclusions", in Statistical Computer Performance Evaluation, ed. by W. Freiberger (proceedings of a conference held at Brown University, Nov. 1971) Academic Press, New York, pp. 103-134.

V1 Varian, L. C., and E. G. Coffman, "An Empirical Study of the Behavior of Multi-programming".

V2 Ver Hoef, E. E., "Automatic Program Segmentation Based on Boolean Connectivity", AFIPS Conf. Proc., Vol. 38, 1971, pp. 491-496.

This empty page was substituted for a blank page in the original document.

CS-TR Scanning Project
Document Control Form

Date : 11 / 16 / 95

Report # LCS-TR-148

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 228 (232 IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:

☐ Single-sided or
☒ Double-sided

Intended to be printed as :

☐ Single-sided or
☒ Double-sided

Print type:

☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number): _____

Photographs/Tonal Material (by page number): _____

Other (note description/page number):

Description : Page Number:

IMAGE MAP: (1-228) UNNUMBERED TITLE & BLANK PAGES, i-ix,
UNNUMBERED BLANK, 1-53, UNNUMBERED BLANK,
54-142, UNNUMBERED BLANK, 55-213, UNNUMBERED BLANK.
(229-232) SCANCONTROL, TRGT'S (3)

Scanning Agent Signoff:

Date Received: 11/16/95 Date Scanned: 12/18/95 Date Returned: 12/21/95

Scanning Agent Signature: Michael W. Cook

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 12/18/1995

M.I.T. Libraries
Document Services