AN IMPROVED CLUSTERING METHOD FOR PROGRAM RESTRUCTURING

This thesis is submitted in conformity
with the requirements for the
degree of Master of Science
in the School of Computer Science
McGill University
May 1981

Abstract

The internal organization of a program can greatly affect the performance of virtual memory systems. There exist many algorithms that reorganize programs to improve their paging performance. This thesis examines several of these methods and proposes a new approach that has produced better experimental results. As well, a brief examination is made of the effect of restructuring on a program that executes on a system with prepaging memory management.

Resumé

L'organisation interne d'un programme peut affecter considérablement le fonctionnement des systèmes de mémoire virtuelle. Il y a beaucoup d'algorithmes qui réorganisent des programmes pour améliorer leur fonctionnement de pagination. Cette thèse examine plusieurs de ces méthodes et propose une nouvelle approche qui a produit de bons résultats expérimentaux. Il y a aussi une brève discussion des effets de la réorganisation d'un programme qui exécute dans un système où la gestion de mémoire se fait avec prépagination.

Acknowledgements

Most of all, I would like to thank my advisor, Prof. Nigel Horspool, for suggesting this thesis topic. His ideas, suggestions, guidance and patience were essential to the success of this research and were greatly appreciated. I also wish to express my appreciation to two fellow graduate students for their contributions. The programming expertise of Larry Dunkelman was an invaluable aid in preparing many of the algorithms. As well, the many hours of stimulating discussion with Ron Huberman were helpful in the formulation of concepts and produced some significant ideas. Finally, I would like to thank my parents for their support and encouragement.

Table of Contents

List of Illustrations

Chapter 1 - Introduction

The influence of program organization on the performance of virtual memory systems is now well known. Over the years, numerous algorithms have been introduced with the goal of restructuring programs so that they run more efficiently. These algorithms usually consist of two phases. The first phase determines the strength of interconnectivity between program sections, while the second phase uses this information to regroup these sections. Although most algorithms focus on the treatment of the first phase of processing, the regrouping is of considerable importance as well.

This thesis will examine program restructuring as a whole with particular attention being paid to the regrouping or clustering problem. A new approach to this problem will be presented along with experimental evidence of its performance. As well, a brief examination of prepaging for restructured programs is made and experimental results discussed.

1.1 Organization of Thesis

Chapter 1 is an introduction to virtual memory systems. A few major concepts including address translation, paging and memory management are presented along with the necessary terminology.

Chapter 2 provides an in-depth analysis of program restructuring. Important algorithms such as those by Hatfield and Gerald, Ferrari and Masuda are described. A general

discussion of clustering and sequencing techniques is also included.

Chapter 3 briefly discusses predictive memory management. Major policies by Joseph, Baer and Sager, Trivedi and Burris are examined.

Chapter 4 presents a critical analysis of existing restructuring algorithms and introduces a new modified approach. This approach is outlined in detail. Experimental results involving simulations of Ferrari's method compared to the new technique are also presented.

Chapter 5 discusses the use of prepaging to enhance the efficiency of restructured programs.

1.2 Introduction to Virtual Memory

One of the major trends in computers has been towards more sophisticated operating systems. This sophistication lies not only in the degree of multiprogramming or the rate of program throughput but in the ability to disassociate the programmer from the computer hardware. By acting as a program-computer interface, these operating systems have freed the programmer from many tedious tasks. An important advancement in this area was the introduction of Virtual Memory (VM).

To properly discuss VM, it is necessary to distinguish the concepts of "address space" and "physical space". An address space is the set of identifiers that may be used by a program to reference information, while a physical space consists of the set of main memory locations in which program information may be stored [DENN70]. Virtual Memory is thus an address space whose size is independent of the physical space. To a programmer, VM can give the illusion that a memory space, much larger or smaller than is physically present, exists.

In early computer systems, the address space and physical space were synonymous, that is, a one-to-one mapping could be made. Once VM was introduced the mapping concept had to be altered since the entire virtual address space could not always be mapped into physical memory at one time. Another factor in this mapping arises when multiprogramming is present. In this environment, there may exist many programs, each with their own virtual memory space, but only one set of physical memory locations. The operating system must not only map virtual addresses into real addresses but in those instances where

program code is not shared among different programs, it must maintain an isolation between these real addresses.

The mapping of addresses is usually referred to as address translation and is commonly performed by a mapping table. A mapping table can be implemented as a linear array that is indexed by a virtual address and where each element is a real address. Theoretically, every single virtual address that is referenced could be mapped into any physical location. In other words, the translated addresses do not need to be consecutive real addresses. The difficulty in this approach is that the mapping table would have to be as large as the physical memory. One method which circumvents this problem but still allows mapping flexibility is "paging".

Paging is a memory management technique in which the virtual memory space is divided into identical fixed-size sections (typically 256-2K words) called "pages" and the physical memory divided into matching-sized sections called "page-frames". The translation process is thus reduced to a mapping of pages into page-frames. This process is performed as follows. The virtual address is divided into two sections where the high-order bits indicate a page number and the low-order bits denote the offset within the page. The page number is used to index a page map table which contains page-frame numbers. The appropriate page-frame number is then recombined with the offset to obtain the real address (Fig. 1).

In a multiprogramming environment it is often desireable to protect against one program invalidly accessing the memory space of another. This protection can easily be accomplished by adding

access control bits to the page map table (or by having seperate page map tables for the different programs). Another important benefit of paging arises when memory requests and releases occur assynchronously. In the case of multiprogramming systems, for example, programs continuously enter and leave the system, yet the free memory space is always in multiples of the page size. The cost of this feature is that programs which are not exact multiples of the page size will waste some memory on the last page. This wastage (<1 page/program), however, is less than what can occur to main memory without some form of paging.

To further discuss paging, it is necessary to introduce certain concepts and terminology. As previously mentioned, only a fraction of VM may be capable of being mapped into real memory. Those pages that have been mapped are said to be "resident" and the group of resident pages is called the "resident set". Inevitably, a resident page will attempt to reference an address of a non-resident page that is being held in secondary storage (i.e. disk, drum, tape, etc.). A reference of this type is called a "page fault" or "page exception". Before program execution can be continued, the missing page must be located and copied into a page-frame. This change to the page map is referred to as "loading", "pulling" or "fetching". If pages are loaded whenever they are required then main memory would most certainly be filled after a short period of time, thus it is necessary to regularly remove certain pages and return them to secondary storage. This removal is called page "replacement", "pushing" or "discarding".

It is the responsibility of the operating system to control the movement of pages from secondary to primary memory. Paging decisions are based on predetermined policies where certain parameters may be dynamically altered in an effort to "fine-tune" the system performance. Some of the decisions that must be made are: selecting which page(s) should be loaded, determining how much main memory should be allocated to each program and choosing which page(s) should be discarded.

The loading of pages falls under one of two categories: demand or non-demand paging. Demand paging systems, as the name implies, load only the page that is requested. On the other hand, non-demand paging systems are usually predictive, in that they attempt to load one or more pages at the same time to try to avoid future page faults. Predictive memory management techniques are discussed in greater detail in chapter 3.

The discarding of pages and the allocation of main memory is handled by a page replacement policy. Policies that maintain a fixed number of page-frames for each program are called "fixed-partition" replacement policies, while those that permit the number of page-frames to vary throughout the lifetime of the program are called "variable-partition" replacement policies. Variable-partition policies are used in multiprogrammed environments where each program's allocation of page-frames may vary. At any time, the system can increase or decrease the number of page-frames allocated to an individual program. By sharing the available page-frames among many programs and by redistributing the memory allocations dynamically, an acceptable system performance can be maintained.

1.2.1 Fixed-Partition Replacement Policies

There exist numerous fixed-partition replacement policies. One of the simplest and easiest to implemement is called First-In First-Out (FIFO). When a page fault occurs and there are no free page frames, the page that was loaded first, that is the oldest resident page, is discarded and the space filled with the requested page. FIFO can be modelled as a bounded-length queue, where new pages enter at the tail and old pages are discarded from the head.

Another popular policy is called Least-Recently Used (LRU). When a page fault occurs, the page that has been unreferenced for the longest time is removed. Least-Frequently Used (LFU) is a policy that discards the page that was referenced the fewest number of times since the program started. As both LRU and LFU require some sort of hardware counter for each page-frame their implementation can be costly compared to other policies.

An inexpensive policy that requires only one bit per page is called Use-Bit. Whenever a page is referenced, its special bit is set. At a fault, Use-Bit clears the bits that are set and discards the first page encountered that has not been referenced since the last fault (bit='0'). If all the pages have been referenced then one is selected at random. A variation on the Use-Bit method is called Second-Chance or Clock. In this method a cyclic pointer is maintained when looking for a zero bit. At each fault, the search is resumed from the position of the last load.

A detailed analysis will reveal that certain policies are better suited to certain program structures. Unfortunately,

program flow is not always predictable and in many cases it will change in nature as the program executes. The selection of a replacement policy thus depends upon the cost of implementation and its capability to handle a wide range of programs with an acceptable fault rate.

1.2.2 Reference Strings and Windows

Before proceeding to variable-partition replacement policies, it is necessary to introduce two concepts that are extensively used. It is possible during the execution of a program to monitor the references made to various pages. If these references are accumulated then the activity of the program can be viewed as a sequence of page references. As a simplifying assumption, it is often assumed that the time interval between references is constant; thus the index of a reference in the sequence can be used as a time measure.

For a program consisting of n pages, let r(t) denote the page referenced at time t, where $1 \leq r(t) \leq n$. Then the page reference string, R, is defined by:

$$R = r(1)\ r(2)\ \ldots\ r(k)\ \ldots$$

In certain instances it is desireable to examine only a small section of the reference string. This can be accomplished by using a "window". A window is an interval of time (or a fixed number of references) over which observations can be made. That is, for a window of size T, at time t, the segment of the reference string that can be examined by looking backwards is:

$$r(t-T+1)\ r(t-T+2)\ \ldots\ r(t-1)\ r(t).$$

Windows that look forward are often used in the analysis of optimal strategies.

1.2.3 Variable-Partition Page Replacement Policies

One important characteristic of programs that is essential in constructing good page replacement policies is "locality" [DENN70]. Locality can be described as the phenomenon whereby a program strongly favours a slowly changing subset of its pages (locality set) over an extended period of time. While this subset changes, its size may also vary, indicating that the memory allocated to a program should fluctuate as well. This is the principle behind variable-partition paging. With a variable-partition policy, pages are added to or removed from a program's resident set, so that an acceptable page fault rate is maintained. Fixed-partition policies have, in general, a disadvantage over variable-partition policies as their fixed memory allocations may restrict the number of pages of a locality set that can be resident.

There are two well-known variable-partition replacement policies: Working Set (WS) and Page Fault Frequency (PFF). Both policies employ a window.

The concept of a working set was defined by Denning in 1966 [DENN66] as the set of pages used during the most recent interval of time. This interval of time is defined by a moving window whose fixed size, T, determines both the size and contents of the working set. At any given time, t, the working set, W, consists of those pages referenced in the interval $[r(t-T+1), r(t)]$.

i.e. $W(t,T) = \{i \in N \mid$ page i appears among $r(t-T+1)...r(t)\}$

where $T \geq 1$ and N is the set of pages allocated to the program.

The Working Set policy is a memory management policy that operates according to the following rules. A program may run if and only if its working set is in main memory and a page may not be removed if it is the member of a working set of a running program. In other words, a page fault can occur when a reference to a page that is not a member of the working set is made. The missing page is loaded into memory and is added to the working set. When a page is not referenced for a period of time greater than the window size, it ceases to be a member of the working set and can be returned to secondary storage.

By lengthening T, the lifetime of pages in the working set is increased as is the average working set size. Smaller values of T decrease the average working set size but at the expense of a greater number of page faults. Thus, fine-tuning of the window size can result in an acceptable compromise between the fault rate and the average memory allocation.

In 1972, Chu and Opderbeck [CHU72] introduced an easily implementable replacement policy called Page Fault Frequency. An acceptable page fault interval, T, is selected as a target. At a page fault, if the time interval since the last fault is less than or equal to T, the missing page is loaded. If the interval is greater than T, all those pages not used since the last fault are discarded and the missing page retrieved. PFF can be implemented with an interval timer and one use-bit per page. It is easily incorporated into most existing computer systems.

Experimental evidence indicates that PFF and NS perform equally well, however, PFF is more sensitive to its control parameter and can display fault rate anomalies for certain programs [GUPT78, FRAN78].

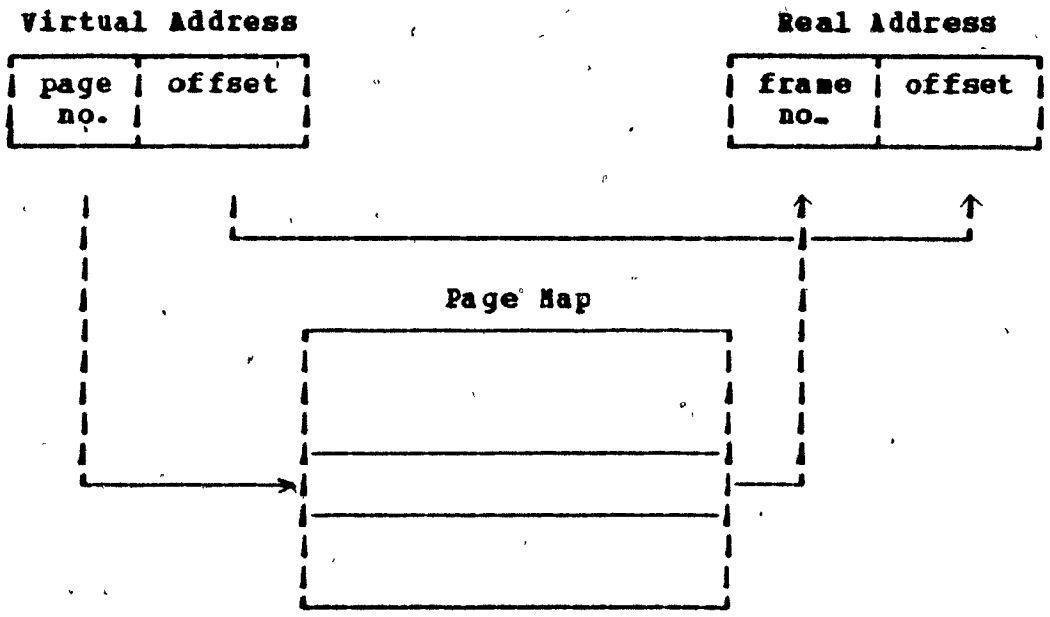


Figure 1a - Address Translation

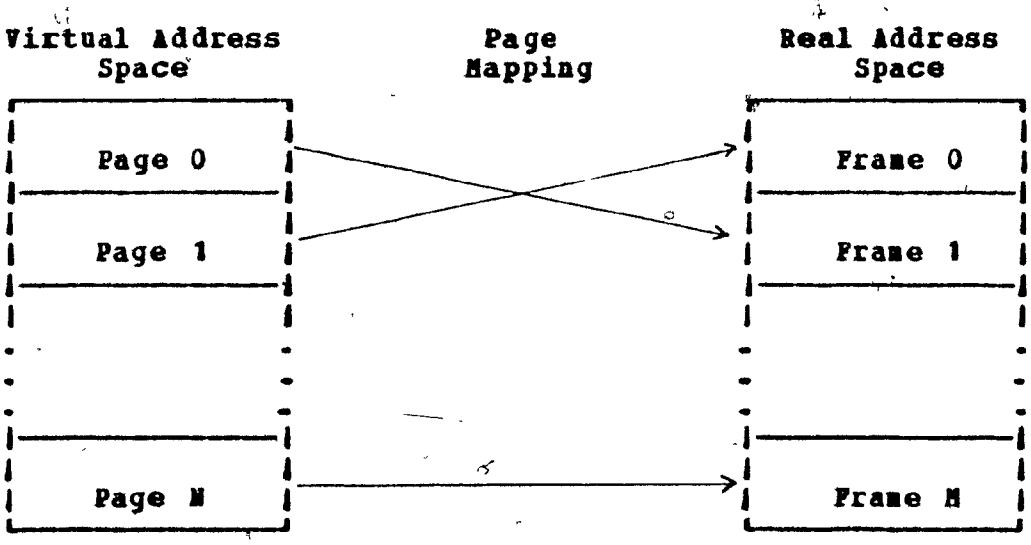


Figure 1b - Page to Frame Mapping

Chapter 2 - Program Locality and Restructuring

As previously discussed, the concept of program locality is the basis for the working set model. The locality exhibited by a program at any given time can stem from a concentration of instruction references in a subroutine, data references in a common storage block or from loops that cover a small set of pages. The working set principle attempts to improve paging performance by allowing groups of pages that constitute a locality set to reside together in memory. The locality set of a program does not, however, have to imply groups of pages but can include smaller units. If a program can be viewed as a collection of independent and relocatable blocks (modules) whose sizes are much smaller than the page size, then the principle of locality indicates that an "intelligent" placement of these blocks onto pages should also improve paging performance. This is the motivation for program restructuring.

In 1967, Comeau performed an experiment to demonstrate the effect that program ordering had on page faults [COME67]. He compared the page exceptions generated when blocks were ordered:

1) alphabetically,
2) randomly,
3) using a knowledge of the block functions and page size,
4) using a knowledge of the blocks and a record of paging performances generated while the job was in execution.

The last reordering showed a 5:1 improvement over the worst case alphabetical ordering and a 2:1 improvement over the second-best intelligent ordering.

McKellar and Coffman, in a study of matrices, showed that considerable improvements could be achieved by proper assignment of matrix elements to pages and by better organization of matrix operations [MCKE69]. Tsau and Margolin performed an analysis of factors that affect paging faults and showed that the ordering of subroutines was more significant than the replacement algorithm used [TSAU72].

Although the benefits of good program structure are now well known, the methods to achieve this goal remain diverse. The obvious solution to the program restructuring problem is to insist that programs be written with their dynamic behavior as local as possible. Several papers have been written to this end [MCKE69, BRAN70], yet it is in general difficult to keep track of the dynamics of normal size programs. In fact, the concept of virtual memory provides little motivation for program locality. The abundance of existing, poorly structured programs is another consideration which must be faced. An alternate approach is to make locality decisions at the compiler and/or loader level. This would imply that the transfer probabilities between subroutines or procedures be known to the compiler. In a modular programming environment, where independent compilation of routines can occur, this interconnection information is not necessarily available.

The most widely accepted forms of program restructuring use special "automatic" algorithms that operate on the entire collection of blocks, and whose only purpose is to assign blocks to pages. Programs can be modelled using a directed graph, whereby the nodes and edges represent modules and

interconnections respectively. Associated with each edge is a numerical value denoting the strength of interconnection. The goal of program restructuring is to group nodes so that the interconnection between different groups is minimized, provided that the groups do not exceed the page size.

Formally, the restructuring process involves four distinct steps. The preliminary step is the division of the program into blocks which are relocatable in virtual memory. For restructuring to be effective, the block sizes should be small, typically one-tenth to one-third the page size [HATF71]. The second step constructs the graph of the program as previously described. It is the assignment of weights to the graph edges which differentiates most restructuring algorithms. The third step takes as input the program graph and using a clustering algorithm, forms clusters of blocks while trying to minimize intercluster connections. The final step assigns each cluster to a page in virtual memory. This last step assumes that clusters fit exactly onto a page, otherwise special consideration must be accorded to blocks that overlap page boundaries.

The second phase of the restructuring process is where the most attention has been paid. There are two classes of algorithms which are used to construct restructuring graphs: static and dynamic. Static algorithms are performed using a program's flow chart prior to execution and assigns weights according to interblock connectivity and cyclic characteristics [BAER72, VERH71]. Dynamic analysis is based on a block reference string generated during the execution of a program and weights are assigned according to statistics accumulated from the

references [HATF71].

There are advantages and disadvantages to both static and dynamic algorithms. The advantages of the static approach are that no reference data is required and that restructuring is performed independently of the data that the program operates on. The disadvantages to static algorithms are that they can be quite complex and experimentally have shown to produce only minor improvements and even anomalous results [FERR74a]. The advantages of the dynamic approach are that no knowledge of the program structure is required, apart from the blocks, and that experimental results have consistently shown significant improvements [HATF71, FERR74a, ACHA78]. The disadvantages of dynamic algorithms are that generating a reasonable length reference string can be expensive and that the program must be monitored while executing "typical" input data. The sensitivity to input data has been a major criticism against dynamic restructuring, however, numerous experiments [HATF71, FERR74a, ACHA78] have indicated that many types of input invariant programs exist. In general, dynamic restructuring algorithms have been shown to be a more profitable area for research.

2.1 Dynamic Restructuring Algorithms

There exists a plethora of dynamic restructuring algorithms, each gathering and interpreting a program's reference string in a different manner. Every algorithm claims some improvement over the unrestructured program yet it is difficult to compare these algorithms and arrive at a "best" one. The early algorithms that

were introduced used simple and easy to implement heuristics and obtained relatively good results. Later algorithms, called strategy-oriented, attempted to get greater improvements by tailoring the restructuring to specific replacement policies under which the restructured program would be run. Still others presented algorithms that provided good results, independent of the future replacement policy. Since the various algorithms are designed to operate on specific types of programs under specific replacement policies and since the published results are all based on experiments using different test programs, it is impossible to reach any global conclusions. The only observation that can be made is that the initial heuristics used in a restructuring algorithm provides the greatest improvements while any refinements require more and more effort to obtain smaller and smaller gains. It is up to the user to decide how much he is willing to pay for the expected benefits.

Dynamic restructuring algorithms base decisions on comparative interconnection strengths between blocks. Therefore, a data structure is necessary to maintain this information as the reference string is processed. The most common data structure used is an nxn interconnection matrix, where n is the number of blocks. According to the heuristics of the restructuring algorithm, elements in the array corresponding to block transfers are incremented. Since references within a block are of little restructuring interest, the diagonal of the matrix is normally zero. In algorithms which require symmetric matrices it is necessary to store only the upper or lower triangular matrix.

2.2 Simple Algorithms

Some very simple restructuring algorithms were introduced by Achard et al. [ACHA78] which make use of two fundamental concepts. The first method is based on the assumption that a program favors a main nucleus of blocks. This Nucleus Constructing Algorithm (NCA) places blocks in virtual memory in decreasing order of their densities, Di,

$$Di = (E(S)/Si) * Wi,$$

where E(S) is the mean size of all blocks, Si the size of block i and Wi the weight of block i. The weight Wi is the number of times block i was involved in a page fault by referencing an absent page or being referenced in an absent page. The first few pages of an NCA reconstructed program contains the nucleus.

A second algorithm attempts to reduce page faults by ensuring that blocks do not overlap page boundaries. The difficulty with this approach is that unused sections of memory are left at the bottom of some pages, spreading the program over a greater expanse of virtual memory. An optimal placement of blocks, so that there is a minimum of wasted spaces, would require the unrealistic chore of enumerating all possible block arrangements. The Greatest Section Algorithm (GSA) is a sub-optimal algorithm which fills a page with the greatest size block (modulo one page) which can be fit in the remaining space. Thus, once the blocks are sorted in decreasing order of size, repetitive scanning of the list is all that is necessary.

A third algorithm was introduced as a compromise between the NCA and GSA methods. The First Section Algorithm (FSA) lists the blocks in decreasing order of density, then it fills the pages

with blocks in the same order except those where an overlap would occur, until the page is filled or the list is completely examined. The process is repeated for all pages until the list is exhausted.

Experimentally, the FSA showed the greatest improvements in page fault reduction compared to the NCA and GSA, but at the cost of increased virtual program space. The FSA produced block-page mappings with an average 2.8% wastage compared with 0.3% for the GSA. The NCA provided better results than the GSA indicating the importance of the nucleus concept.

2.3 The Nearness Method

Historically one of the first algorithms for automatic program restructuring was presented by Hatfield and Gerald [HATF71]. Their Nearness Method (NM) focusses upon the nearness matrix, C. This matrix is generated by incrementing the element c(i,j) each time control is transferred from block i to block j. Once the nearness matrix has been constructed, the blocks of the program are clustered using an "evaluator".

A reasonable evaluator, if all blocks are the same fraction of a page size, might be

$$\sum_{i,j=1}^{n} c(i,j)p(i,j),$$

where n is the number of blocks, and p(i,j) is the probability that blocks i and j are both in physical memory whenever either i or j is in physical memory. If blocks i and j are placed on the same page then p(i,j)=1, so that for each page a,

$$\sum_{i,j \in a} c(i,j)$$ is a term in the evaluator.

Similarly, if block i is on page a and block j on page b then,

$$\sum_{i \in a} \sum_{j \in b} c(i,j)p(a,b)$$ is a term in the evaluator,

where p(a,b) is the probability that pages a and b are both in physical memory whenever either is in physical memory. Unfortunately, as Hatfield and Gerald state, the values for p(a,b) are "computationally exorbitant" to estimate and as a result the simpler,

$$\sum_{i,j \in a} c(i,j)$$ is used as a practical evaluator.

In fact, this evaluator minimizes page faults for a memory size of one page.

Although the NN gives good restructuring results, the narrow window (adjacent block references) used can give poor ordering information. For example [FERR74a], consider two reference strings S1 and S2 containing K occurrences of the pattern 'ij'. In S1 the patterns are spread out while in S2 they are consecutive. The value of c(i,j) in the nearness matrix will be K for both S1 and S2 but c(j,i) is zero for S1 and K-1 for S2. Thus, the probability of clustering blocks i and j together is greater for S2. However, the probability of a fault, if blocks i and j are not clustered together, is greater for S1 (K faults in the worst case) than for S2 (at most one fault with any reasonable replacement algorithm).

In general, the nearness method improves paging performance in terms of a reduction in page faults and necessary memory

allocation. The concept of the nearness matrix provided a solid base from which others have introduced more sophisticated restructuring algorithms.

2.4 Masuda's Method

A restructuring algorithm was introduced by Masuda et al. using an extension of the nearness method, with the goal of reducing the average working set size [MASU74]. This strategy dependent algorithm extends the NM by considering "nearness" to be not only the previous block reference but "recent" references as well.

The approach used is basically a miniature working set where an element r(i,j) of the symmetric working set closeness matrix, R, is incremented each time blocks i and j are both resident in this working set. The set membership is determined from the blocks referenced within a window of size t.

i.e. $R = (r(i,j))$ for $i \neq j$ $i,j = 1,2,\ldots,n$

(n = number of blocks).

Then r(i,j) is defined as follows:

$$r(i,j) = \sum_k Dijk$$

where $Dijk = \begin{cases} 1 & \text{if i,j are referenced in the interval } [k-t,k] \\ 0 & \text{otherwise.} \end{cases}$

It should be noted that in the special case where t=1, the

closeness matrix R is identical to Hatfield and Gerald's nearness matrix C. The selection of a value for t is determined from the main memory size, the degree of multiprogramming, the page replacement algorithm and the program reference patterns. A reasonable value of t would be in the order of the average interval between page faults.

Once R has been constructed, blocks are clustered using the following strength of connection evaluator,

$$Bfg \triangleq (1/(Sf + Sg)) \sum_{i \in f} \sum_{j \in g} r(i,j)$$

where Sf and Sg are the sizes of cluster f and g respectively, that is,

$$Sf = \sum_{i \in f} s(i), \quad Sg = \sum_{j \in g} s(j) \quad \text{and } s(i) = \text{size of block } i.$$

The clustering process selects the most strongly connected pair of clusters (where a cluster consists of one or more blocks) one after another, in a hierarchical manner, as long as the cluster size does not exceed the page size.

Masuda's method produces a significant reduction in the working set size (35-40%) and in general produces better results than the nearness method due to its broader field of observation.

2.5 ARBRE Algorithm

A modified version of Masuda's algorithm was presented by Achard et al. under the name ARBRE [ACHA78]. The evaluator in Masuda's method was of the form:

$$Rfg = \sum_{i \in f} \sum_{j \in g} r(i,j) \quad \text{for clusters f and g.}$$

The ARBRE algorithm uses the Jaccard index,

$$R'fg = Rfg / (Ufg + Vfg) \quad \text{where}$$

$$Ufg = \sum_{i \in f} \sum_{j \notin g} r(i,j) \quad \text{(interconnection between cluster f and all clusters other than g)}$$

$$\text{and } Vfg = \sum_{i \notin f} \sum_{j \in g} r(i,j) \quad \text{(interconnection between cluster g and all clusters other than f).}$$

Achard claims an improvement over Masuda's algorithm of up to 15% and bases this on the relative importance of links between blocks rather than the absolute importance.

2.6 Critical Algorithms

One of the most important concepts in strategy-oriented program restructuring is that of "critical algorithms". This class of algorithms was introduced by Ferrari [FERR74b, 76] as a uniform method for translating a block reference string into relevant locality information. Regardless of the page replacement policy being used, there exists at any given time, a group of resident blocks. If the next reference in the block reference string is to one of the resident blocks then that reference will not cause a page fault. If, however, the next reference is a "critical reference", that is, a reference to a non-resident block, then clustering this block with the resident ones will prevent a fault from occurring. A simple critical restructuring algorithm will increment (by one) the weights of each edge connecting the critical reference to all of the blocks

that are resident according to the paging policy.

In restructuring algorithms, a great deal of processing time is spent interpreting the block reference string and incrementing matrix elements. By using a critical algorithm, only critical references cause matrix updates, often resulting in faster processing. Another significant feature is that critical algorithms can be applied to fixed and variable partition replacement policies as long as the portion of a program resident in main memory at any instant can be derived or estimated from the behavior of the program and the index to be minimized is the page fault rate [FERR76].

For the most part, Ferrari has concentrated his study of program restructuring on the working set environment [FERR73, 74a, 75]. The Critical Working Set (CWS) algorithm is designed to be tuned to the working set page replacement policy and in particular to the window size T being used. The CWS algorithm defines a block working set, Wb(t,T), at time t, as those blocks which are referenced by the block reference string S=r(1),r(2),... in the interval [t-T,T]. A block working set Wb(t,T) is said to be critical if r(t+1) is not in Wb(t,T). In this case, r(t+1) is called a critical reference. The CWS algorithm increments all the weights of the edges connecting a critically referenced block to all the members of Wb when a critical reference occurs. Practically, the edge weights are represented by the CWS matrix C where c(i,j) corresponds to the edge connecting block i to block j. If block i is a critical reference and block j is a member of Wb then c(i,j) is incremented. If blocks i and j were grouped together on the same

page, then c(i,j)+c(j,i) critical references would disappear. By definition, the diagonal elements of C are all zero.

Once the CWS matrix is completed, a clustering algorithm is applied to determine the block-to-page mappings. If each page contains no more than two blocks then the CWS method is optimal. The proof of optimality comes from noting that for any clustering, the sum of intercluster connections is equal to the number of references that are critical in the block reference string and non-critical in the page reference string. Thus, maximizing the intercluster connections is equivalent to minimizing the number of page faults. If there are more than two blocks per page, then this argument is no longer valid since the sum of the intracluster connections differs from the number of critical references which became non-critical due to clustering. There is no way of computing this difference from the CWS matrix [FERR74a]. For example, consider three blocks A, B and C where C is referenced only once. If C is referenced when A and B are resident blocks then two elements of the CWS matrix are incremented, CWS(C,A) and CWS(C,B). If C was then clustered along with blocks A and B, one page fault would be eliminated even though the intercluster connections increased by two (CWS(C,A) + CWS(C,B)). Thus, there is no discernible correlation between the CWS matrix values and the reduction in page faults for more than two blocks per page.

The CWS method can treat data and instructions independently so that they can be restructured into separate pages. This breaks the restructuring into two more manageable problems. The CWS method gives good experimental results and shows an

improvement of 12-35% over the nearness method.

2.7 Bounded Locality Interval Methods

The majority of program restructuring algorithms use some form of the working set concept to isolate program localities. An extensive study by Madison and Batson of program locality, resulted in a model of program behavior that suggests an alternate method to determine localities [MADI76].

The locality model is based on an LRU stack where a block's position in the stack is determined from its most recent use, that is, the most recently referenced block is at depth 1, the second most recently referenced block is at depth 2, etc.. For any position i in the stack, the topmost i blocks are the i most recently referenced blocks. Madison and Batson define an extended LRU stack which maintains not only order of most recent use but the time at which the block at depth i was last referenced, Si, and the time at which a reference was last made to a position greater than i, Ti (Fig. 2b). An "activity set", Ai, at time t, can then be defined as the set of blocks at depth i in the LRU hierarchy in which every set member has been rereferenced since the set was formed, or equivalently, Ai(t) is the topmost i blocks for which Si(t) > Ti(t). In Fig. 2a the activity sets at t=30 are {D}, {C,D}, and {A,B,C,D}. When a reference is made to a position below a particular activity set then it is terminated. The lifetime of the activity set, Li, is thus the interval of time between set formation (Ti) and termination. A "Bounded Locality Interval" is defined as the

2-tuple consisting of an activity set membership, Ai, and its lifetime, Li.

If at time t, two activity sets, Ai(t) and Aj(t) exist where i < j, then by definition Ai(t) is a subset of Aj(t) and Li < Lj. This property means that BLIs at a particular time determine a hierarchy in terms of both set membership and lifetime. The "level" of a BLI is its distance down in the hierarchy, such that a BLI at level 1 contains the largest activity set. The higher the level of a BLI, the smaller its activity set and the shorter its lifetime (Fig. 2). The three main features of BLIs can be summarized as follows:

a) they correspond to most intuitive notions of what constitutes a "locality"

b) the definition is independent of parameters such as windows

c) the hierarchical nature of localities is implemented in the definition.

Since BLIs are determined independent of parameters it is well suited as a basis for strategy-independent restructuring techniques. Kobayashi has proposed a set of restructuring algorithms which attempt to decrease the resident set size of a program by grouping together blocks of high level activity sets [KOBA77]. The Activity Set Algorithm-1 (AS1) increments the edge weights between blocks i and j by the value of the level of the smallest activity set (highest level) to which they both belong. This is accomplished by incrementing the weights by one each time the pair of blocks become members of a new activity set. Due to

the hierarchical property of BLIs, the blocks that are activity set members at level 3, for example, are also members at levels 2 and 1, hence, incrementing one at a time is equivalent to setting the weight equal to three.

If the number of BLI levels or the average size of level 1 activity sets is small, then a simpler algorithm can be used. The Activity Set Algorithm-2 (AS2) focusses on level 1 activity sets only and increments by one the edge weights between blocks which belong to an activity set of level 1.

One factor inherent in both AS1 and AS2 arises from the fact that no time parameter is used in the establishment of activity sets. This deficiency can result in unreferenced members remaining in the activity set. A method suggested by Kobayashi for removing inactive members is to introduce a parameter A similiar to the working set window. This allows the definition of the strict activity set at time t as an activity set, all the blocks of which have been referenced after the time t-A. Strict activity sets do not alter the hierarchical structure of BLIs. If strict activity sets are used in place of regular activity sets in the AS1 and AS2 algorithms, then analagous Strict Activity Set Algorithm-1 (SAS1) and Strict Activity Set Algorithm-2 (SAS2) can be considered.

Kobayashi compared the four BLI-based algorithms to strategy dependent algorithms in both a working set and LRU environment. The results showed that although algorithms AS1 and AS2 did not reduce page faults to the same degree as the strategy dependent algorithms, their performance was "satisfactorily close". The surprising aspect of the experiment was that the SAS1 and SAS2

algorithms were not as effective as the AS1 and AS2 algorithms. This performance was attributed to the relatively long average lifetime of BLIs.

2.8 Clustering Algorithms

In almost all of the restructuring algorithms presented, the emphasis has been on determining the strength of interconnections between program blocks. Once the interdependence of these blocks has been established, the problem of grouping or clustering these blocks is faced. The reason most restructuring papers skim over the clustering aspect is not because it is of minor importance, but rather that the subject is well known and exhaustively documented.

Formally, the clustering problem can be stated as follows [BARR79]:

Let V(Xi,Xj) denote the interconnection between clusters Xi and Xj, then,

$$V(Xi,Xj) = \sum_{k \in Xi} \sum_{l \in Xj} Skl$$

where Skl is the interconnection between blocks k and l. For a particular page mapping of c clusters, the total interconnection between pages is given by,

$$C = C(X1,X2,\ldots Xc) = \sum_{1 \le i < j \le c} V(Xi,Xj)$$

Thus, C is a measure of the page fault rate for a particular mapping. For a memory size of one page, C would be in fact the actual page fault rate.

The ideal goal of the clustering procedure is to minimize: 1) the number of pages required and 2) the interconnection between pages.

If one considers the interconnection of blocks in terms of the restructuring graph the clustering problem is a subset of the well known "graph partitioning problem" [KERN69]. As well, clustering is an extension of the "bin packing problem" [JOHN74]. Given a set of bins of unit height and segments of length $a(i) \in (0,1]$ $i=1,2,\ldots,n$ find the assignment of segments to bins that minimizes the number of bins used provided the sum of the lengths of segments in each bin is at most one.

Regardless of what name is used to describe the distribution of program code, the problem is NP-complete, that is, all known algorithms may require the enumeration of all possible mappings in the worst case. To appreciate the magnitude of this problem consider a program of n pages, b blocks and assume b/n blocks per page. The number of unique partitions is:

$$P = b! \;/\; (n!\,((b/n)!)^{**n}) \qquad \text{[PART79].}$$

If, for example, a 10 page program consists of 40 blocks which are distributed as 4 blocks/page, there are 3.5×10^{27} distinct partitions. It is clearly futile to enumerate all the partitions and the probability of hitting upon an optimal solution at random is exceedingly small, even if hundreds exist. In most cases, programs that warrant restructuring are significantly larger than the example given, which indicates the need for sub-optimal heuristics.

A unifying feature of most heuristics for program restructuring is that of hierarchical clustering. Consider a

sequence of partitions which group n objects into c≤n clusters. The initial partition contains n clusters of one object each. The second partition contains n-1 clusters, the third n-2, and so on until the n-th partition in which all the objects form one cluster. A position in the sequence is denoted by a level number, where level k occurs when c=n-k+1. The level 1 correspond to c=n and level n to c=1. If the sequence has the property that whenever two objects are grouped together at level k they remain together at higher levels then the sequence is called a hierarchical clustering [DUDA73].

Hierarchical clusterings can be graphically displayed using a tree called a "dendrogram" ("arborescence" [ACHA78]). Fig. 3 illustrates a dendrogram for a clustering of six objects. Variations of the dendrogram are often used where the vertical axis denotes the strength of connection between clusters as opposed to the level [MASU74, ACHA78].

Most restructuring algorithms [FERR73, MASU74, ACHA78] use constrained stepwise optimal clustering. Barrese and Shapiro [BARR79] describe this procedure as follows:

Step 1: Set c=n and Xi={block i}, i=1,2,...,n.

Step 2: Find a pair of clusters Xi and Xj that reduce the cost function (page interconnections) as much as possible subject to the constraint that:

$\sum_{k \in (Xi \cup Xj)} s(k) \leq PS$ i.e. the page size is not exceeded (s(k)=size of block k, PS=page size).

If no such pair exists, then halt.

Step 3: Replace Xi by (Xi U Xj) where i<j, delete Xj and set c=c-1.

Step 4: Update block interconnections [Sij].

step 5: Go to Step 2.

The "constraint" of the algorithm is the page size limit and the "stepwise optimality" results from the fact that for each iteration the best clustering is made. Though the clustering heuristics described are sub-optimal, the results of Ferrari and others indicate that their performance is acceptable.

2.9 Sequencing Algorithms

The major constraint in program restructuring is the page size limit imposed on the clusters. A reason for this constraint is to achieve an independence between pages. If blocks do not overlap page boundaries then there is no implied continuity between consecutive pages in virtual memory. This is an important point if the restructured program is to be run under various page replacement policies or under an LRU policy with varying memory allocation. As far as working set policies are concerned, Masuda [MASU74] suggests that allowing clusters to cover several pages without consideration of page size might be a better approach.

If one maintains the page size limit on clusters, as most do, then another problem arises. Since the blocks of programs vary greatly in size it is inevitable that gaps or holes will occur when clusters are mapped onto pages. As Hatfield and

Gerald [HATF71] state, the presence of holes spreads the clusters over a greater virtual space. This requires, on the average, more pages to be in physical memory for the same number of instructions executed without a page fault. Clearly this is detrimental to the goal of restructuring. As an alternative approach, clusters can be packed together across page boundaries leaving no holes. Hatfield and Gerald indicate the relative success of this approach from their experience. Ferrari [FERR73] packs his clusters as well, but permits their sizes to be "slightly larger" than the page size in the hope of balancing things out.

If the blocks are allowed to cross the page boundaries then it is important to sequence the clusters intelligently since a block that crosses a boundary will probably require that both pages be in physical memory within a short period of time. Hatfield and Gerald define the sequencing problem as follows.

If the strength of interconnection between two clusters X and Y can be considered as the sum of interconnections c(i,j) of blocks from one cluster to blocks in another,

$$\text{i.e.} \quad \sum_{i \in X} \sum_{j \in Y} c(i,j) = a(X,Y)$$

then an optimal sequencing can be found by solving the maximal tour (travelling salesman) problem on the matrix A={a}. This means finding a circuit of clusters that maximizes the sum of the transition values a(X,Y) between adjacent clusters. Since the travelling salesman problem is a well known NP-complete problem [GARE79], Hatfield and Gerald suggest a simple heuristic approach which they claim gives good results. The "nearest city" or

"greedy" algorithm begins by selecting an arbitrary cluster, X, and placing after it the cluster, Y, with the largest value of a(X,Y). The process is repeated for each additional cluster, at each step selecting the best candidate. Hatfield and Gerald suggest that the sequencing can be initially biased by selecting the two clusters with the greatest transition value in A.

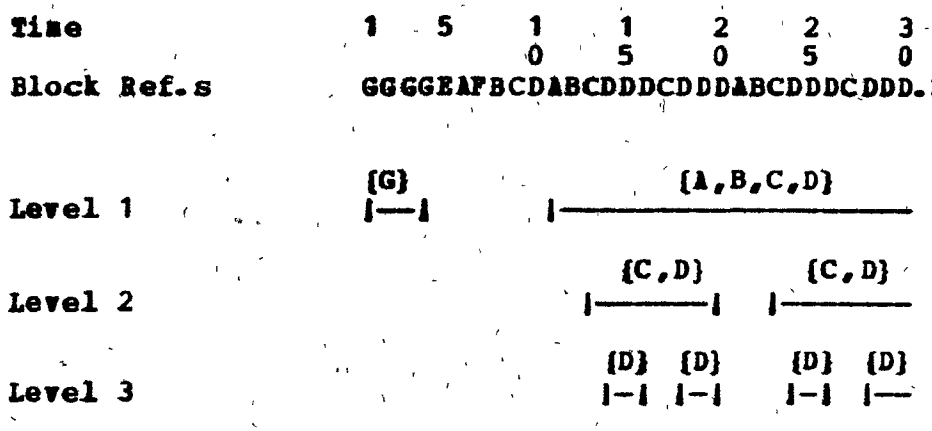


Figure 2a - Example of BLIs

Depth	LRU	Time of Last Ref.	Time of Last Ref. Below Depth I
1	D	30	28
2	C	27	24
3	B	22	24
4	A	21	11
5	F	7	10
6	E	5	10
7	G	4	10

Figure 2b - "Extended" LRU Stack at t=30

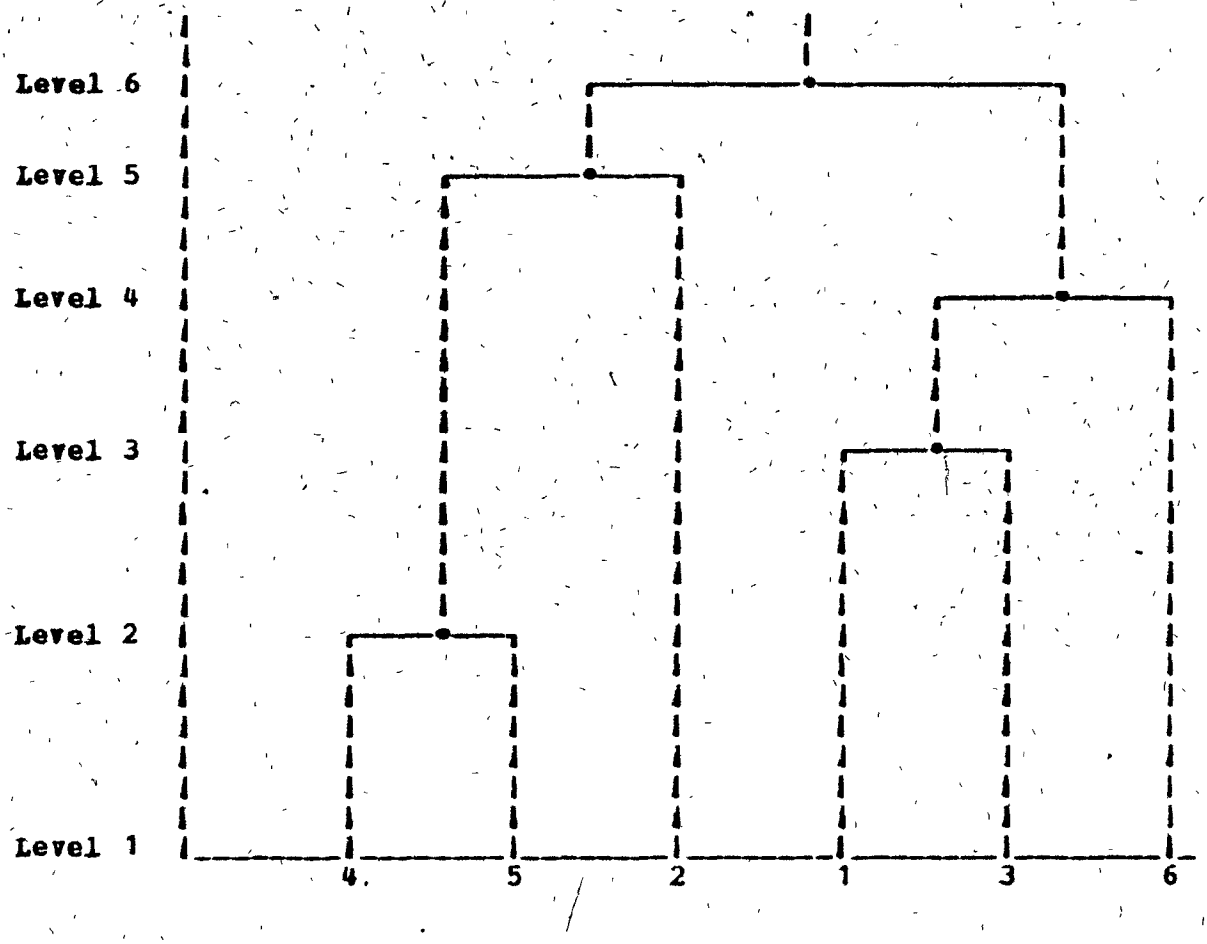


Figure 3 - Dendrogram For Hierarchical Clustering, N=6

Chapter 3 - Predictive Memory Management

In demand paging systems, only the page causing a fault is fetched. Intuitively it would seem reasonable to load not only the missing page but also those pages that will be required in the near future. The ability to determine the future accessing of pages is normally a difficult task. However, program restructuring provides page interconnectivity information. Utilizing this information to predictively reduce page faults appears to be an area worth examining.

Unlike standard memory management techniques that are based on cause/effect actions, predictive memory management uses past experience to predict future program behavior. Predictive elements can be found in even classical demand paging policies such as FIFO, LRU and WS. These policies base resident set membership on the prediction that the members are likely to be referenced in the near future. Predictive strategies can also be found at the CPU level in instruction look-aside buffers and cache memories. Both common devices provide excellent results since the locality principle predicts a slow change in the active section of a program at the microscopic level.

The interest in predictive memory policies and in particular predictive fetching (prefetching, prepaging), stems from observations of program locality characteristics. Programs can be modelled as a sequence of stable, relatively long-lived phases during which a small subset of pages are referenced [MADI76]. These subsets change membership slowly during a phase so that demand paging techniques are sufficient. During transitions from

one phase to another, however, the contents of the resident set can be radically altered. In fact, the majority of page faults occur during phase transitions. The ability to change resident set membership rapidly and correctly by using prepaging algorithms is therefore potentially of value.

When a page fault occurs, prepaging policies attempt to load not only the demanded page, but any other pages that might be referenced soon after. By prefetching a page that is referenced in the near future, a fault is eliminated. If a bad prefetch is made, the cost to the system is that a good page may have been discarded (LRU) or the memory size is increased by one page for the duration of the window (WS). One important assumption used in justifying prepaging is that there is no significant cost differential between loading one page or several pages. Since the access times for existing external memory devices are much longer than the transfer times, this assumption appears to be approximately valid.

Many predictive management strategies have been proposed in the past but none have proved to be practical for implementation on a large scale. Although the improvements made by predictive algorithms are legitimate, it is important to consider them in the proper context. In most cases, the gains made by predictive algorithms are at the expense of increased page traffic (fetches and replacements). Even though a program is not being executed while waiting for a page transfer, it is resident and tying up real resources. The increased page traffic has a direct bearing on the throughput of a computer system and is a major objection to many predictive algorithms. Despite all these factors, the

gains that can be made by prefetching algorithms are still significant. Hence, predictive memory management techniques are of interest in those cases where the benefits outweigh the costs.

3.1 One-Block Lookahead Method

One-Block Lookahead (OBL) is the classical prepaging algorithm introduced by Joseph [JOSE70]. The strategy attempts to load the demanded page into main memory, along with the following page, if that neighbor is not already resident. OBL, being a fixed-partition algorithm, provides a special buffer page-frame into which prefetched pages are loaded. If this page is referenced before the next fault then it is retained in memory, otherwise it is overwritten with the next prefetch. Thus, the space cost of OBL is one extra page.

Under certain conditions such as matrix manipulations, data references in a program which alternate between one group of pages (Matrix A) and another (Matrix B) may cause the OBL buffer to be continuously rewritten, destroying useful prefetches. This deficiency is corrected in another algorithm suggested by Joseph. Simple Prediction (SP) operates the same as OBL except that no special buffer is used. Unlike OBL, a bad prefetch in SP will remain resident until the standard replacement policy discards it. This last factor explains the increase in memory required for SP applications.

In simulations performed by Joseph [JOSE70], the fault rate was found to decrease by 25-35% for OBL and 50-75% for SP. The space-time product, however, increased by a factor of 1-15% for

OBL and 20-30% for SP. Another side-effect of predictive algorithms is the increase in page discards due to failed prefetches. For OBL, this increase was about 50% while SP showed a 20% rise.

3.2 Spatial and Temporal Locality Methods

In 1976, Baer and Sager introduced a pair of prepaging algorithms [BAER76]. Their intention was to improve paging performance by dynamically altering the page predictions during program execution. The rationale behind this approach is that sequential prefetches (OBL) are not always beneficial and for these cases it is desirable to change predictions for future faults. Hopefully, as program execution progresses, the prepaging policy will correct poor prefetches by replacing them with "better" ones.

The Spatial Locality (SL) algorithm keeps track of which page to prefetch by using a predecessor function, PRED. Given a fault to page I, PRED(I) will indicate which page should be prefetched. A variable LAST is also used to record the last page that faulted. Initially, PRED(I)=I+1 for all pages I and LAST is set to some null value. The starting predictions are sequential as in OBL. Consider during program execution, a reference to page q which causes a fault. If preloading did not occur on the previous fault or if the preloaded page was not referenced (bad prefetch) then update PRED(LAST)=q. If PRED(q) is not resident, then preload it into the position of lowest priority according to the replacement policy. Finally, set LAST=q.

The Temporal Locality (TL) method operates the same as SL except that LAST records the page referenced immediately prior to the fault-causing page.

Simulation results [BABR76] indicate the relative superiority of both SL and TL over OBL. The major disadvantage to these algorithms is the overhead required to perform dynamic changes and the cost of fast register storage to implement the PRED table.

3.3 Freeing Demand Prepaging Algorithms

A class of algorithms was presented by Trivedi [TRIV74, 76, 77] with the objective of using predictive paging techniques to supplement rather than replace existing demand paging algorithms. The key to Freeing Demand Paging Algorithms (FDPAs) is that unlike sequential prefetching schemes, the isolation and removal of the old locality set is of equal, if not more importance than the loading of the new locality set. FDPAs require the existance of two special primitives that "suggest" paging actions to the operating system. These primitives are inserted into a program either by the programmer or the compiler, in an attempt to indicate the program's macro-behavior. The primitive FREE(X) is used when page X will not be used in the "near" future, while PRE(X) is inserted when it is desirable to prefetch page X.

The operating system treats the above primitives as advice rather than commands. The primitives may be executed immediately, deferred or even ignored. The system will not allow prefetches to occur at the expense of useful pages already

resident. The available space that arises when requests are made to free pages is used for prefetching purposes. It is assumed that prefetch requests are usually valid, so prefetched pages are not replaced unless they are referenced at least once. These requirements are implemented to prevent the misuse of memory space by naive users.

Simulations using an FDPA in an LRU environment were performed on array manipulations with good results [TRIV76]. Fault rates lower than LRU were attained at large allocations of memory for all matrix operations while page fetch traffic for one simulation approached that of LRU.

3.4 The Dynamic Matrix Model

Studies by Burris and Pooch have resulted in numerous page clustering algorithms which are used to predict a program's locality based upon dynamic behavior [POOC76a, 76b]. One algorithm in particular, the Dynamic Matrix Model (DMM), builds dynamic clusters of pages during program execution according to "time and reference" relations [BURR77]. These clusters are then used to minimize future page faults and overall memory space.

Every page has associated with it, a cluster of pages. Whenever a page faults, all the members of its cluster are loaded into memory if they are not already resident. As well, those resident pages that are not in the current cluster are removed. The physical memory required by a program is thus dependent upon the size of the cluster associated with the most recently referenced page. When control is transferred from one resident

page to another, pages not in the cluster of the newly referenced page may be removed.

Burris goes into great detail about how the clusters are determined. Basically, the algorithm involves a continuous updating of a binary matrix where the matrix elements correspond to page interreferences. Whenever page i faults, the non-zero elements in row i of the matrix indicate cluster members. The actual process is somewhat more complex and includes a method for limiting the size of a cluster.

The simulation results of Burris [BURR77] indicate an improvement in both fault rate and memory size over standard paging algorithms, however, no mention is made of the type of programs being simulated and the reference strings used are quite short (5000 to 9000 references). A major drawback to this and similar algorithms is the tremendous overhead required by the operating system to maintain cluster information for every resident program.

Chapter 4 - A Modified Approach to Program Restructuring

In chapter 2 a number of program restructuring algorithms were presented. For the most part, these algorithms emphasized the accumulation of interconnectivity information as opposed to the clustering of blocks. Although the greatest performance improvements can be attributed to the isolation of localities, further reductions can still be made by a more intelligent clustering.

By far the least well-defined aspects of clustering concern cluster size and the mapping of clusters onto pages. Theoretical analysis usually assumes the unrealistic case of equal-sized blocks that are a small fraction of the page size. This type of analysis might permit a better understanding of the algorithm but it provides no indication of how the algorithm should be applied to "real" programs. Most restructuring algorithms specify a maximum cluster size of one page. If these clusters are mapped directly onto pages, there are no blocks split over page boundaries. This desirable feature has the unfortunate side effect of wasting space at the ends of most pages. Obviously the restructured program must then be stretched over a greater number of pages and will often require a greater average memory size. Hatfield and Gerald's experience has indicated the unsuitability of this approach [HATF71].

One method that could be attempted to minimize the unused memory on pages would be to fill this space with blocks that did not get clustered. Although there are almost always many of these blocks left over, their number is dependent upon the length

of the reference string used for restructuring and the data used by the program. It is not realistic to assume that there are always enough free blocks to reduce the wasted space to a negligible amount.

If this approach is abandoned then clusters must be packed one after another onto pages. The sequencing of clusters then becomes important since blocks will be split over page boundaries, frequently requiring two consecutive pages to be resident within a short period of time. The greedy algorithm outlined by Hatfield and Gerald [HATF71] is a reasonable method for sequencing and has provided adequate results to date. There are, however, certain inadequacies in how this general algorithm pertains to a specific problem.

The greedy algorithm, as described in chapter 2, implies a unidirectional sequencing of clusters. In other words, clusters are added to the sequence at one end only and there exists only one sequence at a time. In the case of clustering, blocks could be added to either end of a cluster and many clusters exist at one time. Clearly the clustering algorithm provides much greater flexibility than the greedy sequencing method. It would therefore seem logical to group together clusters in a method akin to the clustering of blocks. The difference between the two problems is that in sequencing there is no page size constraint. As well, the addition of a cluster to a sequence should be dependent upon the "nearest" clusters and not those clusters at the opposite end of the sequence.

We can therefore propose the following sequencing algorithm (Algorithm SEQ):

Step 1: Consider every cluster of blocks as a sequence.

Step 2: Evaluate the interconnection weights for all pairs of sequences.

Step 3: Select the largest weight from Step 2 and combine the corresponding two sequences.

Step 4: Repeat Steps 2 and 3 while Step 2 returns non-zero weights.

Obviously the key to algorithm SEQ is Step 2. By placing appropriate constraints on Step 2 the algorithm can be reduced to the greedy method. A reasonable evaluation would be to consider only the end clusters in a sequence. For example, consider two sequences S1 and S2 each consisting of many "page-size" clusters. If S2 were to be placed after S1 in memory, the block that would overlap the page boundary would almost certainly occur in the last cluster in S1 or the first cluster in S2. Therefore, it seems sufficient to evaluate the interconnection strengths between the terminal clusters of sequences. This analysis is valid assuming the clusters are all approximately one page in size. Unfortunately this assumption is not always correct.

When dealing with real programs, the sizes of blocks do vary greatly. If clusters are limited in size to exactly one page, they will probably end up being significantly smaller than a page. Ferrari suggests that clusters be allowed to grow slightly larger than the page size in an effort to balance out this

difference [FERR73]. Since the key to cluster sequencing is in the accurate prediction of page boundaries, a small variance in the cluster sizes can accumulate and cause considerable misalignment with respect to page boundaries. With the greedy algorithm, one end of the sequence is fixed so that the page boundaries are always known during the sequencing process. If an algorithm such as SEQ is used, however, the boundaries are not known until the final iteration. The cost of greater flexibility in sequencing is the loss of page boundary information.

Another important aspect of block sizes that is glossed over in most algorithms is the frequent occurrence of large blocks (i.e. larger than a page size). It is not uncommon to have certain blocks, such as initialization or I/O routines, that are many times the average block size. There are essentially two standard ways of handling these blocks. They can be manually split into smaller blocks or they can each be treated as a single cluster [FERR74a]. The former approach requires a knowledge of the source or object code for the block; however, logical boundaries may not always be evident. The problem with the latter method is that these clusters can have any size, destroying the careful alignment of page boundaries. If large blocks are poorly sequenced and frequently executed, the page faults resulting from references within a block can overshadow the gains made through restructuring the small blocks. The effects of these blocks on the restructuring process are not well-known and no algorithm to date has realistically attempted to accommodate them.

4.1 A Combined Clustering/Sequencing Algorithm

It is clear that existing restructuring algorithms contain inherent weaknesses and limitations. To correct these deficiencies, an alternate approach must be taken to perform the clustering and sequencing of blocks. It has been shown that a flexible sequencing method results in a loss of page boundary information. If page boundaries cannot be accurately predicted then the use of page-size clusters becomes questionable. It is therefore proposed that a probabilistic technique be employed which takes into acccunt the page boundaries at the clustering level. Such a technique should permit clusters of any page size but should weight the interconnectivity contributions between blocks according to the probability of the blocks being on the same page.

Consider, for example, the interconnectivity between two clusters A and B containing blocks i and j respectively. While most sequencing algorithms would include the interreference contribution of i and j with an equal weighting compared to other blocks in the clusters, it is proposed that a weighting be used that takes into account the distance between blocks. The farther apart two blocks lie, the greater the probability of a page boundary separating them. Thus, if i and j are far apart, they should contribute less to the total interconnectivity of A and B than blocks that are close together.

Clustering algorithms normally require an interconnection matrix as a basis for connectivity. The following discussion assumes the existance of Ferrari's CWS matrix (the CWS method has consistently provided good restructuring results) although any

similar type of matrix can be substituted. The standard sequencing approach uses a formula of the form:

$$\sum_{i \in A} \sum_{j \in B} CWS(i,j) + CWS(j,i) \text{ .}$$

What is proposed is a weighted version of the above formula:

$$\sum_{i \in A} \sum_{j \in B} W(i,j) * CWS(i,j) + W(j,i) * CWS(j,i) \text{ .}$$

A suitable weighting function W can be obtained from the following observations. Data references can normally be ignored since most programming languages maintain separate data areas. For those languages that maintain local data (e.g. FORTRAN), passed variables must be from recently executed blocks and therefore should not result in critical data references. Critical instruction references from block i tc block j can arise from two sources. Either block i is calling block j or block i is returning from a call made by block j. Unfortunately, the CWS matrix does not provide this information. We will, however, make the assumption that critical references are exclusively calls as opposed to returns. This assumption is necessary for the derivations that follow. It can be justified if the CWS window is relatively large compared to the average block size. In other words, if a critical call occurs and the window size is large, the probability of the routine returning within the window is high. This implies that the vast majority of critical references should be calls.

Figure 4a illustrates critical references from block j to block i for a possible ordering of clusters, namely, cluster B following cluster A. The number of times that block j was resident when a non-resident block i was referenced is given by

CWS(i,j). As we have no information concerning the structure of block j, we can only assume that any instruction in j could have referenced block i. We denote the location of this arbitrary instruction by X. On the other side, we may not know to which instruction in block i control was transferred. We can denote the location of this instruction in block i by E. If, during the execution of the program, a reference is made from X to E and |E-X| is greater than the page size then we are guaranteed that a page boundary separates the two memory locations. For this condition, the CWS(i,j) contribution should be excluded, that is, a zero weighting should be applied. If |E-X| is less than the page size, the probability that a page boundary does not occur between i and j can be used as an appropriate weight. Formally,

$$\text{Prob}\{E \text{ and } X \text{ are not separated by a page boundary}\} = \begin{cases} (PS - |E-X|)/PS & \text{if } |E-X| \leq PS \\ 0 & \text{otherwise} \end{cases}$$

where PS is the page size.

In the following experiments, control transfers are always made to the beginning of blocks. In other words, E was always at the beginning of a block. The derivations that follow also use this constraint. For those cases where this condition does not apply, a modification to the integral formulae would be required. We cannot predict which instruction in block j will reference block i. In the absence of any information, the formula for W(i,j) must be an integral over all the locations in j. Let Dij represent the distance (due to intervening blocks) between the end of block i and the beginning of block j and let Si and Sj denote the size of block i and j respectively. The formulae for

W(i,j) are thus:

If Si+Dij ≥ PS,

$$W(i,j) = 0$$

If Si+Dij < PS ≤ Si+Dij+Sj,

$$W(i,j) = \int_{Si+Dij}^{PS} (PS-x)/(PS*Sj)\ dx$$

$$= (PS-Si-Dij)^2/(2PS*Sj)$$

If Si+Dij+Sj < PS,

$$W(i,j) = \int_{Si+Dij}^{Si+Dij+Sj} (PS-x)/(PS*Sj)\ dx$$

$$= (2PS-2Si-2Dij-Sj)/2PS$$

The derivation of W(i,j) which weights the interreference contribution from block i to block j (Figure 4b) produces similar formulae:

If Dij ≥ PS,

$$W(j,i) = 0$$

If Dij < PS ≤ Dij+Si,

$$W(j,i) = (PS-Dij)^2/PS$$

If Dij+Si < PS,

$$W(j,i) = (2PS-Si-2Dij)/2PS$$

The difference between the formulae for W(i,j) and W(j,i) arises from the assumption that control transfers enter at the beginning of a block.

Using the the weighting formulae, a combined clustering/sequencing algorithm is presented (Algorithm CS):

Step 1: Initially consider every block as a cluster.

Step 2: Evaluate the interconnection strengths between every pair of clusters in both orders.

Step 3: Select the largest value from Step 2 and combine the two clusters to form a new large cluster.

Step 4: Repeat Steps 2 and 3 until only one large cluster remains.

The processing required by this algorithm is not as expensive as it might appear. For each iteration only those pairs involving the new cluster need to be computed. As well, computation of the weighting formulae requires only a scan through the blocks of a cluster until the page size is exceeded.

To summarize this new restructuring approach:

1) It appears that the method can be applied to any type of interconnection matrix (not only the CWS matrix).
2) Separate clustering and sequencing algorithms have been replaced by one process.
3) The weighting of interreferences is intuitively correct in that the farther apart two blocks lie, the less likely they are to be on the same page and the less they contribute to interconnectivity.
4) Large blocks contribute less to interconnectivity than small blocks as the probability of a page boundary occurring within these blocks is significant.

4.2 Analysis Criteria

To be able to judge the performance of different algorithms in a working set environment, it is essential to establish certain criteria. The two most important statistics are the page fault rate and the average working set size. The page fault rate is simply the number of faults divided by the reference string length. The average working set size is the integral of the resident size over the time the program is running (also referred to as the Space-Time Product) divided by the reference string length. There are many variations of these measures, but, these two are usually considered to be sufficient.

Both statistics can be derived from one pass through a program's reference string [DENN72].

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

Let $R = r(1)r(2)...r(k)$ denote a finite sequence of k page references where $r(t) \in N$, $1 \le t \le k$.

If, for a page reference r(t) at time t, there exists a previous reference r(z) where:

$$r(z) = r(t) \quad \text{for } z < t$$

$$\text{and} \quad r(m) \neq r(t) \quad \text{for } z < m < t$$

then the interreference interval $x(t) = t-z$, otherwise $x(t) = \infty$.

Let C(x) be an array defined over the indices $1 \le x \le k$ and $x = \infty$ and initialized to zero. If for every reference in the sequence R, the corresponding C(x(t)) is incremented by one, then C will maintain a set of interreference distance counters. The data in C is accumulated independently of a window size but is sufficient to generate working set statistics for any window size.

The number of page faults with a window of size T is,

$$F(T) = \sum_{t=T+1}^{k} C(t) + C(\infty) .$$

The first term is the number of interreference distances which were greater than the window, while the second term is the number of initial loads.

The Space-Time Product with a window of size T is,

$$ST(T) = \sum_{t=T+1}^{k} C(t) * T + \sum_{t=1}^{T} C(t) * t + \sum_{p} \min(T, B(p))$$

where p is the set of referenced pages and B(p) is the distance between k+1 and the last reference to page p. The first term accounts for all pages that were resident for T time units and then discarded. The second term adds those pages that were rereferenced within the window and thus, had an extended life. The third term is the end correction factor for finite length strings. This term picks up the pages that were discarded from the working set and never referenced again, as well as adding in the final working set.

Slutz and Traiger [SLUT74] have shown that the end correction factor can be ignored in certain cases since the error, E, involved is $0 \leq E \leq nt/k$. If the reference string length is large compared to the number of pages and the window size, then this error becomes negligible.

One disturbing aspect in the area of program restructuring is the lack of uniformity when comparing different algorithms. Many people, including Ferrari and Masuda, have graphed their results in terms of faults vs. window size. This type of analysis can be misleading. As the goal of program restructuring is to reduce both the number of faults and the mean memory size,

all comparisons in this thesis will be based on both criteria.

4.3 Description of Simulation Programs

Simulations of various restructuring algorithms were performed on two programs. The first program is a Pascal compiler written in Pascal and the second is a PFORT Verifier written in FORTRAN. The programs were selected based on their size, number of blocks and the difference in their structures. The blocks of the programs were determined directly from the CSECT map generated by a linkage editor. The CSECTs or Control Sections are groups of relocatable object code which have a unique entry point at one end.

The Pascal program is the first pass of a compiler and is responsible for the generation of P-code which is used by the second pass. Figure 5 provides the salient characteristics of the program as far as restructuring is concerned. A page size of 2048 bytes was selected due to its reasonable block/page ratio.

The PFORT Verifier is a program that examines a FORTRAN program for adherence to PFORT, a portable subset of ANSI FORTRAN [RYDE73]. The Verifer takes as input, a PFORT program and provides various diagnostics such as symbol tables, cross-references and error messages. The Verifier is itself written in PFORT. The program characteristics are outlined in Figure 5. A page size of 4096 bytes was selected.

Excluding the source languages of the programs, the major difference between the two, structurally, is in the handling of data areas. Pascal creates data storage dynamically (i.e. at

execution time) so the CSECT blocks contain instruction information and constants only. FORTRAN, on the other hand, creates data storage at compile time so that the CSECT blocks may also contain data information. This explains the large size of the FORTRAN blocks compared to the Pascal blocks. This also has an effect on the efficiency of restructuring based on instruction references.

A trace program which interprets object code was used to monitor and record the execution of the test programs. Although both instruction and data references can be used in the restructuring process (i.e. in the generation of the interconnection matrix), cost considerations permitted only the instruction references to be monitored. A string of approximately 475,000 references was generated for each program. This corresponded to 21,151 non-repetitive Pascal block references and 14,357 non-repetitive FORTRAN block references.

To generate the CMS matrices for the programs, a window size of 3000 references was selected. This appeared to be a reasonable value considering the reference string length and it provided a sufficient number of critical references to perform the restructuring. There were 1123 critical references in the Pascal reference string and 1016 critical references in the PFORT reference string. When performing Ferrari's clustering algorithm, a margin of overflow was permitted when creating clusters. This margin was determined from the page size and the block sizes. A margin of 128 bytes was allowed for in the Pascal program using a 2K page size. The margin was increased to 256 bytes for the FORTRAN program using a 4K page size.

4.4 Experimental Observations

Throughout this chapter an assumption was made that the sequencing of clusters was an integral part of restructuring. An initial simulation run was made to examine this assumption. Figure 6 contains three curves: the non-restructured Pascal program, the restructured Pascal program using Ferrari's clusters but a random sequence of the clusters and the restructured Pascal program using Ferrari's clusters and the greedy sequencing algorithm. This graph clearly illustrates the tremendous importance that intelligent sequencing has. The graph indicates that intelligent sequencing can account for at least 50% of the improvements.

Many different sequencing algorithms were attempted during this research in an effort to improve upon the greedy heuristic. Although several of these algorithms produced favorable results on certain simulation runs, they did not provide consistent improvements. Their failure can be attributed, in part, to the varying cluster sizes and thus the varying location of page boundaries. Another experiment took the greedy sequencing and biased the page mapping by one-half page, that is, instead of placing the first block at location 0 of page 1, the block was shifted down. This slight bias caused a fault increase of up to 35%. Clearly the location of a page boundary, even within the same block, can affect the results. Another problem in the sequencing algorithms attempted, was the handling of large blocks. In those cases where the restructuring reduced interpage faults to a minimal level, faults due to intrablock references distorted the overall statistics. Poor alignment of large blocks

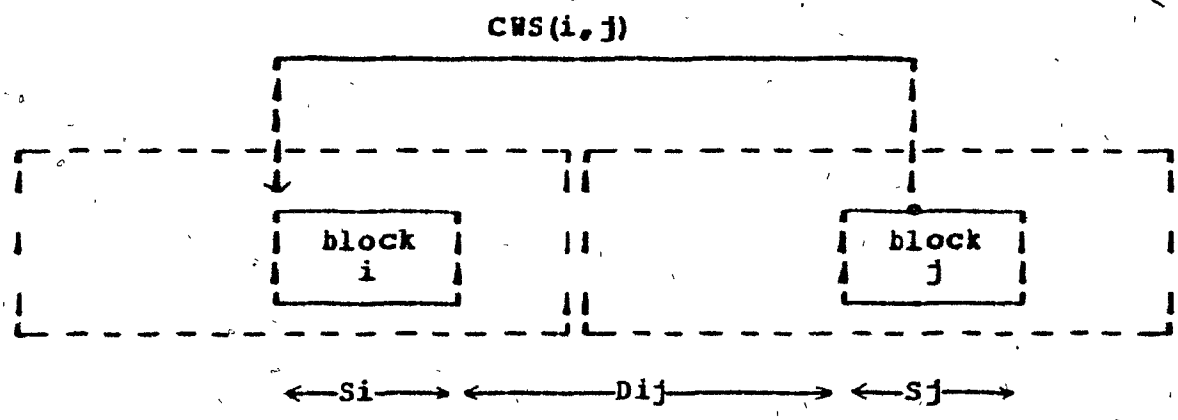
often resulted in more page boundaries within the blocks than appeared acceptable.

The inconsistency of modified sequencing algorithms and the sensitivity of the restructuring to certain parameters required a radical change in the clustering and sequencing. Algorithm CS along with the probabilistic weighting function was proposed and compared against the two test programs. The block/page ratio of the simulation programs are comparable to those of Ferrari's [FERR74a]. If there are more blocks per page, the performance of Ferrari's method improves, as the cluster sizes become more uniform. However, the programs used here are presumably typical and their results should be judged accordingly.

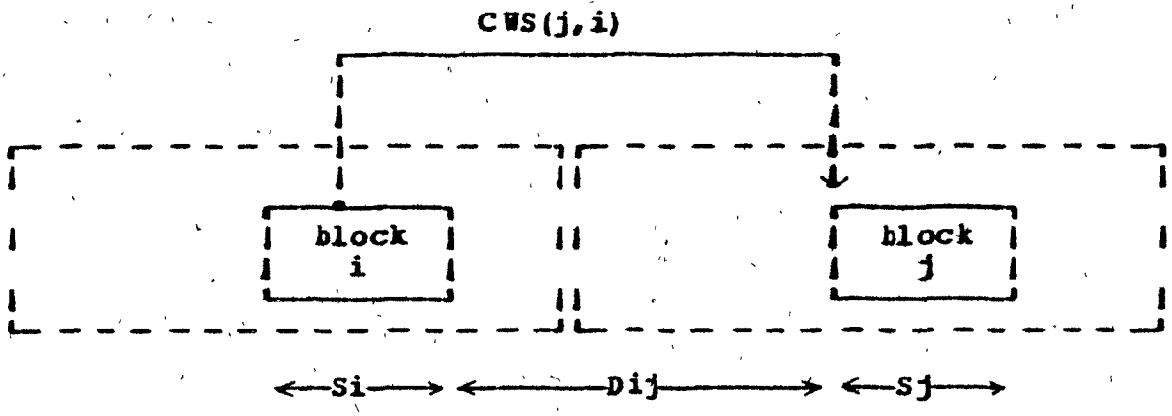
Figure 7 illustrates the restructuring results of the Pascal program. Both restructured versions produce a tremendous decrease in faults compared to the unrestructured program. To the left of the knee, the improved method is definitely superior to the standard approach. In fact, the difference widens as the average working set size decreases or, equivalently, as the working set window approachs the CWS window. To the right of the knee, a slight anomaly takes place. Although this difference might appear as a significant percentage, if the initial load faults were discounted, the actual raw number of additional page faults would be quite small.

Figure 8 illustrates the results of restructuring the PFORT program. The overall characteristics of these curves are quite different from those of Figure 7. This might be due to the different function of the program, an inherent feature of FORTRAN code and/or the presence of many distinct localities. Once

again, the improved method provides the best results, especially for smaller memory sizes. There is also a small anomaly for a mean working set size of approximately 22 pages. This difference is negligible for the same reason as for Figure 7.



(a)



(b)

Figure 4 - Intercluster Referencing

	Pascal	PFORT
Program Size (bytes)	89,136	234,312
Number of Blocks or CSECTs	115	132
Block Size (bytes)		
minimum	72	8
average	775	1775
median	416	1032
maximum	9656	24008
Average Number of Blocks per Page		
2048 bytes/page	2.6	-
4096 bytes/page	-	2.3
Number of blocks > 2048 bytes	11	-
Number of blocks > 4096 bytes	-	8

Figure 5 - Characteristics of Programs

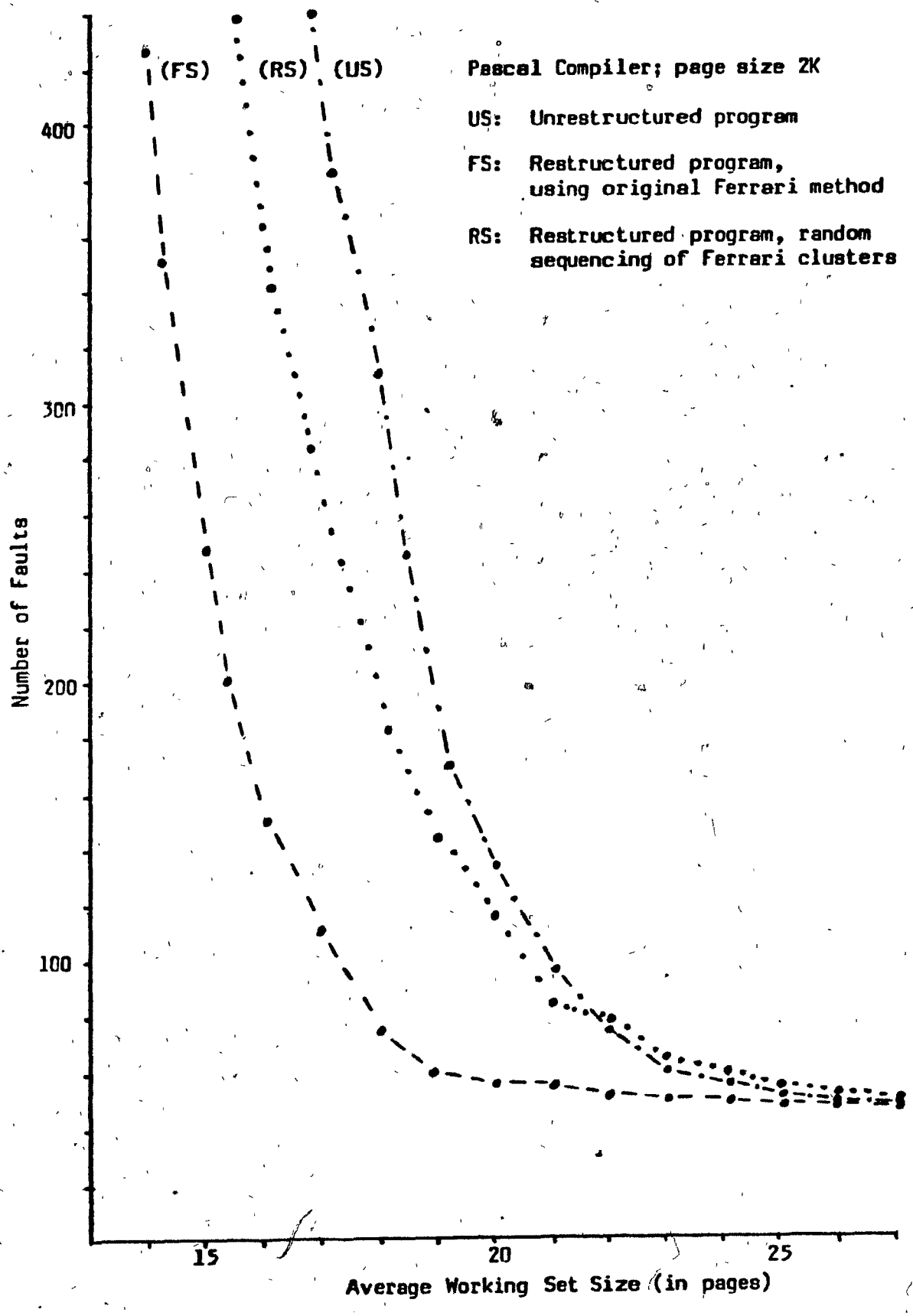


Figure 6 - Random Cluster Sequencing of Pascal Program

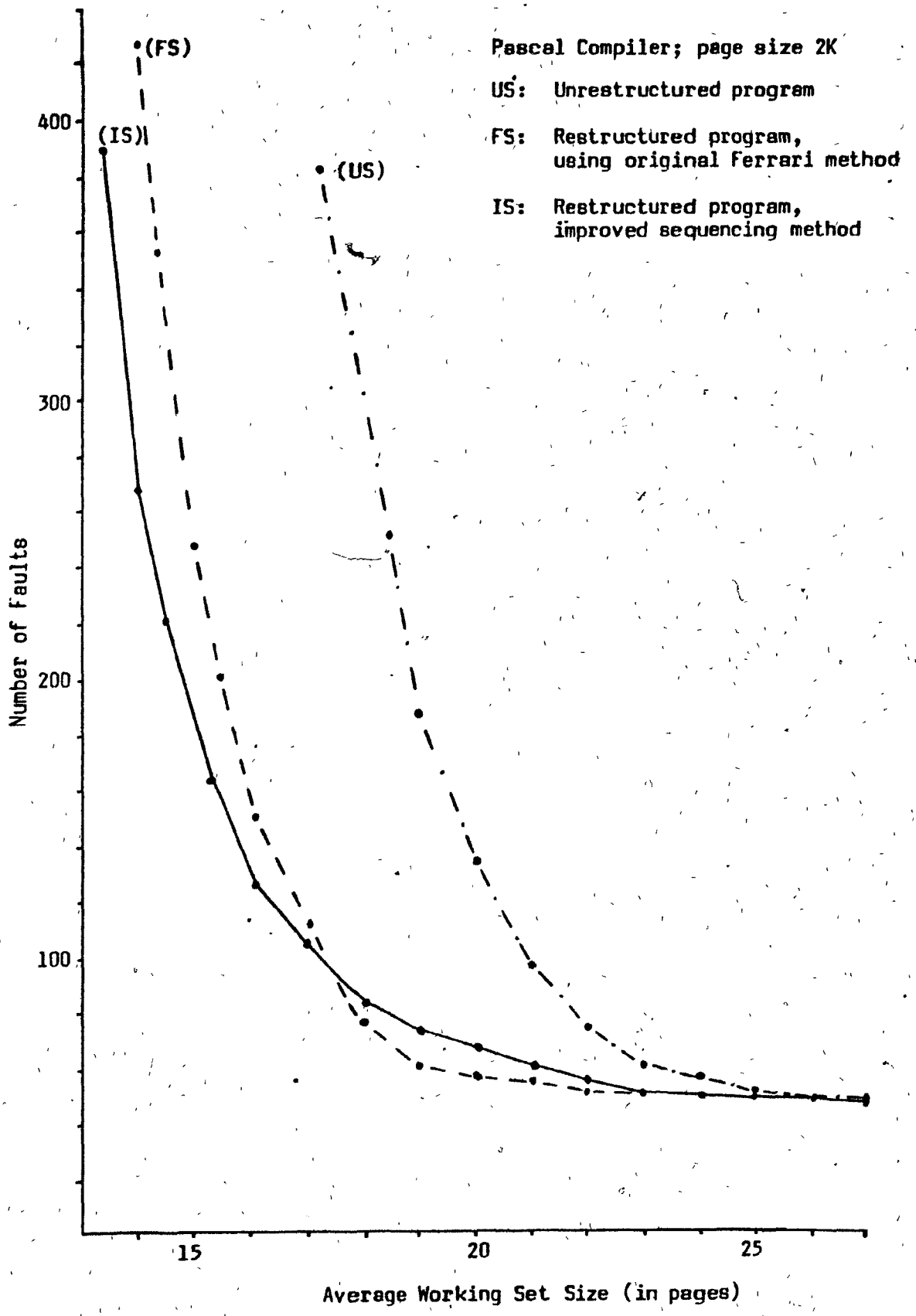


Figure 7 - Improved Restructuring of Pascal Program

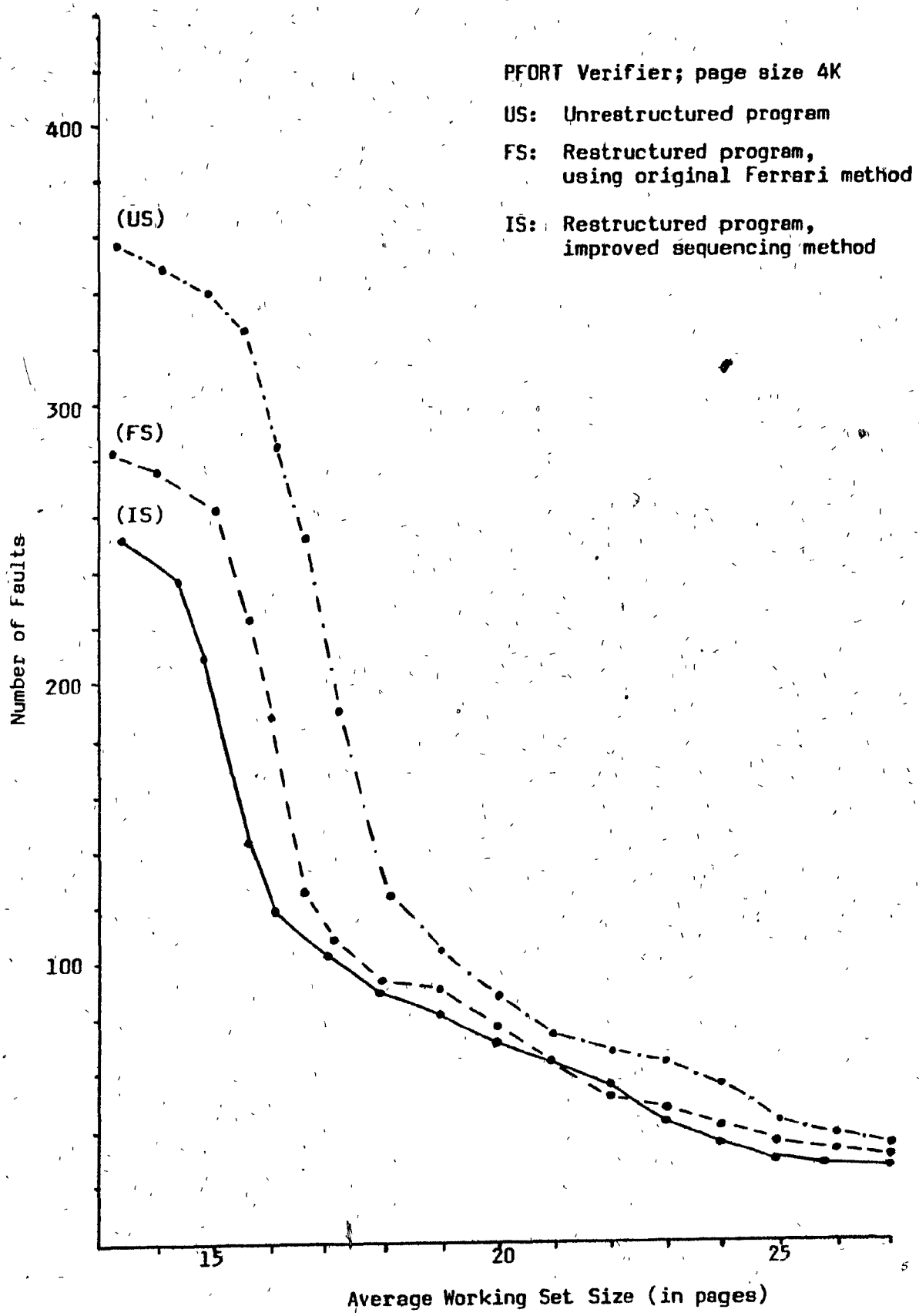


Figure 8 - Improved Restructuring of PFORT Program

Chapter 5 - Sequential Prepaging of Restructured Programs

As discussed in Chapter 3, the main issue in prepaging is determining which pages are most likely to be used in the near future. A method such as Joseph's OBL, prefetches pages sequentially based on the assumption that a block of code on one page may overlap onto the following page. It would be desirable, if the following page is prefetched, that not only the overlapping code be executed, but the rest of the page as well. This would imply some sort of sequential flow from block to block. Since most programs do not follow a strict sequence of routines, we can modify this characteristic. For any block, the following block should be the one most likely to be executed next. If this condition were found in a program, then a prepaging policy that fetches pages sequentially should increase the system performance. Unfortunately, this type of program characteristic does not naturally appear.

The goal of program restructuring is to group together, on one page, program blocks that are most likely to be executed within a short period of time. As well, if blocks overlap pages, a sequencing is used that selects the next page based on the likelihood that blocks on that page will be executed next. Clearly, the restructuring process creates block sequencing that is more desirable for a prepaging environment than non-restructured programs.

To test this hypothesis, a working set version of Joseph's SP algorithm was used. At a fault, the page immediately following the requested page is added to the working set if it is

not already a member. Simulations using this prepaging strategy, unlike the standard working set policy, requires one pass through the reference string for each and every window size desired. When dealing with a half-million references the cost can become quite prohibitive. As a result, only a few data points were accumulated for each simulation run.

Prepaging fault rate statistics were gathered for the non-restructured Pascal program and the restructured Pascal program using the standard Ferrari clustering and sequencing. What is of interest is the decrease in faults for the restructured program as compared to the non-restructured program. In other words, we should expect that the benefits obtained by prefetching the restructured program should be greater than those for the non-restructured program.

Figure 9 illustrates the percent improvement in page faults by using prepaging instead of the normal working set policy for a range of mean memory sizes. The restructured program clearly provides the greatest improvements. Over the range shown, the restructured program is 4% better. This improvement is not as large as might have been expected. A possible reason might be the relatively short reference string length compared to the window size (a slightly larger window size is necessary to ensure that the prefetched pages do not get discarded prematurely). The benefits of efficient prepaging are more significant over a longer period of time when the initial load faults are no longer a factor. Unfortunately, it was not possible to attempt longer reference strings due to the costs involved. Regardless of the magnitude of the improvement, the experiment does indicate that

prepaging of restructured programs can increase system performance and is worth further examination.

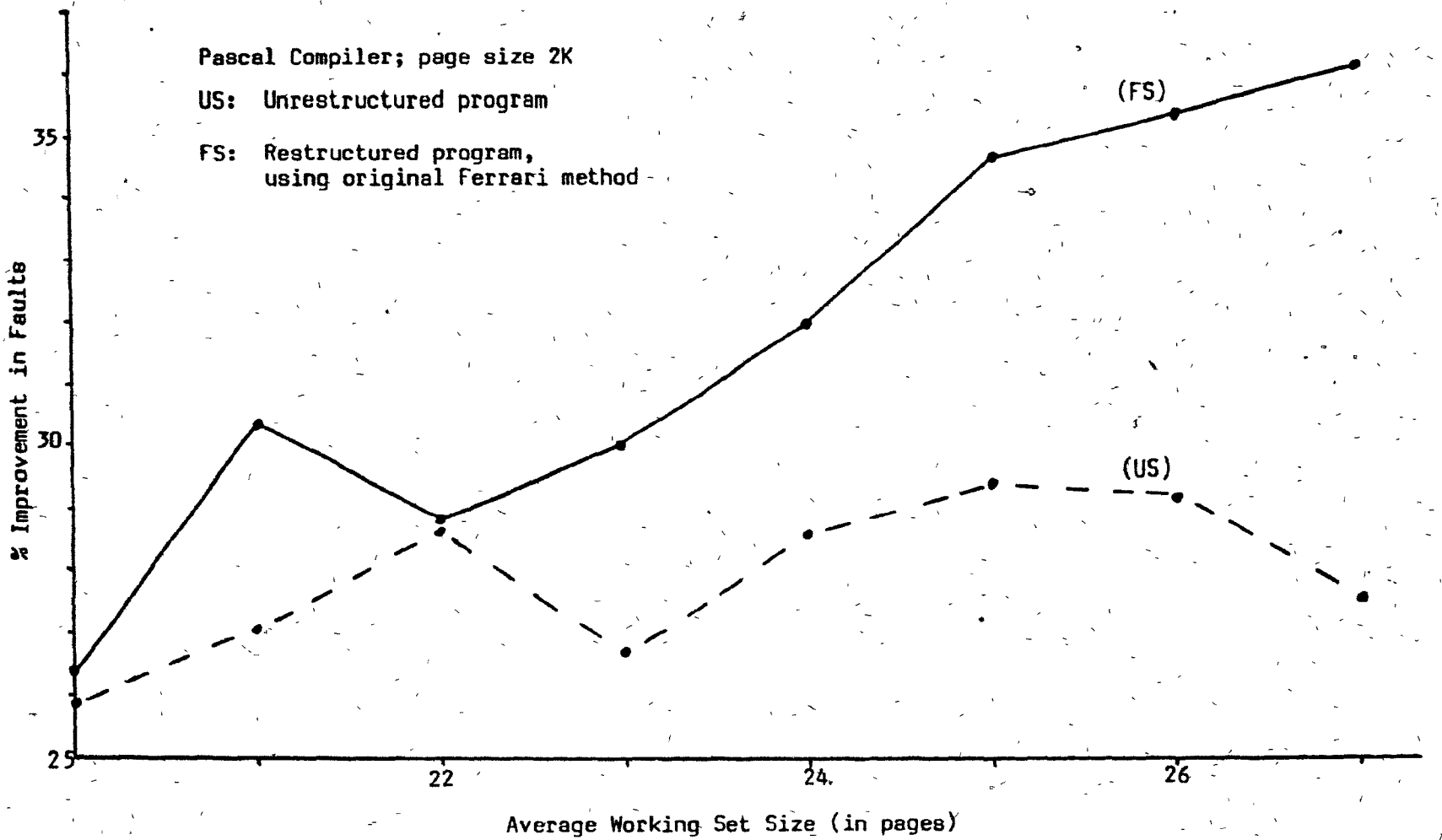


Figure 9 - Fault Improvements Using Prepaging

Conclusions and Proposals for Future Research

Various major restructuring algorithms were presented along with an analysis of inherent deficiencies. Particular attention was paid to the problem of clustering program blocks and to the sequencing of these clusters. An analysis revealed that a flexible sequencing process results in a loss of page boundary information. It was also shown that the alignment of page boundaries is an integral part of restructuring.

As a result, a new approach to the clustering/sequencing problem was introduced. This method begins by considering each individual block as a cluster. Clusters are then iteratively grouped according to the greatest interconnection strengths. The interconnectivity is determined by weighting the interblock contributions according to the probability of a page boundary occurring between the blocks. In other words, the farther apart two blocks lie, the less they contribute to the overall connectivity of two clusters. This method was tested, using Ferrari's CWS matrix as a basis for interconnectivity, and compared against the standard Ferrari technique. Simulations using two test programs showed that the new method produced significantly better results.

A brief examination of prepaging was made and a few well-known strategies were described. The characteristics of restructured programs suggest that a sequential prepaging policy, such as OBL, should enhance the paging performance. A simple experiment was performed comparing a non-restructured program to a restructured version in a prepaged working set environment.

The results indicated that the benefits of prepaging are greater for restructured programs than for non-restructured.

Although the goal of this research was to improve the methods for program restructuring, the new method presented is by no means perfect. There are still certain areas where improvements can be made. One area in particular is the treatment of large blocks. The CWS approach treats blocks as single entities so that the matrix entries provide no information as to which section of the block was most active in interreferencing. If blocks are small compared to the page size, this lack of information provides no handicap. Using the new method on large blocks, however, this information would be useful. What is suggested is that large blocks be split into small blocks for the generation of the CWS matrix. The first step in the clustering process would then be to recombine these small blocks back into large ones. The difference now, is that we can group these large blocks using a weighting of the sub-blocks. This modification might prove beneficial.

Another area worth examining is that of the weighting function for interreferences. The novelty of the algorithm presented is the use of a weighting to reflect the probability of a page boundary occurring between blocks. The function used, however, was a very simple linear weighting. The use of a more sophisticated weighting could produce better results.

The experimental results presented on prepaging provided only a hint of what performance gains can be expected. The prepaging strategy that was selected was simple to simulate and required no unwieldy tables. There are still ways to improve

this strategy so that more of the information provided through restructuring is used. This could involve prefetching the preceding page instead of or as well as the following page since interconnectivity can be bi-directional. Research into a more efficient prepaging policy would also be valuable in terms of simulation costs. Such a policy would permit longer reference strings to be used thereby producing more accurate results.

Bibliography

[ACHA78] Achard, M.S. et al., "The Clustering Algorithms in the Opale Restructuring System," Proceedings of the National Conference on the Performance of Computer Installations, edited by D. Ferrari, North-Holland, pp. 137-153, 1978.

[BAER72] Baer, J.L. and R. Caughey, "Segmentation and Optimization of Programs from Cyclic Structure Analysis," SJCC, Vol. 40, pp. 23-36, 1972.

[BARR79] Barrese, A.L. and S. Shapiro, "Structuring Programs for Efficient Operation in Virtual Memory Systems," IEEE Trans. on Software Engineering, Vol. SE-5, pp. 643-652, Nov. 1979.

[BRAW70] Brawn, B. et al., "Sorting in a Paged Environment," CACM, Vol. 13, pp. 483-494, Aug. 1970.

[BURR77] Burris, D.S. and U.W. Pooch, "The Dynamic Matrix Model," Proc. of ACM National Conference, pp. 386-391, 1977.

[CHU72] Chu, W.W. and H. Opderbeck, "The Page Fault Frequency Replacement Algorithm," FJCC, Vol. 41, pp. 597-609, 1972.

[COME67] Comeau, L.W., "A Study of the Effect of User Program Optimization in a Paging System," Proc. ACM Symp. Operating Systems Principles, 1967.

[DENN66] Denning, P.J., "Memory Allocation in Multiprogrammed Computer Systems," MIT Project MAC, Computation Structures Group Memo 24, Mar. 1966.

[DENN70] Denning, P.J., "Virtual Memory," Computing Surveys, Vol. 2, pp. 153-189, Sept. 1970.

[DENN72] Denning, P.J. and S.C. Schwartz, "Properties of the Working Set Model," CACM, Vol. 15, pp. 191-198, Mar. 1972.

[DENN80] Denning, P.J., "Working Sets Past and Present," IEEE Trans. on Soft. Eng., Vol. SE-6, pp. 64-84, Jan. 1980.

[DUDA73] Duda, R.O. and R.E. Hart, Pattern Classification and Scene Analysis, Wiley, New York, 1973.

[FERR73] Ferrari, D., "A Tool for Automatic Program Restructuring," Proc. ACM Nat. Conf., Atlanta, GA., pp. 228-231, 1973.

[FERR74a] Ferrari, D., "Improving Locality by Critical Working Sets," CACM , Vol. 17, pp.614-620, Nov. 1974.

[FERR74b] Ferrari, D., "Improving Locality by Strategy Oriented Restructuring," IFIP Congress Proc. , Vol. 2, pp.266-270, 1974.

[FERR75] Ferrari, D., "Tailoring Programs to Models of Program Behavior," IBM Journal of Research Dev. , Vol. 19, pp.244-251, May 1975.

[FERR76] Ferrari, D., "Improvement of Program Behavior," Computer , Vol. 9, pp.39-47, Nov. 1976.

[FRAN78] Franklin, M.A. et al., "Anomalies with Variable Partition Paging Algorithms," CACM , Vol. 21, pp.232-236, Mar. 1978.

[GARE79] Garey, M.R. and David S. Johnson, Computers and Intractability- A Guide to the Theory of NP-Completeness , Freeman and Co., San Francisco, 1979.

[GUPT78] Gupta, R.K. and M.A. Franklin, "Working Set and Page Fault Frequency Replacement Algorithms: A Performance Comparison," IEEE Trans. Comput. , Vol. C-27, pp.706-712, Aug. 1978.

[HATF71] Hatfield, D.J. and J. Gerald, "Program Restructuring for Virtual Memory," IBM Systems Journal , Vol. 10, pp.168-192, 1971.

[JOHN74] Johnson, D.S. et al., "Worst Case Performance Bounds for Simple One-Dimensional Packing Algorithms," SIAM J. Comput. , Vol. 3, pp.299-332, 1974.

[JOSE70] Joseph, M., "An Analysis of Paging and Program Behaviour," Computer Journal , Vol. 13, pp.48-54, Feb. 1970.

[KERN69] Kernighan, B.W., Some Graph Partitioning Problems Related to Program Segmentation , Ph.D. Thesis, Princeton University, 1969.

[KOBA77] Kobayashi, M., "A Set of Strategy-Independent Restructuring Algorithms," Software-Practice and Experience , Vol. 7, pp.585-594, 1977.

[MADI76] Madison, A.W. and A.P. Batson, "Characteristics of Program Localities," CACM , Vol. 19, pp.285-294, May 1976.

[MASU74] Masuda, T. et al., "Optimization by Cluster Analysis," IFIP Congress , pp.261-265, 1974.

[MCKE69] McKellar, A.C. and E.G. Coffman, "Organizing Matrices and Matrix Operations for Paged Machines," CACM, Vol. 12, pp. 153-165, March 1969.

[PART79] Partridge, D.R., Measurement, Modelling and Management of Phased Program Behavior, Ph.D. Thesis, Univ. of California, Los Angeles, 1979.

[POOC76a] Pooch, U.W., "A Dynamic Clustering Strategy in a Demand Paging Environment," Proc. of the 4th Annual Symposium on Simulation of Computer Systems, Aug. 1976.

[POOC76b] Pooch, U.W. and D.S. Burris, "A Modified Locality Matrix Model (MLMM) - Dynamic Clustering in a Demand Paging Environment," Proc. ACM Annual Conference, Houston, Texas, pp. 337-343, Oct. 1976.

[RYDE73] Ryder, B.G. and A.D. Hall, "The PFORT Verifier", Bell Laboratories, Murray Hill, New Jersey, May 1973.

[SLUT74] Slutz, D.R. and I.L. Traiger, "A Note on the Calculation of Average Working Set Size," CACM, Vol. 17, pp. 563-565, Oct. 1974.

[TRIV74] Trivedi, K.S., Prepaging and Applications to Structured Array Problems, Ph.d. Thesis, Univ. of Illinois, 1974.

[TRIV76] Trivedi, K.S., "Prepaging and Applications to Array Algorithms," IEEE Trans. on Computers, pp. 915-921, Sept. 1976.

[TRIV77] Trivedi, K.S., "An Analysis of Prepaging," Dept. of Computer Science, Duke University, CS-1977-7, Aug. 1977.

[TSAU72] Tsau, R.F. et al., "A Multifactor Paging Experiment: I. The Experiment and the Conclusions," in Statistical Computer Performance Evaluation, edited by W. Freiberger, Academic Press, New York, pp. 103-134, 1972.

[VERH71] VerHoef, R.W., "Automatic Program Segmentation Based on Boolean Connectivity," SJCC, pp. 491-495, 1971.