IMPROVING PERFORMANCE BY STRATEGY-INDEPENDENT PROGRAM RESTRUCTURING USING BOUNDED LOCALITY INTERVALS

by

BERNARD MING-KI LAW

B.Math., The University of Waterloo, 1978

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

July 1981

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of COMPUTER SCIENCE

The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1W5

Date 17 JULY 1981

ABSTRACT

An efficient strategy-independent program restructuring algorithm based on the empirical studies of phases and transitions in the symbolic reference strings of real programs is developed. The algorithm is formulated on the basis that the majority of the page faults occur during phase transitions. Thus emphasis is placed in grouping those relocatable blocks referenced during phase transitions in the same pages. Some parameters to characterize program behavior are also established. The purpose is to study the relationship between these parameters and the performance of the program.

An experiment to compare the performance improvement of the proposed restructuring algorithm and other major existing algorithms is conducted. The performance indices chosen are the mean working set size and the page fault rate. The problem of data dependency, the cost as well as the portability of program restructuring procedures are discussed.

TABLE OF CONTENTS

LIST OF TABLES AND FIGURES

ACKNOWLEDGMENT

I would like to thank Dr. S. Chanson for his ideas and guidance throughout this work and for his careful readings of the thesis. I also greatly appreciate the efforts of Dr. J. Peck in his reading of the drafts and his comments. Finally, I would like to acknowledge my family, especially my father, for their moral and financial support.

1 Introduction

1.1 Program Behavior

In a virtual memory system, the performance of a program in execution depends, to a large extent, on how its instruction code and data are distributed between the levels of the memory hierarchy, and on how this information is accessed. This is one of the reasons for the interest in program behavior --- the study of the mechanism underlying the observed memory reference pattern of a program. Because of the great influence of program behavior on the performance of systems as well as programs running on them, the study of program behavior is essential to the design of efficient systems and programs.

Various models of program behavior have been developed since the late sixties, for examples: the Working Set Model described by Denning [11], Belady's lifetime function [6], the notion of Bounded Locality Interval (BLI) defined by Madison and Batson [25], the LRU Stack Model [31] and the phase/transiton Model described in Graham's Ph.D. thesis in 1976 [32].

In order to provide the ultimate and indispensable test for the validity of models, measurement is necessary, in all cases, to gather information on how real programs reference their address spaces. From these measurement data, we hope to gain some insight to questions such as: What is the relationship between programming style such as structured programming and reference pattern? What is the relationship between program behavior and program performance How sensitive is program behavior to input data? To this end, it was suggested by Batson [2] that measurement and analysis of program behavior could be done at the symbolic level rather than at the level of machine language. This approach gives us a clearer picture of the flow of the program during execution corresponding to the program's high-level constructs.

1.2 Program Locality

Many measurement experiments have been performed to study the memory reference behavior. A commonly observed property is that a program in execution favors a subset of its segments (blocks) during extended periods of time [18]. This property of reference clustering has come to be known as program locality or locality of reference. Numerous studies [26,5,7,8,29] have indicated that locality is to be found in most programs to at least some degree. Also it has

been reported [20,22,23,1] that by improving the degree of locality of the program through reorganization (see section 1.2) of its relocatable blocks (arrays, procedures etc.), substantial improvements in program and system performance can be obtained.

1.3 Program Restructuring

Programs written under the assumption that main memory is virtually unlimited can, when executing in a paging system, result in situations when much more time is spent performing paging I/O operations than executing the program's instructions. As the rate of paging I/O increases, system performance degrades. It is obvious that it would be highly desirable to minimize the page fault rate. For large programs, some alternatives to complete rewriting of the codes are clearly desirable. The idea of program restructuring was initiated by the experiment performed in 1967 by L.W. Comeau [10]. In his experiment, Comeau pointed out that the rearrangement of relocatable sectors of code over virtual pages can have a profound effect on paging performance. Changing the load-time ordering of the modules in a monitor system resulted in a five-to-one reduction in the number of page faults generated during the course of an assembly.

The program to be restructured is divided into relocatable blocks, such as arrays, procedures, functions etc.. We shall assume that the average size of a relocatable block (the term "block" will be used hereafter to mean a segment of relocatable codes) is small compared to the size of a page (for example, between one-tenth and one-third of the page size) so that several blocks can be packed into a page. The objective of restructuring is to reorganize the blocks of a program such that those that are needed within a relatively short time of one another are found either in the same virtual page or in pages that would otherwise tend to be in physical memory at the same time. This has the effect of reducing the page fault rate.

Program restructuring can generally be broken down into two phases:

(1) A preprocessing phase which defines and computes the "desirability" of grouping blocks together based on examination of the symbolic block reference string. The information gathered can be stored in a desirability matrix M where each entry mij of the matrix represents the desirability of putting blocks i and j in the same page.

(2) A clustering phase which groups the strongly related blocks with high desirability into common pages,

subject to the constraint that each cluster fits into a single page.

1.4 Previous work on Dynamic Restructuring Algorithms

Invariably, dynamic restructuring algorithms are based on a block reference string, (see Appendix II) collected during execution, representing the dynamic behavior of the program to be restructured. In the following examples, we assume that all the entries in the desirability matrix in question are initialized to zero.

1.4.1 Nearness Matrix

The Nearness Matrix was first introduced by Hatfield and Gerald in 1971 [23]. Matrix A = [aij], is a symmetric nxn matrix with indices labelled by block numbers, where n is the total number of blocks in the program. The element aij is the number of times a reference to block j immediately follows a reference to block i. That is to say, aij represents the desirability of placing two blocks i and j in the same page.

1.4.2 Extension of Nearness Matrix

The algorithm proposed by Masuda [27], is based on an extended definition of the notion of nearness: two blocks i

and j are near not only when they are consecutively referenced, but also when reference to them follow each other at a short distance in time. The desirability matrix B = [bij] is a symmetric nxn matrix with rows and columns labelled by the block numbers. At each new reference to block i, issued at virtual time t by the program, bij will be incremented by one, for all j, where j has been referenced during the virtual time interval (t-T,t) where T is a chosen constant known as the window size.

1.4.3 Critical Least Recently Used (CLRU) Matrix

The CLRU restructuring method [22] is one of the restructuring methods that takes the memory management policy into account. They are also known as Strategy-Oriented restructuring methods. In this case, the LRU policy (see Appendix Ia) is the memory policy in question. The CLRU matrix C = [cij] is an nxn matrix with rows and columns labelled by block numbers. By simulating the demand paging LRU page replacement algorithm on the page reference string, we could identify when the page faults would occur and the set of pages that are in main memory at those times. Assuming a program is given m page frames in main memory, the CLRU algorithm will increment all the related elements cij by one, where i is the block which has just generated a page fault and j covers the set of m blocks

at the top of the LRU block stack at the time of the page fault. The idea is that it is desirable to put block i in the same page with one of the blocks already in primary memory as then the reference to block i will not cause a page fault. This philosophy can be used on other memory management policies such as the working set policy described in the next section.

1.4.4 Critical Working Set (CWS) Matrix

The CWS method [22] also belongs to the Strategy-Oriented restructuring family using the Working Set policy (see Appendix Ib) as the page replacement policy. A critical reference is one which causes a page fault to occur. The desirability matrix D = [dij] is a symmetric nxn matrix with rows and columns labelled by block names. Similar to the CLRU algorithm, the CWS algorithm increments all the related elements dij by one, where block i is the critical reference and j is an element in the block working set W(t,T), for all j in W(t,T). (A program's block working set W(t,T) at virtual time t is defined to be the set of distinct blocks referenced in the time interval (t-T+1,t) where T is the window size.)

Although the Nearness Matrix is very simple, the main weakness of the algorithm is that it only considers adjacent

pairs of references. Thus it might have trouble identifying the major localities which requires a more global examination of the reference pattern. The extended version of the Nearness Matrix tries to remedy this problem by considering a larger window of T references (T is usually much greater than 2). The difficulty here is in the choice of T. As localities have different sizes and are generally difficult to predict (often depend on the input data), and the overheads of the algorithms increase approximately exponentially with T, it is easy to see why it will be difficult to choose a proper value of T. The Strategy-Oriented restructuring algorithms work better than the existing non-strategy-specific ones. The major drawback is that for some complex memory management policies, it may not be possible to find the corresponding restructuring algorithms. If the strategy assumed by the algorithm is different from the one actually used by the system, the result could be worse than that resulting from the use of a strategy-independent restructuring algorithm.

1.5 Examples of Clustering Algorithms

The function of a clustering algorithm is to determine, based on the desirability matrix, which blocks should be grouped into common pages, so as to minimize the number of

page faults produced by the program. Various methods used in restructuring experiments can be found in the literature [23,27,19,1]. Most of them, as reported, are reasonably efficient algorithms although none of them is optimal and some cost less than the others. The major constraint of a clustering algorithm is that the sum of sizes of all blocks in a cluster cannot exceed the page size. This gives rise to problems such as aligning blocks to page boundaries, avoiding overlaps of blocks over page boundaries and avoiding excessive fragmentation due to wasted space at the end of pages. Two types of clustering algorithm which have been used in past experiments are presented in the following sections. The first is based on a simple method called nucleus-constructing[1], and the other is a more sophisticated one based on hierarchical classification [1].

1.5.1 Nucleus-constructing

This method presumes that there exists a nucleus of blocks of the program more frequently used than the others and it tries to identify such nucleus. The method defines a weight wi for each block i, such that wi is equal to the sum of all the elements in row i and column i of the desirability matrix. In order to take into account the size of the block, a density function di for each block i is determined from the weight wi as

$$di = \frac{S}{si} * wi$$

where si = size of block i
S = mean size of all blocks

It orders the blocks in a list in decreasing order of their densities. Then it places the blocks into a page in the same order except those which cause an overlap, until the page is filled or the list is completely examined. The next page is then filled in the same way with a new list composed of the remaining blocks ordered in the same manner. This process is repeated until the entire list is exhausted.

The wasted space at the end of a page, as reported in [1],was on the average 2 to 8% with the above algorithm. In general, no compaction is made if the total waste is less than or equal to one virtual page. Hatfield and Gerald [23] pointed out that it may be better to permit some overlaps to avoid excessive fragmentation.

Although this is a very fast and inexpensive clustering algorithm a major pitfall is that it does not lend itself in finding the different nuclei of a program if they exist. This is because blocks belonging to different nuclei are mixed if they happen to have similar densities.

The hierarchical classification technique described in the following section overcomes this problem.

1.5.2 Hierarchical Classification

Let J be the set of blocks to be classified. Let D = [dij] be the desirability matrix obtained in the preprocessing phase. Our goal is to classify J into subsets, each of which represents a cluster of blocks that are strongly associated to each other based on the given desirability measure. Given a list of blocks bi, i=1,..,n with size si and page size P, the clustering algorithm works as follows:

The construction of the hierarchy is to be carried out in (n-1) steps (n is the total number of blocks in J). Progressing from step (m-1) to step m consists in passing from the set Jm-1 to the set Jm such that:

$$Jm = Jm-1 - \{A\} - \{B\} + \{A \cup B\}$$

where: A, B are elments of Jm-1 and subsets of J such that dAB is the largest element in the desirability matrix Dm-1 in step (m-1).

$$J0 = \{ \{ bi \} \mid bi \text{ is an element of } J, i=1,..,n\}$$

In other words, J0 is a set of subsets of block(s) (a set of clusters), with each subset initially containing a single block. In each step, we try to merge two clusters together

based on the reduced desirability matrix Dm with D0 = D, to form a new cluster under the constraint that the sum of sizes of all blocks in the new cluster cannot be greater than the page size P.

In detail, step m consists of the following operations:

(1) From matrix Dm-1, find the largest element dAB, where A and B are two elements of Jm-1.
If the largest element equals to zero then the process is terminated as that indicates no new cluster can be formed based on Dm-1.

(2) If the sum of sA (sum of sizes of all blocks in cluster A) and sB (sum of sizes of all blocks in cluster B) is smaller than or equal to P (the page size) then we merge A and B, i.e. A U B, to form a new subset C and continue with (3). Otherwise, we replace dAB of Dm-1 with zero, indicating that the merge is unsuccessful, and then continue with (1) again.

(3) Construct the reduced matrix Dm from Dm-1 by deleting all the rows and columns associated with A and B. And then add row C and column C to Dm, such that:

$$dCX = dAX + dBX$$

and $$dXC = dXA + dXB$$

where dAX,dXA,dBX,dXB are elements of Dm-1,
for all X in Jm-1 except for A or B

Note that Dm is now the new desirability matrix with rows and columns labelled by cluster names.

In summary, we start off with J0 which is a set of n clusters, each of which consists of a single block in set J; and eventually we end up with Jn which is a set of new clusters, each of which consists of a subset of block(s) and whose size is less than or equal to the page size. An example illustrating the method is described in Appendix III.

Optimization of space usage should be done in order to minimize the wasted space at the end of pages, especially for small clusters. Placement is made with emphasis on the size condition rather than on the desirability between two clusters in this last step.

1.6 Objectives

The main objective of this thesis is to develop an efficient restructuring algorithm which is strategy-independent. The work is based on the empirical studies of phases and transitions (details in section 2.1)

in the symbolic reference strings of real programs, (specifically Pascal programs). Also we wish to establish some parameters to characterize program behavior and to analyze the relationship between these parameters and the performance of the program.

This brief introduction in this chapter hopefully provides sufficient background on the motivation of program restructuring and some of the techniques reported in the literature. We then proceed, in chapter 2, to describe in detail the phase/transition model and the Bounded Locality Interval (BLI) definition which led to the construction of a new algorithm which we call the BLI restructuring algorithm, with the emphasis on phase transitions. In chapter 3, we describe the setup of the experiment, the measurement tools involved and the implementation of the BLI restructuring algorithm. Analysis and discussions of the experimental results are presented in chapter 4.

2 BLI Restructuring Algorithm

This chapter begins with the introduction of the phase/transition model of program behavior. The strengths and the weaknesses of the model are briefly discussed. The problem of identifying all the distinctive phases is raised and the concept of Bounded Locality Interval (BLI) is discussed. A new algorithm based on the concept of BLI is described in the last section of the chapter.

2.1 Phase/Transition Model

From the viewpoint of program locality, a program's execution time can be regarded as a sequence of locality phases (or simply phases) separated by transitions. Informally, a phase is an interval during which the small set of blocks being referenced is constant; and a transition denotes the interval of migrating from one phase to another. By subdividing the program into blocks, the locality set of a phase (phase set) is the set of blocks active in that phase. (A given block is considered active in a phase whenever processing of that phase requires the presence of that block in main memory.) Similarly, the transition set is the set of blocks active during the transition. As proposed

by Spirn [31], the execution of a program may be viewed as a sequence

$$\{ (P1,l1),...(Pi,li),.....\}$$

where Pi is the ith locality set or phase set
and li is the lifetime of the phase for Pi
(informally, the lifetime of a phase is the difference in virtual time between entering and leaving of a phase.)

The phase/transition model of program behavior has been studied in detail by Graham [32] in his doctoral research. His work deals mainly with the model as a synthetic generator of reference strings. Though it is interesting and important in its own right, the emphasis and applications of his work are quite different from ours. One of the major properties of the phase/transition model is that it permits a wide range of detail to be incorporated with the reference string generation process. In one extreme, the model may assume that the lifetimes of all locality sets have length one in which case the model consists of the entire reference string. Obviously this will preserve all the details of the program behavior. In the other extreme, the model may assume that only one locality set, namely the program space itself, exists during execution. In this case, a lot of details will be lost. Clearly, neither of these extremes is reasonable in gaining a better understanding of locality. Madison and Batson [25] offered a method for discerning phases and transitions with

the notion of Bounded Locality Interval (BLI) and they undertook empirical studies of phases and transitions in the symbolic reference strings of real programs.

2.2 Bounded Locality Interval

A basic difficulty with the phase/transition model is the one of formulating a procedure which could identify all the distinctive phases and their corresponding locality sets given a reference string [15]. The notion of the Bounded Locality Interval (BLI) was introduced by Madison and Batson [25] as a mechanism to overcome this problem. The initial idea of BLI was triggered by the obervation that the least-recently-used (LRU) stack (see Appendix Ia) contains, at any time t, the blocks arranged in the order of the times at which they were last referenced, with the most-recently-used block at the top of the stack.

If the top i elements in the LRU stack are { Pi } then we can also record fi, the time of formation of this set { Pi } and ei, the last time of reference to the block currently occupying the ith LRU stack position. At any instant t, an activity set is defined as any { Pi } in which every element of that set has been referenced more than once since the set has been formed at the top of the LRU stack.

The lifetime, li, of such an activity set is defined to be the diffence between fi and ei where ei > fi. A BLI is the pair (Ai,li), such that Ai is the activity set and li is its lifetime.

The notion could be generalized by defining an activity set as one whose elements have been referenced at least k times since the set was formed. In particular, the definition given is for the case when k = 1. Moreover, k is the only parameter of the model and it is independent of any memory management policies.

Another important characteristic of BLI is the implicit hierarchical nature of localities embedded in the definition. The hierarchy of BLI's is illustrated in Figure 2.1, which shows a segment of a reference string and the corresponding BLI structure, with the parameter k = 1. The level of a BLI is defined as its distance down in the hierarchy. Thus, in Figure 2.1, the "level one" BLI's are those at the top of the hierarchical structure, partitioning the reference string into the longest possible subintervals of distinctive referencing behavior.

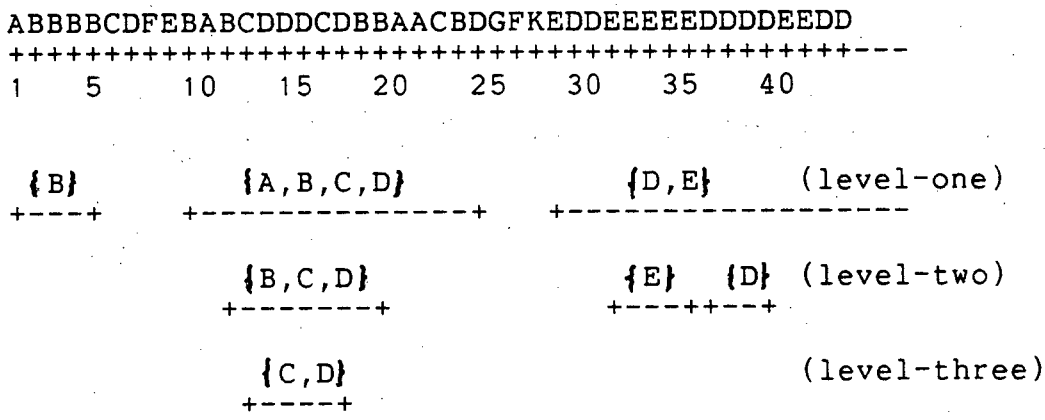

Figure 2.1. Hierarchy of BLI's for an example symbolic reference string.

The development of the concept of the BLI could be used to solve the problem of identifying distinctive major phases in a given reference string, except that we still have to determine whether a phase is really a "major" phase. Intuitively one would expect that a major phase should have a reasonably long lifetime. Batson [3] pointed out that the criterion of "reasonably long" can only be formulated in the context of the particular virtual memory system upon which the program will be executed. He suggested that the mean time required to transfer a block from secondary storage to main memory can be used to determine a sequence of major phases and transitions. If p is the block transfer time, each BLI is regarded as a major phase if its lifetime is

greater than or equal to p.

Besides the importance of the behavior of the major phases and their nesting characteristics, there are significant disruptive transition-intervals between major phases. While the major phases dominate virtual time, the transitions account for a substantial part of the page faults. It was reported by Kahn [24] that phases covered at least 98% of virtual time while the rate of page faults during transitions was 100 to 1000 times higher than that during phases.

A simple way of approximating the BLI concept in the identification of the level-one major and transition phases is as follow. Scan the block reference string from the left. The first repetition of a block reference is the (possible) beginning of a major phase and also the (possible) termination of the current transition phase (depending on whether the eventual length of the phase exceeds p). A major phase terminates if a reference is made to a block not contained in the previous transition phase set. Our empirical data show that this will obtain almost identical level-one major and transition phase sets with much less overheads. Subsequent results reported in this thesis were obtained using this approximation method.

2.3 The BLI Restructuring Algorithm

2.3.1 Motivation

It was the last remark, in the previous section, about the effect of transitions on program performance that led to the idea of the BLI restructuring algorithm. Basically, the goal of program restructuring is to reduce the page fault rate generated by the program and traditionally experiments on program restructuring have concentrated on the major phases or localities. However, the studies of Denning and Kahn [15] indicated that, for executable memory size greater than the mean phase size, performance was much more dependent on the characteristics of phase transitions than either the program behavior within phases or the memory management algorithm in use.

The BLI restructuring algorithm is constructed with the following objectives in mind:

(1) Strategy-independent.

The algorithm should be independent of any memory management policy thus having the widest possible domain of application.

(2) Emphasis on phase transitions.

Besides trying to place all blocks in the major phase into common pages, we emphasize on the strength of "nearness" of blocks during transition intervals. This will achieve better utilization of the information provided by the program behavior model, and more importantly we hope to obtain better performance by reducing the page fault rate.

The BLI model is the perfect choice for our purposes. It is strategy-independent and it provides a clear definition of major phases and transitions given any reference string. In building the desirability matrix, unlike other algorithms described in section 1.4, we put more weights on transitions than on phases. We group the blocks in the transition interval the same way as the locality sets or the activity sets despite their very short lifetimes, as long as the lifetime of each set is greater than the block transfer time from secondary storage to main memory.

In the clustering stage, the major constraint is that the sum of the sizes of all blocks in a cluster cannot be greater than the page size. Wasted space at the end of pages is reduced by combining clusters that may fit into a

page as follows: The clusters are listed in decreasing order of their sizes. They are then placed into a page in the same order skipping those which cause an overlap, until the page is filled or the list is completely examined. The next page is then filled in the same way with a new list composed of the remaining clusters ordered in the same manner. This process is repeated until the entire list is exhausted.

2.3.2 The Algorithm

In detail, the BLI restructuring algorithm consists of the following steps:

(1) Using the definition of BLI, with a chosen value for the parameter k (k is used to determine the activity set) and given the block transfer time p, a sequence of transitions and major phases can be obtained from any given block reference string. In order to keep the cost of the algorithm within reasonable limit, we will only concern ourselves with the "level one" BLI's of the hierarchy. (To obtain an optimal path through the hierarchy is too costly for program restructuring. Moreover, the "level one" BLI's are those at the top of the hierarchical structure (see Figure 2.1),

partitioning the reference string into the longest possible subintervals of distinctive referencing behavior.)

(2) Each of the transitions and major phases composes of a set of distinct blocks which are ordered with respect to their times of first occurrence. Now we have a choice of considering

(a) the phase sets only

(b) the transition sets only

(c) both the phase sets and the transition sets

(3) Then we have to decide how to determine the "desirability" of block pairings in each of the sets chosen from step (2). The desirability matrix M = [mij] is a symmetric nxn matrix with indices labelled by block numbers, where n is the total number of blocks in the program. Two methods are described for the computation of mij, the desirability of placing two blocks i and j in the same page.

(a) the Neighborhood method

For each set of blocks chosen from step (2), arrange the blocks in a list in the order of their occurrence within that phase or transition. Increment mij by one for any two blocks i and j in

the list such that block i is followed by block j, for all i in the list.

(b) the Combination method

For each set of blocks chosen from step (2), and for all combination of block pairings i and j in the set, increment mij by one where i is different from j.

2.3.3 Examples of BLI Restructuring Heuristic

In effect, we have described a whole category of heuristics for program restructuring based on the notion of BLI. The following are two examples of such heuristics from the category which are found to be particularly interesting and hence they are chosen to be used in the experiments.

Example 1: Phase-Transition-Neighborhood-k=1 (PTN1)

This method uses the BLI definition with k=1. It considers both the phase sets and the transition sets. The entries of the desirability matrix are determined by the Neighborhood method described in 3(a). (This is, in effect, similar to Hatfield's nearness matrix except that the weights on block pairings in major phases are greatly reduced.)

Example 2: Transition-Combination-k=2 (TC2)

In this method, we use k=2 to determine the major phases and the transitions. However only the transition sets are examined. (It is observed that a transition set always includes the set of blocks in the next major phase. Hence, in effect, we have already implicitly considered the blocks in the major phases though their weights are further reduced, despite their much longer lifetimes.) The desirability of block pairings is computed using the Combination method described in 3(b).

In the next chapter, we describe the design of the experiment with the BLI restructuring algorithm. The objectives of the experiment are two-folded:

(1) To evaluate the performance of the BLI restructuring algorithm compared to other algorithms used in the literature.

(2) To study the relationship between some proposed parameters of program behavior and the performance of real programs.

3 Description of the Experiment

3.1 Overview

In this chapter, the experiment which was conducted will be described in detail. The main objective of the experiment is to evaluate the performance of the BLI restructuring algorithm compared to the other major existing algorithms. Also we wish to establish some parameters to characterize program behavior and hopefully to gain some insight through the analysis of the relationship between these paramters and the performance of real programs.

Having decided to do a measurement experiment, the first step is to determine what performance measures to use and then to identify the major parameters related to the program behavior model which may influence the selected performance indices. The next step is to develop tools to collect block reference strings from real programs and to implement the various restructuring algorithms (including the Nearness Matrix, the Critical Working Set restructuring algorithm and the BLI restructuring algorithms). The performance of the restructured program is then evaluated by running the reference string on the simulated memory

management program to compute the performance statistics and other parameters. Following this, we discuss the type of program used in the experiment and the variety of data input included to test for data dependency. Finally the design and implementation of the experiment are discussed in the last section of the chapter.

3.2 Approach

The experiment is based on block reference strings (see Appendix II) gathered from real production programs running on an AMDAHL 470V/8 computer. These programs consist of relocatable blocks such as procedures and functions. They were executed on the Michigan-Terminal-System (MTS) Symbolic Debugging System (SDS) from which symbolic traces of procedure calls and returns were collected and extracted to form the resulting block reference strings. At the same time, the block sizes were recorded for use in the clustering phase.

The references were then fed to the restructuring programs based on the different restructuring algorithms to produce the corresponding desirability matrices. Each matrix was then input to the clustering program which output a set of virtual pages each of which contained a cluster of

blocks.

The performance of the restructured program was estimated by simulation. The block reference string was first transformed into the corresponding page reference string according to the mapping suggested by the restructuring and clustering procedures. The page reference string was fed to a memory management simulator which gathered the desired performance statistics. Because of its popularity and the relationship to one of the major restructuring algorithms (i.e., the Critical Working Set algorithm), the working set memory management policy was used throughout the study.

3.3 Parameters

3.3.1 Performance Indices

Traditionally, the page fault rate is an important performance index in virtual memory systems. The reason is that the turnaround time of a program is strongly influenced by the number of page faults generated during its execution. This is because the time needed to transfer a page from secondary storage to main memory is about four orders of magnitude larger than the access time of primary storage.

Apart from the size of main storage, the number of page faults generated during an execution depends mainly on the behavior of the program being executed and on the memory management policy used by the system. Since we have decided to use the working set policy as the replacement policy throughout the experiment, the average working set size is a good indicator of memory usage during the program's execution.

Thus, the performance indices chosen are the page fault rate and the mean working set size which together cover, to certain extent, the space and time components of a computational activity.

3.3.2 Controllable Factors

A controllable factor is a factor whose levels (or values) are under the direct control of (and therefore can be chosen by) the experimenter.

Parameters which may affect the behavior of a given program are its input data and the order in which its blocks are stored in the virtual memory space. The influence of the ordering is due to the fact that the page contents would generally be different for different orderings. Thus,

different parts of the program are maintained in main memory for different ordering of blocks in the virtual address space. As described in Chapter 1, the original ordering of blocks in the virtual storage can be altered by the use of a restructuring algorithm and a clustering algorithm. Thus, the three controllable factors selected for the experiment are:

(1) input data,

(2) the restructuring algorithm,

(3) the clustering algorithm.

Since we have decided to use the working set policy as the replacement algorithm, we might include the window size T (the only parameter of the Working Set policy), as the fourth controllable factor.

These controllable factors are selected with the view to compare the performance of the BLI restructuring algorithm with the others. The four controllable factors and their levels selected are listed in Table 3.1 under section 3.5.

3.3.3 Observable Factors

An observable factor is a factor whose levels can be

measured during the experiment but they are not under the direct control of the evaluator. The observable factors of our experiment depend very much on the program behavior model in use. With the phase/transition model and the notion of BLI, the natural choices are the average cardinality of the phase sets and of the transition sets. Intuitively, we expect: (1) the average phase set size to be approximately equal to the mean working set size; and (2) large average transition set size will give rise to high page fault rate. It is one of our goals to see if there is any correlation between these parameters and the performance of the measured programs.

3.4 Measurement Tools

3.4.1 Data Collection and Reduction

Data collection was done using the Symbolic Debugging System (SDS) on the AMDAHL 470V/8 computer which operates under the Michigan Terminal System (MTS). SDS includes a simulator (originally designed for the IBM 370/168) which simulates the instructions step by step. The SDS simulator provides a TRACE CALL command which can establish breakpoints at the base of all the control sections (subroutines) and at their entry points. During the

simulation of an instruction, if a call intercept (i.e., a breakpoint) is encountered, the symbolic address of the routine being called and the return address of the calling routine are recorded and a return intercept is established at the return address. Similarly, when a return intercept is encountered the address of the returning routine and the return address are recorded and the return intercept is removed. Hence after the program's execution (by the simulator), a trace of subroutine calls and returns is recorded. The cost of this TRACE CALL command is fairly high due to simulation of the instructions and frequent interference for data collection at the breakpoints. The cost is, as expected, roughly proportional to the number of instructions being simulated and the number of subroutine calls made during the execution.

A data reduction program was written to transform the collected information into a more concise and useful form. The program includes a mapping of each subroutine or block name into an unique positive integer and hence reduces the collected trace into a sequence of positive integers which represents the flow of execution of the measured program at the symbolic or subroutine level. This sequence of block numbers is known as the block reference string.

The SDS also produced a list of block names of the measured program and their corresponding lengths. This information was used in the clustering stage.

3.4.2 Restructuring Algorithms

Three types of restructuring algorithms were implemented. They were Hatfield's Nearness Matrix, Ferrari's Critical Working Set and the BLI restructuring algorithms. All of them were coded in Pascal. Each accepts a sequence of positve integers as input and scans the entire string in a single pass, producing the corresponding desirability matrices. In the case of the BLI program, the user can specify the required options (such as PTN1, TC2, etc.) as described in section 2.3.3. In addition to the desirability matrix, a sequence of phases and transitions, each containing a list of distinct block numbers and their lifetimes is also produced by the BLI algorithm.

Two clustering algorithms, as described in section 1.5, were implemented in Pascal. Given a desirability matrix, the block sizes and the size of a page, the clustering program would produce a set of pages and the block numbers associated with each page. An approximation of wasted space is also displayed.

3.4.3 Working Set Policy Simulator

The function of the simulator is to estimate the performance of the restructured program based on the original block reference string collected and the placement ordering of blocks suggested by the restructuring algorithm. Given a set of pages (clusters) and their corresponding block numbers and the window size, the simulator outputs the number of page faults and the average working set size associated with the restructured program.

Note that the simulator implements the "pure" working set replacement policy which means only the pages covered by the window are regarded as the program's working set at any time instant. The simulator was coded in Pascal.

3.5 Design and Implementation

The program experimented on was a Pascal compiler. It was written in Pascal and considered to be well structured. It consisted of 336 blocks (procedures and functions) and about 90% of these had sizes less than 800 bytes. The average block size was about 600 bytes. Thus for a page size of 4K (4096) bytes, each page could hold about 7 blocks

on the average. The compiler consisted of about 54 pages of codes.

In section 3.3 we identified the major factors felt to influence the page fault rate of a program. The different values of a factor (both quantitative and nonquantitative) are called its levels. The four controllable factors selected and their corresponding levels for the experiment are listed in Table 3.1.

The levels for the input data were selected to cover a variety of Pascal programs to be compiled with respect to the number of syntactic errors and the length of the programs. Program P was designed to include most of statement types in Pascal and had about 25 statements. The first four samples of the input data were modified versions of program P ranging from zero to 50 syntactic errors. The fifth and sixth samples were both error-free programs which had about 60 and 355 statements respectively. Hopefully that would allow us to test for data dependence on input of different nature as well as sizes.

Five restructuring heuristics were selected for comparsion. The original ordering of the blocks by the author of the program went through no restructuring and thus

served as the control of the experiment. Hatfield's Nearness Matrix was included partly for historial reason and partly because it is still the simplest, strategy-independent and yet the cheapest restructuring algorithm known. The CWS restructuring method was chosen for obvious reasons: it seems to have the best performance among the algorithms reported in the literature. It is a strategy-oriented algorithm and was purposely chosen since we decided to use the working set policy for memory management.

Both the PTN1 and TC2 are members of the BLI restructuring algorithms. PTN1 resembles the Nearness Matrix but with less weights on the phase sets. While TC2 is the natural product of our intuition and past experience that disruptive phase transitions are the major cause of page faults.

Six block reference strings were gathered from different input data. Each of them was then fed to the five restructuring programs and two clustering programs. A total of 60 sets of clusters was produced. The working set policy simulator went through all 60 sets of clusters twice, each time with a different window size. As a result, 120 runs were made, each of which produced a pair of figures

representing the number of page faults and the average working set size.

Additional runs were performed to test the result of running all the reference strings with a particular set of clusters chosen from TC2 and to compare the results to the ones using the CWS algorithm. This was done to test the robustness of the two algorithms.

Table 3.1 Factors and Levels for the Experiment

Program: a Pascal compiler
Replacement policy: working set policy

FACTORS	LEVELS	
	NAME	DESCRIPTION
Input data (program to be compiled)	P1	program P, 25 statements, 50 errors
	P2	program P, 25 statements, 25 errors
	P3	program P, 25 statements, 5 errors
	P4	program P, 25 statements, no errors
	P5	program Quicksort, 60 stts, no errors
	P6	program BLI,355 statements, no errors
Restructuring Algorithms	ORIG	original ordering by author
	NEAR	Hatfield's Nearness Matrix
	CWS	Critical-Working-Set
	PTN1	Phase-transit-neighbor-k=1 (BLI)
	TC2	Transit-combination-k=2 (BLI)
Clustering Algorithms	NUCL	Nucleus-constructing
	HIER	Hierarchical classification
Window size (reference)	10	small
	50	large

4 Discussion of Results

Six block reference strings were gathered for a Pascal compiler executed with different input data. A number of simulation runs were performed to evaluate the restructuring procedure and several restructuring algorithms as described in section 3.5. In this chapter, we shall summarize what we feel to be the most important results in the following areas: performance improvement, data sensitivity, portability and cost.

4.1 Performance Improvement

Improvements in page fault rate and the average working set size by program restructuring are computed, in terms of percentage reduction in those indices as:

$$\% \text{ Reduction} = \frac{Po - Pr}{Po} * 100$$

where Po, Pr are the original and restructured performance indices respectively.

Before we discuss the performance of different restructuring algorithms, we make the following observations:

(1) Window Size

A reasonable range of window sizes, from 10 to 100 page references, was tested initially for our performance evaluation. It was found that smaller window sizes produced smaller gain (with respect to % reduction in page fault rate). A window size of fifty page reference was found to be the optimal for all restructuring algorithms in the experiment and was therefore chosen for performance comparsion among the restructuring algorithms throughout the study.

(2) Clustering Algorithm

Despite the simplicity of the Nucleus Constructing method (NUCL), our results showed that clustering by Hierarchical Classification (HIER) is much superior to NUCL in all cases. For the same reference string and restructuring algorithm, HIER outperforms NUCL by an average of 20% in our experiments. This can be explained by the inability of isolating distinct nuclei (a set of blocks which are closely related with respect to the program's referencing behavior) by the NUCL method. Thus, we shall concentrate on the results gathered from the use of the HIER method.

In almost all cases and with all four restructuring algorithms tested, significant performance improvement was

obtained in both indices (see Table 4.1 and Table 4.2). The page fault rate was reduced from 22% to as much as 42.5%. The average working set size was reduced to about two-thirds compared to the performance associated with the original ordering in most cases. In general, the original layout of the program has great influence on the magnitude of the improvement. Since our test program, a Pascal compiler, is considered to be well-structured, the performance improvement shown is very significant.

Among the four restructuring algorithms, TC2 (a member of the BLI restructuring algorithm) works extremely well in reducing the page fault rate and outperforms the others by non-neglible amounts. The results of TC2 confirm the claim that most page faults occur at phase transitions. Hence putting more weights on the transition blocks can reduce the page fault rate significantly. While the CWS algorithm comes second to TC2 in the reduction of page faults, in some comparsions it is slightly more effective in reducing the average working set size. As expected, CWS works best when the window size selected for the working set policy simulation matches with the restructuring parameter.

As a matter of fact, both the NEAR and PTN1 algorithms, though inferior to TC2 and CWS, perform reasonably

satisfactorily for their relative simplicity as well as low cost. In the case of page fault rate, NEAR and PTN1 manage to obtain an average reduction of 25% and 31% respectively. The percentage reduction in the mean working set size for all four algorithms turned out to have the same order of magnitude.

Table 4.1. Comparsion of the 4 restructuring algorithms on percentage reduction in page fault rate

reference string	restructuring algorithms			
	NEAR	CWS	PTN1	TC2
P1	23.4	32.3	32.0	37.8
P2	25.3	33.2	35.4	36.1
P3	22.2	28.7	29.9	35.6
P4	26.1	31.5	31.0	41.8
P5	30.8	33.4	33.0	42.5
P6	25.9	35.2	33.7	40.1

Table 4.2. Comparsion of the 4 restructuring algorithms on percentage reduction in average working set size

reference string	restructuring algorithms			
	NEAR	CWS	PTN1	TC2
P1	27.5	31.6	27.5	28.6
P2	27.2	27.8	27.2	23.1
P3	25.2	28.7	25.7	25.6
P4	24.7	30.4	23.7	26.3
P5	28.2	25.2	25.6	30.5
P6	24.8	26.9	26.1	28.9

Table 4.3 Average transition set sizes and average phase set sizes (blocks)

reference string	average phase set size	average transition set size
P1	4.180	8.109
P2	4.302	10.050
P3	4.550	10.027
P4	4.397	11.830
P5	4.445	10.336
P6	4.444	10.281

Next we examined the performance between TC2 and CWS by looking at the distributions of working set sizes during the simulated execution of the restructured program (see Figure 4.1). We observed the same kind of behavior in all cases with various reference strings. Their properties are described as follows:

(1) Both TC2 and CWS reach almost the same maximum in working set size. They perform better than NEAR and PTN1 by 20% and non-restructuring by 40%.

(2) In the CWS curve (see Figure 4.1), the working set size with the highest frequency of occuring were less than or equal to four pages. While TC2 had its majority ranges from 2 to 5.

(3) The CWS curve drops abruptly after the first peak and then maintain a gentle slope and forms a tilted platform at the end. While TC2 starts to climb very sharply to the left of the peak, and slides down smoothly to the bottom as the working set size increases. Thus the probability of having the very large working sets is less in TC2.

The behavior of the two curves can be better understood by studying the different nature and objectives of the two restructuring algorithms. Property (2) can be explained by the fact that CWS tends to predict the major phases very well and is thus able to maintain the high frequency working set sizes to be equal to or less than the average size of the major phase sets. During phase transitions, the fault rate as well as the working set size increase. Since TC2 is designed to group transition blocks together, it manages to cut down the page fault rate and the frequency of large working set size. This is indicated by property (3).

4.2 Data Sensitivity

As mentioned in section 3.3, the dynamic behavior of the program to be restructured is affected by the input data used. Thus it is not clear that a program restructured based on a specific set of data will perform equally well when the input data are different. We are interested to see how sensitive the improvement of program restructuring is with respect to various input. In particular, we would like to study the robustness of TC2 and CWS, i.e., which one is less affected by the variation of input data.

Five different block reference strings of the Pascal compiler were obtained using different sets of input data. The Pascal compiler was restructured using first the CWS algorithm and then the TC2 algorithm based on one set of input data (P2). The performance of the restructured program was simulated using the five different block reference strings corresponding to the five different sets of input data. The results of the study are summarized in Table 4.4 and Table 4.5. Both algorithms appear to adjust to various input very well and manage to maintain satisfactory improvement on performance relative to the original ordering. Although CWS appears to be more robust

than TC2 in terms of reduction in page fault rate, TC2 shows better performance than CWS in all cases. CWS also tends to be more stable in mean working set size reduction.

Using the BLI definition of program behavior we can obtain from a block reference string a sequence of phase and transition sets. From the latter we can compute the average transition set size of a particular block reference string. This observable quantity allows us to predict the performance improvement of the program to be restructured with various data input. We expect large average transition set size to give rise to higher fault rate and also higher frequency of large working set sizes. As shown in Table 4.3, the average transition set sizes of the six reference strings tend to be very close to each other. This is a rough indication that the program is relatively insensitive to input data.

It has been confirmed in the literature [20,23] and from our experiments that most of the programs for which restructuring is convenient (such as compilers) are not very data-dependent. However, to find an optimal layout for a particular program with various input remains a nontrival and expensive task. The task of selecting a "typical" input for restructuring relies very much on the past experience of

the evaluator or the general opinion of the users of the program to be restructured.

Table 4.4. Results of an experiment on input dependency

Performance index: % reduction in page faults

Restructuring Algorithms: TC2 and CWS

note: Ci stands for cluster layout from ref. string Pi where i=1,2,..,6

reference string	TC2		CWS	
	Ci	C2	Ci	C2
P1	37.8	26.1	32.3	30.4
P2	36.1		33.2	
P3	35.6	32.3	31.8	28.7
P4	41.8	35.5	32.6	31.5
P5	42.5	34.5	33.4	30.7
P6	40.1	32.3	35.2	28.5

Table 4.5. Results of an experiment on input dependency

Performance Index: % reduction in mean working set size
Restructuring Algorithms: TC2 and CWS
note: Ci stands for cluster layout from ref. string Pi where i=1,2,..,6

reference string	TC2		CWS	
	Ci	C2	Ci	C2
P1	28.6	17.4	31.6	23.2
P2	23.1		27.8	
P3	25.6	21.4	28.7	26.4
P4	26.3	21.3	30.4	27.5
P5	30.5	23.9	25.2	24.4
P6	28.9	23.5	26.9	22.7

4.3 Portability

Among the four restructuring algorithms, only the CWS algorithm is dependent on the memory management policy in

use. When a tailored program is transported to another system with a different memory policy, it is almost certain that the performance will be degraded. Both the NEAR and the BLI algorithms (TC2 and PTN1 are members of BLI) are strategy-independent. They make use of only the extracted information of the program reference behavior of the program and no system parameter is involved. Thus these algorithms are adaptable to different environment for practical use.

Data collection at the symbolic (or subroutine) level also makes the work of restructuring more portable. This is because we can deal with the source code of the program directly especially for those programs written in high-level languages which encourage structured programming.

4.4 Cost

The costs of restructuring can be analyzed in terms of the following areas: data collection, preprocessing phase (constructing the desirability matrix) and clustering phase.

Data collection is the most difficult and costly part in the procedure, primarily because in most existing systems, tools (hardware or software) for gathering the reference trace are not available. An alternative is to run

the program to be restructured interpretively by a simulator which, needless to say, is very slow and expensive. Nevertheless the latter method was employed in our experiment. However, instead of gathering a complete memory reference string, a trace of subroutine calls and returns was recorded. It is found that restructuring is just as effective at the symbolic level despite the fact that the length of the reference string and the corresponding cost are greatly reduced.

The cost of running the restructuring algorithm varies roughly linearly with the length of the reference string to be examined. Among the four algorithms, the cost (in terms of computer time) of NEAR is the least and CWS costs slightly more than TC2 and PTN1. However, the cost of the restructuring phase is fairly small compared to the cost of data collection. The cost of clustering depends quadratically on the number of blocks in the program. The HEIR method is much more sophisticated and expensive than NUCL but we still think it is more reasonable to choose HIER for the substantially greater improvement.

Table 4.6 Sample Cost of the Restructuring Procedure

cost (CPU sec)	NEAR	CWS	PTN1	TC2
data collection	<----	900.188	---------->	
restructuring phase	11.51	33.38	13.89	13.05
clustering phase	39.76	43.61	41.06	42.61
working set simulation	<----	31.414	---------->	

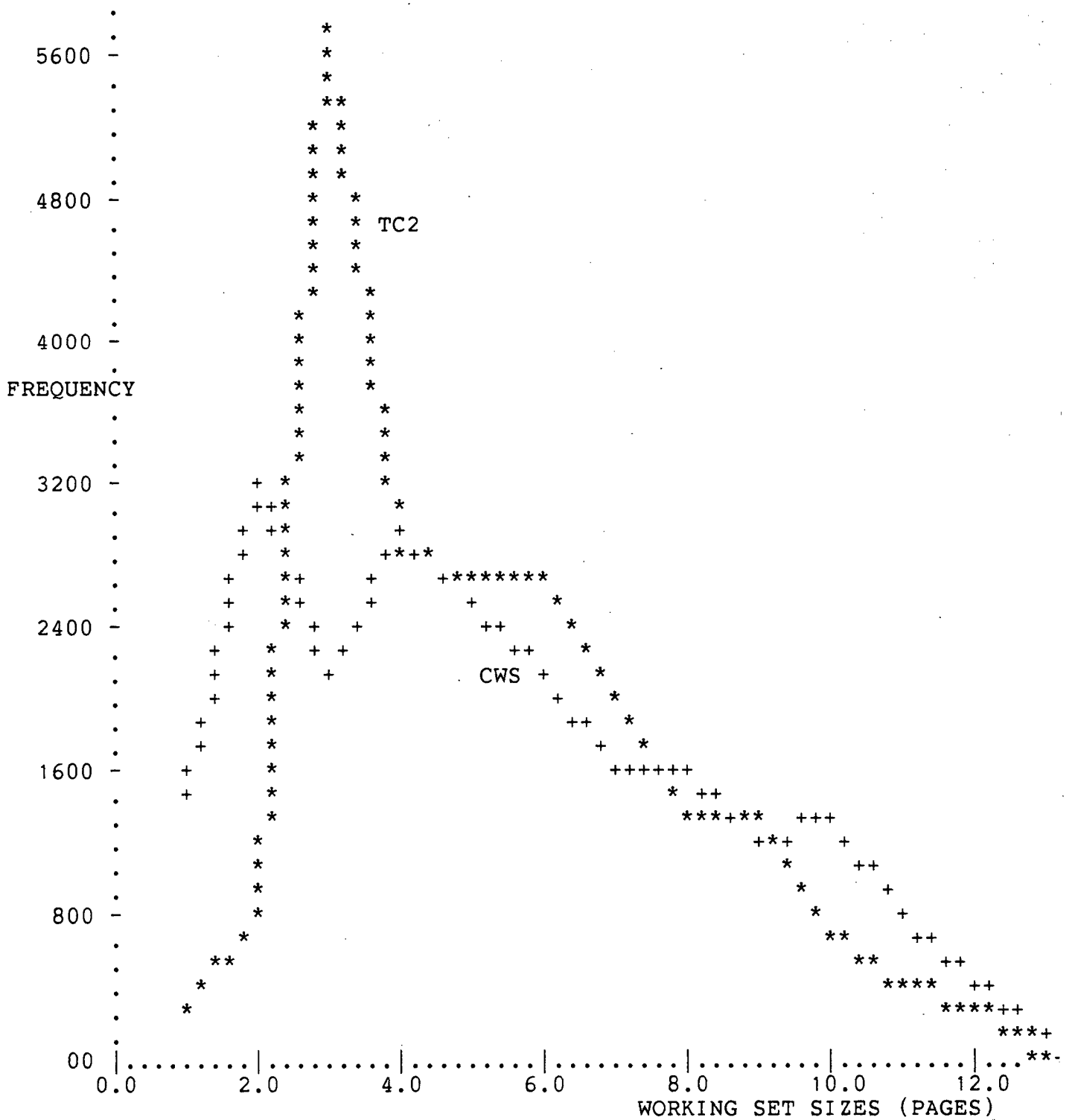

Figure 4.1. Working set size distributions of CWS and TC2 restructuring algorithms

5 Conclusion and Suggestions for Future Research

5.1 Conclusion

Program restructuring has been shown to be effective to improve program performance in virtual memory systems. We have developed a whole category of heuristics for program restructuring based on the notion of Bounded Locality Interval (BLI). In particular, the TC2 algorithm gives very good performance in reducing the page fault rate (an average of 39% in our experiments). It outperforms other algorithms including the strategy-oriented Critical-working-set algorithm by at least 20%. The results of our experiments justify the observation made in the literature that phase transitions during program execution have strong influence on the page fault rate. The objective of developing an efficient and strategy-independent restructuring algorithm is achieved.

Two parameters, the mean transition set size and the mean phase set size, of the phase/transition program behavior model are studied. Unfortunately we are unable to establish any strong correlation between these parameters and the performance indices in our experiments. However, it

is observed that if the mean transition set sizes and the mean phase set sizes of a program running on various sets of input data are similar, we can expect the program to be more or less data-independent and vice versa. Moreover, if the working set paging algorithm is the memory policy in use, choosing a window size which is slightly greater than the mean transition set size will generally result in good performance. That suggests that, if the window size is allowed to vary for different programs, the mean transition set size of the program is a good guideline for the choice of the window size.

5.2 Suggestions for Future Research

Choosing the right parameters to characterize program behavior remains one of the important research topics to be explored. The notion of Bounded Locality Interval (BLI) provides us with rich information about major phases and transitions from a reference string. In our experiment, we have studied the effect of the mean transition set size and mean phase set size on performance improvement. An extension to the study would be a more detail examination of the distributions of the transtion set sizes and the phase set sizes. An application of such research would be to provide guidelines for memory management policy to adapt to

changes in program behavior so as to optimize performance.

As mentioned in the previous chapter, in order to make program restructuring more economical, the cost of collecting the reference string has to be minimized. An alternative solution to intrepreting the program by a simulator is to insert measurement probes into the program to be restructured and execute the program after the modification. The task becomes even simpler if we are interested in obtaining a reference string only at the symbolic level. Such modification can be done with the aid of a clever "compiler" which will reduce the cost of data collection. This is because much of the work required can be carried out during the compilation process with only minor additional effort.

Specifically, an optimization option for programs to be compiled and executed can be provided by the compiler. The user who wants restructuring has to submit the source code and a set of input data if needed by the program, and turns on the option switch. The compiler would be expected to do the following:

(1) As usual, generate assembler codes if the program is error-free. (It is pointless to perform program

restructuring if the program contains errors.)

(2) Insert check points (assembly language statements necessary to update a counter), at the entry point of every relocatable block in the program.

(3) Run the modified program with the given input and collect the block reference string.

(4) Using the reference string from (3), perform the restructuring procedure, specifically the restructuring phase and the clustering phase.

(5) Reorganize the layout of the object code of the program based on information obtained in (4) and remove the checkpoints.

(6) Return the object code to the user and display other measurement information such as the desirability matrix, the resulting clusters etc., if requested.

The work of inserting measurement probes into the object code by the compiler is straight-forward and easy to do. The entire restructuring procedure can be invoked by the compiler as an independent process which takes the modified program to be restructured and produces the

information for the compiler to reorganize the object code accordingly. Considering the reduced cost of data collection and the relatively low cost of restructuring, such optimization option for programs to be executed very frequently would be very useful and cost-effective. Moreover, the near transparency and ease-of-use characteristics of such feature would be greatly appreciated by the computer users.

BIBLIOGRAPHY

[1] M. S. Achard, J.Y.Babonneau, M.Carpentier, G.Morisset, M.B.Mounajjed, "The Clustering Algorithms in the Opale Restructuring System," Performance of Computer Installation, D. Ferrari(ed.) CILEA, North-Holland Publishing Co.,1978.

[2] A. P. Batson and W. Madison, "Measurements of major locality phases in symbolic reference strings," in Proc. Int. Symp. Computer Performance Modeling, Measurement, and Evaluation, ACM SIGMETRICS and IFIP WG7.3, Mar. 1976,pp.75-84.

[3] A. P. Batson, "Program behavior at the symbolic level," Computer, vol.9,no.11,pp.21-28, Nov. 1976.

[4] A. P. Batson, W. E. Blatt, and J. P. Kearns, "Structure within locality intervals," in Proc. Symp. Modeling and Performance Evaluation of Computer Systems, H. Beilner and E. Gelenbe, Eds. Amsterdam, The Netherlands: North-Holland, Oct. 1977, pp.221-232.

[5] L. A. Belady, "A study of replacement algorithms for virtual storage computers," IBM Syst. J., vol.5, no.2, pp.78-101, 1966.

[6] L. A. Belady, and C. J. Kuehner, "Dynamic space sharing in computer systems," Commun. ACM, vol.12, pp282-288, May 1969.

[7] B. Brawn and F. G. Gustavson, "Program behavior in a paging environment," in 1968 AFIPS Conf. Proc., Fall Joint Comput. Conf., vol.33, Washington, DC: Thompson, 1968, pp.1019-1032.

[8] W. W. Chu and H. Opderbeck, "The page fault frequency replacement algorithm," Proc. FJCC, 1972, pp.597-609.

[9] W. W. Chu and H. Opderbeck, "Program behavior and the page fault frequency algorithm," IEEE Computer 9 11, Nov. 1976, pp.29-38.

[10] L. W. Comeau, "A study of the effect of user program optimization in a paging system," in Proc. ACM Symp. Operating Systems Principles, Oct. 1967.

[11] P. J. Denning, "The working set model for program behavior," Commum. ACM, vol.11, pp.323-333, May 1968.

[12] P. J. Denning and S. C. Schwartz, "Properties of the working set model," Commum. ACM, vol.15, pp.191-198, Mar. 1972.

[13] P. J. Denning, "On modeling program behavior," in 1972 AFIPS Conf. Proc., SJCC, vol.40, Montvale, NJ: AFIPS Press, 1972, pp.937-944.

[14] P. J. Denning and G. S. Graham, "Multiprogramming memory management," IEEE Proc., vol.63, pp.924-939, June 1975.

[15] P. J. Denning and K. C. Kahn, "A study of program locality and lifetime functions," in Proc. 5th ACM Symp. Operating Systems Principles, Nov.1975, pp.207-216.

[16] P. J. Denning and K. C. Kahn, "An L=S criterion for optimal multiprogramming," in Proc. Int. Symp.Computer Performance Modeling, Measurement, and Evaluation, ACM SIGMETRICS and IFIP WG7.3, Mar. 1976,pp.219-229.

[17] P. J. Denning, K. C. Kahn, J. Leroudier, D. Potier, and R. Suri, "Optimal multiprogramming," Acta Informatica, vol.7, no.2, pp.197-216, 1976.

[18] P. J. Denning, "Working Sets Past and Present" IEEE Trans. on Software Engin. Vol.SE-6,No.1, Jan 1980.

[19] D. Ferrari, "Improving Locality by Critical Working sets," CACM, vol.17, Nov. 1974, pp.614-620.

[20] D. Ferrari, "Improving Program Locality by Strategy-Oriented Restructuring," Information Processing 74, Proc. IFIP Congress 74, North-Holland, Amsterdam, 1974, pp266-270.

[21] D. Ferrari, "Tailoring Programs to Models of Program Behavior," IBM Journal of Research and Development, vol.19, 3, May 1975, pp.244-251.

[22] D. Ferrari, "The improvement of program behavior," IEEE Computer 9 11, Nov. 1976, pp.39-47.

[23] D. J. Hatfield and J. Gerald, "Program restructuring for virtual memory," IBM Sys. J. vol.10, pp.168-192, 1971.

[24] K. C. Kahn, "Program behavior and load dependent system performance," Ph.D. Dissertation, Dep. Computer Sci., Purdue Univ., W. Lafayette, IN, Aug. 1976.

[25] A. W. Madison and A. P. Batson, "Characteristics of program localities," CACM, vol.19, pp.285-294, May 1976.

[26] T. Masuda, H. Shiota, K. Noguchi, and T. Ohki, "Optimaztion of program locality by cluster analysis," in Proc. IFIP Congress, 1974, pp.261-265.

[27] T. Masuda, "Methods For the Measurement of Memory Utilization and the Improvement of Program Locality", IEEE Trans. on Software Engin., vol.SE-5,NO.6, Nov 1979.

[28] A. J. Smith, "A modified working set paging

algorithm," IEEE Trans. Comput., vol.C-25, pp.907-914, Sept.1976.

[29] J. R. Spirn and P. J. Denning, "Experiments with program locality," in AFIPS Conf. Proc., FJCC, vol.41, Montvale, NJ: AFIPS Press,1972, pp.611-621.

[30] J. R. Spirn, "Distance string models for program behavior," Computer, vol.9., pp.14-20, Nov. 1976.

[31] J. R. Spirn, Program Behavior: Models and Measurement. New York: Elsevier/Noth-Holland, 1977.

[32] G. S. Graham, "A Study of Program and Memory Policy Behaviour," (Ph.D. Thesis), Purdue University, Computer Science Dept, 1976.

[33] R. L. Mattson, J. Gescei, D. R. Slutz and I. L. Traiger, "Evaluation techniques for storage hierarchies," IBM Sys. J. 9 2 (1970), pp.78-117.

Appendix I.

Reference Strings

The reference string is defined as the chronological sequence of the virtual addresses (aj) referenced by the program, each with (or without) an indication of the CPU (or virtual) time tj to the next reference:

$$\{ a_j(t_j) \} \qquad (j=1,2,\ldots,k)$$

where k is the number of references generated by the program.

Several other characterizations can be derived from a reference string. For example, we may group virtual addresses into blocks. Given a set of blocks $B = \{b1,b2,\ldots,bn\}$, we may defined a many-to-one mapping from the set A of virtual addresses to B, so that each aj is associated with one and only one block bi. This mapping when applied to each element of the address trace $\{aj(tj)\}$, transforms it into the block trace or block reference string:

$$\{ b_i(t_j) \} \qquad (j=1,2,\ldots,k)$$

Each block usually consists of information items having contiguous addresses in virtual space, so that blocks are treated as single entities during transfers and for allocation purposes.

Appendix II.

Paging Algorithms

A paged system transfers information to and from main memory in fixed size units called pages. The paging algorithm in such systems is responsible for the fetching, placement and replacement of pages in main memory. A prepaging algorithm tries to anticipate page references of a task in the near future (and brings in those pages before they are referenced) in an attempt to reduce the page fault waiting time. However, such predictions are difficult to make accurately. Thus, except for a few systems (particularly those that use the working set policy for memory management), most virtual memory systems use demand paging (i.e., pages are brought into main memory only when referenced).

Let N = {0,1,2,....,n-1} be the set of pages of a task, and let M(t) = {0,1,2,....m(t)-1} be the set of page frames in main memory allocated to the task. We shall assume that n is constant with 1<=m(t)<n. We call m(t) the memory allocation of the task at time t. If m(t) is a constant this is called fixed allocation. When m(t) is allowed to vary with time, we have a variable or dynamic allocation scheme.

Let the memory references of a task be r(1)r(2)r(3)....r(K), where r(t) is the page referenced at virtual time t. We shall use S(t) to denote the set of pages in main memory just after the reference at time t (the S stands for "storage"). For all t, we have S(t) a subset of N and |S(t)|<=m(t). We shall use S(0) to indicate the initial content of memory, before the first reference.

Formally, the definition of a strict demand paging algorithm gives S(t+1) as a function of S(t):

$$S(t+1) = \begin{cases} S(t) & \text{if } r(t+1) \text{ in } S(t) \\ S(t)+r(t+1)-Z(t+1) & \text{if } r(t+1) \text{ not in } S(t) \end{cases}$$

where Z(t+1), the replaced page set, is a subset of S(t). In other words, if no page fault occurs, the memory content is unchanged. If r(t+1) is not in S(t), a page fault occurs and the missing page is brought into memory, replacing the set of pages Z(t+1). If |S(t)|=m (a constant) for all t, then Z(t) consists of exactly one page z(t), the replaced page.

Stack algorithms [33] are a particularly interesting class of fixed-allocation paging algorithms. At each instant of time, a stack algorithm defines a vector (the "stack") on some or all of the pages of a task. The stack at time t is vs(t)=[s1(t),...,sk(t)], with k<=n, (n is total number of pages needed by a task) such that if the algorithm

operates with memory allocation m, then the memory will contain exactly the set of pages

$$S(m,t) = \begin{cases} \{s1(t),....sm(t)\} & \text{if } m<=k \\ \{s1(t),....sk(t)\} & \text{if } m>k \end{cases}$$

at time t. By convention, s1(t), the first element in the stack vector, is to be at the top of the stack, and sk(t) at the bottom. We say that si(t) is lower in the stack than sj(t) if i>j. The stack distance d(t) is defined to be the position (or index in the stack) of r(t) in stack vs(t-1). If r(t)=si(t-1) then d(t)=i, with 1<=i<=k. If r(t) does not appear anywhere in vs(t-1), then d(t) is equal to "infinity".

At each page reference, the stack vector is updated according to the following three rules:

(1) The page just referenced is moved to the top of stack.

(2) An unreferenced page never moves up the stack. That is to say, pages above the one referenced can either remain at their current position or be displaced downwards according to their priorities.

(3) Pages below the referenced page remain fixed in the stack.

Given a memory allocation of m page frames, the top m

pages in the stack are always maintained in main memory. The stack algorithms have two basic advantages. First, the page fault behavior of a given reference string can be computed effectively for all memory sizes in one scan of the reference string. The other advantage is that for any given reference string, the number of page faults produced by a stack algorithm does not increase as memory allocation (m) increases. Due to the inclusion property of stack algorithms, the top (m+k) pages (k>0) on the stack always include the top m pages. This in turn implies that if a page fault occurs when the memory allocation is (m+k) where (k>0), then a page fault must occur if the memory allocation is reduced to m. The same may not be true for other page replacement algorithms.

Appendix IIa.

The Least Recently Used (LRU) paging algorithm

The LRU paging algorithm is a stack algorithm. This algorithm chooses for replacement that page in memory which has not been referenced for the longest period of time. The stack vs(t) for the LRU algorithm consists of all pages of the task ordered by decreasing time of their most recent reference. Thus, s1(t) is the most recently used page (which is r(t)), s2(t) is the next most recently used page, and sn(t) is the least recently used page. Pages which have never been referenced are grouped together at the bottom of the stack in any arbitrary order.

The procedure for updating the LRU stack is conceptually quite simple. If d(t)=i, then s1(t)=r(t), and sj(t)=sj-1(t) for all j such that 1<j<=i. Each entry in the stack above the point of reference is moved down by one position. Pages below the point of reference do not move in the stack.

The stack vector for this algorithm is time varying, and must be updated at every reference. Suppose vs(t)=[x1,....,xn], and r(t+1)=xi, then:

vs(t+1) = [xi,x1,...,xi-1,xi+1,...,xn].

Appendix IIb.

The Working Set paging algorithm

The working set W(t,T) of a task at time t has been defined by Denning [11,12] to be the set of distinct pages in the T most recent references r(t-T+1)...r(t). Parameter T is known as the window size. The working set algorithm retains exactly the working set in memory at all times. Pages can join or leave the working set at other than page fault times, so this is not a strict demand paging algorithm but a loose one. Since S(t)=W(t,T) at all times, a page can be removed from main memory only when it has not been referenced in T consecutive time instants.

Appendix III.

An example to illustrate the Hierarchical Classification Clustering Algorithm.

Let J = { b1, b2, b3, b4 } be the set of blocks to be classified and the desirability matrix D be given by:

$$D = \begin{array}{c} \begin{array}{cccc} \{b1\} & \{b2\} & \{b3\} & \{b4\} \end{array} \\ \left|\begin{array}{cccc} 0 & 17 & 47 & 24 \\ 0 & 0 & 12 & 68 \\ 0 & 0 & 0 & 8 \\ 0 & 0 & 0 & 0 \end{array}\right| \end{array} \begin{array}{l} \{b1\} \\ \{b2\} \\ \{b3\} \\ \{b4\} \end{array}$$

Our goal is to classify J into subsets, each of which represents a cluster of blocks that are strongly associated to each other based on the given desirability measure.

To simplify the problem, we shall assume that the sizes of the blocks are the same and exactly two blocks can fit into a page.

Step 1:

We start off with J0 = { {b1}, {b2}, {b3}, {b4} }, a set of 4 clusters each containing a single block initially. Let D0 = D, the desirability matrix.

(1) Find the largest element in D0, which is d(2,4)=68.

(2) Since the sum of sizes of the 2 clusters, {b2} and {b4}, is equal to the page size, we merge {b2} and {b4}

to form a new cluster C={b2,b4}. Now J1 = { {b1}, {b3}, {b2,b4} }.

(3) Construct the reduced matrix D1 from D0 as follows:

-- delete all the rows and columns associated with {b2} and {b4}

-- and add row C ({b2,b4}) and column C to D1 such that

d(1,C) (of D1) = d(1,2) + d(1,4) (of D0)

d(2,C) (of D1) = d(2,3) + d(3,4) (of D0)

	{b1}	{b3}	{b2,b4}	
D0 ==> D1 =	0	47	41	{b1}
	0	0	20	{b3}
	0	0	0	{b2,b4}

Step 2:

(1) From D1, d(1,2)=47 is found to be the largest

(2) We merge {b1} and {b3} to form {b1,b3}

(3) J2 = { {b1,b3}, {b2,b4} }

	{b1,b3}	{b2,b4}	
D1 ==> D2 =	0	108	{b1,b3}
	0	0	{b2,b4}

Step 3:

(1) 108 is the largest in D2

(2) Since the sum of sizes of the 2 clusters exceeds the page size, we can not merge the clusters together. Hence we replace d(1,2) of D2 by zero to indicate the

merge is unsuccessful. As a result D3 becomes a zero matrix and J3 = J2.

Step 4:

The largest element of D3 equals to zero, the process is halted since no new cluster can be formed based on D3.

The final result is J3 which is a set of clusters each of which consists of a set of 2 blocks and whose size is equal to the page size.