A study of sampling, granularity and localities in program restructuring

Kwai-Ting Lan

Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation
https://lib.dr.iastate.edu/rtd/7441
INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame. If copyrighted materials were deleted you will find a target note listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in "sectioning" the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.
Lan, Kwai-Ting

A STUDY OF SAMPLING, GRANULARITY AND LOCALITIES IN PROGRAM RESTRUCTURING

Iowa State University  PH.D.  1981

University Microfilms International  300 N. Zeeb Road, Ann Arbor, MI 48106
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark √.

1. Glossy photographs or pages ______
2. Colored illustrations, paper or print ______
3. Photographs with dark background ______
4. Illustrations are poor copy ______
5. Pages with black marks, not original copy ______
6. Print shows through as there is text on both sides of page ______
7. Indistinct, broken or small print on several pages ✓
8. Print exceeds margin requirements ______
9. Tightly bound copy with print lost in spine ______
10. Computer printout pages with indistinct print ✓
11. Page(s) _______ lacking when material received, and not available from school or author.
12. Page(s) _________ seem to be missing in numbering only as text follows.
13. Two pages numbered ________, Text follows.
14. Curling and wrinkled pages ______
15. Other ____________________________________________

University Microfilms International
A study of sampling, granularity and localities in program restructuring

by

Kwai-Ting Lan

A Dissertation Submitted to the Graduate Faculty in Partial Fulfillment of the Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa
1981
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER I. INTRODUCTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Memory System Concept</td>
<td>1</td>
</tr>
<tr>
<td>Memory Management Strategies</td>
<td>8</td>
</tr>
<tr>
<td>LIPO</td>
<td>9</td>
</tr>
<tr>
<td>FIFO</td>
<td>9</td>
</tr>
<tr>
<td>LRU</td>
<td>10</td>
</tr>
<tr>
<td>OPT</td>
<td>10</td>
</tr>
<tr>
<td>WS</td>
<td>11</td>
</tr>
<tr>
<td>PFF</td>
<td>11</td>
</tr>
<tr>
<td>GLRU</td>
<td>12</td>
</tr>
<tr>
<td>VMIN</td>
<td>12</td>
</tr>
<tr>
<td>Measurements of Program Behavior</td>
<td>13</td>
</tr>
<tr>
<td>Reference String</td>
<td>13</td>
</tr>
<tr>
<td>Methods for Reference String Collection</td>
<td>14</td>
</tr>
<tr>
<td>Hardware Monitor</td>
<td>14</td>
</tr>
<tr>
<td>Software Monitor</td>
<td>16</td>
</tr>
<tr>
<td>Reference String Sampling Methods</td>
<td>18</td>
</tr>
<tr>
<td>Program Restructuring</td>
<td>19</td>
</tr>
<tr>
<td>Introduction: Program Restructuring</td>
<td>19</td>
</tr>
<tr>
<td>Benefits from Program Restructuring</td>
<td>24</td>
</tr>
<tr>
<td>The General Program Restructuring Model</td>
<td>24</td>
</tr>
<tr>
<td>Program Restructuring Methods</td>
<td>26</td>
</tr>
<tr>
<td>A Priori Restructuring Method</td>
<td>26</td>
</tr>
<tr>
<td>Ramamoorthy's Approach</td>
<td>27</td>
</tr>
<tr>
<td>Lowe's approach</td>
<td>28</td>
</tr>
<tr>
<td>Ver Hoef's approach</td>
<td>28</td>
</tr>
<tr>
<td>Baer and Caughey's approach</td>
<td>29</td>
</tr>
<tr>
<td>Snyder's approach</td>
<td>30</td>
</tr>
<tr>
<td>A Posteriori Restructuring Method</td>
<td>31</td>
</tr>
<tr>
<td>Strategy-Independent Restructuring Algorithms</td>
<td>32</td>
</tr>
<tr>
<td>Nearness Method</td>
<td>32</td>
</tr>
<tr>
<td>AS1</td>
<td>35</td>
</tr>
<tr>
<td>AS2</td>
<td>35</td>
</tr>
<tr>
<td>SAS1</td>
<td>36</td>
</tr>
<tr>
<td>SAS2</td>
<td>36</td>
</tr>
<tr>
<td>Strategy-Dependent Restructuring Algorithms</td>
<td>36</td>
</tr>
<tr>
<td>CWS</td>
<td>38</td>
</tr>
<tr>
<td>CBLU</td>
<td>39</td>
</tr>
<tr>
<td>CPSI</td>
<td>41</td>
</tr>
<tr>
<td>MWS</td>
<td>42</td>
</tr>
<tr>
<td>MPSI</td>
<td>42</td>
</tr>
<tr>
<td>Dynamic Restructuring Method</td>
<td>44</td>
</tr>
<tr>
<td>Clustering Algorithms</td>
<td>45</td>
</tr>
</tbody>
</table>
Summary of Results ........................................... 103

CHAPTER IV. SAMPLED PROGRAM RESTRUCTURING METHODS .... 106

The Design of the Experiments ............................... 106
Selection of Levels of Each Factor ......................... 106
Sampling Methods ........................................... 108
Sampling Patterns ........................................... 110
Sampling Rate ................................................ 112

The Experiments .............................................. 113
LRU Experiments and Results ................................. 113
Working Set Experiments and Results ....................... 120

Summary of Results ......................................... 135

CHAPTER V. STRATEGY-INDEPENDENT RESTRUCTURING ALGORITHMS. 137

Introduction ................................................. 137

Bounded Locality Interval (BLI) Restructuring
Algorithm ..................................................... 146

Critical Bounded Locality Interval (CBLI)
Restructuring Algorithm .................................... 147

The Design of the Experiments ............................... 152
Selection of Levels of Each Factor ......................... 152

The Experiments .............................................. 153
LRU Experiments and Results ................................. 153
Working Set Experiments and Results ....................... 159

Summary of Results ......................................... 170

CHAPTER VI. CONCLUSION AND RECOMMENDATIONS FOR FURTHER WORK ........................................... 172

Future Work ............................................... 173

BIBLIOGRAPHY ................................................. 174

ACKNOWLEDGEMENTS .......................................... 179

APPENDIX A. PL/1 IMPLEMENTATION OF BLOCK ANALYZER .... 180

APPENDIX B. COMPUTE BLOCK STRING ......................... 183

APPENDIX C. CLLU RESTRUCTURING METHOD ................ 185
APPENDIX D.  CRITICAL WS RESTRUCTURING METHOD  . . . .  186
APPENDIX E.  LRU MEASUREMENT  . . . . . . . . . . . . . . .  188
APPENDIX F.  WORKING SET MEASUREMENT  . . . . . . . . .  190
APPENDIX G.  PROGRAM XREF  . . . . . . . . . . . . . . . . .  192
LIST OF TABLES

TABLE 1. A block reference string and two corresponding page reference strings. .......... 21
TABLE 2. List of Clustering Algorithms. ............................................ 45
TABLE 3. List of Tools. ................................................................. 71
TABLE 4. Syntax difference between the selected programs. ................................. 83
TABLE 5. Names of the twelve reference strings. ............................................ 85
TABLE 6. Parameters of LBU Experiments. ............................................... 88
TABLE 7. Parameters of WS Experiments. ................................................ 91
TABLE 8. Parameters of LBU Experiments. .............................................. 96
TABLE 9. Parameters of WS Experiments. ................................................ 98
TABLE 10. Parameters of Experiments. .................................................... 107
TABLE 11. Parameters of LBU Experiments. ............................................ 114
TABLE 12. Parameters of WS Experiments. .............................................. 120
TABLE 13. An algorithm for updating the Extended LRU stack. ............................ 144
TABLE 14. The BLI Restructuring Algorithm. ............................................. 146
TABLE 15. The CBLI Restructuring Algorithm. .......................................... 151
TABLE 16. Summary of the Basic Factors. ................................................. 152
TABLE 17. Parameters of LBU Experiments. ............................................ 154
TABLE 18. Parameters of WS Experiments. .............................................. 159
LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Description</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIGURE 1</td>
<td>Classification of Replacement Strategies</td>
<td>7</td>
</tr>
<tr>
<td>FIGURE 2</td>
<td>Classification of Reference Strings</td>
<td>13</td>
</tr>
<tr>
<td>FIGURE 3</td>
<td>Hardware Monitor</td>
<td>15</td>
</tr>
<tr>
<td>FIGURE 4</td>
<td>Two different layouts of a program in virtual memory</td>
<td>20</td>
</tr>
<tr>
<td>FIGURE 5</td>
<td>Classification of Restructuring Algorithms</td>
<td>27</td>
</tr>
<tr>
<td>FIGURE 6</td>
<td>The Restructuring Experiment</td>
<td>59</td>
</tr>
<tr>
<td>FIGURE 7</td>
<td>List of Possible Candidates</td>
<td>63</td>
</tr>
<tr>
<td>FIGURE 8</td>
<td>The Monitor</td>
<td>65</td>
</tr>
<tr>
<td>FIGURE 9</td>
<td>Block-Analyzer</td>
<td>66</td>
</tr>
<tr>
<td>FIGURE 10</td>
<td>Compute-Block-String</td>
<td>67</td>
</tr>
<tr>
<td>FIGURE 11</td>
<td>Restructuring-Method</td>
<td>68</td>
</tr>
<tr>
<td>FIGURE 12</td>
<td>Clustering-Method</td>
<td>69</td>
</tr>
<tr>
<td>FIGURE 13</td>
<td>Compute-Page-String</td>
<td>69</td>
</tr>
<tr>
<td>FIGURE 14</td>
<td>Evaluator</td>
<td>71</td>
</tr>
<tr>
<td>FIGURE 15</td>
<td>Program Restructuring</td>
<td>81</td>
</tr>
<tr>
<td>FIGURE 16</td>
<td>Data-Sensitivity Studies</td>
<td>82</td>
</tr>
<tr>
<td>FIGURE 17</td>
<td>LRU Result: Time (Procedure-Block)</td>
<td>90</td>
</tr>
<tr>
<td>FIGURE 18</td>
<td>WS Result: Time (Procedure-Block)</td>
<td>92</td>
</tr>
<tr>
<td>FIGURE 19</td>
<td>WS Result: Space (Procedure-Block)</td>
<td>93</td>
</tr>
<tr>
<td>FIGURE 20</td>
<td>WS Result: Time-Space (Procedure-Block)</td>
<td>94</td>
</tr>
<tr>
<td>FIGURE 21</td>
<td>LRU Result: Time (Basic-Block)</td>
<td>97</td>
</tr>
</tbody>
</table>
FIGURE 22. Relation between block size and restructuring 99
FIGURE 23. WS Result: Time (Basic-Block) 100
FIGURE 24. WS Result: Space (Basic-Block) 101
FIGURE 25. WS Result: Time-Space (Basic-Block) 102
FIGURE 26. Sampling Method 108
FIGURE 27. Parts of the Sampling Intervals 109
FIGURE 28. Effect of Different Sampling Parameters 111
FIGURE 29. LRU Result: Time (WSM.proc) 116
FIGURE 30. LRU Result: Time (XREF.proc) 117
FIGURE 31. LRU Result: Time (WSMerr30.proc) 118
FIGURE 32. LRU Result: Time (QUARK.proc) 119
FIGURE 33. WS Result: Time (WSM.proc) 123
FIGURE 34. WS Result: Time (XREF.proc) 124
FIGURE 35. WS Result: Time (WSMerr30.proc) 125
FIGURE 36. WS Result: Time (QUARK.proc) 126
FIGURE 37. WS Result: Space (WSM.proc) 127
FIGURE 38. WS Result: Space (XREF.proc) 128
FIGURE 39. WS Result: Space (WSMerr30.proc) 129
FIGURE 40. WS Result: Space (QUARK.proc) 130
FIGURE 41. WS Result: Time-Space (WSM.proc) 131
FIGURE 42. WS Result: Time-Space (XREF.proc) 132
FIGURE 43. WS Result: Time-Space (WSMerr30.proc) 133
FIGURE 44. WS Result: Time-Space (QUARK.proc) 134
FIGURE 45. Hierarchy of BLI's for an Example Symbolic Reference String. 141
FIGURE 46. LRU stack at time t=30

FIGURE 47. Hierarchy of LRU stack localities at time t=30

FIGURE 48. Extended LRU stack at time t=30 for the reference string in Figure 45

FIGURE 49. LRU Result: Time (XREF.proc)

FIGURE 50. LRU Result: Time (WSM.proc)

FIGURE 51. LRU Result: Time (QUARK.proc)

FIGURE 52. WS Result: Time (XREF.proc)

FIGURE 53. WS Result: Time (WSM.proc)

FIGURE 54. WS Result: Time (QUARK.proc)

FIGURE 55. WS Result: Space (XREF.proc)

FIGURE 56. WS Result: Space (WSM.proc)

FIGURE 57. WS Result: Space (QUARK.proc)

FIGURE 58. WS Result: Time-space (XREF.proc)

FIGURE 59. WS Result: Time-space (WSM.proc)

FIGURE 60. WS Result: Time-space (QUARK.proc)
CHAPTER I. INTRODUCTION

Virtual Memory System Concept

The main memory available to a processor has rarely been of sufficient capacity to serve as the sole storage medium for a program in execution. This problem motivated the design of computer systems which use memory hierarchies to create an extended addressable space. The use of memory hierarchies has been aided by the appearance of newer memory technologies as well as the development of numerous strategies for managing these memory hierarchies. Today, most modern large computer systems have been implemented using an automatically managed memory hierarchy, termed a virtual memory, to bridge the capacity gap between main memory and secondary storage. Most practical virtual memory systems employ a two-level hierarchy comprised of a fast random access main memory of relatively small capacity and a secondary storage of relatively slow but much larger capacity.

For a number of reasons, we believe that virtual memory systems will continue to be used even though the cost of main memory is rapidly decreasing. One reason is that as the cost of memory decreases larger applications, requiring larger amounts of memory, do appear. For example, interactive time sharing systems supporting a large number
of terminals and large data base systems have become more popular as the cost of memory has decreased. Another reason is the economics of memory hierarchies. The unit cost of fast main memory is significantly higher than the cost of slower secondary storage. It is reasonable to expect that this cost structure will continue to exist. Thus, the same economic reasons for virtual-memory systems which existed in the past will continue to exist in the future.

In a paging implementation of virtual-memory system, the address spaces of programs are divided into fixed-size portions, called virtual page frames, consisting of contiguous virtual addresses. Similarly, the physical main memory is divided into physical page frames having the same sizes as their virtual counterparts. The information stored in a virtual page frame, and occasionally also in a physical page frame, is called a page. The page is the unit of memory allocation and of information transfer between main memory and secondary storage.

Virtual memory systems allow programs to be written as though a virtually unlimited amount of main memory space were available to them. The large virtual memory not only frees users from the need to carry out memory allocation for programs larger than available physical main memory space, but also makes possible more efficient use of computing system resources. Since programs do not use all of their
main storage space at all times during execution, it is possible to keep only small pieces (pages) in main memory at any given time. The virtual memory systems provide automatic address translation, bringing additional pages into physical memory as they are needed. In this way, the pages of many programs can be in main memory at the same time, with the remaining pages kept on high-speed secondary storage. This increases both the utilization of main memory as well as the possibility of overlapping central processing unit (CPU) and input/output (I/O) operations.

Although it is true that most virtual memory systems present the programmer with a logical memory of unlimited capacity and provide automatic memory management, it is not the case that the virtual memory behaves as a randomly accessed store. Typical programs display a highly localized referencing pattern which persists for a significant period of time. Thus, most virtual memory systems [13], [15], [26], [31], [36] use memory management policies which tend to keep the most recently referenced portions of the program in the main memory.

The task of a memory management strategy is to decide which parts of the programs that are ready to run, at any instant, are to be in main memory at that instant. Thus, in a paged system, the memory management strategy determines how many physical page frames are to be allocated to each
program, and which ones of its pages are to be stored into those frames at any given time. In other words, the memory management strategy has to solve the two interrelated problems of size and page identity for the loaded portion of each program. In dealing with these interrelated problems, the goal of a good memory management strategy is to retain in main memory those pages which a program will reference in the near future. In this way, the strategy attempts to improve the performance of the system to such a degree that it will permit efficient sharing of memory space among different users and achieve the high access rates and low cost per bit that is possible with a memory hierarchy.

The predictability of logical memory addresses, which is essential to obtaining reasonable solutions to the size and identity problem alluded to earlier, is based on a common characteristic of computer programs called locality. Locality [16], [17], [27], [34], [42], [46] is a basic property of program behavior which states that during any interval of time the pages of a program are referenced nonuniformly. Locality derives from the observed tendency of programs and programmers to concentrate on small parts of a large problem for moderately long intervals before switching attention to other parts of the problem. One reason for locality of reference is that program instructions are stored sequentially in the computer's
memory in approximately the order in which they are needed during program execution. A second factor is the presence of program loops. A third factor is due to the one-entry and one-exit constructs used in top-down/bottom-up structured programming.

An important characteristic of virtual memory systems is that their performance (in terms of page faults) is very much dependent on the locality of references exhibited by the programs being executed. The performance generally improves as programs with greater locality are executed. But programs written under the assumption that main memory is virtually unlimited can result in a phenomenon called "thrashing" if they have poor locality. Thrashing occurs when much more time is spent performing paging I/O operations than executing instructions. The cost of thrashing is compounded since, as paging I/O increases, the rate of system degradation also increases.

So the goal of a virtual-memory system can be achieved only as a result of the fortuitous fact that for most processing, the references to memory tend to be highly localized in small groups which tend to change relatively slowly during the course of instruction execution.

The problem of achieving reasonable levels of performance in virtual memory systems has received, and is still receiving, a considerable amount of attention.
Tremendous research effort has been invested in the exploration of virtual memory system measurement, evaluation and improvement over the past decade as the use of multiprogramming and virtual memory techniques has become more widespread. Most of the efforts made to improve the performance of memory hierarchies have been spent in devising good memory management strategies that could maximize the probability of finding referenced information in main memory at the time it is requested, thereby minimizing the page fault rate [5]. This approach accepts programs with poor locality as a necessary evil and concentrates its efforts on trying to make the memory management strategies able to deal efficiently with a broad spectrum of behavior.

However, even the best memory management strategy cannot yield acceptable performance in the face of programs with poor locality. Thus, an alternative for obtaining better performance from a memory hierarchy is to increase the degree of locality [13] of the programs to be executed. This result can be achieved by a technique called program restructuring.

In the above discussion, it has been shown that memory management strategies, program behavior (i.e. program locality) and program restructuring are closely related. In the next section, several memory management strategies are
reviewed. In section C, methods to characterize program behavior by different kinds of reference strings are briefly reviewed and, finally, in section D, program restructuring is discussed in detail.

FIGURE 1. Classification of Replacement Strategies
Memory Management Strategies

Because the exact prediction of a program's future locality set is usually an impossible task, the memory management strategy attempts only to estimate this set of needed pages. Such strategies generally do so by making assumptions about the behavior of the program in the near future based on the pages referenced in the recent past. These assumptions can be viewed as defining a model of program behavior. In other words, a strategy is designed to work optimally with programs whose behavior is accurately represented by the model underlying the strategy (i.e., programs which satisfy the behavioral assumptions made by the strategy).

The automatic demand strategies are the most successful memory management strategies in modern computer systems. Automatic replacement strategies are those where the replacement decision is entirely a system function. Demand strategies replace pages of the resident set only at page fault instances. Automatic, demand replacement algorithms can be divided into two categories as shown in Figure 1. Replacement strategies in the first category have a constant sized resident set (i.e. fixed-space algorithms), whereas those in the second category allow a program to have resident sets of different sizes (i.e. variable-space algorithms) during different periods of its execution.
No attempt will be made herein to cover all of the page replacement algorithms in this brief survey. Only the most important and widely used policies are discussed. The fixed-space algorithms LIFO, FIFO, LRU and OPT are discussed first. Then the variable-space algorithms WS, PFF, GLRU and VMIN are discussed in turn.

**LIFO**

The LIFO (Last-In-First-Out) algorithm \([10]\) estimates future references by replacing the page which has been fetched most recently. In some instances, this strategy does poorly, such as when the instructions of a loop are divided between two pages and all of the allocatable page frames are full.

**FIFO**

The FIFO (First-In-First-Out) algorithm \([10]\) replaces the page which has been in memory the longest period of time. It is simple to implement, since the pages are placed in a queue as they arrive in main memory. Unfortunately, the FIFO strategy exhibits an anomalous behavior \([6]\) where the page fault rate can actually increase when the memory allocation is increased.
The LRU (Least-Recent-Used) algorithm [10] replaces the page which has not been referenced for the longest period of time. LRU algorithm belongs to the class of stack algorithm which includes all those algorithms without anomalous behavior. Of the practical fixed-space replacement algorithms, the LRU replacement strategy is the most widely used.

The OPT (Optimal) algorithm [13] can reduce the number of page faults of a program to minimum. According to the OPT algorithm, the page replaced at each page fault time is that page in the current resident set which has the largest forward distance. The forward distance of a page is the number of references (i.e., virtual time) until the next reference is made to that page. Unfortunately, OPT is practically unrealizable, because it requires a knowledge of the future page references of the program every time a page fault occurs. However, the OPT algorithm can be used as a benchmark against which the number of page faults generated by other practical, though suboptimal, algorithms may be compared.
The first of the variable-space strategies is the WS (Working-Set) algorithm [13], [14], which maintains an estimate of the current locality in main memory at all times. The locality is estimated by the working set, which is the set of distinct pages that have been referenced by the program during the interval \([t(i)-T, t(i)]\), where \(T\) is the parameter supplied to the working set algorithm and is referred to as the window size. The resident set at any one instance, then, is the set of pages which has a backward distance less than or equal to \(T\). The working set algorithm is a loose demand paging strategy, since pages not referenced for \(T\) references will leave the working set even though no page fault has occurred.

The PFF (Page-Fault-Frequency) algorithm developed by Chu and Opderbeck [8], [9] seeks to maintain the page fault rate below a given parameter \(P\). At each page fault, the time interval between the current page fault and the last page fault is calculated. If this time interval is less than \(1/P\), then the page that caused the fault is added to the resident set (i.e. increasing the memory allocation by one page). On the other hand, if the time interval exceeds \(1/P\), then all the pages in the resident set which have not been referenced since the last page fault are removed, and
the allocation is decreased by that number. Note that the
PFF algorithm is a demand paging algorithm, since pages are
removed from the resident set only at page fault times.

GLRU

The GLRU [24] (Global-LRU) algorithm is created by
extending the fixed-space LRU algorithm to the entire
contents of memory in a multiprogramming system. All of the
resident set pages of all of the programs are put in one
central LRU stack. As each program references its own
pages, they are moved to the top of the stack and hence
possibly cause the pages of other programs to be moved out
from memory. Strategies of this type, which allow one
program to interfere with another program's memory
allocation, are susceptible to thrashing when too many
programs are active at once.

VMIN

An optimal variable-space algorithm is VMIN [40]. With
the VMIN policy a cost is associated with keeping a page in
memory and a cost is associated with a page fault. The VMIN
policy minimizes the total cost. But VMIN requires precise
knowledge of the future references to make its replacement
decision. So it is an unrealizable algorithm.

In the next section, the use of reference strings to
describe a program's dynamic behavior is discussed.
Reference String

The reference string of a process is the sequence of all the virtual addresses referenced by the program during its execution. This string depends only on the program itself and its input data. It depends neither on the structure of the physical memory nor on the memory allocation strategy. Reference strings are widely used to
describe the dynamic behavior of a program. According to the level of detail of the description, reference strings may be divided into four types as shown in Figure 2. These reference strings may be a sequence of virtual memory addresses, of block references, of page references, or of symbolic instruction references.

Because of its ability to describe important aspects of program behavior, the reference string has become a central element in research when dealing with program restructuring, page replacement strategies, system modeling and, of course, program behavior models [45]. The next section discusses the practical problem of how to obtain the reference string from a program in execution.

Methods for Reference String Collection

The current methods of reference string collection can be categorized into two types: hardware monitor and software monitor.

**Hardware Monitor**

Hardware monitor involves a second processor which traces the first processor. As shown in Figure 3, the second processor has a hardware tap to the address bus of the first processor. Whenever the first processor accesses the main memory, the memory address is sent out on the address bus. The second processor then reads this address via its bus tap.
FIGURE 3. Hardware Monitor

A hardware-lock on the processor guarantees that the processing of the address is completed by the monitor processor before the next memory request is sent out on the bus by the first processor. This lock is controlled by the
monitor processor and prevents the first processor from continuing until the lock is removed. The lock is "on" while the address is being processed by the monitor processor and is not turned "off" until the monitor processor is ready to accept another address. With this method, the execution time of the process on the first processor includes the idle time during the lock.

**Software Monitor** Software monitoring can be accomplished by emulation. The traced process is interpreted by a software emulation of the hardware. The software emulator processes the references of the process being traced during the interpretation of each instruction. This method requires a complicated piece of software to perform the emulation.

Software monitoring can also be accomplished by a method termed process switching. In this method, after the execution of each instruction of the traced process, the processor is switched to a monitor routine which will process the addresses associated with the instruction. The switching can be done by a hardware interrupt which calls the monitor routine. In the monitor routine, the address of the last instruction executed by the interrupted process is retrieved. With this instruction, the addresses of any operands can be determined. The state of the interrupted process is restored after the monitor routine completes its
process, then allowing another instruction of the traced process to be executed.

The current tracing methods described above involve a large overhead in the tracing process. The two important overhead factors are the delayed time in execution and large storage requirement for recording the reference strings. Note that these two overhead properties of tracing are in conflict with each other. In order to reduce the amount of processing time, the reference string can be saved for later processing while the process is being traced. But, on the other hand, measurements can be calculated during the tracing so that the reference string does not have to be stored. For real-time tasks the high overhead adds another problem. The overhead of complete collection of the reference string alters the behavior of time-critical systems. Time-critical systems are defined as systems where the real-time between events is important and cannot be significantly altered without disturbing the normal functioning of the system. Examples of such systems are operating systems, real-time process control systems and interactive systems.

The current methods of reference string collection have been shown to have high cost in time delay and storage requirements, and hence also alter the behavior of time-critical systems. One solution to these problems is given
by Wittneben [49] and independently by Smith [43], Allen and Clark [1], and Orchard [37]. They designed a class of reference string sampling methods that has a low overhead and also preserves the structural characteristics of the original complete reference string throughout the sampling process. The structural characteristics which are preserved include the page fault rate, working set measurement, and lifetime curve.

Reference String Sampling Methods

The sampling methods define which portions of a complete reference string are monitored and which portions are skipped. In [49], these methods are described and discussed in detail. Both the time-delay overhead and the storage overhead are lowered by the sampling technique, since there are fewer references to process and store. Note that the sampling method works for both hardware and software tracing methods. The monitor processor in the hardware tracing method could use the skip time to process or store the references collected during the monitoring period. With the software tracing method, the sampling method cuts the overhead of the tracing. In the next section, we will show how reference strings can be used to improve a program's behavior.
Program Restructuring

Introduction: Program Restructuring

In this section, it will be shown how program restructuring can improve program behavior. We begin with a simple example to introduce the basic idea behind program restructuring.

Consider a program that consists of one main routine "MAIN_a" and four subroutines SUB_b, SUB_c, SUB_d, and SUB_e. Let us assume for simplicity that blocks MAIN_a, SUB_b, SUB_d and SUB_e are one-half a page in size, and that block SUB_c is one full page in size. Thus, the virtual space occupied by the program will consist of three pages. The physical size of the blocks permits them to be placed in pages in different combinations. The layout of the program in its virtual space, generally decided by the linker, is influenced by the order in which programs are presented for linking. Program restructuring is a modification of the program's layout in virtual memory in an attempt to improve its locality. In our example, this modification only entails relinking the modules of the program after having changed the order in which they are presented to the linker. If the blocks had not been chosen so as to be relocatable with respect to each other, the program could still be restructured, but some changes in the source code
FIGURE 4. Two different layouts of a program in virtual memory and the recompilation of some modules would probably be required.

To show that the behavior of a program is sensitive to its layout in virtual memory, let us analyze the performance...
TABLE 1. A block reference string and two corresponding page reference strings.

<table>
<thead>
<tr>
<th></th>
<th>Page Reference String</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CCCBBBDBBBCCDDBAAACCEECCECCAAAECC</td>
</tr>
<tr>
<td>2</td>
<td>222111331122233111122332222111332</td>
</tr>
<tr>
<td>3</td>
<td>22233333332223331112211222211112</td>
</tr>
</tbody>
</table>

of our program with two different layouts. The two layouts we shall consider here are shown in Figure 4. Different layouts generally produce different page reference strings from the same block reference string. An example is given in Table 1, Rows 2 and 3. Assume the memory allocation in this example is a fixed size of two pages. If the replacement algorithm used by the system is LRU (Least Recently Used), then the page reference string in Table 1, Row 2 is generated by the page structure shown in Figure 4(a), and has eleven page faults. The page reference string in Table 1, Row 3 is generated by the page structure shown in Figure 4(b). With this page structure, the same block reference string that caused eleven page faults, now has only three page faults. This example has shown how the
restructuring of the blocks of a program improves the execution performance by decreasing the number of page faults.

Program restructuring is the process of mapping the virtual addresses of a program's address space into pages to reduce the page fault rate or the (mean) memory allocation, or both. The user benefits from restructuring by having a faster execution and, hence, a less costly program. The potential of program restructuring for improving the performance of program running in a virtual memory system is illustrated by numerous research studies reported in the literature. Several of these research studies are surveyed below.

Tsao, Comeau, and Margolin [47] showed that the page fault rate experienced during execution of a program can be reduced much more by a good ordering of code in virtual-memory than by other factors affecting the page fault rate (such as the page replacement algorithm).

Hatfield and Gerald [28] executed a program with a set of "typical" input data. The reference pattern of the program was then recorded and used to formulate a deterministic model of the intermodule relationships. The model reflected the relative desirability of linking modules into the same page in virtual-memory. An intuitively based clustering procedure was applied to the model to partition
the modules into logical pages and the program was then recompiled and run. Hatfield and Gerald noted a substantial reduction in the page fault rate experienced by the program during execution.

The program restructuring procedures developed by Hatfield and Gerald did not utilize an underlying model of program behavior and made no assumptions about the page replacement algorithm used by the virtual-memory system in which the program under study was to be run.

Ferrari [20], [22], [23], [24], Masuda et al. [35], and Johnson [30] developed program restructuring procedures based on the assumption that the program under study was to be run in a virtual-memory system operating under working set memory management. They differ in the specific method for measuring the intermodule relationships and in the clustering procedure applied. The procedures developed are related to those of Hatfield and Gerald [28]. Ferrari [21] also developed a program restructuring procedure based on the assumption that the program under study was to be run in a virtual-memory system operating under LRU or other specific memory management algorithms.

For the restructuring operation to be successful, the mean block size must be much smaller than the page size. A good rule of thumb says that it should be between 1/3 and 1/10 of the page size [25], [28]. In general, the smaller
the blocks, the more effective, but also the more expensive the method.

**Benefits from Program Restructuring**

If we can reduce the number of page faults by program restructuring, then we can improve the performance of a virtual memory system in several areas:

1. Reduced time spent paging (I/O)
2. Less supervisory overhead spent in main memory and paging management
3. Better throughput on the average, because of the increased rate of execution
4. Better paging operation when it is needed, because there will be less contention for the paging device

**The General Program Restructuring Model**

Most of the program restructuring methods proposed so far can be conceptually described by the following steps.

**Step 1:**

The program to be restructured is divided into blocks. A block is a set of contiguous instructions of the program to be restructured, which the restructuring procedure can relocate in the virtual address space.
Step 2:
A restructuring algorithm is applied to the blocks. The result of this step is a matrix called the restructuring matrix R. This matrix may also be viewed as representing a graph, called the restructuring graph. A restructuring graph is a nondirected graph whose nodes are weighted by the size of the block represented by that node, and whose edges have labels quantifying the desirability of grouping into the same page as the two blocks which define that edge. The edge label values are typically derived from an analysis of the reference string.

Step 3:
A clustering algorithm is applied to the restructuring matrix or graph to solve the following problem: Group the nodes of the graph into clusters so as to maximize the sum of intracluster edge labels under the constraint that the sum of the weights of all nodes in each cluster not be greater than the page size. This is a well-known problem in graph theory [21].

Step 4:
The clusters of nodes determined in step 4 specify the restructured block-to-page mapping. In this step, the program is rearranged in virtual-memory according to this mapping. The size of a cluster will not generally equal the
size of a page. If the information in the program is to be compacted to eliminate some (or all) gaps in the virtual space, another algorithm (possibly also based on B) may be applied in this step to determine the best ordering of the pages to be compacted.

Program Restructuring Methods

Methods for automatic program restructuring are divided into three major categories: a priori restructuring, a posteriori restructuring and dynamic restructuring. In the next three sections, we will discuss these three categories. First, we begin with the a priori restructuring method.

A Priori Restructuring Method

In the literature, this method is also called static restructuring [2], [33], [48]. Static restructuring is done at compile time, or at least before run time. Static restructuring maps into the same page those portions of a program that are likely to be the most frequently executed, such as the instructions in the body of a loop. These portions of the program are referred to as basic blocks. In the static method, the program is structurally segmented into basic blocks by graph theory methods.

In the next few sections, we try to survey only the most important and widely used a priori restructuring methods.
FIGURE 5. Classification of Restructuring Algorithms

Ramamoorthy's Approach

Ramamoorthy made the observation that the paths of possible control flow may be represented by a directed graph or a connectivity matrix.
A strongly connected subgraph of a program's graph that is not a proper subset of any strongly connected subgraph is called a maximal strongly connected subgraph. Ramamoorthy presented a method for reduction of inter-block references based on the principle that the blocks represented by a maximal strongly connected subgraph (MSCS) should not be split into two or more pages. The difficulty in this technique is that an entire program could be a maximal strongly connected subgraph and be larger than a page. This problem is tackled by the following two methods.

**Lowe's approach**
A generalization of Ramamoorthy's suggestions is made by Lowe [33] in order to allow their application when the size of the MSCS is greater than the page size. Lowe's method can be briefly summarized as follows. We begin to traverse the directed graph of the program to be restructured, attempting to pack the shortest loop into a single page. Successively slower (more global) loops are, if possible, packed into the page in turn. This method is based on the assumption that the shorter loop be contained in a single page in order to reduce the number of inter-page transfers.

**Ver Hoef's approach**
Ver Hoef's algorithm [48] operates in three phases. In phase one, loops are detected and their component instructions are identified. The smallest loops are found first, followed by successively
larger loops. The size of a loop is measured in terms of the number of instructions that comprise the loop. As soon as a loop has been detected, the instructions comprising the loop are treated as a single unit. Phase one is finished after the largest loop that could fit in a single page has been considered.

When phase two begins, the program consists of a collection of program units forming a linear string, a tree structure, a lattice structure, loops larger than a page or some combination of these. In this phase, a particular path is traversed and the program units along that path are identified and merged until the segment size is exceeded or the end of the path is encountered. In either event, a new path is selected and traversed in like manner. Phase two is complete when all such paths have been traversed and all merging based on connectivity accomplished. However, there may yet remain merged units of less than segment size. Such units are identified in phase three and merged to minimize wasted space.

Baer and Caughey's approach A restructuring method for FORTRAN source programs is given by Baer and Caughey [2]. Their method identifies do-loops and their embeddedness in the program. Their basic philosophy, similar to Lowe's, is to pack first those statements which are in the most nested cycles. Again, this method is based
on the assumption that the statements in the most nested cycles are the most important factor in reducing the number of inter-page transfers. The Baer and Caughey method guarantees that the statements in the most nested cycles are always packed into the same page.

**Snyder's approach** The last method to be discussed in this section was recently proposed by Snyder [44]. This method also works on the source code level. Snyder uses the following simple method to calculate the weight of each edge of the directed graph of the program to be restructured. The weight of an edge is the sum of the raw number of source references from block b(i) to block b(j), plus ten times the number of source references inside a loop in block b(i) to block b(j). References that occurred in nested loops counted geometrically, e.g., as 100 for a double loop, and so on. This gives a coarse approximation to the number of times block b(j) will follow block b(i) in the address trace. Then, he chose to traverse the program graph in a depth-first fashion to cluster those most related blocks together. Depth-first traversals are preferred over breadth-first arrangements because they group a node and its successors together, thereby achieving the desired prefetching.

The static restructuring methods surveyed above do improve the performance of the restructured program, but
these compiled time decisions are only estimates of the true referencing behavior. Greater improvement can be realized by the restructuring methods based on actual run-time data. Several of these methods are reviewed in the next section.

A Posteriori Restructuring Method In the literature, this method is called dynamic-off-line restructuring [3]. This method refers to those restructuring techniques using information that is available only after the program has run to completion (i.e. based the program memory reference behavior, the reference string). These a posteriori restructuring methods, though more expensive, use the real referencing behavior to restructure the program and, therefore, can produce a more improved restructuring than the static approach. This method does have two disadvantages. First, it is more expensive than static methods since a complete reference string must be obtained in order to do the program restructuring. Second, this method can apply to only those programs whose block referencing behavior is not overly sensitive to its input data.

Various restructuring algorithms and clustering algorithms are described in the next two sections. First, we begin with restructuring algorithms.

The a posteriori restructuring algorithms can be divided into two classes as shown in Figure 5.
Restructuring algorithms in the first class take advantage of the knowledge of memory management strategies, whereas those in the second class are independent of memory management strategies. In the next few sections, both strategy-independent algorithms and strategy-dependent algorithms will be discussed.

**Strategy-Independent Restructuring Algorithms**

We begin by presenting Hatfield and Gerald's interblock reference model for defining the strength of connection between blocks.

**Nearness Method**

The Nearness Method (NM) [28] is a classic strategy-independent restructuring algorithm. The NM increments the edge labels of block pairs referenced consecutively. For example, if a reference to block \( b(i) \) is immediately followed by a reference to block \( b(j) \), then the label on the edge \([b(i), b(j)]\) is incremented by one. More precisely, the nearness method constructs the interblock reference matrix \( M \), representing the restructuring graph as follows. Let:

\[
P = [b_1, b_2, \ldots, b_n]
\]

be the program of \( n \) blocks;

\[
B = [b(1), b(2), \ldots, b(L)]
\]

be a block trace of the program.

Then \( M = M(i, j) \) for \( i, j = 1, 2, \ldots, n \)

where \( M(i, j) = \sum f(i, j, t) \) for \( t = 1, 2, \ldots, L \),

where \( f(i, j, t) = 1 \) if \( b(t) = i \) and \( b(t+1) = j \)

\( f(i, j, t) = 0 \) otherwise.
Thus, the matrix $M$ is defined as an $n \times n$ matrix whose entry $a(i,j)$ (for $i,j = 1, 2, \ldots, n$) is the number of transitions from block $b(i)$ to block $b(j)$ during the execution of the program. The label of the edge connecting nodes $b(i)$ and $b(j)$ in the restructuring graph is then computed as $a(i,j) + a(j,i)$. Because of its narrow field of observation (only two adjacent block references at a time), the NH interblock reference model does not contain any information about the length of the time interval between successive references of block $b(i)$ to block $b(j)$. However, paging may depend quite heavily on the length of these time intervals. For example, consider two block reference strings $B1$ and $B2$. Let the reference $b(i), b(j)$ appear $k$ times in both $B1$ and $B2$, but let these appearances occur at long distances from one another in $B1$ and be consecutive in $B2$. The value of $a(i,j)$ is $k$ in both cases, and, if everything else is equal, $a(j,i)$ is zero in $B1$ and $k-1$ in $B2$. Thus, the probability that the clustering algorithm will group $b(i)$ and $b(j)$ together is higher in the case of $B2$. However, the cost of not grouping them together is greater in $B1$, since the number of page faults due to the reference to $b(j)$ immediately following those to $i$ will be at most one for $B2$, but might be much greater than one for $B1$ (it can reach $k$ in the worst case).
The next four program restructuring algorithms are based on a recently proposed behavior model called the bounded locality intervals model, which allows us to give a precise definition of the localities of a program.

Bounded Locality Intervals (BLI) have been proposed recently by Madison and Batson [34] to model dynamic program behavior from the locality's point of view. Two features of their model are that it contains no parameters, so that localities are determined naturally, and that it defines hierarchies of localities. A locality in this context is a proper subset of a program's blocks defined by the BLI model. A set of \( k \) distinct blocks of a program is said to be formed when it consists of the \( k \) most recently referenced blocks of the program, and it is said to be terminated when a block not belonging to the set is referenced for the first time after the set is formed. An activity set at time \( t \), \( A_k(t) \), is defined as any set of \( k \) blocks all of which have been re-referenced since the set was formed. The lifetime of an activity set is defined as the interval between the time the activity set is established and the termination time. A Bounded Locality Interval (BLI) is defined as the pair consisting of an activity set and its lifetime.

A set of strategy-independent restructuring algorithms has been derived from the concept of BLI by Kobayashi [32]. He observed that if a set of blocks belongs to an activity
set of high level, they should be packed close to each other in the virtual address space to decrease the working set size of the program and the number of inter-page transfers.

**AS1** The first of his algorithms, called the Activity Set algorithm-1 (AS1) is defined as a restructuring algorithm that increments the label of the edge between blocks $b(i)$ and $b(j)$ by an amount equal to the level of the smallest activity set to which they belong.

**AS2** The second one, called the Activity Set algorithm-2 (AS2), is a simpler algorithm than AS1. The AS2 algorithm keeps track of the activity sets of level one and may be described as an algorithm that increments by one all the labels of the edges between the blocks which belong to an activity set of level one. AS1 can be used in the situation where the hierarchy has only a few levels or where the average size of the activity sets of level one is small enough to fit in main memory.

Kobayashi also observed that some of the members of an activity set can stay in the set unreferenced. To reduce the activity set size and remove inactive members, he introduced a parameter $T$ that plays a role similar to the window in the working set strategy. This allows him to define a set of blocks called a strict activity set. A strict activity set at time $t$ is defined as any activity set, all the blocks of which have been referenced after the
time $t-T$. By using the strict activity sets, he introduced two more restructuring algorithms analogous to AS1 and AS2.

**SAS1** The Strict Activity Set algorithm-1 (SAS1) is defined as the algorithm that increments the label of the edge between a pair of block $b(i)$ and $b(j)$ by the level of the strict activity set to which they belong.

**SAS2** The Strict Activity Set algorithm-2 (SAS2) is defined as the algorithm that increments by one the label of the edge between a pair of blocks only when these blocks belong to a strict activity set of level one.

**Strategy-Dependent Restructuring Algorithms**

Now, we begin to study the class of strategy-dependent algorithms. First, we would like to introduce those program restructuring algorithms developed by Ferrari [20], [21], [22], [23] and [24]. Before proceeding to describe Ferrari's restructuring algorithms, we would like to introduce some important notations, terms and assumptions related to the algorithm.

In this section, basic blocks are the units of restructuring. Therefore, the reference string is defined in terms of blocks rather than of pages. We assume that references are made at the discrete time instants $1, 2, 3, \ldots, t, \ldots$ to the blocks $b(1), b(2), b(3), \ldots, b(t), \ldots$, respectively. The block sequence generated by the program is called the block reference string. The most important
term is "critical reference," defined as a reference to a block that is not resident in main memory and causes a block fault.

Based on the critical reference concept, Ferrari developed two sets of strategy-dependent restructuring algorithms which he also called program tailoring algorithms. The first set of algorithms is the critical-set algorithms, whose ultimate objective is to reduce the page fault rate of a program.

The basic idea of these algorithms is that if the reference is critical, the mapping should make it noncritical, that is, the corresponding block should be ready in memory when it is referenced. To better fit the program's behavior to the model's behavior, the block involved in a critical reference should be grouped with at least one of the blocks already in memory at the time the reference is issued. Because conflicts usually arise, we shall have to choose those groupings that reduce most drastically the number of page faults. To obtain this reduction, a critical-set algorithm constructs the restructuring graph as follows: For each block reference, determine whether that reference is critical, and, if so, increments by one the labels of all edges connecting the node that represents the critical reference to all those nodes representing the blocks currently in memory.
The algorithms belonging to this family are CWS, CLBU, CFIPO, CLFU, CMRU, CMIN, etc. In the following, the critical algorithms CWS and CLBU, two of the most important ones in the family, are described and explained in detail.

**CWS** The Critical Working Set [20] (CWS) restructuring algorithm is intended to decrease the number of page faults generated by a program executed under the working-set memory management policy.

A working set \( W(t,T) \) is said to be a critical working set (CWS) if \( b(t+1) \) is not resident in \( W(t,T) \). In this case, \( b(t+1) \) is said to be a critical working set reference. A page fault in the page reference string \( P \) always comes from a block fault in the block reference string \( B \), but not all block faults have a corresponding page fault.

The goal of the CWS algorithm is to minimize the number of page critical references in the page reference string or the number of block faults that become page faults. To achieve that goal, the restructuring graph is derived by the following procedure.

A graph \( G \), called the CWS graph, will be constructed from the block reference string \( B \). Noncritical references in \( B \) will be disregarded. CWS increments by one, whenever a block fault occurs, the labels of the edges \([b(i), b(j)]\), where \( b(j) \) is the block reference which caused a block fault at time \( t \), and \( b(i) \) is any block which belongs to the block
working set $W(t, T)$. Note that the labels of the edges $[b(i), b(j)]$ are the number of critical references having $b(j)$ as their critical reference and containing $b(i)$. So $[b(i), b(j)] + [b(j), b(i)]$ is the number of critical references that will disappear if $b(i)$ and $b(j)$ are mapped into the same page. Next, the CLRU algorithm will be discussed briefly.

**CLRU** The critical LRU (CLRU) [23] restructuring algorithm is intended to decrease the number of page faults generated by a program executed under the LRU memory management policy.

Let us assume that a program is running under the LRU policy with fixed allocation of $n$ pages and that the $m$ blocks already in memory occupy, at that instant, the top $m$ positions of the block stack, where $m$ is the largest integer such that the size of the $m$ blocks in memory is less than or equal to the size of $n$ pages. If the block referenced next is already in memory, because its distance in the block stack is not greater than $m$, the reference is said to be noncritical, and nothing is done. If the reference is critical, increment by one the labels of the edges connecting the block, that causes the critical reference to all of the blocks occupying the first $m$ positions of the LRU block stack.
The principle of critical references can be applied to all local replacement policies, since it is always possible with such policies to derive from a block reference string the sets of blocks that are certainly going to be in memory at any given time. There are, for instance, the CFIFO, the CLFU, the CMRU, and even the CMIN restructuring algorithms, whose objectives are to tailor programs to the first-in-first-out, least-frequently-used, most-recently-used, and MIN replacement policies, respectively.

Ferrari [21], [24] observed that the notion of critical set cannot be defined when the contents of primary memory at any given time cannot be predicted solely from the behavior of the program to be restructured (e.g., a program running under the global IRU environment). Fortunately, a single parameter model developed by Bard [4], called the page survival index (PSI), which summarizes the influence of the rest of the workload on the process being considered, can be used to develop a critical set algorithm for global LRU policy.

The execution of a program is usually suspended when
a) a page fault is generated, or
b) the time quantum assigned to the process expires, or
c) the completion of an I/O operation must be waited for.

The value of the PSI represents the mean number of such interruptions that a page of the program can survive (that
is, remain in main memory) without being referenced. High values of the PSI correspond to situations of low paging activity and vice versa. The composition of the working set of pages may be roughly estimated from the page reference string by applying the following PSI method: a page in main memory will remain there unreferenced for a number of interruptions equal to the value of the PSI and will disappear immediately after the last of these interruptions.

It is easy to see that the PSI method can also be used to estimate the sequence of the working sets of blocks by applying it to a block reference string.

Based on the above conclusion, Ferrari [24] developed the CPSI restructuring algorithm.

**CPSI** The Critical PSI (CPSI) [24] restructuring algorithm is intended to decrease the number of page faults generated by a program executed under the PSI policy. CPSI constructed the restructuring graph as follows: If the block \( b(i) \) referenced at virtual time \( t \) is critical, then the CPSI algorithm will increment the labels of all edges connecting \( b(i) \) to the members of working set at time \( t \).

Ferrari [24] also observed that if the main objective of restructuring is to optimize some performance index other than the page fault rate, other program restructuring algorithms, not belonging to the class of critical
algorithms, can be devised. For example, in a working set environment, one concern may be to reduce as much as possible the mean working set size of a program. So, the second class of restructuring algorithm developed by Ferrari, called the minimum-set algorithm, attempts to reduce the mean size of a program's working set.

The Minimum Working Set (MWS) [21] restructuring algorithm is intended to reduce the mean working set size of a program executed under the working set management policy. In order to achieve this goal, the restructuring graph is constructed by the following procedure.

A graph G, called the MWS graph, will be constructed from the block reference string B. At each new reference, MWS increments the labels of the edges connecting each pair of blocks in \( W_b(t) \), to reduce the mean size of the working set of blocks. Note that the value of the edges \( [b(i), b(j)] \) is the number of times that block \( b(j) \) was referenced when block \( b(i) \) and block \( b(j) \) both were in the block working set.

The Minimum PSI (MPSI) [24] restructuring algorithm is intended to reduce the mean working set size of a program executed under the PSI policy. MPSI constructed the restructuring graph from the block reference string in almost the same way as the CPSI.
However, MPSI will increment at every time t the labels of the edges connecting all the pairs of members of the working set.

The strategy-dependent restructuring algorithms are expected to be superior to the strategy-independent algorithm because they use additional information. Several experiments confirmed this intuitive expectation [23], [25].

Note that the above restructuring methods can be applied at two levels: internal and external. In internal restructuring, the atomic unit of reorganization is a group of instruction and/or data within some routine. External restructuring considers entire relocatable routines as the atomic units to be rearranged. These routines are called sectors, blocks, records, or modules in the literature.

The disadvantage of a posteriori restructuring technique is that it requires the preprocessing of a representative memory trace of each subject program, making its use expensive because of the large amount of data involved and also restricting its use to production programs which are not highly data-dependent. To solve these two problems, Baer and Sager [3] proposed a dynamic restructuring method that uses information acquired during run time to change the page structure during the execution of the program [3]. The detail of this method will be discussed in the next section.
Dynamic Restructuring Method  Dynamic restructuring [3] (or dynamic-on-line restructuring), is a technique for virtual memory restructuring that could be useful in a memory hierarchy with more than two levels. This method is especially important for data dependent programs where the referencing of pages is highly dependent on the input data, and where the system is in operation over a long period of time (e.g. on-line database systems).

Unfortunately, it is not possible to restructure a program at execution time in a straightforward manner due to the overhead of virtual address modification. Note that the replication of heavily used pieces of code is trivial with a priori or a posteriori restructuring, but this is not true in the case of dynamic restructuring. However, a restricted form of dynamic restructuring can be realized whenever two contiguous levels of memories in a hierarchy have a different frame size. Assuming that the higher levels have a larger frame size, pages must coalesce as they move up the hierarchy (farther from the executable store). Likewise, as pages move down the hierarchy, they pull their partners down with them. With this arrangement, a movement down the hierarchy is in fact preloading the lower levels.

At all levels of the hierarchy except the lowest one, the coalescing of pages can be handled entirely by software. At lower levels, especially in the case of cache memory, the
coalescing of pages must be handled by hardware because construction, manipulation, and use of the directories must be done at speeds which compare to that of the main memory.

**Clustering Algorithms**

**TABLE 2. List of Clustering Algorithms.**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>CNN</td>
</tr>
<tr>
<td>2.</td>
<td>CFN</td>
</tr>
<tr>
<td>3.</td>
<td>CAN</td>
</tr>
<tr>
<td>4.</td>
<td>CANV</td>
</tr>
<tr>
<td>5.</td>
<td>SIP</td>
</tr>
<tr>
<td>6.</td>
<td>ICB</td>
</tr>
<tr>
<td>7.</td>
<td>UNN</td>
</tr>
<tr>
<td>8.</td>
<td>UPN</td>
</tr>
<tr>
<td>9.</td>
<td>UAN</td>
</tr>
<tr>
<td>10.</td>
<td>UANW</td>
</tr>
<tr>
<td>11.</td>
<td>HG</td>
</tr>
</tbody>
</table>

After a restructuring graph has been built, a clustering algorithm is applied to it in order to determine how the nodes of the graph can be grouped together so that the sum of the labels of edges connecting nodes belonging to different groups is minimized, and the size of all nodes in each group does not exceed the page size. This is a well-known, classical NP-complete problem in graph theory. But
hopefully, finding an optimal solution is not essential to the practical success of the restructuring procedure. Fast and approximate clustering algorithms [30] have been experimentally found to be so effective as to suggest that the further refinements obtainable by optimum algorithms would probably require much longer execution time for relatively little gain over an approximate clustering algorithm. The total improvement produced by the approximate clusterings are consistently close to that produced by an optimal clustering algorithm. Note that the running time of any clustering algorithms grows rapidly with the number of nodes in the graph. Thus, this number cannot be made as large as the increasing improvements which can be obtained as block size decreases would encourage us to make.

The purpose of this section is to survey those automatic clustering methods which have appeared in the literature. The clustering methods presented in this section may be applied to any of the restructuring graph of section 4.

Since an optimal clustering algorithm is an NP-complete problem, several clustering procedures based on heuristic approaches have been developed with the following significant properties. First, they are completely automatic. Second, all these clustering methods tend to group the blocks with the strongest interblock strengths
into the same page and they tend to minimize the interaction of blocks clustered into different pages. Third, all these clustering procedures are computationally fast and suitable for practical purpose. Fourth, all these clustering procedures are easy to implement. Fifth, all these clustering procedures produced restructured programs which showed substantial improvements in their paging performance.

The technique of the following clustering procedures is to take a restructuring graph and group blocks into pages such that the sum of the labels of edges connecting nodes within pages tends to be maximized. We begin by presenting the Nearest Neighbor Methods.

Johnson [30] developed two classes of hierarchical clustering methods that successively cluster the nearest two clusters under a specified bond strength definition. They are the Constrained Nearest Neighbor methods and the Unconstrained Nearest Neighbor methods.

In the following, several intercluster bond definitions are given, and a clustering procedure is defined over each of these definitions. A cluster is a collection of one or more blocks grouped together by the clustering algorithm. The cluster size is less than or equal to the page size in the case of constrained algorithms. In the unconstrained case, the cluster size is unlimited. The intercluster bond between cluster $x$ and $y$ is denoted by $B(x,y)$. 
The algorithms belonging to the constrained family are CNN, CFN, CAN, and CANW. The CNN, CFN, and CAN clustering procedures are variations of clustering procedures widely used in the field of multivariate analysis. According to Johnson [30], the CANW procedure is the best one in this family.

**CNN** The Constrained Nearest Neighbor bond $B(x,y)$ between two clusters $x$ and $y$ is represented by the label of the edge with the largest value between the two clusters, and under the condition that the sum of the size of the two clusters is not greater than the page size.

This definition of $B(x,y)$ is applied using the following procedure: First, partition the $n$ blocks of a program into exactly $n$ clusters, where each cluster contains one block. Then, at each step in the clustering process, the nearest two clusters are combined to form a new cluster provided that the new cluster size does not exceed the page size. The nearest two clusters are defined as the clusters $x$ and $y$ with the largest value of $B(x,y)$. The process comes to an end when new clusters cease to appear.

**CFN** The Constrained Farthest Neighbor bond $B(x,y)$ between two clusters $x$ and $y$ is represented by the label of the edge with the smallest value between the two clusters, and under the constraint that the sum of the size of the two clusters is not greater than the page size. The CFN
The clustering algorithm is identical to the CNN except that the bond definition is different.

**CAN** The Constrained Average Neighbor bond $B(x,y)$ between two clusters $x$ and $y$ is represented by the average value of the labels of the edges between the two clusters, and also under the constraint that the sum of the size of the two clusters is not greater than the page size. By changing the bond definition in the clustering algorithm of CNN to the above definition, the clustering algorithm CAN is obtained.

**CANW** The Constrained Average Neighbor Weighted bond $B(x,y)$ between two clusters $x$ and $y$ is represented by the sum of the labels of the edges between the two clusters, and also under the constraint that the sum of the size of the two clusters is not greater than the page size.

The above clustering algorithms require that the size of a cluster never exceeds the page size, but in reality, natural clusters of blocks may be larger or smaller than a page size. By removing the page size constraint of the constrained algorithms, we get a set of unconstrained clustering algorithms. The clustering algorithms belonging to this family are UNN, UFN, UAN and UAWN, where UNN, UFN, UAN, and UAWN are defined to be exactly the same as CNN, CFN, CAN, and CANW, respectively, with the exception that the page size constraint is removed. These algorithms try
to make clusters covering several pages without any consideration of page sizes and to assign each of them to several contiguous pages. That is, in the unconstrained cases, clusters may be combined independently of their sizes.

\[ \text{UNN} \quad \text{The Unconstrained Nearest Neighbor bond } B(x,y) \]

between two clusters \( x \) and \( y \) is represented by the label of the edge with the largest value between the two clusters.

With a slight modification, the clustering procedure for the constrained clusters becomes the clustering procedure for the unconstrained clusters. The clustering procedure for the unconstrained clusters is defined below.

Partition the \( n \) blocks of a program into exactly \( n \) clusters, where each cluster contains one block. Then, at each step in the clustering process, the nearest two clusters \( C(1) = a(1), a(2), a(3), \ldots, a(i) \) and \( C(2) = b(1), b(2), b(3), \ldots, b(j) \) are concatenated to form a new cluster. It is important to note that such a cluster is defined as an ordered list of blocks. It is obvious that the concatenation of two ordered lists is also an ordered list. In this way, the clustering algorithms insure that the relative order in which blocks are clustered is preserved. This is important because the clustering procedure ends when all the connected clusters are grouped into one giant cluster, which could be the whole program.
But the order of the blocks in the constrained clusters is not important, because a cluster will always fit into a page.

**UPN** The Unconstrained Farthest Neighbor bond \( B(x,y) \) between two clusters \( x \) and \( y \) is represented by the label of the edge with the smallest value between the two clusters. By changing only the bond definition of the clustering algorithm in UNN to the above definition, we will get a new clustering algorithm **UPN**.

**UAN** The Unconstrained Average Neighbor bond \( B(x,y) \) between two clusters \( x \) and \( y \) is represented by the average value of the labels of the edges between the two clusters. The clustering algorithm **UAN** is obtained by changing the bond definition in the clustering algorithm of **UNN** to the above definition.

**UANW** The Unconstrained Average Neighbor Weighted bond \( B(x,y) \) between two clusters \( x, y \) is represented by the sum of the labels of the edges between the two clusters. The clustering algorithm **UANW** is defined in the same way as the clustering algorithms **UPN** and **UAN**.

**SIP** In his thesis, Johnson [30] developed a Block (or Sector) Interchange Procedure (SIP) which he uses as a clustering method.

**SIP** is an efficient clustering algorithm that can be considered as a further refinement procedure for other
clustering methods. SIP is particularly useful in improving the performance of the class of unconstrained clustering methods. Another application of SIP is in the evaluation of reprogramming to break up large blocks into smaller parts.

The basic strategy of the SIP is to reassign blocks to clusters by exchanging two blocks of different clusters when the exchange provides a positive contribution to the sum of the block connections within clusters.

HG Hatfield and Gerald [28] developed a matrix-oriented clustering procedure. Their clustering procedure can be applied to any interblock reference matrix \( M = M(i,j) \) representing the restructuring graph. Their goal is to cluster blocks into submatrices with each submatrix corresponding to one page of virtual memory. They try to accomplish this goal by ordering the rows and columns of \( M \) to bring the largest \( m(i,j) \) values into square submatrices along the diagonal. We summarized briefly the HG clustering procedure below. For further detail, refer to [28].

Let

\[
E = E(i,j) \quad i,j = 1,2,\ldots,m \quad (m \text{ is the number of blocks})
\]

where

\[
E(i,j) = \begin{cases} 
-M(i,j) & \text{if } i <> j \\
\sum_{j=1}^{m} M(i,j) + 2m & \text{if } i=j
\end{cases}
\]
The inverse matrix of $E$ is calculated, then a row in the inverse is chosen, and a set of blocks in that row is clustered into a page, and the process is iterated until all blocks are assigned.

**ICB** Johnson [30] developed a method to identify natural clusters of dense block interactions. This is accomplished by permuting the rows and columns of an interblock reference matrix representing the restructuring graph in such a way as to group the numerically larger matrix elements together.

Johnson defines an Intercluster Bond model (ICB) to measure the interblock reference matrix $M = M(i,j)$ for $i,j = 1,2,...,m$. He defined the ICB in [30].

The ICB is defined so that a matrix $M$ that has dense clusters of numerically large elements will have a large ICB when compared with the same matrix whose columns and rows are permuted such that numerically large elements are more uniformly distributed over the array cells.

The ICB clustering method can be divided into four steps:

1. Compute and save the set of intercolumn bonds for all pairs $(i,j)$ of columns.
2. Pick any one of the columns, put it into a list, and let $k=1$. 
3. For each of the remaining \( m-k \) columns, compute the contribution to the ICB measure for each of the \( k+1 \) possible positions to the left and to the right of each of the \( k \) columns already placed in the list. Place the column that gives the largest incremental contribution to the ICB measure in its best location in the list.

4. If \( k=m \) then stop, else \( k=k+1 \) and go to Step 3.

After carefully examining various restructuring methods, the a posteriori program restructuring method was found to be a very attractive and powerful tool to improve the performance of the virtual memory systems.

Statement of Research and Motivation

Perhaps the most serious objection that may be raised against a posteriori restructuring methods has to do with the role of the input data. The restructuring algorithm is applied to a block reference string which represents the behavior of a (program, input) pair. If the behavior of the program is very sensitive to the input data, the mapping suggested by the restructuring procedure might produce, with input data different from those used in the instrumented run, a behavior worse than that of the nonrestructured program. However, several experiments [23], [28] have shown that the behavior of programs such as compilers, assemblers, editors, operating systems, and other production programs
for which restructuring is most advantageous, tends to be remarkably insensitive to the input data, at least from the viewpoint of restructuring efficiency [23], [28].

Another objection is that a posteriori restructuring methods are costly to use because a complete reference string must be obtained. Even when a dramatic improvement in performance is possible because of the poor locality of the initial program, the cost of the restructuring procedure is not justified if the program is to be executed only a few times. But for application or system programs that are run frequently, the performance improvement may offset this cost. All types of improvement studies described in this thesis should be considered only for programs to be run more than a certain minimum number of times.

In the case of locality improvement via restructuring, this number depends primarily on the quality of the program, on the specific restructuring algorithm used, and on the choice of the blocks. For all production programs that constitute a large fraction of the work load in most installations, restructuring is certainly advantageous. Note also that, beyond the break-even point, all of the savings of each run contribute to the total payoff, since the procedure is generally applied only once. Even so, it is natural that one try to reduce the cost of obtaining a complete reference string as much as possible. The recent
work by Wittneben [49], the reference string sampling method, probably can be used to reduce the restructuring cost considerably.

So one of the objectives of this research is to try to do program restructuring based on the sampled reference string instead of the conventional method using a complete reference string, (i.e. a less expensive program restructuring technique) to produce a program with improved locality. When we say less expensive, we mean reduce the cost by 50% or more compared to the traditional program restructuring cost.
CHAPTER II. THE DESIGN OF THE EXPERIMENT

Overview

This chapter discusses the structure and organization of the restructuring experiment. The intent is to give a broad overview, rather than discussing the details. The experiment is a conceptually simple one. The heart of the experiment is designed to quantitatively evaluate the influence of the various sampling rates on the performance improvements resulting from program restructuring. Figure 6 represents schematically the stages of the restructuring experiment which will be performed in this research.

Note that the general scheme presented in Figure 6 is valid for any program; it is both machine-independent and language-independent.

The tool developed for this experiment consists of nine modules: the Monitor, Block-Analyzer, Compute-Block-String, Restructuring-Method, Clustering, Compute-Page-String, Evaluator and Sampling-Method respectively (see Figure 6).

After a brief overview and a discussion of the experiment, each one of the nine modules will be discussed and described in the next few sections.
Input test data (Pascal program)

MONITOR

The program to be restructured
(PASCAL/8000 Compiler)

Address string

Sampling Rate

Block-Analyzer

Block Map

Compute block string

Block string

Sampled block string

Sampled address string
FIGURE 1. The Restructuring Experiment
The experiment can be summarized briefly as follows: first, the program to be restructured is executed with a typical set of input data and at the same time the reference string is collected by the monitor. Then, using the block-map-table, the program Compute-Block-String transforms the reference string to block string. From the block reference string, a restructuring algorithm (which can be any restructuring algorithm we have described in Chapter One) generates a restructuring matrix. This restructuring matrix is then processed by a clustering algorithm (which may be any clustering method we have mentioned before), that produces the page-map-table to be used, either by the programmer or by a program, to reallocate the blocks in virtual memory. This completes the list of the essential stages of any restructuring procedure based on dynamic behavior.

In order to compare the improvement in locality due to restructuring based on sampled reference string, it is sufficient to determine the paging performance of the nonrestructured program, the restructured program based on complete reference string, and that of the restructured programs based on various sampled reference strings and compare them, provided that the parameters of the paging policy in the different executions are equal. (See the solid line, dashed line and the dotted line, respectively,
in Figure 6). Here, the general discussion of the experiment is ended. In the next section, the criteria of the program to be restructured are discussed. Following that, the nine modules mentioned above will be described.

The Program to be Restructured

The selection of the program on which the experiment would be performed was made by taking a number of criteria into account. First of all, the program had to be relatively large and one which was heavily used in order for the results to have greater practical interest. Second, the program had to be representative of an entire class of programs in order to demonstrate that the results of this study are valid for more than just the single program under study. Finally, for practical purposes, the program had to be available and, preferably, one which was written in a high level language, such as Pascal. Of course, the program should not be sensitive to its input data. This is in fact the class of programs for which restructuring is likely to produce the most substantial benefits.

Based on the criteria mentioned above, a list of possible candidates is given in Figure 7.

After carefully examining all the candidates in Figure 7, the PASCAL/8000 compiler was found to best fit the requirements. PASCAL/8000 was one of the first production
compilers for Pascal to become available for the IBM 360/370 range of computers. PASCAL/8000 is now in use in more than 250 sites around the world, and has proved to be a most reliable and versatile compiler.

The compiler is itself written in Pascal and produces standard IBM object modules that may be linked with object modules produced by other language compilers such as FORTRAN and ASSEMBLER.

The language accepted by the compiler conforms closely to the ISO Draft Standard for Pascal. Extensive testing of the compiler using the Sale/Wichman Validation Suite has been performed with excellent results.

The PASCAL/8000 Compiler will be restructured, based on virtual address reference strings that have been generated from either a complete instruction reference string or sampled instruction reference strings created at various sampling rates. This restructured program will be compared on the basis of page fault rate and average working set size to see if the improvement based on the sampled reference strings is as great as restructuring based on the complete reference string. It is hoped that program restructuring based on sampling rate as low as ten percent can result in improvement of efficiency close to those that could be gained from restructuring based on a complete reference string. Further characteristics of the PASCAL/8000 compiler
FIGURE 7. List of Possible Candidates

will be discussed in the next chapter.

In each of the following sections, the input and output of each module in the restructuring experiment are identified, and function of the module is briefly described. The reason why a particular algorithm or policy is selected in each of the following modules will be discussed in the next chapter.
Monitor: (Reference String Collection)

For a number of technical reasons, the Monitor is the most complex and difficult module of the entire experiment. The original PASCAL/8000 compiler runs under IBM VS/370. But, unfortunately, this operating system does not provide any reference string monitoring facility. On the other hand, the Stanford University time-sharing operating system ORVYL/370 does support a limited program tracing facility which we find adequate for our purpose. Thus, the PASCAL/8000 compiler was modified to run under ORVYL/370. However, ORVYL/370 only provides a strictly limited amount of file space that is not sufficient to hold the reference strings we would like to collect.

To solve this file space problem, we ran the monitor program under ORVYL/370 to collect the reference string and at the same time passed the reference string to the program Receiver which was running under IBM VS/370. Receiver accepts the reference string and outputs it to a tape file. A detailed diagram of the Monitor is given in Figure 8.

Block-Analyzer

It is straightforward to monitor the sequence of instruction address but, unfortunately, the monitor does not generate block references. On the other hand, the restructuring algorithm only takes a block reference string
FIGURE 8. The Monitor

as its input. Therefore, some kind of mechanism is required to convert the address reference string to a block reference string. One method of conversion is to subdivide the program into logical blocks, either manually or automatically. This division results in a block-map. Using the block map, an address trace collected from the monitor can be easily transformed into the corresponding block
reference string. The input to the Block-Analyzer is a stream of address reference strings and its output is a block-map as shown in Figure 9. This block analyzer determines the single-entry, single-exit (basic) blocks of the program. A fast algorithm for constructing a block-map from an address reference string was developed. A PL/1 implementation of this algorithm is given in Appendix A.

FIGURE 9. Block-Analyzer

Compute-Block-String

The function of Compute-Block-String is to convert an input address reference string to a block reference string using the block-map. Figure 10 displays a block diagram of this module. A PL/1 implementation of this module is given in Appendix B.
This module takes a block reference string as input, and the output of this module is a restructuring matrix as shown in Figure 11. This matrix actually represents the restructuring graph. The labels of the edge in this restructuring graph quantify, in a way that depends on the restructuring algorithm used, the desirability that the two corresponding nodes be grouped together into the same page. In this research, the restructuring matrix will be generated either by the critical LRU or critical WS method. Both of the above restructuring methods are strategy-dependent.
PL/1 implementations of both the critical LRU and the critical WS restructuring methods are given in Appendices C and D, respectively (both from Ferrari [19]).

A relatively simple and easily implemented clustering algorithm, the Constrained Average Neighbor (CAN), is used in this research. Its function is to group closely related blocks into pages to reduce the page fault rate and/or the mean working set size. This module takes a restructuring matrix as input and produces a page-map as an output.
Restructuring Matrix $\rightarrow$ CAN $\rightarrow$ Page-map

Clustering Method

FIGURE 12. Clustering-Method

Compute-Page-String

This module is almost identical to the module compute block string. A block diagram of this module is given below.

FIGURE 13. Compute-Page-String
Tools for Measurement and Evaluation

In this section, methods for measuring the performance of the restructuring PASCAL/8000 compiler are described.

The general function of an evaluator can be best described by the picture in Figure 13.

Two sets of measurements are used in this research. The first set, dealing with fixed-space policies, is based on the LRU (Least-Recent-Used) stack policy. The page fault rates for all possible memory allocation are measured. A PL/1 implementation of the LRU stack page fault measurement algorithm is given in Appendix E (from Spirn [45]).

The second set of performance measurements, dealing with variable-space strategies, is based on the working set policy. Both the page fault rates and the mean working set sizes are calculated for specific window sizes. A PL/1 implementation of the Denning, one-pass algorithm to measure the mean working set size and page fault rates (from Spirn) is given in Appendix F. For complete detail, please refer to [45].

Sampling-Method

The discussion of sampling methods will be deferred until Chapter IV.
List of Tools Used in This Research

TABLE 3. List of Tools.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Monitor</td>
</tr>
<tr>
<td>2.</td>
<td>Block-Analyzer</td>
</tr>
<tr>
<td>3.</td>
<td>Compute-Block-String</td>
</tr>
<tr>
<td>4.</td>
<td>Restructuring-Method</td>
</tr>
<tr>
<td>5.</td>
<td>Clustering-Method</td>
</tr>
<tr>
<td>6.</td>
<td>Compute-Page-String</td>
</tr>
<tr>
<td>7.</td>
<td>Evaluator</td>
</tr>
<tr>
<td>8.</td>
<td>Sampling-Method</td>
</tr>
</tbody>
</table>

A list of names of all the modules mentioned in this chapter is given in Table 3. Further discussion of these modules can be found in the next chapter.
CHAPTER III. INPUT-DATA SENSITIVITY AND GRANULARITY

The principal objective of the experiment described in this chapter is to determine the influence of the input data on the performance improvements of the PASCAL/8000 compiler resulting from program restructuring. Also, the issue of granularity (i.e. the relationship between block size and program restructuring) is investigated in this chapter.

On most machines, including ours (ITEL AS/6), an instruction reference string is much easier to obtain than a data or complete (instruction and data) reference string. So, it was decided to restructure only the instruction portion of the PASCAL/8000 compiler, leaving the data part untouched.

It is useful to observe that the decision to gather and restructure only instruction blocks does not lessen the significance of these experimental results. In a working-set environment, as mentioned in Ferrari [23], restructuring an instruction block (or a data block) string does not cause the results to be distorted with respect to those which would be obtained by restructuring the corresponding complete reference string, provided that the instructions and data are kept in separated pages.

Having decided to concentrate on the restructuring of instruction blocks, we needed a program whose restructuring would have significant practical interest. As mentioned in
Chapter I, such a program must meet the following five criteria:

1. The program had to be representative of an entire class of programs in order to demonstrate that the results of this study are valid for more than just the single program under study.

2. The program had to be heavily used.

3. The program's instruction space had to be relatively large.

4. The program had to be insensitive to input data.

5. The program's performance had to admit improvement by restructuring techniques.

It is obvious that the PASCAL/8000 compiler satisfies the first and second criteria since, on many systems, the compiler is one of the heavily used system programs. Furthermore, the PASCAL/8000 compiler represents the class of compiler-programs. In terms of the third criterion, the PASCAL/8000 compiler has more than 100K bytes object code and has about 8000 source statements (written in PASCAL/8000). The PASCAL/8000 compiler has 209 procedures (i.e. 209 instruction blocks). For the purpose of this research, PASCAL/8000 compiler is a suitably large program. The experiments required to check conditions four and five are described in the rest of this chapter.
The Design of the Experiment

The performance indices used to measure the effectiveness of the restructuring procedure are the page fault rate and the mean working set size. These two performance measures depend mainly on the following factors:

1. The restructuring algorithm used.
2. The clustering algorithm used.
3. The input data chosen for the restructuring procedure.
4. The memory management policy under which the program is run.
5. The input data used to evaluate the restructured program.
6. The relative sizes of blocks and pages.

The main goal of the experiment required factor five and factor six to be varied.

Ideally, a complete experiment would have varied all of the six factors listed above. However, such an experiment would have been too expensive to perform. Its cost can be reduced only by restricting the number of levels considered for each factor, including factor five and factor six, or by reducing the number of combinations of levels with which the
experiment can be performed. It was decided to concentrate on the impact of factors five and six. Since we were in fact interested in their influence on performance improvement, we also considered two levels for factor one: no restructuring and restructuring. Other factors were kept constant throughout the experiment. This obviously means that our conclusions are strictly valid only for the particular levels selected for each of the fixed factors.

However, it is believed that such a conclusion is likely to have much more general validity. Based on the experience of previously published empirical studies of program restructuring, the only factor likely to have an overriding influence on the improvements due to restructuring is the program to be restructured [25]. If this program is already well-structured (i.e. good locality), only minor improvements can be obtained by restructuring it. A detailed discussion of the selections of these levels of each factor is given in the next section.

Selection of Levels of Each Factor

In this section, selection of levels of each factor of the experiments is discussed. First, we begin with the discussion of restructuring and memory policy. The restructuring procedure introduced in Chapter I contains several components which, in a more detailed study, could be introduced as additional factors. For instance, the
restructuring algorithms, the clustering algorithms, the parameters of these algorithms, the choice and size of the blocks, and the page size could be varied during such experiments. Following the program-tailoring philosophy [22], the selection of the restructuring algorithm to be used was made in conjunction with the choice of the memory management policy.

Since the program restructuring algorithm and memory management policy are closely related to each other, it was decided to use strategy-dependent restructuring methods. For the study described in this chapter, only the critical LRU (Least-Recently-Used) and critical WS (Working-Set) restructuring methods are employed. In both cases, the objective is to minimize the number of page faults generated by the given program under the LRU (Least-Recently-Used) and WS (Working-Set) policies, respectively. Of course, the memory management policy was chosen to be either the LRU (Least-Recently-Used) or the WS (Working-Set) policy.

The assignment of blocks to pages is the function performed by the clustering procedure. The effectiveness of the clustering procedure affects the behavior of the restructured program. A good block assignment scheme (cluster algorithm) can, for example, reduce the number of page faults by placing in the same page program blocks that are frequently used together.
Note, however, that the overall improvement obtained by the restructuring depends also on other factors, such as the restructuring algorithm, the input data used in the restructuring procedure, the memory management policy, the block size and the page size, etc. In the experiments described in this chapter, there is no intention of answering the question, "How close to the optimum is a layout suggested by a restructuring algorithm?" Instead, the purpose is specifically to determine the input data sensitivity and granularity effects of the PASCAL/8000 compiler. Thus, a relatively simple and easy-to-implement clustering algorithm, the Constrained Average Neighbor (CAN) [30], was selected as the constant clustering algorithm used in all of the following experiments. The CAN method is a simple, suboptimal heuristic. In general, the optimal clustering can only be determined by a prohibitively expensive, exhaustive search.

Note that when blocks have been assigned to pages, one problem remains: What to do about page boundaries? Holes in pages can occur if blocks do not fit evenly into pages. For most real programs, two alternatives have been considered. First, blocks are not allowed to cross page boundaries, which may cause empty space within the pages. Second, blocks are packed one after another into the virtual address spaces, leaving no holes but allowing the blocks to
cross page boundaries. For these experiments, we do not allow blocks to cross page boundaries. So the total number of pages increased from 182 to 185 because of this wasted space. In the next paragraph, the problem of how to choose the input data for the restructuring is analyzed.

One means for suppressing the data sensitivity is to select input data which are "representative" of the total program workload. For a compiler, however, it is difficult to determine a "representative" program to serve as the input data. We attempted to choose an input program that contained most of the features of the Pascal language and that was also relatively long. The selected program is a cross reference generator program (XREF). It has about 190 Pascal source statements. A copy of this program is given in Appendix G. The instruction reference trace of this nontrivial program contained more than a hundred thousand references. In the next few paragraphs, the inputs to the PASCAL/8000 compiler for data sensitivity study are discussed.

The most important factor in the data sensitivity analysis is the input program of the experiment. The choice of the levels for this factor was difficult since the number of different inputs to be experimented with had to be small enough as to make the experiment economically feasible. On the other hand, it had to provide us with a sufficiently
wide spectrum of compilations and compilation performances. We thought six levels, corresponding to six different programs, would be economically acceptable and technically sufficient. Therefore, six different Pascal programs were selected. How could we measure the magnitudes of their "differences," and what amount of difference would we consider satisfactory? The simplest criterion was that the inputs should have different application areas and also be syntactically different from each other, so that they would exercise the different parts of the compiler as dissimilarly as possible. Based on this criterion, the following six programs were selected:

1. XREF program - a cross reference generator program, about 190 statements, predominantly character manipulation, sorting and searching.

2. WSM program - a program which computes the page fault rate and the mean working set size of any reference string, about 160 statements, predominantly arithmetic statements.

3. WSM.err program - the same as program WSM but with some errors inserted in the source code.

4. WSM.err30 program - the same as program WSM but with numerous errors inserted in the source code.
5. QUARK program - a queueing simulation program, about 200 source statements, predominantly arithmetic statements.

6. QUARK.err - the same as program QUARK but with errors inserted in the source code.

A comparison of the syntactic differences among these selected programs is given in Table 4.

The above two types of input (the input data chosen for the restructuring procedure and the input data chosen for data sensitivity studies) can lead to some confusion. For ease of understanding and to avoid unnecessary complication and lengthy explanation, two diagrams are presented in Figure 15 and Figure 16, to show the difference between the input data chosen for the restructuring procedure and the input data chosen for data-sensitivity studies. Both of the diagrams are much simplified in order to illustrate the difference between the above two types of input. As shown in Figure 15, the input data chosen for the restructuring procedure are used to restructure the PASCAL/8000 compiler. The input data chosen for data-sensitivity studies are used to measure and evaluate the performance of the restructured PASCAL/8000 compiler. Note that all six of the programs are used for data sensitivity study, but only one program, the XREF, among the six programs is used for restructuring.
FIGURE 15. Program Restructuring
FIGURE 16. Data-Sensitivity Studies

The performance improvement obtainable by restructuring depends also on the relative size of blocks and pages. The larger page sizes have in fact been found to increase the effectiveness of restructuring [28].

The page size used in this research was fixed at 512 bytes per page. Two different block sizes have been considered in this experiment. First, each procedure was
<table>
<thead>
<tr>
<th></th>
<th>XREF</th>
<th>WSM</th>
<th>QUARK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) type-stmts</td>
<td>some</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>2) array</td>
<td>one</td>
<td>many</td>
<td>many</td>
</tr>
<tr>
<td>3) record type</td>
<td>some</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>4) repeat-until</td>
<td>many</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>5) for-loop</td>
<td>none</td>
<td>many</td>
<td>many</td>
</tr>
<tr>
<td>6) if-stmts</td>
<td>many</td>
<td>few</td>
<td>none</td>
</tr>
<tr>
<td>7) case-stmts</td>
<td>none</td>
<td>few</td>
<td>none</td>
</tr>
<tr>
<td>8) character string</td>
<td>many</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>9) integer and real</td>
<td>few</td>
<td>many</td>
<td>many</td>
</tr>
<tr>
<td>10) recursive</td>
<td>very</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>11) link-list (tree)</td>
<td>many</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>12) pointer</td>
<td>many</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>13) comment</td>
<td>none</td>
<td>many</td>
<td>none</td>
</tr>
<tr>
<td>14) i/o</td>
<td>few</td>
<td>few</td>
<td>many</td>
</tr>
</tbody>
</table>

considered as a separate block. There is a total of 209 procedure blocks in the PASCAL/8000 compiler. The minimum block size is 32 bytes. The maximum block size is 3232 bytes. Most of the blocks are small except a few of very large size. Second, we use basic-block. Basic blocks are considerably smaller than procedure blocks in the PASCAL/8000 compiler. There is a total of 605 basic blocks in the PASCAL/8000 compiler. The minimum block size is five bytes. The maximum block size is 512 bytes. And of course most of them are small.

---

1 The word basic-block is used for the executable object code. A basic block is a sequence of instructions, containing, at the most, one branch point (at the end) and one entry point (at the beginning).
There are several disadvantages which result if the basic-block is used as the unit of restructuring. They are:

1. The basic blocks are not relocatable blocks (i.e., since program instructions are stored sequentially in a fixed order in the computer's memory). After clustering, the relative address position relation between instructions is altered. Therefore, a special program is required to modify the existing machine code and rebuild this relative address relationship between instructions.

2. Debugging the restructured program from its machine dump becomes very difficult or impossible because the machine code was internally modified by the special program described above. The code sequence and code relative position between instructions do not make any sense to the user anymore.

3. The restructuring procedure becomes very expensive, because the number of basic blocks is considerably more than the number of procedure blocks. Hence, during the restructuring process, a huge matrix, which occupies a substantial amount of memory space, is required.

4. It was initially suspected that the basic-block would be more sensitive to input data than procedure-block.

5. A special compiler is needed to produce the basic-block map.
Therefore, unless the basic-block-level has significant performance improvement over the procedure-block-level, its use will not be justified.

The Experiments

TABLE 5. Names of the twelve reference strings.

<table>
<thead>
<tr>
<th>Procedure-Block</th>
<th>Basic-Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. XREF.proc</td>
<td>1. XREF.blk</td>
</tr>
<tr>
<td>2. WSM.proc</td>
<td>2. WSM.blk</td>
</tr>
<tr>
<td>3. WSM.err.proc</td>
<td>3. WSM.err.blk</td>
</tr>
<tr>
<td>4. WSM.err30.proc</td>
<td>4. WSM.err30.blk</td>
</tr>
<tr>
<td>5. QUARK.proc</td>
<td>5. QUARK.blk</td>
</tr>
<tr>
<td>6. QUARK.err.proc</td>
<td>6. QUARK.err.blk</td>
</tr>
</tbody>
</table>

Two sets of experiments have been designed and executed to analyze the performance improvements of the restructuring process as measured by the page fault rate and mean working set size. The experiments were performed on reference strings generated by the PASCAL/8000 compiler while
compiling the six different programs described previously. These six programs are named XREF, WSM, WSM.err, WSM.err30, QUARK and QUARK.err as mentioned before.

Note again that the above six reference strings are instruction reference strings only. For restructuring purposes, block reference strings are needed instead of instruction reference strings. So the above six full instruction reference strings are mapped into six procedure-block reference strings and six basic-block reference strings by using a procedure-block-map and a basic-block-map.² From now on, we name these twelve reference strings as in Table 5.

The next two sections report on the results of the restructuring experiments performed under a fixed-spaced strategy, Least-Recent-Used (LRU) and variable-spaced strategy, Working-Set (WS), on both of the procedure-block reference strings and the basic-block reference strings. The basic structure of these two sets of experiment is as follows:

² Both of the procedure-block-map and the basic-block-map could be generated by the compiler itself. Unfortunately, the PASCAL/8000 compiler does not provide these two types of block maps. The basic-block Analyzer program, as mentioned in Chapter II, was developed to get the basic-block-map. The procedure map was generated internally by the PASCAL/8000 compiler for its own procedures.
1. A restructuring matrix is constructed from the XREF block reference string.

2. The clustering procedure "CAN", based on the above restructuring matrix, is used to partition the blocks into pages.

3. Then, the nonrestructured version and the restructured ones were run in the simulated LRU (Least-Recently-Used) and WS (Working-Set) environment, and the values of indices such as the total number of page faults and the mean working set size were computed.

First Set of Experiments

In order to determine the data sensitive and granularity effects, two sets of experiments were performed. The first set of experiments uses procedure-block reference strings. The second set of experiments uses the basic-block reference strings. For each of these experiments, the experiments performed, the measurements taken, the comparisons made, and the conclusions derived will be presented.

In each set of experiments, the measurements obtained will be presented in two parts. In the first part, dealing with the fixed-space strategy - the LRU (Least-Recently-Used) measurement, the results are presented in terms of
percentage of performance improvement of page fault rate. The second part, involving the variable-space strategy, the WS (Working-Set) measurement, includes the page fault rate and the mean working set size comparisons.

**LRU Experiments and Results**

The purpose of this section is to report the paging performances under the LRU environment. The evaluation consists of two basic parts. First, the paging performances produced by different reference strings are related and contrasted with one another. Second, the improvements in paging performance
produced by the restructuring methods are computed and evaluated within the desired range of memory size. Table 6 gave a listing of those parameters used in the experiment. Note, the performance index in this section is the page fault rate only since LRU (Least-Recent-Used) is a fixed-spaced policy.

Performance improvement indices of the restructured programs over the original one (the nonrestructured version) of input program XREF, WSM, WSM.err30 and QUARK are plotted in Figure 17. Those for the other two input programs were similar to those presented in this figure and have therefore been omitted.

The results displayed in Figure 17, in terms of page fault percentage of improvement, are very encouraging. All of them are closed to the best curve XREF.proc. And exhibit a similar and stable behavior.

In this case, we can conclude that the PASCAL/8000 compiler is not sensitive to input data. It should also be observed in Figure 17 that a considerable margin of performance improvement was obtained in the memory size range between \( M=30 \) to \( M=60 \). The improvement was from approximately 30% to 60%. So it seems that the compiler's performance can be significantly improved by program restructuring.
Figure 17. LRU Result: Time (Procedure-Block) vs. % Improvement (Memory Size)
TABLE 7. Parameters of WS Experiments.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Restructuring algorithm</td>
<td>CWS</td>
</tr>
<tr>
<td>2) Clustering algorithm</td>
<td>CAN</td>
</tr>
<tr>
<td>3) Program chosen for restructuring procedure</td>
<td>XREP</td>
</tr>
<tr>
<td>4) Memory management policy</td>
<td>WS</td>
</tr>
<tr>
<td>5) Input-data (programs)</td>
<td>XREP, WSM, WSM.err, WSM.err30, QUARK, QUARK.err</td>
</tr>
<tr>
<td>6) Block type</td>
<td>Procedure-block</td>
</tr>
<tr>
<td>7) Page size</td>
<td>512 bytes</td>
</tr>
</tbody>
</table>

Working Set Experiments and Results

An experimental study of the performance of the PASCAL/8000 compiler under the WS (Working-Set) environment is reported in this section. We would like to see how the performance of the compiler is affected by changes in the data which are input to the compiler.

The measurement of performance index in this experiment is in terms of page fault rate, mean working set size and also time-space product (i.e. page fault rate times mean working set size). Table 7 shows the parameters of this experiment.
FIGURE 18. WS RESULT: TIME (PROCEDURE-BLOCK)

% OF IMPROVEMENT (TIME)

0.00 2.00 4.00 6.00 8.00 10.00 12.00

(x10^4) WINDOW SIZE

-2.00 -1.00 0.00 1.00 2.00 3.00

O : XREF.PROC
△ : WSM.PROC
△ : WSMERR30.PROC
O : QUARK.PROC
FIGURE 19. WS RESULT: SPACE (PROCEDURE-BLOCK)
FIGURE 20. WS RESULT: TIME-SPACE (PROCEDURE-BLOCK) 

% OF IMPROVEMENT (TIME-SPACE) 

0.00 2.00 4.00 6.00 8.00 10.00 12.00 WINDOW SIZE 

XREF.PROC 
WSM.PROC 
HSMERR30.PROC 
QUARK.PROC
In Figure 18, it is easy to see that the page fault rate improvement of input programs XREF, WSM and WSM.err are positive. But the input program QUARK shows a negative result.

Figure 19 shows the mean working set size improvement. All the curves show positive results. Figure 20 shows the time-space product result. In this figure, the performance improvements are between 15\% to 30\%.

By comparing LRU (Least-Recently-Used) results and WS (Working-Set) results, we found that the LRU (Least-Recently-Used) result is better than WS (Working-Set). This comes from the fact that the WS (Working-Set) strategy continuously adapts to the demand of the reference string locality more easily than the LRU (Least-Recently-Used) policy. Thus, it is more difficult to improve the WS (Working-Set) performance. However, in both cases useful improvement has been obtained by the restructuring process. Furthermore, the improvements for the different programs showed no evidence of data sensitivity.

Second Set of Experiments

The second set of experiments is identical to the first set of experiments except that the basic-block reference strings are used rather than procedure-block reference strings. The purpose of this set of experiments is to
investigate the problem of granularity effects and to further study the data sensitivity effects.

TABLE 8. Parameters of LRU Experiments.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Restructuring algorithm</td>
<td>CLRU</td>
</tr>
<tr>
<td>2) Clustering algorithm</td>
<td>CAN</td>
</tr>
<tr>
<td>3) Program chosen for restructuring procedure</td>
<td>XREP</td>
</tr>
<tr>
<td>4) Memory management policy</td>
<td>LRU</td>
</tr>
<tr>
<td>5) Input-data (programs)</td>
<td>XREF, WSM, WSM.err, WSM.err30, QUARK, QUARK.err</td>
</tr>
<tr>
<td>6) Block type</td>
<td>Basic-block</td>
</tr>
<tr>
<td>7) Page size</td>
<td>512 bytes</td>
</tr>
</tbody>
</table>

**LRU Experiments and Results** Table 8 shows the parameters of this experiment. As can be seen in Figure 21, the results do not show any evidence of data sensitivity. However, by comparing with Figure 18 the performance results are not better than the LRU (Least-Recently-Used) result on the procedure-block-level.
FIGURE 21. LRU RESULT: TIME (BASIC-BLOCK)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Restructuring algorithm</td>
<td>CWS</td>
</tr>
<tr>
<td>2) Clustering algorithm</td>
<td>CAN</td>
</tr>
<tr>
<td>3) Program chosen for restructuring procedure</td>
<td>XREF, WSH, WSM.err, WSM.err30, QUARK, QUARK.err</td>
</tr>
<tr>
<td>4) Memory management policy</td>
<td>WS</td>
</tr>
<tr>
<td>5) Input-data (programs)</td>
<td>XREF, WSH, WSM.err, WSM.err30, QUARK, QUARK.err</td>
</tr>
<tr>
<td>6) Block type</td>
<td>Basic-block</td>
</tr>
<tr>
<td>7) Page size</td>
<td>512 bytes</td>
</tr>
</tbody>
</table>

Working Set Experiments and Results

Table 9 displays the parameters of this experiment. As is shown in Figures 23, 24 and 25, the results again are insensitive to the input data and, by comparing with Figures 18, 19 and 20, are not better than the WS (Working-Set) result on procedure level.

One possible explanation of this phenomenon is that the procedure-block is a natural choice as the unit for restructuring. When a programmer is designing or coding a program, there is a strong tendency to put strongly related material into the same procedure. Another reason is due to
the one-entry and one-exit constructs used in top-down/bottom-up structured programming techniques. So the localities are determined naturally within each procedure. Even though procedure blocks can be further broken down into smaller rearrangeable basic blocks, this only results in disturbing their natural locality instead of improving it. From this evidence, we conclude that the block size is related to improvement as generally shown in Figure 22.

![Figure 22. Relation between block size and restructuring](image)

That is, the performance will decrease if either the blocks are too small or the blocks are too large.
FIGURE 23. WS RESULT: TIME (BASIC-BLOCK)

% OF IMPROVEMENT (TIME)

0.00 2.00 4.00 6.00 8.00 10.00 12.00 (x10^4) WINDOW SIZE

XREF.BLK
WSM.BLK
WSMERR30.BLK
QUARK.BLK
FIGURE 24. NS RESULT: SPACE (BASIC-BLOCK)
FIGURE 25. HS RESULT: TIME-SPACE (BASIC-BLOCK)
Summary of Results

The results of our experiments, however, do not represent the full potential of the restructuring procedure used, since substantially greater improvements would presumably be obtained if the data portion of the program were also restructured.

It is believed that these two sets of experiments are sufficient to establish the following conclusions:

1. The nonrestructured program is normally outperformed by its restructured version using the CLLU (Critical Least-Recent-Used) and CWS (Critical Working-Set) methods; the relative improvement of the number of page faults is about 30% to 60% in LRU (Least-Recently-Used) and 15% to 25% in WS. The mean working size is about 10% better.

2. As should be expected, the LRU (Least-Recently-Used) performance can be improved more than the WS (Working-Set) performance. Program restructuring is more useful for a more rigid paging policy (such as LRU) than a more robust paging policy (such as WS). In situations where only LRU (Least-Recently-Used) type policies are provided, this implies that program restructuring is more important and necessary.

3. It was expected that the basic-block-level would exhibit stronger performance improvements, in terms of mean
working set size and page fault rate, than the procedure-block-level. However, the results of experiments in this chapter show quite the contrary to be true. In addition, the basic-block-level also displays greater sensitivity to input data. So from now on, all the experiments in this research were performed on the procedure-block-level.

The influence of the input data on the performance improvements resulting from program restructuring is found to be nonsignificant for the PASCAL/8000 compiler, especially in the procedure-block level. It is speculated that such a conclusion is likely to have much more general validity.

By comparing the results of the experiments under different input programs and two different paging environments, the results of this chapter support the claim of Hatfield and Gerald that, "Many commonly used programs are rather insensitive to input data."

Based on the results of this chapter, it is suggested that procedure size units be used in program restructuring. Especially in the situation like LRU (Least-Recent-Used), tracing the sequence of execution of the procedures in a program is conceptually straightforward: a trace routine which records the block number whenever a procedure is entered. This tracing is much cheaper compared to a full reference string collection in terms both of the time and
space expended. Unfortunately, this simple method only works in a particular situation, such as the LRU (Least-Recent-Used) environment.
CHAPTER IV. SAMPLED PROGRAM RESTRUCTURING METHODS

In the last chapter, experimental results indicated that program restructuring based on complete reference strings may significantly improve the behavior of programs in virtual memory systems. As mentioned before, one of the objections to a posteriori restructuring methods is that they are costly to use due to the high overhead in capturing and analyzing a complete reference string. Therefore, the reference string sampling method, intended to reduce this cost, is proposed and studied in this chapter.

The principal objective of the experiment described in this chapter is to determine the influence of different sampling rates on the performance improvements of the PASCAL/8000 compiler resulting from program restructuring. Also, the problem of sampling method (i.e. the relationship between sampling patterns and program restructuring) is investigated in this chapter.

The Design of the Experiments

Selection of Levels of Each Factor

The experiments in this chapter are very similar to those presented in the last chapter. For completeness, a summary of the basic factors used in these experiments is given in Table 10. The experiments in this chapter differ
TABLE 10. Parameters of Experiments.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Restructuring algorithm</td>
</tr>
<tr>
<td>2</td>
<td>Clustering algorithm</td>
</tr>
<tr>
<td>3</td>
<td>Program chosen for restructuring procedure</td>
</tr>
<tr>
<td>4</td>
<td>Memory management policy</td>
</tr>
<tr>
<td>5</td>
<td>Input-data (programs)</td>
</tr>
<tr>
<td>6</td>
<td>Block type</td>
</tr>
<tr>
<td>7</td>
<td>Page size</td>
</tr>
</tbody>
</table>

from the earlier experiments in two ways. First, program restructuring is based on a sampled reference string. Second, the experiments are only performed at the procedure-block level.

The main goal of the experiments described in this chapter required two factors to be varied: the sampling rate and the sampling pattern. Since there is an infinite number of different parameters that can be used for these factors, careful selection of the number of experiments to be performed is required. In the next section, a detailed discussion of this problem is presented.
**Sampling Methods**

The sampling methods define which portions of a complete reference string are monitored and which portions are skipped. These methods are fully described in [49]. Here, only a short review is given.

The general function of the sampling method is described by the following block diagram:

```
   (K, S)
  /     |
↓      ↓
Complete reference string → SAMPLING → Sampled reference string
```

**FIGURE 26. Sampling Method**

A complete reference string is input to the sampling algorithm. Some of the references of this complete reference string are selected to form a sampled reference string. The rest of the references (address) which do not appear in the output string are simply ignored. These latter references are the skipped references. Note that the sampling algorithm depends on two parameters: K and S.
The two parameters \((K, S)\) of the sampling method divide a complete reference string into intervals of consecutive references. Each interval has the same size. The size of the interval is determined by the sum of the two parameters of the sampling method. This is shown in Figure 27.

Each interval is divided into two parts, the monitored part and the skipped part, by the parameter \((K, S)\). The first parameter \(K\) (keep) is the number of references selected (i.e. put in the sampled reference string) from the monitored part of each interval. The second parameter, referred to as \(S\) (skip), is the number of references to be skipped in the skipped part of the interval. Figure 27 shows this division of the interval by the parameters. The sampling rate is the percentage of references monitored over the length of the total reference string. That is, the sampling rate is \((K / (K + S)) \times 100\%\). For example:
if K=10 and S=90
then the sampling rate is calculated as follow:

\[ SR = \frac{10}{10+90} \times 100\% \]
\[ = \frac{10}{100} \times 100\% \]
\[ = 10\% \]

In the next section, the question of sampling patterns is examined, and following that the question of sampling rates is discussed.

**Sampling Patterns**

First, let us examine the structure of a procedure-block reference string which was used to do program restructuring. A typical procedure block reference string would look like this:

...b1 b1 b1... b1 b2 b2... b2 b3 b3... b3...

Because the procedure-block is considerably larger than the basic-block, it is expected that each block occurrence would appear consecutively for a long period in the procedure-block reference string. A number of real procedure-block reference strings was examined, and this rereferencing pattern was observed to be true. This characteristic of procedure-block-level reference string will determine the sampling pattern as shown in the following example.

Consider the piece of a procedure-block reference string shown in Figure 28. Suppose we hold the sampling rate
constant at twenty percent while varying both the K and the S parameters. The examples in Figure 28(a)-(c) illustrate the effect of different sampling patterns.

\[
\begin{align*}
\text{(a): } & \quad \text{If sampling parameters are } K=1 \text{ and } S=4 \\
& \quad \text{then the resulting sampled reference string is } b_1 b_2 b_3 b_4 \ldots \\
\text{(b): } & \quad \text{If sampling parameters are } K=2 \text{ and } S=8 \\
& \quad \text{then the resulting sampled reference string is } b_1 b_1 b_3 b_3 \ldots \\
\text{(c): } & \quad \text{If sampling parameters are } K=3 \text{ and } S=12 \\
& \quad \text{then the resulting sampled reference string is } b_1 b_1 b_1 b_4 \ldots 
\end{align*}
\]

FIGURE 28. Effect of Different Sampling Parameters

It can be observed from these three examples that as the value of K increases, the probability of missing some of the referenced blocks is also increased sharply. The lowest value of K is most likely to preserve the structural characteristics of the original complete reference string. Also note that, for \( k > 1 \), the consecutive portion of the reference string contains no information for restructuring purposes.
From the above examples and analysis, a $K=1$ sampling pattern will be used in the sampling experiments.

**Sampling Rate**  The objective of this research is to achieve the lowest overhead by sampling the fewest number of references. On the other hand, we want to keep the performance improvement of the restructuring program, based on this sampled reference string, as close as possible to the one achieved from a complete reference string. The sampling rates of one, five, ten, and twenty percent are used in each of the following experiments. Since we fixed $K=1$, the sampling parameters will be as follows:

1. $(K=1, S=99)$ -------------- 1%
2. $(K=1, S=19)$ -------------- 5%
3. $(K=1, S=9)$ -------------- 10%
4. $(K=1, S=4)$ -------------- 20%

These sampling rates were selected arbitrarily, but we believe that they are sufficient to achieve our purpose in understanding the effects of sampling on program restructuring.
The Experiments

Two sets of experiments have been designed and executed to analyze the performance improvements of the restructuring process based on sampled reference strings as measured by the page fault rate and mean working set size. In the first set of experiments, dealing with fixed-space policy, the LRU (Least-Recent-Used) is employed. The second set of experiments, using a variable-space policy, is based on the working set strategy. In both sets of experiments, the page fault rate is calculated and, in the second set, the mean working set size is also calculated. For each of these experiments, the experiments performed, the measurements taken, the comparisons made, and the conclusions derived will be presented.

LRU Experiments and Results

Table 11 gives a listing of the parameters used in the experiment. Note, the measurement of performance index in this section is in terms of page fault rate only since LRU (Least-Recent-Used) is a fixed-spaced policy.

The purpose of this section is to report the paging performance under the LRU environment. The evaluation consists of two basic parts. First, the paging performances
TABLE 11. Parameters of LRU Experiments.

<table>
<thead>
<tr>
<th>Parameter Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Restructuring algorithm</td>
<td>CIBD</td>
</tr>
<tr>
<td>2) Clustering algorithm</td>
<td>CAN</td>
</tr>
<tr>
<td>3) Program chosen for restructuring procedure</td>
<td>XREF</td>
</tr>
<tr>
<td>4) Memory management policy</td>
<td>LRU</td>
</tr>
<tr>
<td>5) Input-data (programs)</td>
<td>WSM</td>
</tr>
<tr>
<td>6) Block type</td>
<td>Procedure-block</td>
</tr>
<tr>
<td>7) Page size</td>
<td>512 bytes</td>
</tr>
<tr>
<td>8) Sampling methods</td>
<td>1%-(K=1, S=99)</td>
</tr>
<tr>
<td></td>
<td>5%-(K=1, S=19)</td>
</tr>
<tr>
<td></td>
<td>10%-(K=1, S=9)</td>
</tr>
<tr>
<td></td>
<td>20%-(K=1, S=4)</td>
</tr>
</tbody>
</table>

produced by different sampled reference strings are related and constrained with one another. Second, the improvements in paging performance produced by the restructuring methods are computed and evaluated within the desired range of memory size.

Performance improvement indices of the restructured programs, based on different sampling reference strings, over the original one (the nonrestructured version) of input program WSM are plotted in Figure 29.
The results displayed in Figure 29, in terms of page fault percentage of improvement, are very encouraging. The curves with sampling rate 5%, 10% and 20% are all close to the one without sampling, and also exhibit a similar and stable behavior. The curve with 1% sampling rate seems a little bit apart from the rest of the curves but still shows stable behavior. Note that the curve with sampling rate 5% (i.e. \(K=1, S=19\)) is almost identical to the one for the complete reference string.

It should also be observed in Figure 29, that a considerable margin of performance improvement was obtained in the memory size range between \(M=30\) to \(M=60\) with sampling rate at 5%, 10% and 20%.

The same experiments were repeated for input data (procedure reference string) XREF, WSM.err30 and QUARK. The results of these three different input data are shown in Figure 30, Figure 31 and Figure 32, respectively. All of these figures show the same results as in Figure 29. So, it seems that the compiler's performance can be significantly improved by program restructuring based on reference strings from even very low sampling rates, such as 5%. From this data, we can conclude that the effect of a program restructuring based on at least a 5% sampled rate is as good as that resulting from restructuring based on a complete reference string.
FIGURE 30. LAU RESULT: TIME (XREF_PROC)

% IMPROVEMENT 5,000 6,000

MEMORY SIZE

O = NO SAMPLING
+ = K154
@ = K159
* = K1519
\= = K1529
FIGURE 31. LAV RESULT: TIME (WSMERA30.PROC)

Legend:
- O: NO SAMPLING
- +: K154
- @: K159
- *: K1519
- A: K1599

% OF IMPROVEMENT (TIME) x 10^1

MEMORY SIZE (x10^1)

0.00 2.00 4.00 6.00 8.00 10.00 12.00
FIGURE 32. LAU RESULT: TIME (QUARK.PROC)
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Restructuring algorithm</td>
<td>CHS</td>
</tr>
<tr>
<td>2) Clustering algorithm</td>
<td>CAN</td>
</tr>
<tr>
<td>3) Program chosen for restructuring procedure</td>
<td>XREP</td>
</tr>
<tr>
<td>4) Memory management policy</td>
<td>WS</td>
</tr>
<tr>
<td>5) Input-data (programs)</td>
<td>WSM</td>
</tr>
<tr>
<td>6) Block type</td>
<td>Procedure-block</td>
</tr>
<tr>
<td>7) Page size</td>
<td>512 bytes</td>
</tr>
<tr>
<td>8) Sampling methods</td>
<td>1%((K=1,S=99))</td>
</tr>
<tr>
<td></td>
<td>5%((K=1,S=19))</td>
</tr>
<tr>
<td></td>
<td>10%((K=1,S=9))</td>
</tr>
<tr>
<td></td>
<td>20%((K=1,S=4))</td>
</tr>
</tbody>
</table>
Working Set Experiments and Results

An experimental study of the performance of the PASCAL/8000 compiler under the WS (Working-Set) environment is reported in this section. The purpose of this experiment was to determine how the Working Set performance of the compiler is affected by restructuring based on different sampled reference strings and also to find the lowest sampling rate for practical program restructuring.

The performance indices in this experiment are the page fault rate, mean working set size and the virtual time-space product (i.e. page fault rate times mean working set size). Table 12 shows the parameters of this experiment.

In Figure 33, it is easy to observe that the page fault rates of the program WSM which has been restructuring based on different sampling rates are very stable. In particular, the curve with sampling rate 5% shows a promising result. It indicates that program restructuring using only 5% of the reference string is as good as program restructuring using the entire reference string. Figures 34, 35 and 36 show the results for input programs XREF, WSM.err30 and QUARK, respectively, by repeating the same experiment. All of them show the same results.

The second performance index used in the working set experiments is the mean working set size. The mean working
set size results are shown in Figures 37 through 40. In
Figure 37, the curve with sampling rate 1% is apart from the
5%, 10% and 20% sampling curves. The performance
improvement of program restructuring based on 5% sampled
reference string is not as good as the one based on complete
reference string, but it is very close. Figures 38, 39 and
40 are the results of input programs XREF, WSM.err30 and
QUARK, respectively, by repeating the same experiment.

For fixed-space memory management policies, such as LRU
(Least-Recent-Used), the page fault rate index is a
reasonable way to measure and compare different policies.
However, that is not the case for variable-space memory
management policies. It would appear reasonable to compare
the time-space product of different variable-space memory
management policies. Figures 41, 42, 43 and 44 show the
time-space product results for input programs WSM, XREF,
WSM.err30 and QUARK, respectively. As you can see in these
figures, the time-space product performance improvements are
about 20% in our experiments.

In all of these WS (Working-Set) measurements, the
sampling rates of 1% and 5% appear to delimit the range of
practical sampling rates for restructuring purposes.
FIGURE 33. WS RESULT: TIME (WSM.PROC)
FIGURE 32. HS RESULT: TIME (XREF. PROC)

% OF IMPROVEMENT (TIME), \( \times 10^1 \)

-2.00
-1.00
0.00
1.00
2.00
3.00

0.00 2.00 4.00 6.00 8.00 10.00 12.00 (x10^1) WINDOW SIZE

O : NO SAMPLING
+ : K154
x : K159
* : K1519
A : K1599
FIGURE 36. WS RESULT: TIME (QUARK.PROC)

% OF IMPROVEMENT (TIME)

0.00 1.00 2.00 3.00

0.00 2.00 4.00 6.00 8.00 10.00 12.00

WINDOW SIZE

O : NO SAMPLING
+ : K194
- : K159
* : K1919
X : K1999
FIGURE 37. WS RESULT: SPACE (WSM.PROC)

% IMPROVEMENT (SPACE) (x10^3)

0.00  2.00  4.00  6.00  8.00  10.00  12.00

WINDOW SIZE

O : NO SAMPLING
+ : K154
* : K159
* : K1519
A : K1599
FIGURE 38. NS RESULT: SPACE (XREF.PROC)

% OF IMPROVEMENT (SPACE) (x10^4)

O : NO SAMPLING
+ : K154
* : K159
X : K1519
A : K1599

WINDOW SIZE (x10^4)
FIGURE 39. WS RESULT: SPACE (WSMERA3D.PROC)
FIGURE 40. WS RESULT: SPACE (QUARK.PROC)

- O: NO SAMPLING
- +: K1S4
- : K1S9
- *: K1S19
- A: K1S99

% OF IMPROVEMENT (SPACE) (x10^1)

WINDOW SIZE (x10^4)
FIGURE 41. WS RESULT: TIME-SPACE (HSM.PROC)
FIGURE 42. WS RESULT: TIME-SPACE (XREF.PROC) x10^6 WINDOW SIZE
FIGURE 43. WS RESULT: TIME-SPACE (WSMERA30.PROC)
FIGURE 44. WS RESULT: TIME-SPACE (QUARK.PRO)
Summary of Results

It is believed that the LRU and Working Set experiments are sufficient to establish the following conclusions:

1. The performance of a restructured program based on sampled reference string is just as good as its restructured version based on complete reference string under both the fixed-spaced LRU (Least-Recent-Used) strategy and variable-spaced working set strategy.

2. The lowest sampling rate suitable for program restructuring seems to be in the range of 1% to 5%.

3. The sampling method \((k=1, S=19)\), five percent sampling rate, is highly recommended for practical program restructuring purposes.

4. The influence of the input data on the performance improvements resulting from program restructuring based on sampled reference strings is insignificant for the PASCAL/8000 compiler.

The results of this chapter, comparing different sampling rates and two different paging environments, indicate that the cost of program restructuring can be substantially reduced by reference string sampling. It is suggested that for program restructuring purposes, it is not necessary to use a complete reference string. Hence, the
cost can be considerably reduced in terms of both the space and time expended.
CHAPTER V. STRATEGY-INDEPENDENT RESTRUCTURING ALGORITHMS

Introduction

As mentioned in Chapter I, the a posteriori restructuring algorithms can be divided into two categories: strategy-dependent algorithms and strategy-independent algorithms. Restructuring algorithms in the first category assume a particular target memory management strategy and take advantage of the known operation of that strategy, while those in the second category are oriented toward any specific memory management strategy. It is expected that the restructuring algorithms in the first category are more effective than those in the second category since significant additional information is available to the restructuring algorithms in the first category. This intuitive expectation has been confirmed by several experiments [20, 22]. However, with strategy-dependent restructuring, programs should be restructured for each virtual memory system in which they are executed. Therefore, in certain cases, it would be desirable to use a restructuring algorithm in the second category, provided that its improvement was not unacceptable when compared to that of the algorithms in the first category. Such cases arise, for example, when little is known about the memory management strategy to be used or when a program is to be
executed on a number of different systems with different memory management policies.

This chapter presents and evaluates two new approaches to strategy-independent restructuring algorithms. The first approach is based on the concept of locality. The second approach is based on the critical reference principle. The paging activities of the PASCAL/8000 compiler restructured with the new algorithms under LRU (Least-Recent-Used) and WS (Working-Set) memory management strategies are simulated to evaluate the new algorithm.

The basis of the first restructuring algorithm is the locality of a reference string. We assume that references are made at the discrete time instants $1, 2, 3, \ldots, t, \ldots$, to the blocks $b(1), b(2), b(3), \ldots b(t), \ldots$, respectively. In this section, blocks are the units of memory management. Therefore, the Working-Set is defined in terms of blocks rather than of pages. The term 'locality' has been used to denote that subset of a program's blocks which are referenced during a particular phase of its execution. A program's behavior can be characterized in terms of its residence in localities of various sizes and lifetimes, and the transitions between these localities.

Denning [16] suggested in 1972 that program execution could be modeled as a sequence of pairs:

$$[L(1), t(1)], [L(2), t(2)], \ldots, [L(i), t(i)], \ldots$$
where \( L(i) \) is the set of blocks or information units which are referenced in the \( i \)th 'locality', which exists for time \( t(i) \). This model can be best depicted by the following diagram:

The execution of a program proceeds as residencies in a sequence of 'major phases' in which only some subset of its information blocks or pages are referenced. In the above diagram, phase \( i \) only involves references to the blocks of the set \( L(i) \), and the program remains in this phase of its activity for time \( t(i) \). When the program leaves this 'stable' phase of behavior, it may display some transitional activity for time \( t \), referencing the set \( s \) of blocks, before entering its next phase of stable behavior. In order to partition the reference string into distinctive major phases, Madison and Batson [34] introduced the concept of
the Bounded Locality Interval (BLI) defined as follows:

Definition: An activity set \([34]\) at time \(t\), \(A_k(t)\), is defined as any set of \(k\) blocks, all members of which have been re-referenced since the set was formed.

Definition: The lifetime \([34]\) of an activity set is defined as the interval between the time the activity set is established and its termination time.

Definition: A Bounded Locality Interval (BLI) \([34]\) is defined as the pair consisting of an activity set and its lifetime.

The Bounded Locality Interval model contains no parameters, such as window set size or stack depth, so that localities are determined naturally (i.e. without reference to a particular memory management policy). It can also be seen that the BLI definition establishes a hierarchy of localities. Figure 45 shows an example reference string and the corresponding hierarchy of BLI's. These hierarchies of BLI's reflect the nested and iterative structures which exist in many real programs.

A locality in this context is a proper subset of a program's blocks defined by the BLI model. This model also allows one to model dynamic program behavior from the locality's point of view.

Spirn and Denning \([46]\) observed that the Least-Recent-Used (LRU) stack may be used to construct sets of blocks

which possess many of the characteristics associated with a locality. The LRU stack can be represented as the ordered vector

\[ L'(t) = (L(1,t), L(2,t), \ldots, L(n,t)) \]

where \( n \) is the number of blocks in the program and \( L(i,t) \) is the block identifier for the \( i \)th most-recent-referenced block at time \( t \). For example, the corresponding LRU stack for the reference string in Figure 45 at time \( t \) will have the configuration shown in Figure 46.

If we cut the stack at any position \( i \), then the topmost \( i \) blocks in the stack \( S(i,t) = (L(1,t), L(2,t), \ldots, L(i,t)) \) are the \( i \) most recently referenced blocks at time \( t \). Such a set does in some sense correspond to the usual concept of a
locality, and furthermore, since the LRU stack can be cut at any point, it defines a hierarchy of localities as illustrated in Figure 47.

The problem with this approach is that, at each instant of time, the number of such localities can have any value in the range 1 and n. All localities are significant, but some locality sets will more accurately describe the program's behavior than others. For example, the block name on the top of the stack may have been in that position for a long period of time. The LRU stack simply does not contain enough information to allow one to select certain of these locality sets as being more distinctive than others. Some mechanism is needed for selecting those localities which
FIGURE 47. Hierarchy of LRU stack localities at time t=30

correspond most closely to our intuitive notion of a locality from the hierarchy defined by the LRU stack.

To select localities from the hierarchy defined by the LRU stack, Madison and Batson [34] introduced the Extended LRU stack mechanism. Two new stacks are added to the normal LRU stack:

\[ V'(t) = (V(1,t), V(2,t), \ldots, V(n,t)) \]

\[ T'(t) = (T(1,t), T(2,t), \ldots, T(n,t)) \]

where \( n \) is the number of blocks in the program. \( V(i,t) \) is the time at which the block in the \( i \)th stack position was last referenced, and \( T(i,t) \) is the time at which a reference was last made to a stack position greater than \( i \). Thus, \( T(i,t) \) corresponds to the formation time of the set \( S(i,t) \).

The algorithm for updating the Extended LRU stack as time progresses from time \( t \) to \( t+1 \) is given in Table 13.
TABLE 13. An algorithm for updating the Extended LRU stack.

<table>
<thead>
<tr>
<th>If the reference at time t+1 is to the block in L(i,t), then:</th>
</tr>
</thead>
</table>
| \[
L(1,t+1) := L(i,t); \\
V(1,t+1) := t+1; \\
\text{For } j := 1 \text{ to } i-1 \text{ do } \\
\begin{align*}
L(j+1,t+1) &= L(j,t); \\
V(j+1,t+1) &= V(j,t); \\
T(j,t+1) &= t+1; \\
\text{End;}
\end{align*}
|-------------------------------------------------------------|

For the ith member of the LRU hierarchy of sets, \( T(i,t) \) is the time of the sets formation, and \( v(i,t) \) is the time of the most recent reference to its least recently referenced member. Therefore, we can redefine an activity set at time \( t \), \( A(i,t) \), as any \( S(i,t) \) for which \( V(i,t) > T(i,t) \). Figure 48 shows the state of the Extended LRU stack at time \( t=30 \) for the reference string in Figure 45. The activity sets are \{D\}, \{C,D\} and \{A,B,C,D\}.

The lifetime of an activity set is defined as its duration at the top of the LRU stack. Thus, when an
activity set is terminated at time $t'$ by a reference to a block below it in the LRU stack, then the activity set lifetime $h(i)$ of the activity set $A(i)$ is given by:

$$h(i) = t' - T(i, t'-1)$$

Finally, a Bounded Locality Interval is defined as the 2-tuple $(A(i), h(i))$ where $A(i)$ is an activity set and $h(i)$ is its associated lifetime. For complete details of the BLI method, refer to [34].

In the next section, the new restructuring algorithm BLI will be described. In the following section, the restructuring algorithm CBLI will be introduced. Then, the experimental results produced by the new restructuring algorithms will be compared with both the CLRU and CWS restructuring algorithms.
In this section, a new strategy-independent restructuring algorithm based on the BLI concept is presented. The basic idea behind this algorithm is that if a set of blocks belongs to an activity set, they should be put close to each other in the virtual address space in an attempt to decrease the number of page faults and the mean working set size. The number of page faults is reduced by forming the activity set in memory with fewer page fetches. The mean working set size is reduced by eliminating from the WS blocks which are not part of the activity set. The BLI restructuring algorithm is outlined in Table 14.

**TABLE 14. The BLI Restructuring Algorithm.**

1. Scan the reference string and identify all the possible Bounded Locality Intervals.
2. Sort the Bounded Locality Intervals into ascending order based on their activity set size. Among activity sets with the same size, priority is given to the one with longer holding time.
3. First, cluster blocks in the smallest activity set into pages, and then the next smallest one, and so on until the last one has been clustered.

This algorithm is very similar to the one developed by Kobayashi [32]. The main difference between these methods
is in BLI, blocks in the smallest activity set are grouped together first. Among activity sets with the same size, priority is given to the one with longer holding time. For example, the order of activity sets for the reference string in Figure 45 would be as follows:

<table>
<thead>
<tr>
<th>Activity Set Size</th>
<th>Activity Set</th>
<th>Holding Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{D}</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>{G}</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>{C,D}</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>{A,B,C,D}</td>
<td>20</td>
</tr>
</tbody>
</table>

Critical Bounded Locality Interval (CBLI) Restructuring Algorithm

In this section, a new strategy-independent restructuring algorithm, based on the concept of BLI and the principle of critical reference, is presented. In general, a critical reference is defined as:

**Definition:** A critical reference [20] is a reference to a block which is not resident in main memory and which causes a block fault.

Previously, the principle of critical reference has been associated only with strategy-dependent restructuring.
algorithms. However, the strategy-independent algorithm described in this section shows that the principle of critical reference can also be applied to strategy-independent restructuring methods.

Given a reference string
\[ S = r(1), r(2), r(3), \ldots, r(t), \ldots, \]

where \( r(t) \) is the block or page reference made by a program at the \( t \)th reference, for \( t=1, 2, 3, \ldots \), the Current Locality \( \text{CL}(t) \) is defined as the largest activity set, if one exists, whose lifetime interval includes \( t \). In this context a critical reference may be defined as follows:

Definition: A critical reference is defined as a reference at time \( t \) to a block which is not in \( \text{CL}(t) \).

The objective of this restructuring effort is to minimize the number of page faults generated by the program during execution. The question then arises as to what relationship exists between the critical references in the block reference string (which can be called block faults) and the page faults which occur during program execution. Suppose that a memory management policy always retains the current activity set in main memory. This indeed is the goal, though not perfectly realized, of successful memory management policies. For a policy which retains the activity set in main memory, the number of page faults
generated by the program is equal to the number of critical references in its page reference string $Sp$. Each of these page faults (critical reference) is also a block fault (critical reference) in the block reference string, $St$. However, a noncritical reference in the block reference string $Sb$ never becomes critical in page reference string $Sp$. Thus, a page fault in $Sp$ always comes from a block fault in $Sb$, but not all block faults result in a corresponding page fault.

The goal of the CBLI algorithm is to minimize the number of critical references in $Sp$, or the number of block faults which become page faults. Therefore, we will try to maximize the number of critical references in $Sb$ which are not critical in $Sp$. A matrix $C$, called the CBLI matrix, will be constructed from $Sb$. Those references which are noncritical in $Sb$ will be disregarded in constructing the matrix. $C$ is an $N \times N$ matrix whose entry $C(i, j)$ ($1 \leq i \leq N$, $1 \leq j \leq N$, $i \neq j$) is the number of times that block $i$ is a critical reference when the activity set contained block $j$. So, $C(i, j) + C(j, i)$ is the number of critical references which will be made noncritical if $i$ and $j$ are mapped into the same page. By the definition of CBLI, $C(i, i)$ will be zero, for all $i$. The CBLI algorithm consists of extracting the $C(i, j)$'s from the block reference string and using them to label the edges of the restructuring graph. More precisely,
the label of the edge connecting nodes $i$ and $j$ will be $C(i,j)+C(j,i)$.

The principle of CBLI is that if the block referenced at time $t$ is already in $CL(t)$, the reference is noncritical, and no restructuring is necessary. If, however, the reference is critical, then the restructuring should attempt to make it noncritical by insuring that the block accessed by the critical reference at time $t$ is already included in $CL(t)$ when it is referenced. This inclusion can be achieved by grouping the block involved in a critical reference with at least one of the blocks already in $BLI(t)$. To obtain this grouping, we can construct the CBLI matrix as follows: increment by 1 the label of each edge connecting the node which represents the critical reference to the nodes representing the blocks which are in $CL(t)$.

To repeat, the philosophy behind the CBLI is that the block involved in a critical reference should be grouped with at least one of the blocks in the current $BLI$ at the time the reference is issued. This approach tends to reduce the number of page faults which occur between adjacent localities (i.e. try to minimize the number of page faults during phase transition time). The CBLI restructuring algorithm is outlined in Table 15.
The CBLI restructuring method has the following advantages:

1. It is a natural approach since it uses the locality set as the basis for the restructuring.
2. It is strategy-independent.
3. It does not depend on any parameters.
4. It is potentially less expensive on the average since a program need be restructured once for any kind of memory management strategy.

TABLE 15. The CBLI Restructuring Algorithm.

For t:=1 to ... do
Begin
If r(t) is a critical reference
Then
Begin
Increment the entry (r(t),b(i)) of the CBLI matrix C by 1
(where b(i) belongs to current activity set).
Compute largest activity set
End
Update Extended LRU stack
End.

The CBLI method is similar to the CRLU and CWS methods except that the CBLI method uses an exact locality in the restructuring and it does not depend on any other parameters. However, both the CRLU and CWS methods use only an estimated locality in the restructuring. Furthermore, CWS depends on the window size parameter T and CRLU depends on a stack depth number.
The Design of the Experiments

Selection of Levels of Each Factor

The principal objective of the experiments described in this chapter is to evaluate the BLI and CBLI restructuring algorithms. The experiments in this chapter are very similar to those presented in the last two chapters. For completeness, a summary of the basic factors used in these experiments is given in Table 16.


<table>
<thead>
<tr>
<th>Number</th>
<th>Factor</th>
<th>BLI</th>
<th>CBLI</th>
<th>CLRO</th>
<th>CBS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Restructuring algorithm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1)</td>
<td>BLI</td>
<td>CBLI</td>
<td>CLRO</td>
<td>CBS</td>
</tr>
<tr>
<td></td>
<td>2) Clustering algorithm</td>
<td>CAN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3) Program chosen for restructuring procedure</td>
<td>XREF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4) Memory management policy</td>
<td>WS (LRU)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5) Input-data (programs)</td>
<td>XREF</td>
<td>WSM.</td>
<td>QUARK.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6) Block type</td>
<td>Procedure-block</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7) Page size</td>
<td>512 bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Experiments

Two sets of experiments have been designed and executed to analyze the performance of the BLI and CBLI restructuring algorithms as measured by the page fault rate and mean working set size. In the first set of experiments, the fixed-space, LRU (Least-Recent-Used) policy is used. The second set of experiments, using a variable-space policy, employs the working set strategy. In both sets of experiments the page fault rate is calculated and, in the second set, the mean working set size is also calculated. For each of these experiments, the experiments performed, the measurements taken, the comparisons made, and the conclusions derived will be presented.

**LRU Experiments and Results**

Table 17 gives a listing of the parameters used in the experiments. Note that the performance index used in this section is the page fault rate since LRU (Least-Recent-Used) is a fixed-spaced policy.

The purpose of this section is to report the paging performance under the LRU environment. The improvements in paging performance produced by the restructuring method BLI and CBLI are compared with the result of the CLRU method over a wide range of memory size.
TABLE 17. Parameters of LRU Experiments.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>BLI</th>
<th>CBLI</th>
<th>CLRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Restructuring algorithm</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>2) Clustering algorithm</td>
<td></td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>3) Program chosen for restructuring procedure</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>4) Memory management policy</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>5) Input-data (programs)</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>6) Block type</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>7) Page size</td>
<td>.</td>
<td>.</td>
<td>512 bytes</td>
</tr>
</tbody>
</table>

Performance improvement indices of the restructured programs, based on the BLI, CBLI and CLRU restructuring method over the original one (the nonrestructured version) of input programs XREF, WSM and QUARK are plotted in Figures 49 to 51.

The results for CBLI displayed in Figures 49 to 51, showing the percentage of improvement in the page fault rate, are very encouraging. The CBLI method is as good as the CLRU method and also exhibits a similar and stable behavior. On the other hand, the results for the BLI method shown in Figures 49 and 50 are dissatisfying. Other
experiments using variations of the BLI algorithm failed to show any significant difference from the results show in Figures 49 to 51. Both the BLI and Kobayashi's restructuring methods are based on locality principle. While Kobayashi's method is better than BLI, both of these methods are not as effective as those restructuring algorithms based on the critical principle, such as CLEO or CWS. This leads us to believe that eliminating page faults which occur during phase transitions is more effective than eliminating page faults which occur within a locality. Thus, the critical principle seems to be a more appropriate basis for program restructuring than does the locality principle. It was, therefore, decided that further investigation of the BLI method would be dropped.

It should also be observed in Figures 49 to 51 that the CBLI method can result in improvement better than those obtained by the CRLU method even when, as in this experiment, the memory management policy is LRU itself. From this data, we can conclude that the strategy-independent restructuring method CBLI is as good as the CRLU strategy-dependent restructuring method. Based on the results of the above experiments, we also conclude that the BLI approach is not satisfactory.
FIGURE 1. LRU RESULT: TIME (XREF.PROC)

MEMORY SIZE

% OF IMPROVEMENT (TIME)
FIGURE 50. LRU RESULT: TIME (HSM.PROC) vs. MEMORY SIZE

% OF IMPROVEMENT (TIME) (x10^4)

0.00 2.00 4.00 6.00 8.00 10.00 12.00

MEMORY SIZE

O : CBLI
* : BLI
+ : CLAU
FIGURE 51. LAU RESULT: TIME (QUARK.PROC)
An experimental study of the performance of the CBLI restructuring method under the WS (Working-Set) environment is reported in this section. The purpose of these experiments was to evaluate the performance of the CBLI restructuring method under the Working-Set environment.
The performance indices in these experiments are the page fault rate, mean working set size and the virtual time-space product (i.e. page fault rate times mean working set size). Table 18 shows the parameters of this experiment.

Figures 52, 53 and 54 show the results for input programs XREF, WSM and QUARK, respectively. All of them show that program restructuring using the CBLI strategy-independent restructuring algorithm is as good as program restructuring using the CWS strategy-dependent restructuring algorithm. In particular, the curves in Figures 53 and 54 show a promising result.

The second performance index used in the working set experiments is the mean working set size. The mean working set size results are shown in Figures 55 through 57. The results of these measurements also show that the CBLI method is somewhat better than the CWS method.

Figures 58, 59 and 60 show the time-space product results for input programs XREF, WSM and QUARK, respectively. As can be seen from these figures, the time-space product results are consistent with the results of the previous two sets of measurements. In all of these WS (Working-Set) measurements, the CBLI method has proven itself to be a sound strategy-independent restructuring algorithm.
FIGURE S2. WS RESULT: TIME (XREF.PROC) vs WINDOW SIZE

% IMPROVEMENT (TIME)

0.00  2.00  4.00  6.00  8.00  10.00  12.00

(X x 10^4)

○: CBLI
+ : CWI
FIGURE S3. WS RESULT: TIME (NSM.PROC)
FIGURE 5. WS RESULT: TIME (QUARK.PROC)
FIGURE 55. WS RESULT: SPACE (XREF.PROC) vs. WINDOW SIZE
FIGURE 56. WS RESULT: SPACE (WSM.PROC)

% OF IMPROVEMENT (SPACE)

0.00  5.00  10.00  15.00  20.00  25.00
0.00  2.00  4.00  6.00  8.00  10.00  12.00

(x10^4)  WINDOW SIZE

○: CBLI
+: CWS
H5 RESULT: SPACE (QUARK.PROC)

% OF IMPROVEMENT (SPACE)

FIGURE 57. WS RESULT: SPACE (QUARK.PROC) (×10^4) WINDOW SIZE

○ : CBLI
+ : CNS
FIGURE 5B. WS RESULT: TIME-SPACE (XREF_PROC)

% OF IMPROVEMENT (TIME-SPACE)

WINDOW SIZE

O : CBLJ
+ : CMS
FIGURE 59. WS RESULT: TIME-SPACE (WSM.PROC)
FIGURE 60. WS RESULT: TIME-SPACE (QUARK.PROC) (×10^4) WINDOW SIZE
Summary of Results

Two major observations can be made based on the data presented in Figures 49 to 60. First, over a wide range of input data, CBLI is at least as good as CLRU when the LRU policy is used and also at least as good as CWS when the WS policy is used. This means that the CBLI combines into one restructuring algorithm all of the power of both CLRU and CWS. Secondly, for each of three WS measurements (page fault rate, mean working set size, virtual time-space product), the CBLI improvement is always better than the CWS improvement except in one case. This exceptional case occurs when the reference string used for the performance measurement was the same reference string used for the restructuring. In all other cases, CBLI out-performed CWS, suggesting that, overall, CBLI produced a more effective restructuring and that CBLI might be less sensitive than CWS to differences in the input data. It is believed that the LRU and Working-Set experiments reported in this chapter are sufficient to establish the following conclusions:

1. The CBLI restructuring method can reduce the cost of program restructuring by eliminating the necessity to restructure a program for each memory management policy under which it will be run.

2. The critical reference principle can also be applied to strategy-independent restructuring method.
3. A strategy-dependent restructuring is not necessarily superior to a strategy-independent restructuring.

4. The critical principle is a correct approach to program restructuring. The locality principle does not appear to be as effective as the critical principle.

5. The CBLI is less sensitive to its input data compared to either the CLRU or the CWS method.

A major advantage of the CBLI method over the CLRU and CWS methods is that the CBLI method can produce useful information, such as the stack depth (average or largest activity set size) for LRU memory management policy or the window size $T$ (the holding time) for Working-Set policy, for the restructured program. Thus, the CBLI method can also determine the memory management parameters which should be used to enhance the program's performance. By contrast, both CLRU and CWS expect such parameters as inputs rather than produce them as by-products.

Another interesting possibility is that the performance of the CBLI can be further improved by combining CBLI and BLI together. Such a combination is impossible for either the CLRU or CWS method since the locality sets are not computed during restructuring.
CHAPTER VI. CONCLUSION AND RECOMMENDATIONS FOR FURTHER WORK

The results of experiments presented in this dissertation demonstrate the feasibility of utilizing the sampling reference strings approach for program restructuring. Based on the experimental results of Chapter IV, it is shown that the performance of several restructuring methods using 5% sampled reference string is comparable to restructuring using the complete reference string. These results indicate that the cost of program restructuring can be substantially reduced by reference string sampling. Also, the sampling method \((K=1, S=19)\), at a five percent sampling rate, is highly recommended for practical program restructuring. In Chapter III, it was found that the restructuring based on the basic blocks is not superior to the one based on the procedure blocks. Furthermore, the results in both of these chapters show that program restructuring for variable-space policies yields less improvement than for fixed-space policies.

Finally, the new restructuring algorithm, Critical Bounded Locality Interval, presented and analyzed in Chapter V, combined the critical and locality principles into a single robust restructuring algorithm. This new algorithm is found to be better than any other existing restructuring method.
Future Work

Only instruction blocks have been considered in this research. It is possible to measure the page fault rate of data blocks by the same method as the one for instruction blocks. However, new clustering principles will be required to reorganize data blocks into structures with low page fault rates. Future investigation of this problem seems desirable.

A second area for future work related to this research is the investigation of the relationship between program restructuring and a prefetched paging policy. It appears possible to utilize the information gathered from a reference string by a restructuring method to predict future reference under such a prefetched paging policy.

A third possible extension of this research is to apply the critical reference principle to the static restructuring method and incorporate this new method into a compiler to provide automatic restructuring.
BIBLIOGRAPHY


ACKNOWLEDGEMENTS

In this section, I would like to use this opportunity to express my appreciation to my thesis supervisor, Professor Dennis G. Kafura, for the substantial time and effort he spent supervising the thesis and in particular for his enthusiasm throughout the course of the research and the writing.

I also thank my committee members for their help and critique of this work: Thank you,

Dennis Kafura
Arch Oldehoeft
Ramachandran Krishnaswamy
Roy Zingg
Terry Smay

I also wish to single out Carol Kelley of the Iowa State University computation center for his assistance and many suggestions for using the Itel AS/6 computer system. My appreciation also goes to Ron Wolf, also from the computation center, who provided me with the PASCAL/8000 compiler and its complete documentation.

My appreciation goes to an understanding and patient person, my wife Melody Fung who typed this thesis.

Again, I especially thank my wife, Melody Fung, for her patience and understanding throughout my years of graduate study at the Iowa State University.
APPENDIX A. PL/1 IMPLEMENTATION OF BLOCK ANALYZER

GETBLK:PROC OPTIONS(MAIN); /* COMPUTE BASIC BLOCK MAP */
DCL CH4004 CHAR(4004);
DCL CH4000 CHAR(4000) DEF CH4004 POS(5);
DCL REF_STR(0:1000) FIXED BIN(31) BASED(P);
DCL CH28000 (7000) FIXED BIN(31);
DCL MAP(0:100) BIT(2) BASED(P1);
DCL BMPA(2000) FIXED BIN(31);
DCL BLKSIZE(2000) FIXED BIN(31);
DCL #B FIXED BIN(31);
DCL BLKNAP(5000) FIXED BIN(31);
DCL EOF BIT(1) INIT('0'B);
DCL O FIXED BIN(31);
DCL I FIXED BIN(31);
DCL J FIXED BIN(31);
DCL AVGBLKSIZE FIXED BIN(31);
DCL #BLK FIXED BIN(31);
DCL SUM FIXED BIN(31);
DCL MINBLK FIXED BIN(31);
DCL MAXBLK FIXED BIN(31);

P=ADDR(CH4004);
P1=ADDR(CH28999);
REF_STR(0)=0;
CH28999=0;
ON ENDFILE(SYSIN) EOF='1'B;
GET EDIT(CH4000) (A(4000));
MAP(0)='1'B;

DO WHILE( ~EOF);

DO I=0 TO 999;
IF (ABS(REF_STR(I))-REF_STR(I+1)) > 6) & (REF_STR(I) > 0)
THEN
   DO;
      MAP(REF_STR(I))='01'B;
      MAP(REF_STR(I+1))='10'B;
   END;
END;
REF_STR(0)=REF_STR(1000);
GET EDIT(CH4000) (A(4000));
END;

#BLK=0;
DO I=0 TO 98000;
IF #BLK > 5000
THEN
DO;
PUT SKIP LIST(' *** #BLK OVERFLOW ***');
STOP;
END;

IF MAP(I)="10"B
THEN
DO;
#BLK=#BLK+1;
BLKMAP(#BLK)=I;
END;
ELSE
IF MAP(I)="01"B & MAP(I+1)="00"B
THEN
DO;
#BLK=#BLK+1;
BLKMAP(#BLK)=I+1;
END;
END;

SUM=0;
MINBLK=100000;
MAXBLK=0;
I=2;
#B=1;
BLKSIZE(1)=0;
BMAP(1)=BLKMAP(1);

DO WHILE (I <= #BLK);
  IF (BLKMAP(I)-BMAP(#B)) > 512
  THEN
  DO;
    DO WHILE ((BLKMAP(I)-BMAP(#B)) > 512);
      #B=#B+1;
      BMAP(#B)=BMAP(#B-1)+512;
      BLKSIZE(#B)=512;
      SUM=SUM+512;
    END;
    #B=#B+1;
    BMAP(#B)=BLKMAP(I);
    MAXBLK=512;
  END;
  ELSE
    IF (BLKMAP(I)-BMAP(#B)) < 32
    THEN
    DO;
      IF (BLKMAP(I)-BMAP(#B-1)) <= 512
      THEN
        DO;
        END;
      ELSE
        IF (BLKMAP(I)-BMAP(#B-1)) <= 512
        THEN
        END;
      END;
    END;
  END;
END;
SUM = SUM - (BMAP(#B) - BMAP(#B-1));
BMAP(#B) = BLKMAP(I);
END;
ELSE
  IF (BLKMAP(I+1) - MAP(#B)) <= 512
    THEN
      DO;
        #B = #B+1;
        I = I+1;
        BMAP(#B) = BLKMAP(I);
      END;
  END;
ELSE
  DO;
    #B = #B+1;
    BMAP(#B) = BLKMAP(I);
  END;
END;
BLKSIZE(#B) = BMAP(#B) - BMAP(#B-1);
SUM = SUM + BLKSIZE(#B);
IF BLKSIZE(#B) > MAXBLK
  THEN
    MAXBLK = BLKSIZE(#B);
  END;
IF BLKSIZE(#B) < MINBLK
  THEN
    MINBLK = BLKSIZE(#B);
I = I+1;
IF #B > 2000
  THEN
    DO;
      PUT SKIP LIST(' **** TOO MANY BLOCK ***');
      STOP;
    END;
END;
AVGBLKSIZE = SUM/#B;
PUT SKIP DATA (#BLK, #B, MINBLK, MAXBLK, AVGBLKSIZE);
DO I = 1 TO #B;
  PUT SKIP EDIT(' ' BLKMAP('',I)' = ' ' ' ' BMAP(I),
                 ' ' BLKSIZE('',I)' = ' ' ' ' BLKSIZE(I))
            (A,F(3), A,F(6), A,F(3), A,F(4));
END;
END;
APPENDIX B. COMPUTE BLOCK STRING

ATOB:PROC OPTIONS(MAIN);
/** ADDRESS STRING CONVERT TO BLK STRING */

DCL CH4000 CHAR(4000) ;
DCL REF_STR(1000) CHAR(4000) BASED(P) ;
DCL BLKMAP(0:605) FIXED BIN(31) ;
DCL EOF BIT(1) ;
DCL Q FIXED BIN(31) ;
DCL I FIXED BIN(31) ;
DCL J FIXED BIN(31) ;
DCL AVGBLKSIZE FIXED BIN(31) ;
DCL #BLK FIXED BIN(31) ;
DCL BLKFI OUTPUT FILE ;
DCL MAPFI INPUT FILE ;

OPEN FILE(BLKFI) ;
OPEN FILE(MAPFI) ;
ON ENDFILE(SYSIN) EOF='1'B ;
EOF='0'B ;
P=ADDR(CH4000) ;
AVGBLKSIZE=164 ;
#BLK=605 ;
BLKMAP(0)=0 ;
GET FILE(MAPFI) DATA(BLKMAP) ;
GET EDIT(CH4000) (A(4000)) ;
Q=300 ;

DO WHILE (~EOF) ;
   DO I=1 TO 1000 ;
      IF REF_STR(I) < 0 | REF_STR(I) > 93226 THEN DO ;
         PUT SKIP LIST( " ** REF_STR ",I,REF_STR(I)) ;
         STOP ;
      END ;
      IF BLKMAP(Q) <= REF_STR(I) & REF_STR(I) < BLKMAP(Q+1) THEN 
         REF_STR(I)=Q+1 ;
      ELSE DO ;
         Q=REF_STR(I)/AVGBLKSIZE+1 ;
         IF Q >= #BLK THEN Q=#BLK-1 ;
      IF REF_STR(I) >= BLKMAP(Q)
THEN
  DO J=Q TO #BLK WHILE (REF_STR(I) >= BLKHAP(J+1));
  END;
ELSE
  DO J=Q TO 1 BY -1 WHILE (REF_STR(I) < BLKHAP(J));
  END;
  Q=J;
  REF_STR(I)=Q+1;
  END;
END;
PUT FILE(BLKFI) EDIT(CH4000) (A(4000));
GET EDIT(CH4000) (A(4000));
END;
END;
APPENDIX C. CLRU RESTRUCTURING METHOD

CLRU: PROC OPTIONS (MAIN);
DCL #BLOCKS FIXED DEC (5, 0);
DCL #PAGES FIXED DEC (5, 0);
DCL EOF BIT (1) INIT ('0'B);

UPDATE_MATRIX: PROC;
END;
ON ENDFILE (SYSIN) EOF='1'B;
GET LIST (#BLOCKS, #PAGES);
BEGIN;
DCL MATRIX (#BLOCKS, #BLOCKS) FIXED DEC (5, 0);
DCL STACK (0: #BLOCKS) FIXED DEC (5, 0);
DCL SP FIXED DEC (5, 0);

MATRIX = 0;
STACK = 0;
GET LIST (STACK (0));

DO WHILE (~EOF);
  IF STACK (0) > #BLOCKS THEN DO;
    PUT SKIP LIST ('***ERROR***');
    STOP;
  END;
  DO I = 1 TO #BLOCKS WHILE (STACK (0) = STACK (I) & STACK (I) = 0);
  END;
  IF I > #PAGES THEN CALL UPDATE_MATRIX;
  DO J = 1 TO 1 BY -1; /* UPDATE STACK */
    STACK (J) = STACK (J - 1);
  END;
  GET LIST (STACK (0));
END;
END;
END CLRU;
APPENDIX D. CRITICAL WS RESTRUCTURING METHOD

CWS: PROC OPTIONS (MAIN):

DCL #BLK FIXED BIN(31);
DCL EOF BIT(1) INIT('0'B);
DCL CH4000 CHAR(4000);
DCL PAGE#(1000) FIXED BIN(31) BASED(P);

ON ENDFILE(SYSIN) EOF='1'B;

BEGIN;
DCL STACK(0: #BLK) FIXED BIN(31);
DCL SP FIXED BIN(31);
DCL T FIXED BIN(31);
DCL K FIXED BIN(31);
DCL TIME(300) FIXED BIN(31);

P=ADDR(CH4000);
STACK=0;
T=1000;
TIME=-999999;
K=0;

GET EDIT(CH4000) (A(4000));

DO WHILE (-EOF); /* LRU MEMORY POLICY */
    DO LOOP=1 TO 1000;
        K=K+1;
        STACK(0)=PAGE#(LOOP);
        IF STACK(0) > #PAGES /* SAVE GUARD */
            THEN
                DO;
                    PUT SKIP LIST('***ERROR***');
                    STOP;
                END;
            IF K-TIME(STACK(0)) > T
                THEN
                    CALL UPDATE_MATRIX;
    /* SEARCH LRU STACK */
    DO I=1 TO #PAGES WHILE(STACK(0) -: STACK(I) &
        STACK(I) -: 0);
    END;

    DO J=1 TO 1 BY -1; /* UPDATE STACK */
        STACK(J)=STACK(J-1);
    END;
END;

GET EDIT(CH4000(A(4000)));
END;

UPDATE_MATRIX:PROC;
DO I=1 TO #BLK WHILE (STACK(I) <= 0 & (K-TIME(STACK(I))) <= T);
MATRIX(STACK(I),STACK(0)) = MATRIX(STACK(I),STACK(0)) + 1;
END;
TIME(STACK(0)) = K;
END;
END CWS;
APPENDIX E. LRU MEASUREMENT

LRU:PROC OPTIONS(MAIN);

DCL PAGEFI INPUT FILE;
DCL #PAGES FIXED BIN(31);
DCL PP FIXED BIN(31);
DCL EOF BIT(1) INIT('0'B);
DCL CH4000 CHAR(4000);
DCL PAGE#(1000) FIXED BIN(31) BASED(P);

ON ENDFILE(PAGEFI) EOF='1'B;
#PAGES=200;

BEGIN;
DCL D(0:#PAGES) FIXED BIN(31);
DCL STACK(0:#PAGES) FIXED BIN(31);
DCL SP FIXED BIN(31);

P=ADDR(CH4000);
D=0;
STACK=0;

GET FILE(PAGEFI) EDIT(CH4000) (A(4000));

DO WHILE(~EOF); /* LRU MEMORY POLICY */
DO LOOP=1 TO 1000;
STACK(0)=PAGE#(LOOP);
IF STACK(0) > #PAGES /* SAVE GUARD */ THEN
DO;
PUT SKIP LIST('***ERROR***');
STOP;
END;
/* SEARCH LRU STACK */
DO I=1 TO #PAGES WHILE(STACK(0) == STACK(I) & STACK(I) == 0);
END;

IF STACK(I)=0 /* COMPUTE DISTANCE STRING */ THEN
D(0)=D(0)+1;
ELSE
D(I)=D(I)+1;
DO J=I TO 1 BY -1; /* UPDATE STACK */
    STACK(J)=STACK(J-1);
END;
END;

GET FILE(PAGEFI) EDIT(CH4000) (A (4000));
END;

/* COMPUTE PAGE FAULT RATE */

PF=D(0);
DO M=*PAGES TO 1 BY -1;
    PUT SKIP LIST(M,PF);
    PF=PF+D(M);
END;

END;
END LRUM;
APPENDIX P. WORKING SET MEASUREMENT

WSH:PROC OPTIONS (MAIN);
   DCL PAGEFI INPUT FILE;
   DCL I FIXED BIN(31);
   DCL K FIXED BIN(31);
   DCL N FIXED BIN(31); /* #PAGES */
   DCL J FIXED BIN(31);
   DCL DELTA FIXED BIN(31);
   DCL W FLOAT BIN(53);
   DCL EOF BIT(1) INIT ('0'B);
   DCL CH4000 CHAR(4000);
   DCL R(1000) FIXED BIN(31) BASED(P);

ON ENDPILE(PAGEFI) EOF='1'B;
N=250; /* UPDATE THIS NUMBER */
J=100;

BEGIN;
   DCL XX FLOAT BIN(53);
   DCL G(J) FLOAT BIN(53);
   DCL P(0:J) FLOAT BIN(53);
   DCL W(0:J) FLOAT BIN(53);
   DCL ADJ(J) FLOAT BIN(53);
   DCL G1(J) FIXED BIN(31);
   DCL TIME(0:N-1) FIXED BIN(31);
   DCL T(0:J-1) FIXED BIN(31);

   P=ADDR(CH4000);
   H=1000;
   T(0)=0;
   XX=1;

   DO I=1 TO J-1;
      T(I)=H*I;
   END;

   TIME=-T(J-1);
   G=G1=ADJ=0;
   I=0;

   GET FILE(PAGEFI) EDIT(CH4000) (A(4000));

   DO WHILE (~EOF);
      DO LOOP=1 TO 1000;
         IF R(LOOP) < 0 THEN GO TO L1;
         IF R(LOOP) > N /* SAVE GUARD */
THEN
DO;
PUT SKIP LIST( "**ERROR**",N,R(LOOP));
STOP;
END;

I=I+1;
DELTA=I-TIME(R(LOOP));
IF DELTA <= T(J-1)
THEN
   J=(DELTA+H-1)/H;
ELSE
   J=J;
   G(J_)=G(J_)+1;
   G1(J_)=G1(J_)+DELTA;
   TIME(R(LOOP))=I;
END;

GET FILE(PAGEFI) EDIT(CH4000) (A(4000));
END;
L1:
K=1;

DO I=0 TO N-1;
   DELTA=I+1-TIME(I);
   IF DELTA <= T(J-1)
      THEN
         DO;
            J=(DELTA+H-1)/H;
            ADJ(J_)=ADJ(J_)+1/K;
            G1(J_)=G1(J_)-T(J_)+DELTA;
         END;
   END;
   W(0)=0;
   F(0)=1;
   W1=0;

   DO I=1 TO J-1;
      G(I)=G(I)/K;
      F(I)=F(I-1)-G(I);
      W(I)=W(I-1)+X*X*G1(I)/K-
           T(I-1)*G(I)+(T(I)-T(I-1))*(F(I)-W1);
      W1=W1+ADJ(I);
   END;

   PUT SKIP DATA(T,F,W);
END;
END;
APPENDIX G. PROGRAM XREF

PROGRAM XREF(FI,OUTPUT);
(* CROSS REFERENCE GENERATOR USING BINARY TREE *)

CONST C1 = 20; (* LENGTH OF WORDS *)
C2 = 18; (* NUMBERS PER LINE *)
C3 = 6; (* DIGITS PER NUMBER *)

TYPE ALFA = PACKED ARRAY(.1..C1.) OF CHAR;
WORDREF = ITEM;
ITEMREF = ITEM;
WORD = RECORD
  KEY : ALFA;
  FIRST, LAST : ITEMREF;
  LEFT, RIGHT : WORDREF
END;
ITEM = PACKED RECORD
  LNO : INTEGER;
  NEXT : ITEMREF
END;

VAR ROOT : WORDREF;
K, K1 : INTEGER;
N : INTEGER;
CH : CHAR;
ID : ALFA;
FI : TEXT;
A : ARRAY(.1..C1.) OF CHAR;

PROCEDURE SEARCH(VAR W1 : WORDREF);
VAR W : WORDREF;
X : ITEMREF;
BEGIN
  W := W1;
  IF W = NIL THEN BEGIN
    BEGIN
      NEW(W);
      NEW(X);
      WITH W DO
      BEGIN
        KEY := ID;
        LEFT := NIL;
        RIGHT := NIL;
        FIRST := X;
        LAST := X
      END;
      WITH W1 DO
      BEGIN
        KEY := ID;
        LEFT := NIL;
        RIGHT := NIL;
        FIRST := X;
        LAST := X
      END;
    END;
  END;
END;
Xa.LNO := N;
Xa.NEXT := NIL;
W1 := W
END
ELSE
IF ID < W2.KEY
THEN
SEARCH(W2.LEFT)
ELSE
IF ID > W2.KEY
THEN
SEARCH(W2.RIGHT)
ELSE
BEGIN
NEW(X);
Xa.LNO := N;
Xa.NEXT := NIL;
W2.LASTa.NEXT := X;
W2.LAST := X
END
END (* SEARCH *)
;

PROCEDURE PRINTTREE(W:WORDREF);
PROCEDURE PRINTWORD(W:WORD);
VAR L : INTEGER;
X : ITEMREF;
BEGIN
WRITE(' ',W.KEY);
X := W.FIRST;
L := 0;
REPEAT
IF L = C2
THEN
BEGIN
WRITELN;
L := 0;
WRITE(' ':C1+2)
END;
L := L+1;
WRITE(Xa.LNO:C3);
X := Xa.NEXT
UNTIL X = NIL;
WRITELN
END (* PRINTWORD *)
BEGIN
IF W <> NIL
THEN
BEGIN
PRINTTREE(WA.LEFT);
PRINTWORD(WA);
PRINTTREE(WA.RIGHT);
END
END (* PRINTTREE *) ;

BEGIN
ROOT := NIL;
N := 0;
K1 := C1;
PAGE(OUTPUT);
RESET(PI);
WHILE ¬ EOF(PI) DO
BEGIN
    N := N + 1;
    WRITE(N: C3); (* NEXT LINE *)
    WRITE(' ');
    WHILE ¬ EOLN(PI) DO
    BEGIN (* SCAN NON-EMPTY LINE *)
        IF ('A' <= FIA) AND (FIA <= 'Z')
        THEN
            BEGIN
                K := 0;
                REPEAT
                    IF K < C1
                    THEN
                        BEGIN
                            K := K + 1;
                            A(-K_) := FIA;
                            END;
                        WRITE(FIA);
                        GET(PI)
                    UNTIL (FIA < 'A') OR (FIA > '9');
                IF K >= K1
                THEN
                    K1 := K
                ELSE
                    REPEAT
                        A(-K1_) := ' ';
                    K1 := K1 - 1
                    UNTIL K1 = K;
                    PACK(A, 1, ID);
                    SEARCH(ROOT);
            END;
        ELSE
            (* CHECK FOR QUOTE OR COMMENT *)
    END;
    END;
END
IF FIA = '****'
THEN
BEGIN
REPEAT
WRITE(FIA);
GET(FI)
UNTIL FIA = '****';
WRITE(FIA);
GET(FI)
END
ELSE
IF FIA = '(' THEN
BEGIN
WRITE(FIA);
GET(FI);
IF FIA = '**'
THEN
BEGIN
REPEAT
WRITE(FIA);
CH := FIA;
GET(FI)
UNTIL (CH = '*') AND (FIA = ')');
WRITE(FIA);
GET(FI)
END;
ELSE
BEGIN
WRITE(FIA);
GET(FI)
END;
END;
WRITELN;
GET(FI)
END;
PAGE(OUTPUT);
PRINTTREE(ROOT);
END.