INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. 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 original submitted.

The following explanation of techniques is provided to help you understand markings or patterns 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 thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

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

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from “photographs” if essential to the understanding of the dissertation. Silver prints of “photographs” may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

Xerox University Microfilms
300 North Zeeb Road
Ann Arbor, Michigan 48106
SU, (David) Hui-Yang, 1939-
PAGINATION OF PROGRAMS FOR VIRTUAL MEMORY
SYSTEMS.

The Ohio State University, Ph.D., 1974
Computer Science

Xerox University Microfilms, Ann Arbor, Michigan 48106

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED.
PAGINATION OF PROGRAMS FOR VIRTUAL MEMORY SYSTEMS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of the Ohio State University

By
(David) Hui-Yang Su, B.S., M.B.A.

* * * *

The Ohio State University
1974

Reading Committee:  
Dr. Clinton R. Foulk
Dr. Ming-Tsan Liu
Dr. James B. Randels

Approved
Clinton R. Foulk
Adviser
Department of Computer
and Information Sciences
ACKNOWLEDGEMENTS

For their contributions to the successful completion of this dissertation I express my gratitude to my thesis advisor, Professor Clinton R. Foulk, and to my readers, Professors Ming-Tsan Liu and James B. Randels. I especially thank Professor Foulk for his patient and invaluable guidance throughout this research project.
VITA

August 30, 1939 Born - Tainan, Taiwan

1965 B.S. in Accounting, Cheng Kung University Tainan, Taiwan

1968 M.B.A., University of Rochester, N.Y.

1968 - 1972 Teaching and Research Associate, Department of Computer and Information Sciences, The Ohio State University.

1973 - Assistant Professor, Department of Mathematical Sciences, Florida International University.

FIELDS OF STUDY

Major field: Computer Operating Systems and Programming Languages
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>ii</td>
</tr>
<tr>
<td>VITA</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF SYMBOLS</td>
<td>x</td>
</tr>
<tr>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Organization of the Thesis</td>
<td></td>
</tr>
<tr>
<td>Virtual Memory Systems</td>
<td></td>
</tr>
<tr>
<td>2. Program Behavior and Paging Performance</td>
<td>7</td>
</tr>
<tr>
<td>Page Interreference Distance Density Function f(x)</td>
<td></td>
</tr>
<tr>
<td>LRU Replacement and the Shape of f(x)</td>
<td></td>
</tr>
<tr>
<td>Page Size, Superfluous Space and Paging Performance</td>
<td></td>
</tr>
<tr>
<td>Program Restructuring</td>
<td></td>
</tr>
<tr>
<td>Representation of Programs as Directed Graphs</td>
<td></td>
</tr>
<tr>
<td>3. Assignment of Basic Blocks to Pages</td>
<td>28</td>
</tr>
<tr>
<td>Basic Blocks</td>
<td></td>
</tr>
<tr>
<td>Pagination of Programs</td>
<td></td>
</tr>
<tr>
<td>Loops</td>
<td></td>
</tr>
<tr>
<td>Loop Nesting</td>
<td></td>
</tr>
<tr>
<td>Pagination Algorithm PO</td>
<td></td>
</tr>
<tr>
<td>The Priority Rule</td>
<td></td>
</tr>
<tr>
<td>An Example</td>
<td></td>
</tr>
<tr>
<td>Comparison with Other Pagination Algorithms</td>
<td></td>
</tr>
<tr>
<td>The Simulator</td>
<td></td>
</tr>
<tr>
<td>The Paging Performance</td>
<td></td>
</tr>
<tr>
<td>Page Size and Paging Performance</td>
<td></td>
</tr>
</tbody>
</table>

iv
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>A Step by Step Trace of an Application of Algorithm P0.</td>
<td>49</td>
</tr>
<tr>
<td>4.1</td>
<td>A Step by Step Trace of an Application of Algorithm P1.</td>
<td>82</td>
</tr>
<tr>
<td>5.1</td>
<td>Differences in Page Fault Rates Between Sequential Allocation and Packed Page Assignments</td>
<td>133</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 A page interreference distance density function $f(x)$</td>
<td>10</td>
</tr>
<tr>
<td>2.2 An estimated page interreference distance density function $f'(x)$ for an execution of the XPL compiler</td>
<td>11</td>
</tr>
<tr>
<td>2.3 Two different page interreference distance density functions, $f^a(x)$ and $f^b(x)$</td>
<td>15</td>
</tr>
<tr>
<td>2.4 Average number of bytes/page referenced when pages are pushed out of main memory during an execution of the XPL compiler</td>
<td>17</td>
</tr>
<tr>
<td>2.5 Changes of page interreference density function when page size is changed from $p$ words to $\frac{1}{2}p$ words</td>
<td>22</td>
</tr>
<tr>
<td>3.1 A PL/I program and it's basic blocks</td>
<td>30</td>
</tr>
<tr>
<td>3.2 Loop nesting patterns</td>
<td>35</td>
</tr>
<tr>
<td>3.3 A graph with three loops and it's loop nesting tree</td>
<td>36</td>
</tr>
<tr>
<td>3.4 Two different page assignments</td>
<td>47</td>
</tr>
<tr>
<td>3.5 A page assignment using pagination algorithm P0</td>
<td>47</td>
</tr>
<tr>
<td>3.6 Page assignment using algorithms given in [Bar 1] and [Ver 1]</td>
<td>51</td>
</tr>
<tr>
<td>3.7 A test program</td>
<td>56</td>
</tr>
<tr>
<td>3.8 Sequential page assignment of the program in Fig.3.7</td>
<td>57</td>
</tr>
<tr>
<td>3.9 Packed assignment using P0 under the shortest path priority rule</td>
<td>58</td>
</tr>
<tr>
<td>3.10 Packed assignment using P0 under the highest probability rule</td>
<td>59</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>3.11 Estimated page interreference density function of the program in Fig.3.7 under different page assignments</td>
<td>61</td>
</tr>
<tr>
<td>3.12 Average page fault rate of the program in Fig.3.7 under different page assignments (LRU replacement)</td>
<td>62</td>
</tr>
<tr>
<td>3.13 Relative paging performance of the program in Fig.3.7 under different page assignments</td>
<td>63</td>
</tr>
<tr>
<td>3.14 Page fault rate of the program in Fig.3.7 under shortest path assignment for different page sizes</td>
<td>67</td>
</tr>
<tr>
<td>3.15 Relative paging performance of sequential and shortest path page assignment for different page sizes</td>
<td>68</td>
</tr>
<tr>
<td>4.1 A subroutine reference tree with 23 routines</td>
<td>71</td>
</tr>
<tr>
<td>4.2 A flowchart of algorithm Pl</td>
<td>80</td>
</tr>
<tr>
<td>4.3 A subroutine reference tree and its page assignment for p=3</td>
<td>81</td>
</tr>
<tr>
<td>4.4 Structure of routines of the program in Fig.4.1</td>
<td>96</td>
</tr>
<tr>
<td>4.5 Packed page assignment of the program in Fig.4.1</td>
<td>98</td>
</tr>
<tr>
<td>4.6 Ordered page assignment of the program in Fig.4.1</td>
<td>99</td>
</tr>
<tr>
<td>4.7 Random page assignment of the program in Fig.4.1</td>
<td>99</td>
</tr>
<tr>
<td>4.8 Inter-routine reference frequency of the program in Fig.4.1</td>
<td>100</td>
</tr>
<tr>
<td>4.9 Page fault rate of the program in Fig.4.1 under four different page assignments</td>
<td>102</td>
</tr>
</tbody>
</table>
Figure Page

4.10 Relative paging performance of the program in Fig.4.1 under different page assignments... 104

4.11 Page fault rate of the program in Fig.4.1 under packed assignment using algorithm P1 for different page sizes ..................... 106

4.12 Relative paging performance of the program in Fig.4.1 under packed assignment for different page sizes ......................... 107

4.13 The subroutine reference tree of a program with no conditional references ............... 110

4.14 Inter-routine reference frequency of the program in Fig.4.1 with a new set of probability of inter-routine references ........ 110

4.15 Page fault rate of the program in Fig.4.14 under four different page assignments for 2K bytes page size ..................... 112

4.16 Relative paging performance of the program in Fig.4.14 under different page assignments for 2K bytes page size ..................... 113

5.1 A subroutine reference graph and its page assignment for page size p=4 ................. 121

5.2 Page fault rate of the XPL compiler under sequential and packed (P2) assignments (instructions only) ....................... 127

5.3 Relative paging performance of the XPL compiler under sequential and packed (P2) assignments for different page sizes (instructions only) ....................... 129

5.4 Page fault rate of the XPL compiler under different page assignments for page size of 2048 bytes (instructions and data combined) ... 134

5.5 Relative paging performance of the XPL compiler under different page assignments and for different page sizes (instructions and data combined) ....................... 135
LIST OF SYMBOLS

AN  A list of nodes in loop cl with some immediate successors not in \( L_{cl} \).

B_i  A basic block of a program.

c(x,y)  The highest nesting level of loops in which routine x references routine y.

C(x,y)  The number of times routine x references routine y.

cl  Current loop in which nodes are being processed.

cp  The page number of the current page to which nodes are assigned.

d_i(t)  The page interreference distance of reference \( r_t \) where \( r_t=\text{page } i \).

d_i  The page interreference distance of any reference to page i.

d(t)  The page interreference distance of reference \( r_t \).

E  The set of edges in G.

EN  A list of nodes in loop cl with at least an immediate successor as the entry node of another loop.

\( f_i(x) \)  The page interreference distance density function for page i.

\( f(x) \)  The page interreference distance density function of a program.

\( \hat{f}(x) \)  The estimated page interreference distance density function.

\( F_{LRU}(M) \)  The number of page faults with M pages of memory under LRU replacement.

G  A graph representing a program Z.

\( |G| \)  The size of a graph G, the sum of sizes of nodes in G.

IS(v_i)  The set of immediate successors (descendants) of \( v_i \) in G.

\( |IS(v_i)| \)  Number of immediate successors of \( v_i \), i.e. number of elements in the set IS(v_i).

IP(v_i)  The set of immediate predecessors (ancestors) of \( v_i \) in G.

\( |IP(v_i)| \)  Number of immediate predecessors of \( v_i \).
\( L_i \)  
The set of nodes in loop \( i \).

\(|L_i|\)  
The sum of sizes of nodes in \( L_i \).

\( \text{LE}_i \)  
The entry node of \( L_i \).

\( \text{LE} \)  
The set of loop entry nodes.

\( \text{ln}(v) \)  
The loop number of the inner-most loop in which node \( v \) is located.

\( \text{LS} \)  
A stack of loops, top entry of the stack is the loop number of the inner most loop currently being processed.

\( \text{NS} \)  
A stack of nodes in the \( L \) with more than one immediate successor (descendants).

\( p \)  
The page size.

\( \text{P}_i \)  
The set of nodes assigned to page \( i \).

\(|\text{P}_i|\)  
The sum of sizes of nodes in \( \text{P}_i \).

\( \text{pageno} \)  
The next available page for assignment.

\( \text{PE} \)  
The set of program entry nodes.

\( \text{pg}(v) \)  
The page number to which node \( v \) is assigned.

\( R \)  
A reference string from an execution of a program.

\( r_{t} \)  
\( t=1,2,\ldots,n \). Sequence of references in \( R \).

\( \text{rn}(x) \)  
The reference nesting level of routine \( x \), it is the highest nesting level of references in which routine \( x \) is referenced.

\( s(v) \)  
The shortest distance from node \( v \) to the end of a program. The distance between any two nodes, \( v_1 \) and \( v_2 \),is the sum of sizes of nodes in the path from \( v_1 \) to \( v_2 \).

\( T \)  
A set of terminal nodes in a subroutine reference tree or graph.

\( U \)  
A list of partially filled pages.

\( V \)  
The set of nodes in \( G \).

\( v_1 \)  
\( v_1 \in G \).

\( w_i \)  
The size of node \( v_1 \).
Chapter 1  Introduction

Organization of the Thesis

This thesis is primarily concerned with the restructuring of programs to reduce the page fault rate of a virtual memory system. In chapter 2, the relationships of the behavior of programs and the page fault rate are discussed, and a brief review of the representation of programs as directed graphs is also given. In chapter 3, a pagination algorithm to segment programs whose subroutine sizes are greater than the page size is given. In chapter 4, the subroutine packing problem is discussed. The problem concerns the assignment of pages to subroutines when the subroutine sizes are less than the page size. The effects of packing on paging performance are also analyzed. In chapter 5, a case study of the optimization of a production program, the XPL compiler, is presented.
Virtual Memory Systems

Computer systems with large overall memory capacity
normally use a hierarchical memory system because of the
high cost of fast access memory. Most modern memory systems
consist of at least two levels, a high cost fast "main
memory" such as core memory and a set of low cost slow
"auxiliary memories" such as drums, disks and tape units.
The central processor can only access information in the
main memory, and information in auxiliary memories must be
brought into main memory before being processed.

In most computer systems, the program address space
is limited by the available physical main memory space.
When the size of a program exceeds available space, the
programmer has to divide it into overlays and to control
the movement of these overlays between main memory and
auxiliary memory.

In some newer systems, a virtual memory system
[Den l] is used which provides a programmable virtual address
space many times greater than the real main memory
space so that programs can be written as though a very
large main memory space were available. During the exe-
cution of a program, only a fraction of the program is in
the main memory. The system decides for users what parts
of a program should be in main memory, when it should load
them in, and where to put them.
Many virtual memory systems use an addressing scheme called paging, in which programs and main memory are divided into fixed equal size blocks which are called pages and page frames respectively. A page may be placed in any page frame; the system keeps track of page placement by means of a page map. When a program is in execution, each reference to a virtual memory address is automatically mapped to a physical memory address. If a program references a page which is not in the main memory, an interrupt called a page fault will be generated and the program will wait until the system locates the page in auxiliary memory, places it in a free page frame, and updates the page map; these actions are called a "pull". If there is no free page frame, the system must locate a page in main memory and push it back to auxiliary memory; this action is called a "push".

The access time of auxiliary memory is usually $10^4$ to $10^5$ times the access time of main memory. Each page fault takes considerable time and overhead; therefore, attention has been focused on methods of minimizing the number of page faults, especially on paging policies which govern the paging process. These policies consist of a fetching policy to decide when to fetch a page, a placement policy to decide where to place a page, and a replacement policy to select which page is to be replaced when all page frames are filled. If pages are fetched when and only
when they are needed, then the fetching policy is a demand paging policy; otherwise, it is a nondemand, preloading policy. Under demand paging, placement policy is a subsidiary of replacement policy because the location of the page frame being freed is the place to put the new page.

Replacement Algorithms

As mentioned in the last section, the replacement policy determines which page to push whenever space is needed in the main memory. To minimize the number of page faults, we attempt to first replace those pages that have a good likelihood of not being referenced again. Conversely, we try to retain those pages that have a good likelihood of being referenced again in the near future. Algorithms for page replacement have been the subject of many studies, e.g. [Bel 1, Mat 1, Aho 1].

Most replacement algorithms may be grouped into the following two classes:

**Class 1:** The replacement decisions are based on the assumption that all pages are equally likely to be referenced at any time. Examples: random and first-in-first-out (FIFO) replacement.

**Class 2:** The replacement decisions are based on the history of the most recent use in main memory for each page. Examples: least recently used (LRU) and working set principle (WS) replacement [Den 2].
Following is a brief description of Random, FIFO, LRU and WS replacement algorithms.

**Random.** Since it is assumed that all pages have equal probability of being referenced, a page in main memory is randomly selected to be pushed out.

**FIFO.** The page that has been in the main memory for the longest time is pushed out.

**LRU.** The page that has not been referenced for the longest time is pushed out of main memory.

**WS.** Any page that has not been referenced during a fixed preceding period, $t$, can be pushed out. $t$ is called the window size.

All the above algorithms except WS have a fixed memory size. LRU and WS replacement are related to each other in the following sense: WS is an LRU replacement whose memory size is dynamically adjusted; conversely, LRU is a WS replacement whose window size $t$ is dynamically adjusted so that the number of pages referenced during preceding window size $t$ is fixed.

Class 2 algorithms are also called Stack algorithms [Mat 1]. A Stack algorithm is an algorithm in which pages in main memory are ordered according to some priority rule, and the page with lowest priority is pushed out of main memory. The priority rule is such that the number of page faults is a monotonic decreasing function of available
memory size; the larger the memory size, the lower the number of page faults. FIFO and Random are not stack algorithms because an increase in available memory size may not necessarily decrease the page fault rate [Bel 2]. In fact, any algorithm whose page fault rate is a monotomic increasing function of the page interreference distance (to be defined in next chapter) for any given memory size is a stack algorithm.
Chapter 2 Program Behavior and Paging Performance

There are many factors affecting the performance of virtual memory systems, such as the speed and size of main memory, the page size, the paging algorithms, and most important of all, the behavior of programs.

Most paging algorithms predict the behavior of programs according to the principle of locality [Den 1] which says that programs tend to use a small subset of pages in a given period, and that members in this subset change slowly over time. Therefore, the pages that are used in the recent past are most likely to be used in the near future and they should be kept in the main memory.

Other factors being held equal, the program behavior influences strongly the performance of virtual memory systems, as has been shown by several experiments. McKellar and Coffman [McK 1], and Brawn and Gustavson [Bra 1] have shown that by some simple changes of algorithms in writing a program; for example, storing matrices in submatrices instead of by rows or by columns, the page fault rate of the program is significantly reduced. Comeau [Com 1], and Hatfield and Gerald [Hat 1] have also shown that by rearranging the order in which system routines are organized, paging activity is reduced many fold.

7
Page Interreference Distance

The paging behavior of programs will be characterized in terms of page interreference distance. Consider a program of \( n \) pages which are numbered \( 1, 2, \ldots, n \). The dynamic paging behavior of the program can be represented by its reference string \( R = r_1 r_2 \ldots r_t \ldots \), which is a sequence of page numbers. \( r_t = i \) implies that page \( i \) is referenced at the \( t \)-th reference.

Definition 2.1 Let \( r_t = i \) and let the last reference to page \( i \) before the \( t \)-th reference be \( r_j \), then the page interreference distance for page \( i \) at time \( t \), denoted by \( d_i(t) \), is the number of distinct pages, excluding page \( i \), in the sequence \( r_j r_{j+1} \ldots r_t \). For convenience, we shall say that the first time page \( i \) is referenced, \( d_i(t) = n \).

If we now make the assumption that the underlying stochastic process generating \( R \) is stationary over time, then we may make the following definitions.

Definition 2.2 The page interreference distance density function, \( f_i(x) \), for page \( i \), is defined as

\[
f_i(x) = \lim_{k \to \infty} \left\{ \frac{\text{No. of times the page interreference distance } d_i(t) = x \text{ in } r_1 r_2 \ldots r_k}{\text{No. of references to page } i \text{ in } r_1 \ldots r_k} \right\}
\]

(2.1)
Definition 2.3 The page interreference distance density function, \( f(x) \), for a program is defined as

\[
    f(x) = \frac{1}{n} \sum_{i=1}^{n} a_i f_i(x),
\]

(2.2)

where \( a_i \) is the fraction of times page \( i \) was referenced, that is, \( a_i = \lim_{k \to \infty} \frac{1}{k} \) (No. of references to page \( i \) in \( r_1r_2\ldots r_k \)).

(2.3)

The function \( f(x) \) represents the probability that a given reference has page interreference distance \( x \).

Fig. 2.1 shows the density function \( f(x) \) for a typical program with a given page size. According to the principle of locality, the curve should be skewed to the left with a long flat tail on the right. A highly "localized" program which references a small subset of pages of the program in any given period will have a peak occurring very close to the origin of the \( x \)-axis. The higher the peak, the stronger the locality the program has.

Since the actual distribution of \( f(x) \) for programs is not available, we may use \( \hat{f}(x) \) obtained from monitoring an actual execution of a program as an approximation to \( f(x) \). Fig. 2.2 shows the distribution \( \hat{f}(x) \) from an execution of the XPL compiler in a 4.5 million-reference period. This is the same trace which is used in chapter 5 for analysis of a page assignment algorithm.

The importance of function \( f(x) \) will be shown in the
following theorem. It states that given a replacement algorithm and its associated parameter (memory size) the page fault rate is completely determined by the shape of the page interreference distance density of a program.

![Page Interreference Distance Density](image)

**Fig. 2.1** A page interreference distance density function $f(x)$. 
Fig.2.2 An estimated page interreference distance density function $f(x)$ for an execution of the XPL compiler. Page size = 4096 bytes.
LRU Replacement and the Shape of \( f(x) \)

**Theorem 2.1** Let \( k \) be the number of references for an execution of a program with page interreference distance density function \( f(x) \), then the expected number of page faults with \( M \) pages of memory under LRU replacement is

\[
F_{\text{LRU}}(M) = k \sum_{x=M}^{n} f(x). \tag{2.4}
\]

**Proof.** By the definition of page interreference distance, a reference \( r_t \) to page \( i \) having a distance \( d_i(t) \) means that \( d_i(t) \) distinct pages have been referenced since the last reference to page \( i \). Under LRU replacement, if memory size \( M \leq d_i(t) \), then page \( i \) will have been pushed out of memory before it is referenced at time \( t \). Since \( f(x) \) represents the probability of a reference having page interreference distance \( x \), \( \sum_{x=M}^{n} f(x) \) is the probability of a reference to a page not in memory, hence

\[
F_{\text{LRU}}(M) = k \sum_{x=M}^{n} f(x). \quad \text{Q.E.D.}
\]

Note that equation (2.4) can be rewritten as

\[
F_{\text{LRU}}(M) = \sum_{x=M}^{n} k f(x). \quad \text{We will denote } k f(x) \text{ by } f'(x),
\]

which is the expected number of times the page interreference distance is equal to \( x \).

In the following chapters, we will discuss various
algorithms for restructuring programs. The page fault rate under LRU replacement is used to compare the effectiveness of these algorithms. In order to find the page fault rate under LRU replacement, the quantity \( f'(x) \) is obtained by counting the number of references having page interreference distance \( x \) in the reference string for all \( x \); the number of page faults under all possible memory sizes is then computed by \( \sum_{x=M}^{n} f'(x) \) for \( M = 1, 2, \ldots, n \). Thus only a single pass through the reference string is required.

The page interreference distance density function of a program may have different shapes under different page sizes or page assignments as shown in Fig. 2.3(a). By page assignment we mean the assignment of page numbers to sections of a program. \( f^a(x) \) and \( f^b(x) \) are the curves for different organization of the same program. Fig. 2.3(b) shows the quantity \( \sum_{x=M}^{n} f^a(x) \) and \( \sum_{x=M}^{n} f^b(x) \) for \( M = 0, 1, 2, \ldots \).

Note that \( \sum_{x=0}^{n} f^a(x) = \sum_{x=0}^{n} f^b(x) = 1 \). In this example, by Theorem 2.1, if the memory size \( M \) is larger than four pages, then the LRU page fault rate for \( f^a(x) \) is less than that for \( f^b(x) \), but when \( M < 5 \) pages, the situation is reversed. If we know the memory size will always be less than 5 pages, then we should organize the program to behave like
We are interested in ways of changing the shape of the page interreference distance density function $f(x)$ so that the program produces the least number of page faults. Changing the page size may change $f(x)$; we will discuss its effect on page fault rate in the next section. Another way of changing $f(x)$ is to restructure the program, and we will devote chapters 3 and 4 of this thesis to that topic.
(a) Page interreference density functions

(b) Summation of $f(x)$

Fig. 2.3 Two different page interreference distance density functions, $f^a(x)$ and $f^b(x)$. 
Page Size, Superfluous Space and Paging Performance.

In a virtual memory system with large page size, each time a page is loaded into main memory, only a fraction of the words in the page are referenced during its residence. Those words that are not referenced are called superfluous space by Kuck in [Kuc 1]. Large superfluous space means less useful information being contained in a given main memory space, so more page faults are needed to bring in the required information. Fig.2.4 shows a trace of memory utilization during an execution of the XPL compiler [Mckm 1] compiling a 250-statement program. The trace driven simulation of the compiler was run with 20k bytes of main memory under LRU replacement. Every time a page is pushed out of main memory, the number of bytes referenced during its residence in main memory is recorded. In Fig.2.4, every point on the curve represents the average number of bytes referenced in pages being pushed out of main memory in a 5,000 instruction interval. Fig.2.4(a) shows the trace for page size 2048 bytes and Fig.2.4(b) for page size 1024 bytes. The gaps in the curve represent the periods in which no page is pushed out because only pages in main memory are referenced. Note that the gap is immediately followed by a peak, showing that those pages have been heavily referenced in the preceding period. In
Fig. 2.4 Average number of bytes/page referenced when pages are pushed out of main memory during an execution of the XPL compiler. The averages were computed in 5,000-instruction intervals under LRU replacement with 20K of memory space.
summary, the results show that, in this example each time a page is in main memory, on the average only about 50 bytes in a 2048-byte page, and 75 bytes in a 1024-byte page, are referenced; the rest of the page is superfluous space.

Normally, such superfluous space may be reduced by decreasing the page size. Let us assume page size is changed from $p$ words to $\frac{1}{2}p$ words. Let $f^P(x)$ and $f^{\frac{1}{2}P}(x)$ be the page interreference distance density function for page size $p$ and $\frac{1}{2}p$ respectively. By Theorem 2.1, given a memory size of $Mp$ words ($M =$ number of pages) and $k$ references, the expected number of page faults under LRU replacement are

$$F_{\text{LRU}}^P(M) = k \sum_{x=M}^{n} f^P(x),$$

(2.6) for page size $p$ and

$$F_{\text{LRU}}^{\frac{1}{2}P}(2M) = k \sum_{x=2M}^{2n} f^{\frac{1}{2}P}(x)$$

(2.7) for page size $\frac{1}{2}p$ and $M = 1, 2, \ldots n$. $n$ is the number of pages for page size $p$.

On the one hand, if all references are in the same half of the pages of size $p$, then the page interreference distance density functions $f^{\frac{1}{2}P}(x) = f^P(x)$ for $0 \leq x < n$, $f^{\frac{1}{2}P}(x) = 0$ for $n \leq x < 2n$ and $f^{\frac{1}{2}P}(2n) = f^P(n)$. Since $f^P(x+1) \leq f^P(x)$ for $0 \leq x < n$, it can be shown that
For memory size of Mp words by comparing equations (2.6) and (2.7). This indicates that reducing the page size will cut the page fault rate at least in half. A typical curve is shown in Fig.2.5(a).

On the other hand, if both halves of the \( ^{1p} \) words pages are referenced in succession and alternately, i.e. references to page \( i \) always come in pairs, so that a reference to the first half of page \( i \) is always followed immediately by a reference to the second half of page \( i \), then the page interreference distance for smaller pages will always be larger than that for larger pages, as shown by the following reference string \( R^p \) and \( R^{^{1p}} \), where \( i \) is divided into \( i_a \) and \( i_b \). The number below each page number is the page interreference distance \( d_i(t) \). We will only show references to page \( i \). The numbers inside parentheses are the number of references to other pages, which may be to one page or to several different pages. The number of distinct pages are given in \( d_i(t) \) for \( R^p \).

\[
R^p = i \quad i \quad i \quad i \quad i \quad (4) \quad i \quad i \quad (2) \quad i \quad i \quad i \quad i \quad (6) \quad i \quad i \quad i \quad i \quad i \quad i \quad i \quad i \quad i \quad i \quad i \quad ...
\]

\[
d_i(t) = n \quad 0 \quad 0 \quad 0 \quad ... \quad 2 \quad 0 \quad ... \quad 1 \quad 0 \quad ... \quad 1 \quad 0 \quad 0 \quad 0 \quad ...
\]

\[
R^{^{1p}} = i_a \quad i_b \quad a \quad b \quad (4) \quad i_a \quad i_b \quad (2) \quad i_a \quad i_b \quad (6) \quad i_a \quad i_b \quad (4) \quad i_a \quad i_b \quad i_a \quad i_b \quad ...
\]

\[
d_i(t) = 2n \quad 2n \quad 1 \quad 1 \quad ... \quad 5 \quad 5 \quad ... \quad 3 \quad 3 \quad ... \quad 7 \quad 7 \quad ... \quad 3 \quad 3 \quad 1 \quad 1 \quad ...
\]

If the page interreference distance of a reference \( r_t \) to page \( i \) in \( R^p \) is \( x \), \( 0 < x < n \), then the page interreference distance for page \( i_a \) or \( i_b \) in \( R^{^{1p}} \) is \( 2x + 1 \), for there are \( 2x \) distinct pages plus page \( i_b \) or \( i_a \) between successive.
references to either page $i_a$ or $i_b$; if $x=n$, then the interreference distance in $R^{kp}$ is $2n$, for neither page $i_a$ nor $i_b$ has been referenced; if $x=0$, then interreference distance in $R^{kp}$ depends on the distance of the preceding reference, $d_i(t-1)$ in $R^P$: it is $2n$ if $d_i(t-1)=n$, and $2d_i(t-1)+1$ if $d_i(t-1)\neq n$.

Therefore,
\[
\begin{align*}
    f^{kp}(2x+1) &> f^P(x), \\
    f^{kp}(2x) & = 0,
\end{align*}
\]
for $0 \leq x < n$ and
\[
    f^{kp}(2n) = 2f^P(n).
\]

It can be shown that $F_{LRU}^{kp}(2M) > F_{LRU}^P(M)$, that is, reducing the page size will not decrease the page fault rate. Fig.2.5(b) shows the page interreference distance density functions for the type of reference string given above.

In practical situations, the change in the shape of $f(x)$ lies between the two extremes. If a program references words in pages evenly, then reducing the page size will increase the page interreference distance for most references in its reference string, and the curves for $f^P(x)$ and $f^{kp}(x)$ will look like Fig.2.5(c). In Fig.2.5(c), smaller page size has higher page fault rate, i.e. $F_{LRU}^{kp}(2M) > F_{LRU}^P(M)$ for any memory size greater than $p$ words (one page of size $p$ words). In [Hat 2], Hatfield shows
several programs having this type of behavior.

If a program references words in pages unevenly, we may have curves for $f^P(x)$ and $f^P(x)$ as in Fig.2.5(d). In such a case, $F^{LRU}(2m) < F^{LRU}(m)$ for memory size less than $mp$ words, and $F^{LRU}(2m) > F^{LRU}(m)$ for memory size greater than or equal to $mp$ words. The size of $m$ depends on the behavior of the program. Under LRU algorithm, all programs in the machine are given a fixed $M$ pages of memory space, if the behavior of these programs all have the property that $m > M$, then it will not be advantageous to reduce the page size.
Fig. 2.5 Changes of page interreference density function when page size is changed from $p$ words to $\frac{1}{2}p$ words.
In the previous section we have shown that reducing the page size may not be advantageous for all programs. In addition, the optimal page size is also determined by other factors, such as the access speed of the secondary memory and overhead associated with paging activity. This page size is fixed, however, once the system design has been completed. The practical way to change the page interreference density function $f(x)$ of a program is to restructure the program, either by the compiler or some other means.

The distribution of page interreference distances is primarily determined by those pages that are referenced repeatedly. If we consider a program as a directed graph, and let page $i$ contain the instructions in the acyclic part of the graph, then references to page $i$ will normally have a very small page interreference distance. But as $k$ approach $\infty$ in the reference string $R = r_1r_2\ldots r_k$, the fraction of time page $i$ is referenced becomes very small and its influence on the overall behavior $f(x)$ is negligible. If page $i$ contains instructions in a cyclic part of the graph, its contribution to the overall behavior $f(x)$ depends on how many times the cycle is executed. It is very obvious that in order to move the peak of $f(x)$ to
the left as close to the origin of the x-axis as possible, we must minimize the interreference distances for pages inside loops of a program, especially for the most frequently executed loops. This is the same as minimizing the number of pages required to execute a loop.

In the discussion of restructuring algorithms in the following chapters, we will use the following two rules.

1. Minimize the superfluous space in pages brought into main memory. Since we do not want to reduce the page size, the way to achieve this goal is to put instructions or data which are most likely to be referenced in the short time interval in the same page.

2. Minimize the number of pages required to execute a loop. We want to minimize the number of pages occupied by a loop, and by the subroutines referenced inside that loop.

We will approach the program restructure in two areas by asking the following questions:

1. When a program consists of routines with several pages of instructions (i.e. routine size is greater than page size), how do we segment these routines into pages?

2. When a program consists of routines less than the page size, how do we assign routines to pages?
Representation of Programs as Directed Graphs

The prediction of program behavior will be done by analyzing the program control flow. The application of graph theory to program flow analysis has been widely used, e.g. [All 1, Low 1]. Some terminology of directed graphs is defined in this section.

Definition 2.4 A directed graph $G$ can be denoted as $G=(V,E)$ where $V$ is a set of nodes $v_1, v_2, v_3, ..., v_n$ and $E$ is a set of directed edges represented by ordered pairs $(v_i, v_j)$, $v_i$ and $v_j \in V$. The pair $(v_i, v_j)$ indicates that the directed edge goes from $v_i$ to $v_j$.

Definition 2.5 For any pair $(v_i, v_j) \in E$, $v_j$ is called an immediate successor of $v_i$, and $v_i$ is called an immediate predecessor of $v_j$. The set of immediate successors of $v_i$ in $G$, $IS_G(v_i)$, is defined as $IS_G(v_i) = \{v_j | (v_i, v_j) \in E\}$. Similarly, the set of immediate predecessors of $v_j$ is $IP_G(v_j) = \{v_i | (v_i, v_j) \in E\}$. When there is no possibility of ambiguity, subscript $G$ will not be written.

Each node in the graph may represent a section of a program, e.g. a basic block [All 1], or a subroutine.

Definition 2.6 A subgraph of a directed graph $G$ is a directed graph $G'=(V', E')$ in which $V' \subseteq V$, $E' \subseteq E$ and for each $v_i \in V'$, $(v_i, v_j') \in E'$ if $v_j' \in V'$. 
Definition 2.7  A path from node $v_{i_1}$ to $v_{i_j}$ is a sequence of nodes $v_{i_1}, v_{i_2}, \ldots, v_{i_j}$ such that $v_{i_k} \in IS(v_{i_{k+1}})$ for $k=1, 2, \ldots, (j-1)$. If none of the nodes in the sequence appears more than once, then the path is a simple path. If $v_{i_1} = v_{i_j}$ then the path is cyclic and if the path from $v_{i_1}$ to $v_{i_1}$ is a simple path, then the cyclic path is a simple cyclic path or simple cycle.

Henceforth, the terms directed graph, simple path and simple cycle will be referred to simply as graph, path and cycle respectively.

Definition 2.8  The size of a node in a graph, denoted by $w_i$ for $v_i \in V$, is the number of storage words required by the block of program which the node represents. The size of a graph $G$ is the sum of the sizes of the nodes contained in $G$, and is denoted by $|G|$. Note that a subgraph or a path may be thought of as a graph and hence the size of a subgraph or a path is the sum of the sizes of nodes contained in the subgraph or path.

Definition 2.9  A directed tree is a directed graph such that (1) there exit no cycles in the graph, (2) there exist a node $v$ such that $IP(v) = \Lambda$, and (3) for every node $v_j \in V$, there exist a path from $v$ to $v_j$. Node $v$ is called
the root of the directed tree.

Immediate descendants and ancestors are synonyms of immediate successors and predecessors respectively. In the following chapters, a node in a graph may represent a sequence of statements in a routine, or an entire routine. In order to avoid confusion, immediate successors and predecessors will be used when nodes represent statements; immediate descendants and ancestors will be used when nodes represent routines. In both cases, the same notations IS(v) and IP(v) will be used.
Chapter 3 Assignment of Basic Blocks to Pages

In this chapter, we will examine the problem of dividing a routine into several pages when the size of the routine is greater than page size. The division of programs into pages will be called "Pagination of Programs".

Basic Blocks

Let \( Z = S_1, S_2, \ldots, S_i, \ldots, S_h \) represent a program where \( S_i, i = 1, 2, \ldots, h \) is a string of source statements (or sections of source statements) in the order in which they are presented to the compiler. If \( S_i \) may transfer control to \( S_j \), then \( S_j \) is said to be an immediate successor statement of \( S_i \), and \( S_i \) an immediate predecessor statement of \( S_j \).

Definition 3.1 A basic block \( B_k \) in \( Z \) is a sequence of statements in \( Z \), where

\[
B_k = S_{k_1}, S_{k_2}, \ldots, S_{k_j}, \ldots, S_{k_k}
\]

where \( k = 1, 2, \ldots, g \) and \( S_{k_j} \) in \( Z \), such that

1. \( S_{k_j} \) is the only immediate successor statement of \( S_{k_{j-1}} \), for \( j = 2, 3, \ldots, k \).

2. None of immediate predecessor statements of \( B_{k_1} \) is \( B_{k_j} \), \( j = 2, 3, \ldots, k-1 \).
3. $S_{k_l}$ must have (a) no immediate successor statement, (b) at least two immediate successor statements, or (c) an immediate successor statement having more than one immediate predecessor statement.

By condition 2, $S_{k_1}$ is the only entry point to $B_k$ from any path; by condition 1, all statements in $B_k$ are to be executed sequentially once the control enters $B_k$; by condition 3, $B_{k_k}$ is the last statement before another block entry point, hence $B_k$ contains a maximum number of statements satisfying condition 1.
Fig. 3.1 shows a sample PL/I program and its basic blocks. Statements not explicitly written are statements that do not have explicit transfer of control, and declaration statements are not included.

(a) Source program

(b) List of basic blocks

Pagination of Programs

Let $G = \{1, 2, \ldots, n\}$ be the set of nodes in a directed graph representing a program $Z$. Each node $k \in G$ represents the basic block $B_k$ in $Z$. An edge $(i, j)$ in the directed graph indicates that control may be transferred from
basic block $B_i$ to $B_j$.

**Definition 3.2** Pagination of a program $Z$ is to partition the directed graph representing $Z$ into subgraphs, $P_i$, $i = 1, 2, \ldots, m$ such that

1. $G = \bigcup_{i=1}^{m} P_i$
2. $|P_i| \leq p$ for $i = 1, 2, \ldots, m$
   where $|P_i| = \sum_{k \in P_i} w_k$, $p$ is the page size and $w_k$ is the size of node $k$.
3. The expected page fault rate is minimum for a given memory size of $M$ pages, $M \geq m$.

In order to find the best partition which minimizes the expected page fault rate, we have to know the execution frequency of nodes. Since page assignment is to be done at compilation time, the compiler can only estimate the execution frequency either from information supplied by the programmer, or from the analysis of program control flow. In the absence of other information, the compiler may analyze the loop structure and assume that inner loops are executed more often than outer loops.

One important consideration in compiler optimization is the tradeoff between increased compilation time and decreased execution time. How much time should the compiler spend on collecting information? How much time
should it spend on the optimization process? Even if the compiler had all the information it needed, there is still no simple algorithm to find an "optimal" partition which minimizes the page fault rate; therefore we will approach the problem based on loop analysis of a program.

Loops

Recall that in Chapter 2, we defined a cycle in a directed graph as a sequence of nodes

\[ v_{i_1}, v_{i_2}, \ldots, v_{i_j} \]

in the directed graph such that

1. \( v_{i_k} \in IS(v_{i_{k+1}}) \) for \( k=1,2,\ldots,(j-1) \) and
2. \( v_{i_1} = v_{i_j} \).

We may consider the sequence as an ordered set where \( v_{i_1} \) is the first node entered when control is transferred into the cycle, and \( v_{i_{j-1}} \) is the last node executed before going back to \( v_{i_1} \). We will denote a cycle \( CY \) as an ordered set

\[ CY = \{ v_{i_1}, v_{i_2}, \ldots, v_{i_{j-1}} \}, \]

and call \( v_{i_1} \) an entry node. If a cycle has more than one entry node, then the representation of the cycle
is not unique, because we may have different starting points. We will restrict our discussion only to single entry cycles.

A Loop contains all cycles with the same starting point and ending point. Since all cycles have a single entry node, a loop also has a single entry node.

The program in Fig.3.1 has two nested loops. Since statements $S_4$ through $S_8$ form an if statement, the inner loop consists of two distinct cycles:

$$ CY_1 = \{B_2, B_3, B_4, B_6\}, $$

and $$ CY_2 = \{B_2, B_3, B_5, B_6\}, $$

and so does the outer loop:

$$ CY_3 = \{B_1, B_2, B_3, B_4, B_6, B_7\}, $$

and $$ CY_4 = \{B_1, B_2, B_3, B_5, B_6, B_7\}. $$

Let $L_i$ denote the set of nodes in loop number $i$. If a loop contains inner loops, then those nodes in inner loops will be represented only by their entry nodes. In the above example, if the outer loop is loop 1 and the inner loop is loop 2, then

$$ L_2 = \{B_2, B_3, B_4, B_5, B_6\}, $$

and $$ L_1 = \{B_1, B_2, B_7\}. $$

Since $B_2$ is the entry node of $L_2$, it is used in $L_1$ to represent all nodes in $L_2$. We will use $LE_i$ to denote the loop entry node of loop $i$. Thus in the above example, $LE_2 = B_2$ and $LE_1 = B_1$. 
Loop Nesting

Since our underlying assumption is that an inner loop will be executed more often than its outer loops, it is important to identify the nesting level of loops.

Definition 3.3 Loop \( j \) is an inner loop of loop \( i \) if (1) \( LE_j \) is in loop \( i \) and \( LE_j \neq LE_i \) or (2) \( LE_i = LE_j \) and each \( CY_z \) in loop \( j \) is also enclosed in any cycle, \( CY_x \), in loop \( i \); that is, \( CY_z \subseteq CY_x \).

Fig. 3.2(a)-(d) show the nesting of loop \( j \) in loop \( i \) satisfying condition (1) of the above definition. Fig. 3.2(e) shows the nesting satisfying condition (2) above. Fig. 3.2(f) is a special case where \( LE_i = LE_j \) but neither one is completely enclosed in the other. The nesting relationship in such a case will be determined as follows:

1. Let \( X_i \) be the set of nodes in loop \( i \) but not in loop \( j \), and \( X_j \) be the set of nodes in loop \( j \) but not in loop \( i \). If \( X_i \) contains a loop exit node and \( X_j \) does not, then loop \( j \) is an inner loop of \( i \).

2. If neither \( X_i \) nor \( X_j \) contains a loop exit node, or both contain a loop exit node, then the loop that ends first in the source program is the inner loop.

The loop nesting relationships can be represented by a tree structure. Let the part of a program that is not
Fig. 3.2 Loop nesting patterns

(a)  
(b)  
(c)  
(d)  
(e)  
(f)
in any loop be the root of a tree, and all loops that are not enclosed in any other loop be its immediate successors in the tree. For each of these nodes, i, let all loops that are enclosed only in loop i be its immediate successors; apply this process repeatedly. Eventually, those nodes with no immediate successors are the inner most loops. If the root of the tree has a nesting level of 0, all of its

Fig. 3.3 A graph with three loops and its loop nesting tree
immediate successors have a nesting level of 1, and all immediate successors of nodes with nesting level 1 have a nesting level of 2, and so forth. The graph in Fig.3.3(a) shows a program with three loops. Fig.3.3(b) is a tree showing their nesting relationships.

In the following discussion of pagination algorithms, we assume that all cycles, and hence all loops, have been identified in a previous program flow analysis. There have been many articles on the identification of cycles, for example, see [Lowe 1, Tie 1, Wei 1]. If we restrict our attention to smooth programs [Fou 1], or well-formed programs [Pet 1], then the identification problem becomes trivial.

Pagination Algorithm P0

The algorithm is given in a pseudo-Algol language in this section; following is a brief description of the algorithm.

It is assumed that page size p is fixed and is given. LE is a set of loop entry nodes, and LE\_i denotes the entry node of loop i. Associated with each node v, \ln(v) indicates the inner-most loop in which node v is located.

The pagination algorithm traverses the graph along all possible paths, as each node v in a path is processed, it is assigned to the current page being filled, pg(v)
38

denotes the page number to which node v was assigned.

The algorithm traverses the graph in the following way. Beginning with a program entry node, the node being processed is assigned, and one of its immediate successors is selected as the next node to be processed. In addition, if a node has more than one immediate successor, it is pushed into a stack NS. The algorithm stops and backtracks when it reaches one of the following points in a path: a node with no immediate successors, a node with all of its immediate successors processed, or a node which is the end point of a path in a loop.

When backtracking takes place, another immediate successor of the node on top of the stack NS is selected and a new path is then traversed. When all immediate successors of the top node of NS have been processed, the node is popped up, an immediate successor is selected from the new node on top of NS and the traversing continues. The algorithm stops when all nodes are processed.

Whenever a new path is taken, if the page being filled is not full, then it is added to a list of partially filled pages, U. As many as possible of those pages in U that contain nodes in the paths originating from the top node of NS are merged together.

If the node being processed is a loop entry node, the loop number is pushed into a stack LS and the size of the
loop is checked to decide whether to assign nodes in the loop a new page or to the current page. When all nodes in a loop have been processed, LS is popped, and an attempt is made to merge the partially filled pages in U that contain nodes in the loop.

Each entry in the stack NS is an ordered pair (u,v), where v is a node number and u points to the last page being added to U at the time v was pushed into NS. u is used to indicate that pages from the (u+1)th entry to the last entry in U contain nodes in the paths originating from node v, and that these pages are to be merged on backtracking to node v. The pair of numbers on top of NS are denoted by NS.U_{tn} and NS.N_{tn}.

For the same reason, each entry in the stack LS contains a triple, (u,n,l), where l is a loop number, and u,n are pointers to the top of U and NS respectively at the time loop l was pushed into LS. The triples on top of LS are denoted by LS.U_{ts}, and LS.NS_{ts} and LS.L_{ts}.

It is assumed that the size of a given node is not greater than the page size. If a basic block is larger than a page, it can always be divided into smaller sub-blocks and be represented by several nodes on the graph. It is further assumed that nodes are divisible, and hence a node may be divided and assigned to different pages.
Algorithm P0

procedure P0;

begin comment All variable declarations are omitted;

procedure ASSIGN(x);

begin comment Assign node x to page cp;

if \(|Pcp| + wx \leq p\) then

begin

\(Pcp := Pcp \cup \{x\} \); \(|Pcp| := |Pcp| + w_x\)

end

else begin

if all \(I_p(x)\) are assigned to page cp then

begin comment Divide node x;

Divide node x into two halves, \(x'\) and \(x''\)

such that \(|Pcp| + wx'_x = p\);

\(Pcp := Pcp \cup \{x'\} \); \(|Pcp| := |Pcp| + w_{x'}\);

\(v := x''\)

end

else \(v := x\);

comment Assign either \(x\) or \(x''\) to a new page;

\(cp := \text{pageno} \); \(\text{pageno} := \text{pageno} + 1\);

\(Pcp := \{v\} \); \(|Pcp| := w_v\)

end

end ASSIGN;

procedure MERGE(k,j);

begin comment Merge page j to page i if possible;

if \(|P_i| + |P_j| \leq p\) then

begin

\(P_i := P_i \cup P_j\); \(|P_i| := |P_i| + |P_j|\);

\(P_j := \Lambda\); \(|P_j| := 0\); Delete page j from U;

Change the page number of nodes assigned to j;

if \(j = \text{pageno}\) then \(\text{pageno} := \text{pageno} - 1\);

if \(|P_i| = p\) then

begin

Delete page i from U;

if \(i = \text{pageno}\) then \(\text{pageno} := \text{pageno} - 1\)

end

end end MERGE;
procedure CLEANUP(k);
begin comment Add a page j to U if \(|P_j| < \rho\) and \(j \not\in U\);
    Add page cp to U if necessary;
    comment Try to merge the last \((\tau u-k)\) pages in U;
    for \(j := \tau u - 1\) until \(k+1\) do
        for \(x := j + 1\) step -1 until \(k+1\) do
            call MERGE(U_x, U_j)
end CLEANUP;

procedure LOOPENTRY(x);
begin comment Process a loop entry node;
L1: Select an IS \((x)\) which is the entry node of an inner
    loop of the current loop \(\ell \ell\) according to the priority
    rule, let it be \(b\);
    if \(cp \neq pg(x)\) then
        L2: begin Add page cp to U if necessary; \(cp:=pg(x)\) end;
    Let \(\ell_1, \ell_2, \ldots, \ell_h\) be the loops having node \(b\) as their
    loop entry node in the increasing order of loop nesting
    levels;
    if \(|P_{cp}| + |L_{\ell_h}| > \rho\) then
        begin
            if loop \(\ell_h\) has only one path then
                if \(|L_{\ell_h}| \mod \rho + |P_{cp}| < \rho\) then goto L4;
            Add pg(x) to U if necessary;
            comment Start a new page;
            L3: \(cp :=\text{pageno}; \text{pageno} := \text{pageno} + 1\)
        end;
    comment Push loop numbers into stack LS;
    L4: for \(k := \ell_1, \ell_2, \ldots, \ell_h\) do push \((\tau u, \tau n, k)\) into LS;
    \(\ell \ell := \ell_h; \ cn := b\)
end LOOPENTRY;
comment The main routine of P0 starts here;

I: Set stacks NS and LS, lists U,AN and EN empty;
pageno := 1;

comment tn, ts, tu, ta and te are pointers to the top or
the last entry of NS, LS, U, AN and EN respectively.
When they are empty, tn, ts, tu, ta, te = 0. Any operation
(addition of deletion) on stacks or lists implies an
adjustment of the corresponding pointer;

II: for x := each node in PE do
begin comment PE is the set of program entry nodes;
  cn:=x; cp:=pageno; pageno:=pageno+1; c£:=0;
  Push (tu, tn, c£) into stack LS;

III: call ASSIGN(cn);

IV: if |IS(cn)| = 0 then goto VII;

comment Check if any IS(cn) has been processed;
for b := each IS(cn) do
if b has been assigned then
  if b ∈ L_c£ \ cp ≠ pg(b) then call MERGE(pg(b), cp);
if |IS(cn)| > 1 then push (tu, cn) into stack NS;

V: comment Select next node to be processed from IS(cn);
if all IS(cn) have been assigned then
V1: begin if cn = NS.Ntn then pop NS; goto VII end;
if there exists an assigned node in IS(cn) which is
also in L_c£ then
V2: begin cn := an IS(cn) in L_c£ selected according
to the priority rule;
  if cp ≠ pg(NS.S ts) \|P cp| < p then
    begin call CLEANUP(NS.U ts); cp:=pg(NS.N ts) end;
goto III
  end;
VI: comment Either none of unassigned nodes in IS(cn) is in $L_{cl}$, or some IS(cn) are the entry node of inner loops of loop $cl$. In the former case, add cn to AN, in the latter case, add cn to EN;
    if cn=NS.Ntn then pop NS;
    if there exists an unassigned node in IS(cn) which is the entry node of an inner loop of $cl$ then
        add cn to EN;
    else add cn to AN;

VII: comment Any node in $L_{cl}$ still in stack NS?;
    if tn>LS.NS.ts then
        begin comment Yes, NS.Ntn $\in L_{cl}$. Loop $cl$ not done;
            cn := NS.Stn; goto V
        end;

VIII: if EN not empty and $\exists b \in EN$ then
    begin comment Select an inner loop of $cl$;
        b := EN.te; Delete b from EN;
        push (LS.Uts,b) into NS; call LOOPENTRY(b);
        goto III
    end;
    for K := AN.ta,AN.ta-1,...,AN, while k $\in L_{cl}$ do
        begin push (LS.Uts,k) into NS; Delete k from AN end;

IX: call CLEANUP(LS.Uts);
    if ts>1 then
        begin pop stack LS; $cl := LS.Lts$
            if NS empty then goto VIII
        end;
    if NS not empty then goto VII;
    comment If control gets here, it is the end of all paths from the last program entry node x. Try next node in PE;

end II

end P0
The Priority Rule

In step IV and procedure LOOPENTRY, when several nodes in the set of immediate successors can be chosen for the next node to be processed, a priority rule is needed to decide which node to process first. There are three possible rules:

1. **Highest Probability**: The node with highest probability of being executed is selected first. Associated with each node $i$, let $Pr(i,j)$ be the conditional probability of control being transferred from node $i$ to $j$, given that node $i$ has been entered. If $j \in IS(i)$ then $Pr(i,j) \neq 0$, otherwise, $Pr(i,j)=0$. Each time when a decision is to be made, select the node with highest $Pr(i,j)$ among all eligible candidates. The goal is to minimize the number of pages along high probability paths. One major obstacle is that the probabilities, $Pr(i,j)$, are often not known at compilation time.

2. **Shortest Path**: The node in the path that requires the smallest amount of memory space is selected first. Let $s(i) = \min_{X \in X(i)} \left\{ \sum_{k \in X} w_k \right\}$ for any node $i$ in the graph where $X(i)$ is the set of all paths from node $i$ to a program exit node (i.e. a node with no immediate successors). $s(i)$ can be computed by the following procedure. Starting from a program exit node $v$, let $s(v)=w_v$. For each $j \in IP(v)$,
s(j) = \min_{k \in IS(j)} [s(k)] + w_j; \text{ if not all of } s(k), k \in IS(j) \text{ have been computed, then wait until all of them are computed.}

When a decision is to be made, select a node k whose s(k) is the smallest among all candidates. Use of this rule guarantees that there exists a path with a minimum number of pages. For example, assume Fig.3.4(a) is a page assignment given by the algorithm using the Shortest Path rule, and assume Fig.3.4(b) is another page assignment given by another algorithm. In Fig.3.4(a), whenever the loop is repeated, in the best case, it will require only one page, and in the worst case, 3 pages; while in Fig.3.4(b), it will always require 3 pages, regardless of which path is taken. However, care must be taken or nodes in other paths may be scattered in different pages. Use the following program segment as an example and assume that a, c, e and g are assigned to the same page. Nodes b, d and f may be assigned to any other pages, but PO tries to assign them to the same page (if possible).

```plaintext
if a then b
if c then d
if e then f
  g
```

\[ .
. 
. 
]
Note that if this rule is to be used, it requires another scanning of nodes in the graph to compute $s(i)$ for all $i$.

3. **Earliest Appearance**: The node that appears first in the program is selected first. This rule requires no additional overhead but tends to give a poorer assignment than Shortest Path rule as we can see from Fig.3.4(b).

In the following discussions, the second rule, Shortest Path will be used. We will further present the effects of Shortest Path and Highest Probability rules in the section concerning paging performance.
Fig. 3.4 Two different page assignments

Fig. 3.5 A page assignment using pagination algorithm PO
An Example

The graph in Fig. 3.3(a) will be used as an example to show how the algorithm works. A detailed step by step trace is given in Table 3.1. Fig. 3.5 shows the final assignment of pages for page size $p=64$ words. Nodes 2 and 3, node 6, and node 8 are originally assigned to three different pages, they are merged to page 2 in a clean up process when all nodes in loop 1 have been processed. Note that when loop 3 is processed, node 11 is assigned to page 1 instead of a new page, because the loop has only one path, and it requires two pages regardless of whether node 11 is assigned to a new page or not. Node 12 is divided into two parts (the first part containing 3 words, and the second part containing 61 words), the first and second half are denoted by $\text{12}^1$ and $\text{12}^2$ respectively in Table 3.1. The size, loop number and page number of nodes on the graph are summarized as follows:

<table>
<thead>
<tr>
<th>node $i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>$12^1$</th>
<th>$12^2$</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_i$</td>
<td>35</td>
<td>5</td>
<td>30</td>
<td>8</td>
<td>30</td>
<td>10</td>
<td>26</td>
<td>8</td>
<td>6</td>
<td>11</td>
<td>5</td>
<td>3</td>
<td>61</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>$\text{ln}(i)$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>$\text{pg}(i)$</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>
Table 3.1 A Step by Step Trace of an Application of Algorithm P0

<table>
<thead>
<tr>
<th>Steps</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Steps</td>
<td>Actions</td>
</tr>
<tr>
<td>I</td>
<td>NS,LS,U,AN,EN=A,pageno=1</td>
</tr>
<tr>
<td>II</td>
<td>cn=1, cp=1, pageno=2, ci=1, LS={(0,0,0)}</td>
</tr>
<tr>
<td>III,A,IV</td>
<td>P1={1},</td>
</tr>
<tr>
<td>V,V2,III,A,IV</td>
<td>ch=10, P2={(1,10)},</td>
</tr>
<tr>
<td>V,V2,III,A,IV</td>
<td>cn=14, P3={(1,10,14)},</td>
</tr>
<tr>
<td>V,VI, VII,</td>
<td>VI,EN={(0,1)},</td>
</tr>
<tr>
<td>V,VI,VII, VIII</td>
<td>NS=Λ,</td>
</tr>
<tr>
<td>L1,L3,L4</td>
<td>U={1}, cp=2, pageno=3, LS={(0,0,0),(1,1,1)}, ci=1, cn=2</td>
</tr>
<tr>
<td>III,A,IV</td>
<td>P2={(2)},</td>
</tr>
<tr>
<td>V,V2,III,A,IV</td>
<td>ch=3, P3={(2,3)},</td>
</tr>
<tr>
<td>V,VI, VII, VIII</td>
<td>EN={(10,4)}, cn=2</td>
</tr>
<tr>
<td>V,VI, VII, VIII</td>
<td>NS={(0,1)}, AN={(2), b=4,</td>
</tr>
<tr>
<td>III,A,IV</td>
<td>P3={1},</td>
</tr>
<tr>
<td>V,V2,III,A</td>
<td>cn=5, P3={(4,5)},</td>
</tr>
<tr>
<td>IV,V,V2</td>
<td>NS={(0,1),(1,3),(2,4),(2,5)}, cn=7</td>
</tr>
<tr>
<td>III,A,IV,V,V1</td>
<td>P3={(4,5,7)},</td>
</tr>
<tr>
<td>VII,V,V2</td>
<td>cn=5, cn=6</td>
</tr>
<tr>
<td>III,A,IV,M</td>
<td>cp=4, pageno=5, P4={(6)},</td>
</tr>
<tr>
<td>V,VI, VII</td>
<td>NS={(0,1),(1,3),(2,4)}, cn=4</td>
</tr>
<tr>
<td>V,VI, VII</td>
<td>NS={(0,1),(1,2)}, AN={2,4}</td>
</tr>
<tr>
<td>VIII,IX,C</td>
<td>NS={(0,1),(1,3),(2,4),(2,2)}, AN=Λ, CLEANUP(2), LS={(0,0,0),(1,1,1)}, ci=1</td>
</tr>
<tr>
<td>VII,V,VI, VII</td>
<td>cn=2, NS={(0,1),(1,3),(2,4)}, AN={(2), cn=4</td>
</tr>
<tr>
<td>V,V2,C</td>
<td>cn=8, CLEANUP(2), U={(1,2,4)}</td>
</tr>
<tr>
<td>III,A,IV,M</td>
<td>cp=5, pageno=6, P5={(8)},</td>
</tr>
<tr>
<td>V,VI, VII</td>
<td>NS={(0,1),(1,3)}, cn=3</td>
</tr>
<tr>
<td>V,VI, VII</td>
<td>NS={(0,1),NS={(0,1),(1,2)}, AN=Λ</td>
</tr>
<tr>
<td>IX,C</td>
<td>CLEANUP(1),</td>
</tr>
<tr>
<td>cp=2, pageno=4, U={(1,2), LS={(0,0,0)}, ci=0</td>
<td></td>
</tr>
<tr>
<td>VII,V,V2,III,A</td>
<td>cn=2, cn=9, P3={(2,3,8,6,9)},</td>
</tr>
<tr>
<td>IV,VII, V,VI</td>
<td>cn=2, NS={(0,1)}</td>
</tr>
<tr>
<td>VII,V, VI, VIII</td>
<td>cn=1, NS=Λ, b=10,</td>
</tr>
<tr>
<td>L1,L2,L4</td>
<td>b=11, cp=1, LS={(0,0,0),(2,1,3)}, ci=3, cn=11</td>
</tr>
<tr>
<td>III,A,IV,V,V2</td>
<td>P1={(1,10,14,11)},</td>
</tr>
<tr>
<td>III,A,IV,V,V2</td>
<td>P4={(10,14,11,12)},</td>
</tr>
<tr>
<td>III,A,IV,M</td>
<td>P4={(12,13)},</td>
</tr>
<tr>
<td>VII,V,VI, V,VI</td>
<td>cn=10, NS=Λ</td>
</tr>
<tr>
<td>VII,IX,C</td>
<td>CLEANUP(0)</td>
</tr>
<tr>
<td>VII,IX,C</td>
<td>CLEANUP(0)</td>
</tr>
<tr>
<td>I</td>
<td>STOP</td>
</tr>
</tbody>
</table>
Comparison with Other Pagination Algorithms

Similar pagination algorithms are reported in [Bar 1] and [Ver 1]. These algorithms work as follows: nodes in the inner most cycles are assigned first, and whenever the size of a cycle is greater than the space available in the page in which nodes are currently being assigned, a new page is started. Nodes are assumed to be non-divisible except that in [Bar 1], nodes in the inner most cycles may be divided. In [Bar 1], partially filled pages are merged if any two nodes in two different pages are connected (i.e. one node is an immediate successor of the other), and if their total size is less than the page size. In [Ver 1] pages are denoted by the smallest node number assigned to the page, and partially filled pages are merged to any other pages in the ascending order of node numbers without considering the connectivity of nodes in the pages to be merged.

Fig.3.6(a) and (b) show the page assignments of the program on Fig.3.3(a) with a page size of 64 words, using algorithms given in [Bar 1] and [Ver 1] respectively.

The following is a comparison of the page fault rate under three different page assignments, P0, [Bar 1] and [Ver 1]. If the left path is taken, and if loop 1 is executed 50 times, while loop 2 is executed 200 times
(a) Page assignment given by Bar 1.

(b) Page assignment given by Ver 1.

Fig. 3.6 Page assignments using algorithms given in Bar 1 and Ver 1.
taking the path 4,5,7 and 4,5,6,7 alternatively, then the reference strings and page interreference distances, \( d(t) \), can be shown as follows:

\[
\begin{align*}
\text{P0} : & \quad 1-2-3-2-3-2-3- \quad \text{50 times} \\
\text{d}(t) : & \quad \{3\ \ 3\ \ 3\ \ 1\ \ 1\ \ 1\ \ 1\ \} \\
\text{Barl} : & \quad 1-2-3-4-3-4-3-4-3- \quad \text{50 times} \\
\text{d}(t) : & \quad \{4\ \ 4\ \ 4\ \ 4\ \ 1\ \ 1\ \ 1\ \ 1\ \} \\
\text{Verl} : & \quad 1-2-3-2-3-2-3-1- \quad \text{50 times} \\
\text{d}(t) : & \quad \{3\ \ 3\ \ 3\ \ 1\ \ 1\ \ 1\ \ 1\ \ 2\ \} \\
\end{align*}
\]

The first line of each \( d(t) \) is the distances in the initial pass of loop 1, and the second line is the distances after the first pass. The number of page faults are shown in the following table:

<table>
<thead>
<tr>
<th>Memory size (page)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>301</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Bar 1</td>
<td>401</td>
<td>151</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Ver 1</td>
<td>351</td>
<td>151</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

The Simulator

In order to test the effects of pagination algorithms on the paging performance of programs, a simulator has been written to simulate the execution of any arbitrary programs with single entry loops.

There are three inputs to the simulator: graphs representing routines in the program to be simulated, a parameter which controls the loop execution frequency,
and the number of times the program is to be executed.

Each subroutine in the program is called a module and is represented by a directed graph. For each node in the graph, the following information is provided.

1. Number of instructions in the node (we assume these instructions are executed sequentially).

2. The name of the subroutine (the module number), if any, referenced in this node.

3. The page number to which the node is assigned.

4. A set of immediate successors and the conditional probability of a particular immediate successor being selected after exit from the node.

We assume that each node references at most one subroutine. Module number 1 is considered as the main routine of the program.

The loop control parameter is of the following two types.

1. Each time a loop is entered, the number of times the loop will be executed is specified as a constant multiple of its enclosing loop's execution frequency. If a loop is not enclosed in any loop, then its execution frequency is a number selected randomly between zero and a maximum number. The multiple and the maximum are given as inputs. Note that under this type of control, the program must be a well formed program [Pet 1] so that
loops have a single entry node and that the loop nesting level can be determined.

2. The loop execution frequency is determined by the probability of going back to the loop entry node from a loop exit node. The probability is given in the graph as input.

The simulator starts "executing" the program by selecting the first node of module 1. For each node, the page number is fetched, and the page interreference distance is computed. If a node references a subroutine, then a branch to that routine is taken. After the node is executed or after returning from the subroutine, an immediate successor node is selected according to the probability given, and the simulated execution continues. If the loop control parameter is of the first type, the selection of an immediate successor node for the loop exit node is more complicated than the second type. Upon entering a loop, the number of times the loop is to be executed is computed, then every time a loop exit node is encountered, if the loop has not been executed the required number of times, an immediate successor node which is inside the loop is selected; otherwise, an immediate successor node which is outside the loop is selected.

The simulator halts when it reaches a node in module 1 having no immediate successor; it then computes the
page fault rate under LRU replacement for all memory sizes and prints out the statistics. The execution of the program is repeated a given number of times, or until the change in average page fault rate falls below a preset number.

The Paging Performance

Several programs consisting of a single module are simulated, Fig.3.7 represents one of these programs. Numbers inside the circles are node numbers, numbers beside the nodes are node sizes, and numbers beside the edges are conditional probability of control transfers.

The page assignments for page sizes of 256-bytes using Sequential Assignment, and algorithm P0 with Shortest Path and Highest Probability priority rules are shown in Fig.3.8, 3.9 and 3.10 respectively.

Since in Chapters 3 and 4 we are primarily interested in the page assignment of the instruction part of a program, it is assumed that the instruction and data parts are considered separately by the paging system. Only the page fault rate of the instruction part will be presented in the next two sections.

The estimated page inter-reference distance density function, \( \hat{f}(x) \), for different page assignments is shown in Fig.3.11. Under Sequential Assignment, 95.9% of references will be in the same page, but most of the
Fig. 3.7 A test program. Numbers on the right of nodes are node sizes, and numbers on edges are probability of transfers.
Fig. 3.8 Sequential page assignment of the program in Fig. 3.7. Page size = 256 bytes.
Fig. 3.9 Packed assignment using PO under the shortest path priority rule. Page size = 256 bytes.
Fig. 3.10 Packed assignment using PO under the highest probability rule.
Page size = 256 bytes.
remaining 4.1% will have a page interreference distance of 2 to 6 pages, with 4 pages having the highest probability, 2%.

As we have stated in Chapter 2, our goal is to move the curve closer to the Y-axis, and to lower the height of the right tail of the curve. Those curves marked Shortest Path and Highest Probability in Fig. 3.11 indicated that indeed the optimization moved the curve in the direction we hoped. Only 0.8% of references have a page interreference distance longer than 4 pages under both Shortest Path and Highest Probability assignment.

Fig. 3.12 shows the page fault rates under LRU replacement, and Fig. 3.13 shows the relative paging performance based on the ratio of page fault rate between Sequential and Shortest Path assignment. Note that the program requires 12 pages under Sequential and 13 pages under both Shortest Path and Highest Probability assignment. Sequential assignment has a higher page fault rate than either Shortest Path or Highest Probability assignment for all memory sizes except 1 and 12 pages. Initially, the page fault rate of Sequential assignment is 1.5 times higher than that of Shortest Path assignment when memory size is 2 pages. It increases to 3 times when memory size is 3 pages (Fig. 3.12). As the memory size increases, the effectiveness of optimization decreases. When memory
Fig. 3.11 Estimated page interreference density function of the program in Fig. 3.7 under different page assignments. Page size = 256 bytes.
Fig. 3.12 Average page fault rate of the program in Fig. 3.7 under different page assignments. Page fault rate is computed under LRU replacement for page size of 256 bytes.
Fig. 3.13 Relative paging performance of the program in Fig. 3.7 under different page assignments. Page fault rate under Sequential or Highest Probability assignment
Ratio = Page fault rate under Shortest Path assignment
Page size = 256 bytes.
size is large enough to contain all of the required pages for efficient execution of the program, Sequential assignment will generate a lower page fault rate, because it divides the program into a smaller number of pages than any other assignment method. It is clear that when there is sufficient memory space available, optimization of page assignment is not necessary, but when the memory size is small, optimization does improve the paging performance.

It is expected that the page assignment using the Highest Probability rule will generate fewer page faults than that using the Shortest Path rule. Fig.3.12 shows that this is indeed the case for memory size greater than 6 pages. However, with smaller memory size, the Shortest Path rule may even generate less page faults because there always exists a path in a loop with a minimum number of pages such that they may all be kept in the memory. It can be concluded that if a program's behavior satisfies our assumption that an inner loop will be executed more often than any other loop having a lower nesting level, then the Shortest Path rule is as good as the Highest Probability rule.

Page Size and Page Assignment

In Chapter 2, we have discussed the effects of changing page size on paging performance under Sequential
assignment. In general, smaller page size decreases the external fragmentation (space wasted on the partially filled page) and the superfluous space (space not referenced when the page is in memory), and thus decreases the page fault rate. We also discussed that when a program is very well packed, reducing the page size may increase the page fault rate. This is true when the page assignment of the program on Fig.3.7 is optimized using different page sizes. In Fig.3.14, a 512-byte page size has lower page faults than the 256-byte page size, and the 256-byte page size in turn has lower page faults than the 128-byte page size.

Fig.3.15 shows the changes in the effectiveness of the optimization of page assignment for different page sizes. The curves show the ratio between the average page fault rate of Sequential and Shortest Path assignment. The ratio is higher for larger page sizes: the ratio for the 512-byte page size is higher than that for the 256-byte page size; this is also true for the 256-byte page size vs. the 128-byte page size.

Optimization of page assignment tries to minimize the superfluous space and hence increases the utilization of memory by packing related basic blocks into the same page. Since there is less superfluous space to start with, optimization for a smaller page size (relative to
the sizes of basic blocks) is less effective than for larger page size. When page size is large, there is more space for the optimization process to assign more related blocks to a given page, but under Sequential assignment, a larger page size increases the probability that unrelated blocks will be assigned to the same page. Therefore, the improvement of paging performance under the optimized assignment over the Sequential assignment for larger page sizes will be greater than that for smaller page sizes. However, such improvement has a limit, for optimization increases the number of partially filled pages while Sequential assignment has only one partially filled page, the last page in the program.
Fig. 3.14 Page fault rate of the program in Fig. 3.7 under Shortest Path assignment for different page sizes.
Fig. 3.15 Relative paging performance of Sequential and Shortest Path assignment for different page sizes.
Chapter 4 Assignment of Subroutines to Pages

In this chapter we discuss the problem of assigning subroutines to pages during the compilation of a program, or the problem of "subroutine packing". We are interested in large programs which will be run very often. Normally these programs are written in a modular fashion in which programs are divided into subroutines or modules. There are two ways to treat these subroutines: one way is to apply the concept of segmentation [McC 1, Dens 1] to each of these subroutines and consider it an independent segment, and let the system loader link them dynamically during execution time; the other way is to link all these subroutines into a single segment at compilation time. The second approach is applicable under the following conditions:

1. Execution time linkage causes excessive external memory fragmentation [Ran 1], because segments are not multiples of the page size. This is especially true when many segments are referenced, or when segment sizes are less than the page size.

2. Modules or subroutines are not to be shared by other programs.

3. All modules or subroutines are to be compiled at the same time.
Condition 2 and 3 are not serious restrictions because we can always make another copy of sharable routines available for use by other programs, and consider all subroutines referenced by the program to be optimized as the internal routines of that program.

In the past, the page assignments in a segment have been done sequentially in the order in which statements and routines are compiled. We have pointed out in previous chapters that this is not the best assignment. In this chapter, we will discuss the problem of improving page assignment within a segment containing a number of modules.

The Subroutine Packing Problem

Let $G = \{1,2,\ldots,n\}$ be the set of nodes in a tree representing the inter-routine reference relationships of a program. Nodes in $G$ represent routines in the program, and an edge from node $i$ to node $j$ indicates that routine $i$ calls routine $j$ and implies a return from $j$ to $i$. We will call this tree a subroutine reference tree. The root of the tree is the main routine of the program, and the leaves represent routines which reference no other routine. Fig.4.1 represents a program consisting of 23 routines.

The size of node $i$, $w_i$, represents the size of routine $i$, which may include both instructions and data
Fig. 4.1 A subroutine reference tree with 23 routines. The number beside each node is the size of the node in 100 bytes.
in the routine. However, in this chapter, we assume that nodes represent only the instruction part of the routines, and that the compiler divides the memory space into the instruction part and the data part. By optimizing the page assignment of the instruction part, there will be more memory space available for data pages during execution time, thus reducing the overall page fault rate.

**Definition 4.1  The subroutine packing problem**

Let $G = \{1,2,\ldots,n\}$ be the set of nodes in a subroutine reference tree, then given the page size $p$, and memory size of $M$ pages, the subroutine packing problem is to partition the tree into $m$ subtrees $P_i$, $i = 1,2,\ldots,m$ such that

1. $G = \bigcup_{i=1}^{m} P_i$;
2. $|P_i| \leq p$ for $i = 1,2,\ldots,m$, where $|P_i| = \sum_{k \in P_i} w_k$;
3. The expected page fault rate is minimum for the given memory size of $M$ pages, $M \leq m$.

We make no restriction on the nodes in $P_i$: that is, nodes in $G$ may appear in more than one page, or may be divided into several subnodes and put into different pages. Although it is not absolutely necessary that the size of nodes be less than the page size $p$, we assume this is true for most of the nodes in the tree. If the size of each node in the tree is greater than $p$ we should
consider the problem of pagination as well as subroutine packing.

As we have discussed in Chapter 2, it is difficult to find an efficient optimal solution for such a problem, for lack of information on the execution frequency of programs. In the next section a subroutine packing algorithm which uses only the loop nesting level of a program is presented.

Subroutine Packing Algorithm P1

Algorithm P1 is based on the following two assumptions about the structure of programs:

1. Each routine is referenced by only one other routine.

2. Let c(z,x) be the highest nesting level in routine z in which routine x is referenced, then if c(z,x) > c(z,y) for some routine y, routine x will be referenced more often than routine y, regardless of whether y and x are referenced in the same loop or not.

In Theorem 4.1 we add three more assumptions and claim that algorithm P1 will give an optimal page assignment for such a subset of programs for memory size of one page. The three assumptions are:

1. Each routine is referenced by only one other routine at only one place.
2. There is no conditional reference, all routines will always be referenced.

3. Let \( k \) be the nesting level of a loop in routine \( x \), then the reference frequency of any routine referenced in this loop is higher than the total reference frequencies of all other routines referenced in loops in \( x \) whose nesting level is less than \( k \).

With such a restriction, the problem is similar to a graph partitioning problem discussed in [Ker 1], hence algorithm \( P_1 \) is an extension of an algorithm given in [Ker 1], and Theorem 4.1 is also a generalization of a theorem in [Ker 1].

In Theorem 4.2 we also claim that if we further restrict the set of programs that we will optimize, then algorithm \( P_1 \) gives an optimal page assignment for all possible memory sizes.

The following is an intuitive explanation of algorithm \( P_1 \). A list of loops in all routines is assumed to have been created at an earlier stage of program analysis, and for each loop a list of subroutines referenced inside the loop is kept. The first part of the algorithm is to build a reference tree whose edges are ordered from left to right in the descending order of nesting level of loops in which routines are referenced. The second part is to assign nodes to pages from bottom
up and from left to right. It begins with the left most leaf of the tree and "climbs" upward toward the root of the tree, assigning nodes to pages sequentially if each of these nodes has only one immediate descendant. If a node \( y \) has more than one immediate descendant, the algorithm starts with a new page on a new leaf and repeats the same process until all edges of \( y \) have been assigned. Node \( y \) is then assigned to a page, and this page and all unfilled pages containing the immediate descendants of \( y \) are merged together, if possible, according to their nesting level. This process is repeatedly applied to each subtree until all nodes in the tree are assigned. Algorithm PI is presented below in a pseudo-Algol language. A flowchart of the main routine is given in Fig.4.2.
procedure P1;
begin
I: begin comment Section to build a subroutine reference tree. T is a queue containing terminal nodes of the tree being built, it is represented as an ordered set. T' is a stack containing nodes whose successor nodes are yet to be added;
procedure BUILDTREE(x);
comment Procedure to add subroutines referenced by x to the subroutine reference tree;
begin comment The physical representation of the tree will not be given here;
B1: Add all routines referenced by x as branches (immediate descendants) of x from left to right in the descending order of the loop nesting levels, c(x,y), where y is referenced by x;
Comment For routines referenced at the same nesting level, the routine with highest static reference frequency is placed on the left, if they still have equal static reference frequency, then the routine that appears first in the source program is placed on the left;
B2: Push these nodes into stack T' rightmost node first
end of BUILDTREE;
I2: for cn := top of T' while T' is nonempty do
begin Pop T';
I21: if cn calls another routine then call BUILDTREE(cn),
I22: else add cn to the queue T
end
end;
II: begin comment This section does page assignment;

procedure NEWPAGE(x);

begin comment NEWPAGE assigns node x to next available page;

    cp := pageno; pageno := pageno + 1;

comment pageno points to next available page, cp points to current page to which nodes being assigned;

    P_{cp} := \{x\}; \quad |P_{cp}| := w_x;

Mark x processed

end NEWPAGE;

procedure ASSIGN(x);

begin comment ASSIGN assigns node x to page cp;

    if |P_{cp}| + w_x \leq p then

        begin P_{cp} := P_{cp} \cup \{x\}; |P_{cp}| := |P_{cp}| + w_x;

            mark x processed

        end;

    else begin

        Divide node x into two halves at the last reference to IS(x) and let w_x be the size of the last half;

        if |P_{cp}| + w_{x'} \leq p then

            begin Assign the last half and as many instructions as possible from the end of first half to P_{cp} and update the size of P_{cp};

                call NEWPAGE(the un-assigned portion of x)

            end;

        else call NEWPAGE(x);

    end

end ASSIGN:

end
procedure MERGE(i, j);
begin comment MERGE page j to page i if possible;
    if |P_i| + |P_j| < p then
        M: begin P_i := P_i \cup P_j; |P_i| := |P_i| + |P_j|;
            P_j := \emptyset; |P_j| := 0; Delete page j from
            the list of unfilled pages;
            if |P_i| = p then delete page i from the
            list of unfilled pages;
            if pageno - 1 = j then pageno := j
        end
    end MERGE;
end MERGE;

procedure MERGIS(x);
begin comment MERGEIS(x) tries to merge unfilled pages
containing IS(x). let P_j, k = 1, 2, ..., r,
\[ \begin{array}{c}
\text{M}_j \text{ k}
\end{array} \]
be these pages, where k indicates the left to right
ordering of IS(x) in the tree.
MS1: Sort those pages having the same loop
    nesting level in the ascending order of
    their page size and relabel all pages
    P_j, k = 1, 2, ..., r, k' indicates the sorted order.
    \[ \begin{array}{c}
\text{M}_j \text{ k}'
\end{array} \]
MS2: if |P_{j_1}| + w_x < p then
    begin cp := j_1; call ASSIGN(x) end
else call NEWPAGE(x);
    comment start merging pages;
MS3: for k' = 2 step 1 until r do call MERGE(cp, j_k,);
MS4: for any two remaining unfilled pages,
    x and y, do call MERGE(x, y)
end MERGEIS;
comment Actual page assignment starts here. Process the terminal nodes in queue T in their left to right order;

\[ \text{pageno} := 1; \]

**III:** for \( \text{cn} := \) each element in T do

begin

\text{call NEWPAGE}(\text{cn});

**II2:** for \( \text{x} := \) IP(\text{cn}) while IP(\text{cn}) is nonempty do

begin

**II21:** if \( |\text{IS}(\text{x})| = 1 \) then call ASSIGN(\text{x});

else begin

**II22:** if \( |\text{P}_{\text{cp}}| < p \) then add \( \text{P}_{\text{cp}} \) to list of unfilled pages

**II23:** else if all IS(\text{x}) have been processed then call MERGEIS(\text{x});

**II24:** else go to II3;

end;

**II25:** \( \text{cn} := \text{x}; \)

end;

**II3:** end

end of assignment section;

end of P1;
Set queue T empty, push the main module into T' and let it be the root of the tree.

Add cn to T. No cn references other routine? Yes

BUILD_TREE(cn)
Add all routines referenced by cn to the tree and order these nodes from left to right in the descending order of their loop nesting levels. Push them into T' right most node first.

STOP T empty

NEWPAGE(cn)
Assign cn to a new page.

x ← next IP(cn). IP(cn) empty
yes No. of immediate descendants of x is 1?

ASSIGN(x)
Assign x to P(cp) and mark x processed.

[If IP(cp) empty]

MERGEIS(x)
Assign node x and try to merge all unfilled pages containing IS(x).

Fig. 4.2 A flowchart of algorithm P1
An Example

Fig. 4.3 shows a subroutine reference tree constructed by algorithm P1. The numbers beside the edges are nesting levels, $c(x, y)$. Node numbers also represent the sequence in which nodes are added to the tree. A step by step trace is given in Table 4.1. The symbol $U$ in the table contains the list of unfilled pages. The page assignments given are for page size of 3 ($p=3$), the final page assignment pages are $P_1 = \{2, 4, 5\}$, $P_3 = \{1, 6\}$, and $P_4 = \{3, 7, 8\}$.

![Fig. 4.3 A subroutine reference tree and its page assignment for $p = 3$. Node sizes $w_i = 1$, $i=1, 2, \ldots, 8$.](image-url)
Table 4.1 A step by step trace of an application of algorithm PI

<table>
<thead>
<tr>
<th>Steps (labels)</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>( T = \Lambda, T' = {1} ), tree:</td>
</tr>
<tr>
<td>I2</td>
<td>cn=1, ( T' = \Lambda )</td>
</tr>
<tr>
<td>I21,B1,B2</td>
<td>( T' = {2,3} ), tree:</td>
</tr>
<tr>
<td>I2</td>
<td>cn=2, ( T' = {3} )</td>
</tr>
<tr>
<td>I21,B1,B2</td>
<td>( T' = {4,5,6,3} ), Tree:</td>
</tr>
<tr>
<td>I2</td>
<td>cn=4, ( T' = {5,6,3} )</td>
</tr>
<tr>
<td>I22</td>
<td>( T = {4} )</td>
</tr>
<tr>
<td>I2,I22</td>
<td>cn=5, ( T' = {6,3}, T = {4,5} )</td>
</tr>
<tr>
<td>I2,I22</td>
<td>cn=6, ( T' = {3}, T = {4,5,6} )</td>
</tr>
<tr>
<td>I2</td>
<td>cn=3, ( T' = \Lambda )</td>
</tr>
<tr>
<td>I21,B1,B2</td>
<td>( T' = {7,8} ), Tree:</td>
</tr>
<tr>
<td>I2</td>
<td>cn=7, ( T' = {8} )</td>
</tr>
<tr>
<td>I2,I22</td>
<td>cn=8, ( T' = \Lambda, T = {4,5,6,7,8} )</td>
</tr>
<tr>
<td>II</td>
<td>pageno=1</td>
</tr>
<tr>
<td>I11,N</td>
<td>cn=4, cp=1, pageno=2, ( P_1 = {4},</td>
</tr>
<tr>
<td>I12,I122,I124</td>
<td>x=2, ( U = {P_1} )</td>
</tr>
<tr>
<td>I13,I11,N</td>
<td>cn=5, cp=2, pageno=3, ( P_2 = {5},</td>
</tr>
<tr>
<td>I12,I122,I124</td>
<td>x=2, ( U = {P_1, P_2} )</td>
</tr>
<tr>
<td>I13,I11,N</td>
<td>cn=6, cp=3, pageno=4, ( P_3 = {6},</td>
</tr>
<tr>
<td>I12,I122,I123</td>
<td>x=2, ( U = {P_1, P_2, P_3} )</td>
</tr>
<tr>
<td>MS1,MS2,M</td>
<td>cp=1, ( P_1 = {4,2},</td>
</tr>
<tr>
<td>MS3,M</td>
<td>( P_1 = {4,2,5},</td>
</tr>
<tr>
<td>I125</td>
<td>cn=2</td>
</tr>
<tr>
<td>I12,I124,I13</td>
<td>x=1</td>
</tr>
<tr>
<td>I11,N</td>
<td>cn=7, cp=4, pageno=5, ( P_4 = {7},</td>
</tr>
<tr>
<td>I12,I122,I124</td>
<td>x=3, ( U = {P_3, P_4} )</td>
</tr>
<tr>
<td>I13,I11,N</td>
<td>cn=8, cp=5, pageno=6, ( P_5 = {8},</td>
</tr>
<tr>
<td>I12,I122,I123</td>
<td>x=3, ( U = {P_3, P_4, P_5} )</td>
</tr>
<tr>
<td>MS1,MS2,M</td>
<td>( P_4 = {7,3},</td>
</tr>
<tr>
<td>MS3,M</td>
<td>( P_4 = {7,3,8},</td>
</tr>
<tr>
<td>MS4,M</td>
<td>( P_3 = {6,1},</td>
</tr>
<tr>
<td>II25</td>
<td>cn=3</td>
</tr>
<tr>
<td>II2,I123</td>
<td>x=1</td>
</tr>
<tr>
<td>MS1,MS2,N</td>
<td>cp=5, pageno=6, ( P_5 = {1},</td>
</tr>
<tr>
<td>MS4,M</td>
<td>( P_3 = {6,1},</td>
</tr>
<tr>
<td>I125,II1</td>
<td>cn=1, stop as IP(1) = \Lambda and end of queue T.</td>
</tr>
</tbody>
</table>
Algorithm Pi and Paging Performance

In this section we will show analytically that if a program satisfies certain conditions, algorithm Pi is sufficient to give an optimal partition. When some of the conditions are not met, the paging performance of the optimized program is more difficult to analyze, and we will examine the effect of Pi on paging performance by means of simulated execution of programs in the next section.

Theorem 4.1 If a program satisfies the following conditions, then the partition given by algorithm Pi will generate the least number of page faults when memory size M=1.

a). Each routine is referenced by only one other routine at only one place.

b). There is no conditional reference, that is, every time a routine is referenced, all its descendants will be referenced.

c). Let C(z,x) denote the number of times z references x, then c(z,x)=c(z,y) implies that C(z,x)=C(z,y), and c(z,x)=j implies that C(z,x)⩾\sum C(z,y) for all y where c(z,y)<j.

We will prove the following propositions and then use them to prove the theorem.

1. The number of page faults for an execution of a
program with memory size $M=1$ is equal to or less than two times the sum of inter-routine reference frequencies.

The number of page faults for memory size 1 can be computed by counting the number of times the page inter-reference distance $d_i(t)>1$ for all pages, $i$, and references $t$ in the reference string. This is the same as counting the number of times the page number changes in the reference string, which in turn is the same as the number of cross page references because every time a page references another page or transfers control to the other page, it is a change of page number in the reference string. Since $C(i,j)$ represents the number of times routine $i$ calls routine $j$, the number of cross page references is less than or equal to

$$\sum_{(i,j)} 2C(i,j)q(i,j) \quad (4.1)$$

where $(i,j)$ is an edge in the subroutine reference tree and $q(i,j)=1$ if nodes $i$ and $j$ are not in the same page, $q(i,j)=0$ otherwise.

Immediately from the proposition, it is clear that in order to prove the theorem, it is sufficient to show that the quantity in expression (4.1) is a minimum.

2. Let $G$ be a program satisfying the conditions in the theorem, then the paths from the root of $G$ to its leaves have monotonically increasing reference frequency,
that is, if $v_1, v_2, \ldots, v_k$ is the sequence of nodes in a path, then

$$C(v_1, v_2) \leq C(v_2, v_3) \leq \cdots \leq C(v_{k-1}, v_k). \quad (4.2)$$

Since condition b) assumes no conditional reference, once a routine $v_i$ is referenced, all of its descendants will be referenced. If $c(v_i, v_{i+1}) = 0$, $v_{i+1}$ is not in any loop in $v_i$, hence $v_{i+1}$ will be referenced exactly once whenever $v_i$ is executed, thus $C(v_{i-1}, v_i) = C(v_i, v_{i+1})$. If $c(v_i, v_{i+1}) > 0$, $v_{i+1}$ is referenced inside some loops in $v_i$, hence $C(v_{i-1}, v_i) \leq C(v_i, v_{i+1})$. This is true for all $v_i$ in the path, hence

$$C(v_1, v_2) \leq C(v_2, v_3) \leq \cdots \leq C(v_{k-1}, v_k).$$

3. Let $v_i$ be the root of a subtree of types described below in a program $G$ satisfying conditions in the theorem, then the process of assigning pages from leaves to root by algorithm Pl minimizes the cross page reference frequency for the pages containing the subtree.

**Type I Subtree.** The size of the subtree is less than the page size $p$.

Algorithm Pl assigns the entire subtree to a single page because it assigns all nodes in a subtree before processing other subtrees. The cross page reference frequency for the subtree is zero. If there is another algorithm which divides the subtree into more than one page and assigns nodes in other subtrees to these pages,
the number of cross page references for nodes in the subtree under the second algorithm is certainly greater than zero.

**Type II Subtree.** The size of the subtree is greater than $p$, but it has only one branch.

Let $v_i, v_{i+1}, \ldots, v_k$ be the nodes in the subtree, by proposition 2 above

$$c(v_i, v_{i+1}) \leq c(v_{i+1}, v_{i+2}) \leq \ldots \leq c(v_{k-1}, v_k).$$

Hence the best partition is to cut edges as close to the root $v_i$ as possible. $P_1$ does this because it assigns nodes from $v_k$ to $v_i$. Nodes in a type II subtree may be divided and assigned to two different pages if $v_j$ is too big to put into the page containing $v_{j+1}$, i.e. $j < k$. Node $v_j$ will be divided at any point where $v_{j+1}$ is referenced, or at any point outside loops before the last reference to $v_{j+1}$, and the last half of $v_j$ will be assigned to the page containing $v_{j+1}$. The division of nodes in this way will not increase the page fault rate for the following reasons:

1. Since node $v_j$ and $v_{j+1}$ cannot be in the same page, if memory size cannot hold all pages containing $v_j$, $v_{j+1}$ and descendants of $v_{j+1}$, then every time $v_j$ references $v_{j+1}$, it will cause a page fault to bring in the page containing $v_{j+1}$, and a page fault to return to $v_j$.

2. If $v_j$ is divided at the point where $v_{j+1}$ is referenced, and the last half is assigned to the page
the number of page faults is at most the same as 1 above (the page fault for returning to \( v_j \) may be saved if \( v_{j+1} \) is not referenced in a loop).

3. If \( v_j \) is divided at the point before \( v_{j+1} \) is referenced, no page fault will be generated to branch to \( v_{j+1} \), but there will be one page fault when the last half of \( v_j \) is entered.

4. Since part of \( v_j \) is in the page containing \( v_{j+1} \), there will be more room in the page containing the first half of \( v_j \) to hold the predecessors of \( v_j \).

**Type III Subtree.** The size of the subtree is greater than \( p \), and the root of the subtree \( v_i \) has several branches, each of them being a type I or type II subtree.

Let \( y_1, y_2, \ldots, y_j \) be the immediate descendants of \( v_i \), where each of them is the root of a subtree of type I or II; then the partitions for these subtrees already have the minimum cross page references. Let \( P_{y_1}, P_{y_2}, \ldots, P_{y_j} \) be the pages containing nodes \( y_1, y_2, \ldots, y_j \) respectively and \( P_{v_i} = \{ v_i \} \). The cross page reference frequency between these pages is

\[
\sum_{x=1}^{j} C(v_i, y_x) \tag{4.3}
\]

To minimize the cross page reference frequency for pages in the subtree containing \( v_i \), we simply merge \( P_{y_1} \),
to $P_{v_i}$ in such a way that equation (4.3) is decreased by the largest amount. For simplicity, we assume that each of these pages $P_{Y_1}, P_{Y_2}, \ldots, P_{Y_j}$ has enough room for $v_{i-1}$, so that the problem is to select the best page to merge with $P_{v_i}$. If

$$c(v_i, y_1) > c(v_i, y_2) > c(v_i, y_3) > \cdots > c(v_i, y_j),$$

then $P_{Y_1}$ should be selected for $c(v_i, y_1) > \sum_{x=2}^{j} c(v_i, y_x)$ by condition c) of the theorem. If

$$c(v_i, y_1) = c(v_i, y_2) = c(v_i, y_3) > c(v_i, y_4) > \cdots > c(v_i, y_j),$$

then the smallest page among $P_{Y_1}, P_{Y_2}, P_{Y_3}$ should be selected so that $P_{v_i}$ will have more room to merge other pages or more room for node $v_{i-1}$. Procedure MERGEIS of algorithm PI merges these nodes in the way described above. In MERGEIS the above process is repeatedly applied to $P_{Y_1}, P_{Y_2}, \ldots, P_{Y_j}$ until there is no more room in $P_{v_i}$ to contain any of the pages not yet merged.

Proof of Theorem 4.1 By proposition 1, we need only prove that the page assignment given by algorithm PI will have the minimum cross page reference frequency, which can be proved using proposition 3. We know that the pages containing subtrees of the types given in proposition 3
have the minimum cross page reference frequency. We can delete these subtrees from further consideration by creating a new tree as follows:

1. If the page containing the root of a subtree is partially filled, replace the subtree with a node of size equal to the size of the page.

2. Otherwise, delete the subtree.

The new tree will have another set of subtrees of type III; by proposition 3, algorithm Pl will assign these subtrees in such a way that the cross page reference frequency between pages in the subtree is a minimum. The same arguments are repeated until the root of the tree is assigned, then the pages containing the root and all its subtrees have the minimum cross page reference frequency.

Q.E.D.

Discussion The conditions in theorem 4.1 are necessary to avoid a large number of choices when assigning nodes to pages. Note that algorithm Pl prunes the subroutine reference tree from leaves to the root. If condition a) is satisfied, that is, each routine is referenced by only one other routine, then there is only one choice to "climb up" the tree. If condition b) is satisfied, then the higher an edge is on the tree, the lower the routine cross reference frequency, so we may assign as many nodes as possible from bottom up without
worrying whether the next node up the tree has higher execution frequency or not. If condition c) is satisfied, then the root of a subtree may be assigned to the page containing the immediate descendant node referenced in the highest nesting level, because the total execution frequency of all other routines referenced in a lower nesting level is less than the execution frequency of this routine. The subroutine reference tree in Fig. 4.8 does not satisfy conditions b) and c). For examples, C(18, 19) > C(19, 20) (condition b); c(2, 3) > c(2, 6) > c(2, 8) but C(2, 6) < C(2, 8), even though C(2, 3) > (C(2, 6) + C(2, 8)) (condition C).

Theorem 4.2 Let G be a program satisfying the conditions in theorem 4.1 and |G| be the size of G. If algorithm P1 partitions the graph into \(\lceil G/\text{page size} \rceil\) pages, where \(\lceil x \rceil\) is the notation for the smallest integer greater than x, and if, during the application of algorithm P1, it is true that for each node \(v_i\) in G having more than one immediate descendant, \(v_i\) and all of its immediate descendants on unfilled pages can be merged into a single page by the procedure MERGIS, then the partition will generate the least number of page faults for all possible memory sizes.

Lemma 4.1 Let L be an inner-most loop occupying \(k\) pages in which no subroutine is referenced. If instructions
in these pages are packed in such a way that once control exits from a page, the page will not be referenced until the next iteration of the loop, and if memory size \( M < k \), then there will be at most \( k \) page faults on each iteration of the loop.

**Proof** Let the pages in \( L \) be \( P_1, P_2, \ldots, P_k \). Since we assume that once control exits from a page, the page will not be referenced again until the next iteration, a typical reference string for one iteration would be

\[ P_1 P_1 \cdots P_2 P_2 \cdots P_3 P_3 \cdots P_k \]

Since the page interreference distance for consecutive references to a given page is 0, these references will have no effect on the number of page faults. If we ignore these references, the reference string may be condensed to

\[ P_1 P_2 P_3 P_4 \cdots P_k \]

**Case 1.** All \( k \) pages are referenced sequentially on each iteration.

There will be exactly \( k \) pages in the condensed reference string (i.e. \( k \) distinct references). When the loop is initially entered, if none of these pages has been referenced before, there will be \( k \) page faults on the first iteration; if some of these pages have been referenced before, then the number of page faults will be less than or equal to \( k \), because they may be still in the main memory. After the first iteration, the page
interreference distance for these k distinct references are \( k-1 \), i.e. \( d^i = k-1 \). By Theorem 2.1, each of these references will generate a page fault; and hence, a total of \( k \) page faults will be generated on each iteration.

**Case II.** Not all pages are referenced on each iteration.

Since the number of distinct references is less than \( k \), the number of page faults is always less than \( k \).

Q.E.D.

**Lemma 4.2** Let \( L \) be a loop of size \( k \) pages and let the memory size \( M \) be less than \( k \). (1) The number of page faults on each iteration of \( L \) is the number of references whose page interreference distance \( d^i > M \). (2) If the number of distinct references on each iteration is less or equal to \( M \), then there will be at most \( M \) page faults on each iteration.

**Proof.** Immediate from Theorem 2.1.

**Proof of Theorem 4.2** Lemma 4.1 states that if we pack the instructions of a loop in such a way that references to pages in the loop are in consecutive order, viz., \( P_1, P_2, \ldots, P_k \), and such that the same page will not be referenced until the next iteration, then the number of page faults on each iteration will not be greater than the number of pages occupied by the loop. To minimize the number of page faults, we need only to pack instructions in the way stated above into the least possible
Lemma 4.2 implies that even if we cannot pack instructions so that they reference pages consecutively, we can still minimize the page interreference distance for all references on each iteration by minimizing the number of pages required in each path of a loop. Furthermore, if we minimize the number of pages occupied by a loop, page interreference distance will also be minimized.

Since we assume no conditional reference, all pages in a loop will be referenced and the reference string will be the same for all iterations. In order to prove that the partition given by algorithm Pl generates the least number of page faults, we only need prove that Pl minimizes the number of pages occupied by any loop.

If a loop contains an inner loop, the inner loop should occupy a minimum number of pages so that the outer loop (including its inner loop) is in a minimum number of pages. If a loop references other routines, then these routines are part of the loop and hence these routines and their descendants should also be in a minimum number of pages.

Since the theorem assumes that the program has been partitioned into a minimum number of pages, viz., \(|G|/\text{page size}\), we need only show that each branch in the subroutine reference tree occupies a minimum number of pages.
As stated in theorem 4.1, a subroutine reference tree consists of subtrees of type I, II and III. Since all nodes in type I subtrees are assigned to a single page, it is a minimum. Type II trees must also have been partitioned into minimum number of pages, otherwise, the overall partition cannot have minimum number of pages.

Let $v_i$ be the root of a type III subtree and $y_1, y_2, \ldots, y_j$ be the immediate descendants of $v_i$. Each of the subtrees containing descendants of $v_i$ has been assigned to a minimum number of pages because they are of type I or II. Since the partially filled pages containing $y_1, y_2, \ldots, y_j$ can be merged to the page containing $v_i$ by procedure MERGEIS by the assumption of the theorem, the entire type III subtree occupies a minimum number of pages.

Next we want to show that loops in $v_i$ occupy a minimum number of pages. Let $y_k$ and $y_k'$ be referenced in loop L in $v_i$. The subtrees below $y_k$ and $y_k'$ already have been assigned to a minimum number of pages, and any partially filled pages containing $y_k$ or $y_k'$ are merged to the page containing $v_i$ by procedure MERGEIS, therefore L occupies a minimum number of pages.

We can repeat the above arguments by deleting type I, II or III subtrees from the subroutine reference tree or replacing them with a node representing the partially
filled page containing the root of the subtree. Each repetition produces a new set of type III subtrees, hence the above argument can be applied repeatedly until the root of the entire tree is processed. Q.E.D.

Simulated Executions

In the previous section we have shown that algorithm P1 gives the best partition under specified ideal conditions. However, if these conditions are not satisfied, P1 may not produce an optimal partition. In order to analyze the paging performance of programs partitioned by algorithm P1, the simulator described in Chapter 3 is used to simulate the execution of programs and compute the page fault rate.

The results presented here are from simulation of the program in Fig. 4.1.

Fig. 4.4 shows the internal structure of modules of the program. Dots represent nodes in each module and the number beside some of the nodes is the module referenced by the node. It is assumed that each routine is referenced by only one routine. Note that some of these routines may not be referenced because a particular path in which a routine is referenced may not be taken. Given a page assignment of a program, the number of page faults depends on the probability each path will be taken and
Fig. 4.4 Structure of routines of the program in Fig. 4.1. Numbers on top of graphs are routine numbers, numbers beside nodes are the subroutines referenced.
the memory size. In order to compare the effect of algorithm Pl on paging performance with the effect of other algorithms, the program in Fig.4.1 is partitioned using the following three page assignment algorithms:

1. Packed: algorithm Pl.

2. Ordered: modules are assigned to pages sequentially in the order which is considered by the programmer to be a good ordering. A module may be divided and assigned to different pages as long as it is not cut in the middle of a loop.

3. Random: modules are randomly ordered and then assigned to pages sequentially. Each module may be divided and assigned to any number of pages.

Following is a list of pages for page size of 2k bytes. \( P_i, O_i, \) and \( R_i \) represent the set of modules assigned to page \( i \) under Packed, Ordered and Random page assignment respectively. If module \( j \) is divided and assigned to different pages, then \( j' \) refers to its first half and \( j'' \) to its second half. (In this example no module is assigned to more than two pages).
\[ P_1 = \{1, 10, 14\}, \quad P_2 = \{14', 11, 12, 13\}, \quad P_3 = \{2, 3, 4, 5\}, \]
\[ P_4 = \{6, 7\}, \quad P_5 = \{8, 9\}, \quad P_6 = \{15, 16, 17\}, \]
\[ P_7 = \{18, 23\}, \quad P_8 = \{19, 20, 21, 22\}. \]
\[ O_1 = \{1, 2, 3\}, \quad O_2 = \{3', 4, 5, 6'\}, \quad O_3 = \{6", 7, 8, 9'\}, \]
\[ O_4 = \{9", 10, 11, 12, 13\}, O_5 = \{13", 14, 15, 16, 18'\}, \]
\[ O_6 = \{18", 19, 20, 21\}, \quad O_7 = \{21', 22, 23, 17\}. \]
\[ R_1 = \{1, 2, 10'\}, \quad R_2 = \{10", 3, 4, 11, 15\}, R_3 = \{12, 14, 16, 17'\}, \]
\[ R_4 = \{17", 5, 6'\}, \quad R_5 = \{6", 9, 13, 18, 7'\}, R_6 = \{7", 20, 21, 22, 23\}, \]
\[ R_7 = \{8, 19\}. \]

Fig. 4.5 Packed page assignment of the program in Fig. 4.1 for page size of 2048 bytes.
Fig. 4.6 Ordered page assignment of the program in Fig. 4.1

Fig. 4.7 Random page assignment of the program in Fig. 4.1
Fig. 4.8 Inter-routine reference frequency of the program in Fig. 4.1.
Test No. 1

Since the only information available to the compiler is the nesting level of loops, the following assumptions are necessary.

Basic Assumptions

1. The execution frequency of each inner loop is a constant multiple of its outer loop.

2. If a node is not a loop exit node and if it has more than one immediate descendant, then each of these descendants has equal probability of being executed.

What we are interested in is that if a program behaves this way, what is the effect of algorithm P1 on paging performance?

Fig. 4.8 shows the intermodule reference frequencies of a typical execution of the program. The numbers inside parentheses are nesting levels. Note that a higher loop nesting level does not always have a higher execution frequency since the path in which the loop is located may not be taken sometimes.

Fig. 4.9 presents the average page fault rate of the program under three page assignment algorithms: PACKED, RANDOM and ORDERED. Note that the number of pages of the program is 7 under RANDOM and ORDERED assignment, and 8 under PACKED assignment. The page fault rate under
Fig. 4.9 Page fault rate of the program in Fig. 4.1 under three different page assignments.
PACKED assignment is consistently less than that of RANDOM when memory size is less than 7 pages (14K bytes), and less than that of ORDERED when memory size is less than 6 pages (12K bytes). When memory size in pages = "program size/page size", PACKED assignment is no longer useful because the program can be kept in the main memory all the time and hence it does not matter how the program is internally partitioned.

Fig.4.10 shows the relative paging performance of three assignment algorithms relative to PACKED assignment. The curves show the page fault rate of RANDOM or ORDERED assignment divided by the page fault rate of the PACKED assignment. When the ratio is 1, it means that both have the same number of page fault rate.

When the modules are randomly ordered, the page fault rate is 10 times more than that of PACKED assignment when memory size is 1 page, M=1; 25 times more when M=2; and 8.5 times more when M=3. As memory size increases, the ratio decreases.

When modules are pre-ordered, the page fault rate ratio of ORDERED/PACKED is lower than the ratio of RANDOM/PACKED, and the ratio falls below 1 with a smaller memory size. The improvement of paging performance of the PACKED algorithm (P1) over the ORDERED algorithm depends on how well the programmer arranges his program; if he
Fig. 4.10 Relative paging performance of the program in Fig. 4.1 under different page assignments.
understood perfectly the behavior of his program, the ordering he would give would be optimal; and hence the ratio of ORDERED/PACKED should be below 1 for all memory sizes. Fig.4.10 shows that when the behavior of a program satisfies the basic assumptions 1 and 2, then the partitioning obtained by algorithm P1 is as good as, or even better than, the pre-ordered algorithm.

Page size and Packing

The effects of different page sizes on paging performance have been discussed in chapter 2. In this section we will examine the following two questions: (1) is P1 more effective under large page size or small page size, and (2) if a program has been packed using P1 with page size p, what is the page fault rate under page size \( \frac{h}{p} \) without repacking the program?

Fig.4.11 shows the page fault rate of the program in Fig.4.1 packed under 4K-byte, 2K-byte and 1K-byte page sizes. The page fault rate for 1K-byte pages is the lowest, 2K-byte in the middle, and 4K-byte the highest. However, if we look at the ratio of improvement over random ordering, 4K-byte pages have the highest improvement, as shown in Fig.4.12. When memory size is 8K bytes, the page fault rate of the program under RANDOM ordering is 83 times more than that of PACKED for 4K-byte
Fig. 4.11 Page fault rate of the program in Fig. 4.1 under packed assignment using algorithm P1 for different page sizes.
Fig. 4.12 Relative paging performance of the program in Fig. 4.1 under packed assignment for different page sizes.
pages; it is 6 times more for 2K-byte pages; and 2.5 times more for 1k-byte pages. Packing is more effective under large page size because these pages contain more superfluous space and more closely related segments of the program can be packed into the same page. When page size is small, the superfluous space is smaller and hence there is less room for rearranging instructions.

Since the program in Fig.4.1 has conditional references; that is, some of the paths may not be taken on iterations of loops, the packing done by algorithm P1 may not completely eliminate superfluous spaces on pages; hence when pages are fetched into main memory, parts of some pages may not be referenced. Under this condition, reducing the page size after packing may reduce the page fault rate, even though the reduction may not be as great as it would be if the program were repacked using the new page size.

If a program has no conditional references, or if a program can be packed in such a way that all instructions in a page are sequentially referenced, then reducing the page size will only increase the page fault rate. Fig.4.13 is an example of such programs and the following table shows the page fault rate of the program packed under 4K-byte page size and executed under 2K and 4K byte page sizes.
If a program satisfies the conditions stated in Theorem 4.1, and if the program is packed under page size \( p \), then reducing the page size to \( \frac{1}{4}p \) without repacking will not decrease the page fault rate.

Test No. 2

In this test, we deliberately make the behavior of the program in Fig. 4.1 violate the basic assumptions on test no.1. More specifically, the probabilities of taking a path containing loops are set very low so that the path will be executed less frequently than what is expected by algorithm Pl. Fig. 4.14 shows the execution frequency of a typical execution.

Fig. 4.15 shows the average page fault rate under three packing algorithms. Fig. 4.16 shows their relative paging performance. The curve marked ORDERED is the ordering given in test no.1, and OPT is the new ordering which the programmer considers an optimal ordering under the new set of probabilities. The page fault rate of the
Fig. 4.13 The subroutine reference tree of a program with no conditional references. Numbers beside nodes are node sizes in K-bytes, numbers beside edges are typical execution frequencies, and numbers in parentheses are loop nesting levels.

Fig. 4.14 Inter-routine reference frequency of the program in Fig. 4.1 with a new set of probability of inter-routine references.
PACKED program is still less than that of randomly ordered, but is more than optimally ordered by the programmer.

The result indicates that packing of the program by the compiler based on loop analysis is better than arbitrary ordering, but it may not be better than careful organization of the program by a programmer who has more information about the behavior of the program than just the loop nesting levels.

**Conclusions**

The following is a summary of the simulation results. Note that each routine is assumed to be referenced by only one other routine.

1. The effects of the packing algorithm P1 is more significant for large page size (relative to the module size) than small page sizes.

2. When the memory size is large enough to hold most of the pages of a program, packing is less effective, especially when the program is already well organized.

3. When the memory size is small, packing of the program using algorithm P1 is better than random ordering of modules, even though the execution frequency of paths and modules of a program may not be exactly as predicted by algorithm P1.

4. When the probability of a path being executed
Fig. 4.15 Page fault rate of the program in Fig. 4.14 under four different page assignments for 2K bytes page size.
is such that the execution frequency of any path from the root of the subroutine reference tree to its leaves is monotonically increasing, then algorithm Pl performs as well as the best pre-ordering of modules given by the programmer.

Fig. 4.16 Relative paging performance of the program in Fig. 4.14 under different page assignments.
Chapter 5 A Case Study -- The XPL Compiler

In the previous chapters, page assignments are based on the assumption that modules are either all less than or larger than the page size. When module sizes are less than the page size, algorithm P1 is used; when module sizes are equal to or greater than the page size, algorithm P0 is used. In a practical case, however, such a condition is not always true, and the assignment and pagination of modules cannot be considered separately. There must be some interaction between algorithms P0 and P1. When the page size is very small, algorithm P0 is in a predominant role; as the page size increases, algorithm P1 plays a more and more important role than P0.

In the next section we present an algorithm P2 which combines the functions of P0 and P1 and which includes a modification of P1 to handle the case where subroutines are referenced by several other routines. We also present in this chapter the results of an application of P2 to a practical program -- the XPL compiler.

Algorithm P2

Algorithm P1 accepts programs in which subroutines can be referenced by only one other routine. We remove this restriction in Algorithm P2 provided that the program
contains no recursive routines. P2 is essentially an extension of P1. It starts with a 'macro' view of program structure -- subroutine reference tree -- and assigns routines to pages. If the size of a routine is larger than the page size or the space available in the current page to which routines are being assigned, then a more detailed 'micro' view of the structure of the routine is taken and algorithm P0 is used to assign basic blocks in the routine to pages. We will list the necessary modifications to P0 and P1 below, then present P2 in an Algol-like language.

A. When building the subroutine reference graph

Similar to P1, the first part of P2 is to build a subroutine reference tree using procedure BUILDTREE. If a routine is referenced by several routines, then the original BUILDTREE procedure will add several duplicated subtrees, one under each routine referencing it. Recall that in P1, if x is the routine on top of stack T', a call to BUILDTREE will add all routines referenced by x to the tree. If y is referenced by x and if y is already in the tree, then y is also referenced by some other routine(s). BUILDTREE is modified so that in such a case, it will add only the edge (x, y) to the tree and that it will push only those nodes that have just been added to the tree into stack T'. Furthermore, if all routines
referenced by x are already in the tree, then add x to the queue T. The resulting graph is no longer a tree, we will call it a subroutine reference graph. BUILDTREE is renamed BUILDGRAPH.

If on each execution of routine x, routine y is always referenced by x, then y is unconditionally referenced; if y is not always referenced by x, then y is conditionally referenced. When adding routines referenced by x to the subroutine reference graph we order them left to right in descending order of loop nesting levels. If two routines are referenced in the same nesting level, the following tie breaking rules are applied in the order in which they are listed:

1. routine unconditionally referenced is on the left;
2. routine with higher static reference frequency is on the left;
3. routine appearing first in the source program is on the left.

B. When a node has more than one immediate descendant

Having built the subroutine reference graph, P2 starts assigning nodes to pages bottom-up from the graph. Since each routine may be referenced by several routines, a node may have more than one immediate ancestor. One of the following selection rules may be used to select the next node to 'climb' up the graph:
1. Highest probability - The immediate ancestor that references this node most often or has the highest probability of referencing this node is selected next. The difficulty is that such information is not normally available.

2. Highest reference nesting level - Define the reference nesting level of a node $x$, $r_n(x)$, as follows:

$$r_n(x) = \left\{ \begin{array}{ll} \max_{y \in IP(x)} r_n(y) + 1; \\ 0 & \text{if } x \text{ has no immediate ancestor.} \end{array} \right.$$ 

Select the immediate ancestor that has the highest reference nesting level as the node to be processed next. The argument for using this rule is that the higher the reference nesting level of a node, the more often the node is referenced. For example, let $x$ and $y$ be routines referenced by the main routine in a loop of nesting level 2, and let $z$ be referenced by $y$ in a loop of nesting level 1, then $r_n(x) = r_n(y) = 1$ and $r_n(z) = 2$. If both $x$ and $y$ are referenced $m$ times, then very probably $z$ will be referenced in a loop of nesting level 3.

3. Leftmost immediate ancestor - The first immediate ancestor that causes this node to be added to the graph. Left to right ordering of nodes in the graph is maintained in descending order of loop nesting levels in which routines are being referenced. It is hoped that nodes on the left will be referenced more often than nodes on the
right, hence this rule tries to assign the leftmost node first. Use of this rule has the least overhead because the ordering has been established as the graph was created.

Since rules 2 and 3 are used as an approximation of rule 1 based only on local information, the prediction may or may not be true depending on the program. We can only hope that the prediction is correct in most cases.

Algorithm P2 uses a mixed strategy as follows: Let x be the node being processed and let cp be the page containing x. The only thing that matters at this moment is which nodes should be assigned to page cp, so P2 tries to assign to page cp all ancestors of x having only one immediate descendant. First P2 selects an immediate ancestor of x which has only one immediate descendant, namely x, and assigns it to page cp. It keeps on climbing up this path, assigning nodes to page cp until it reaches a node having more than one immediate descendant or ancestor, then P2 backs up to node x and repeats the same process for other immediate ancestors of x until either page cp is full or no more nodes satisfy the condition. At this time P2 discards node x and continues processing of the graph by climbing up the path on the leftmost immediate ancestor of x. Rule 2 or 3 may be used when selecting one of IP(x) that has only one immediate descendant. In the application of P2 to the XPL compiler
to be presented below, rule 2 is used.

The actions to be taken when a node has more than one immediate descendant are the same as in P1.

C. Coordination between P0 and P2

Algorithm P0 is used whenever (1) the size of a node is larger than the page size and (2) the remaining available space in the page to which nodes are being assigned is not large enough to contain the next node and the node has only one immediate ancestor. These are checked in procedure NEWPAGE and ASSIGN before calling P0.

Let x be the node to be processed by P0, the following information is available on entry: (1) the page number, pn, to which nodes are being assigned and (2) the immediate descendant of x that is assigned to page pn. Note that in algorithm P0, nodes in the graph representing routine x are the basic blocks in x. We assume that associated with each basic block, there is a list of routines referenced in that block. Regardless of the space available in page pn, P0 starts with the next page, page pn+1, when it begins assigning nodes in x. Whenever a node that references routine x is assigned to page pn+1, P0 calls procedure MERGE to merge page pn+1 to page pn, if possible. When P0 finishes processing x, if the last page contains nodes referencing routine x, then P0 tries to merge this
Fig. 5.1(a) shows an example of a subroutine reference graph created by P2. Node numbers also indicate the order in which the nodes are added to the graph, numbers beside nodes are the reference nesting levels, and numbers on edges are loop nesting levels. The final set of nodes in T are \{5, 7, 8, 9, 10\}. Fig. 5.1(b) shows the page assignment given by P2, assuming node sizes are 1 unit per node and page size $p = 4$ units.
(a) A subroutine reference graph.

(b) Page assignment for p=4.

Fig. 5.1 A subroutine reference graph and its page assignment for p=4.
Algorithm P2
I: begin comment Build a subroutine reference graph;

procedure BUILDGRAPH(x);
begin
B1: Add all routines referenced by x as IS(x) and
    order them from left to right in descending
    order of their loop nesting levels (tie breaking
    rules are given in the text);
B2: if all IS(x) have already been added to the graph
    then add x to the queue T;
    else push all nodes that have just been added to
    the graph, rightmost node first, to T';
end BUILDGRAPH:

I1: Push the main module into T', set T empty and let
    the main module be the first node of the graph;
I2: for cn := top of T' while T' is non-empty do
    begin Pop T';
    I21: if cn calls another routine then
        call BUILDGRAPH(cn);
    I22: else add cn to the queue T
    end I2;
end I;

II: begin comment Assign nodes to pages;

procedure NEWPAGE(x);
begin
    cp := pageno; pageno := pageno + 1;
    if $w_x < p$ then begin $p_{cp} := (x); |p_{cp}| := w_x$ end;
    else call P0(x,cp,leftmost IS(x));
    Mark x processed;
end NEWPAGE;

procedure MERGE(i,j); begin Same as P1 end;
procedure MERGEIS(x); begin Same as Pl end;

procedure ASSIGN(x);
begin
if $|P_{cp}| + w_x \leq p$ then
begin
$P_{cp} := P_{cp} \cup \{x\}$; $|P_{cp}| := |P_{cp}| + w_x$
end;
else if $|IP(x)| \leq 1 \vee w_x > p$ then
call PO(x, cp, leftmost IS(x));
else begin
$cp := \text{pageno}$; \text{pageno} := \text{pageno} + 1;
$P_{cp} := \{x\}$; $|P_{cp}| := w_x$;
end;
Mark x processed
end ASSIGN;

II0: \text{pageno} := 1;

II1: for cn := each element in queue T do
begin
if cn has not been processed then
begin
if $|IS(cn)| = 0$ then call NEWPAGE(cn);
else if $|IS(cn)| > 1$ then call MERGEIS(cn);
else begin
Let i$\in$IS(cn); $cp := pg(i)$;
call ASSIGN(cn)
end;
end;

II2: for x := leftmost IP(cn) while IP(cn) is non-empty
do begin
if $|IP(cn)| > 1$ then

II20: begin comment More than one routine reference
\text{cn}, each is in a path from root to \text{cn}. Assign
$z \in IP(cn)$ to page $cp$ only if $z$ references only
one routine, \text{cn};

for $z :=$ each IP(cn) in the order specified
by the Selection Rule do

II201: begin
for y := leftmost IP(z) while
$|IS(z)| = 1 \land |P_{cp}| + w_z < p$ do
begin comment Keep on climbing up a path until reaching a node having more than one immediate descendant or ancestor, or until not enough space in page cp;
call ASSIGN(z);
if |IP(z)| = 1 then
  begin comment If this is the path to the leftmost IP(cn), keep track of last node processed;
    if x = z then x := y;
    z := y;
  end;
else goto II202
end;
II202: end II201;
end II20;
II21: if |IS(x)| = 1 then
  begin
    if x has not been processed then
      call ASSIGN(x)
    end;
else if all IS(x) have been processed then
  call MERGEIS(x);
else goto II3;
  cn := x;
end II2;
II3: end III;
end II;
The XPL Compiler

The XPL compiler is a one-pass compiler for a dialect of PL/I: the XPL language [Mckml]. It is used for testing here for several reasons: (1) it is the type of programs that needs optimization for it may be used very often; (2) it is written in a modular fashion consisting of more than 60 routines; (3) the XPL system has a built-in trace facility to generate an execution trace; (4) it is written in a high level language, the XPL language.

In the first step of our testing, the XPL compiler is modified to give a list of basic blocks in the program being compiled, together with their sizes, starting addresses and immediate ancestors and descendants. The original compiler is then recompiled to obtain its list of basic blocks.

In the second step, a trace of an execution of the original compiler compiling a program is generated. The trace tape records the addresses of each instruction and its operands, and the type of each instruction.

Finally a page assignment table is prepared manually by applying P2 to the original XPL compiler source program. The table contains the starting and ending addresses of blocks and the page number to which the block is assigned. Note that a block in the table may
not be same as a basic block, for several consecutive basic blocks may be assigned to the same page and hence are combined.

The trace tape and page assignment table are fed into a simulation program which simulates the execution of the program under LRU replacement. The simulation program computes an estimated page interreference distance function \( f(x) \) [Chapter 2], and prints out statistics at the end of simulation.

Paging Performance - Instructions Only

This section presents the effects of page assignment given by P2 on the paging performance of the XPL compiler when only the instruction part is considered.

Page fault rate vs. memory sizes

Fig.5.2(a), (b) and (c) show the page fault rate per thousand instructions with page sizes 1024, 2048 and 4096 bytes respectively. The curves marked Sequential are the page fault rates under sequential assignment in the original order of compiler source text; the curves marked Packed are the page fault rates under the page assignment given by P2. The vertical distance between two curves is the difference of page fault rate between the two page assignments which is shown in Fig.5.3(a). The figure
Fig. 5.2 Page fault rate of the XPL compiler under sequential and packed (P2) assignments (instructions only). $p$=page size.
shows that the difference in page fault rate is a decreasing function of memory size, i.e. the larger the memory size, the lower the difference. Note that the size of the instruction part is 54,800 types, or 54 1024-byte pages under sequential allocation. In all three page sizes, 1024, 2048, and 4096 bytes, when the available memory size is more than two-thirds of the number of pages required under sequential allocation, the page fault rate under packed assignment is either the same or higher than that of sequential assignment. When the available memory size is large enough to hold most of the pages referenced, the effect of optimization is insignificant as it does not matter how a program is divided. However, what we are interested in is the case when available memory size is small; in the case of XPL compiler the interesting range is 10K-36K bytes.

Fig.5.3(b) shows the relative difference in paging performance in terms of the ratio of page fault rate between sequential allocation and packed assignment under P2. When memory size is in the range of 10K-36K bytes, the page fault rate under sequential allocation is close to, or in most cases, higher than 2 times that under packed assignment. The highest ratio is 20 for 4096-byte pages with a memory size of 10 pages.
Fig. 5.3 Relative paging performance of the XPL compiler under sequential and packed (P2) assignments for different page sizes (instructions only).
Page fault rate vs page sizes

The XPL compiler is a program such that, under sequential page assignment, reducing the page size will also decrease the page fault rate, as shown in Fig.5.2. This is also true under packed assignment; however, there is a cross-over point in memory size when this is no longer true. The page fault rate with a page size of 1024-bytes is higher than that with a page size of 2048-bytes when memory size is equal to or greater than 36K bytes. The same is true between page sizes of 2048-byte and 4096-byte when memory size is equal to or greater than 40K bytes.

In chapter 4, we observed that when algorithm Pl is used for page assignment, if a routine can be referenced by only one other routine then the rate of improvement in paging performance is higher for large page size than for smaller page sizes. In the case of the XPL compiler, the differences in the effectiveness of page assignment between large page size and small page size (e.g. 4096 vs 2048) are not as significant as that observed in the last chapter. This is shown in Fig.5.3(b). Since a routine may be referenced by more than one other routine, all of its ancestors may be assigned to the same page, thus increasing the chance of assigning to the same page routines that may not be referenced together within a short
time interval. Note that in terms of absolute differences, larger page sizes still have higher reductions in page fault rate.

We should also point out that the arrangement of routines in the original XPL compiler is already good. Routines are presented together in groups: lexical analysis, syntax analysis, code emitting, symbol table handling and miscellaneous service routines. The same is true for variables. Our testing of the XPL compiler amounts to a fine-tuning of routines within functional groups; still we achieved higher than two to one improvement. The result shown here is very close to that reported in [Hat 1], both show a similar rate and pattern of improvements with respect to different memory sizes. In [Hat 1] page assignment is based on the actual reference frequency between modules, while in algorithm P2, page assignment is based only on the loop nesting levels.

Paging Performance - Both Instructions and Data

So far we have ignored the page assignment of data and its effects on paging performance. We have done this in the hope that a reduction in the working set size of instructions will provide more memory space for program data and thus reduce the overall page fault rate. In order to check how much the optimization of page assignment
for instructions affects the over-all paging performance, the following tests were performed. Two page assignments are used for the testing: (1) Seq: both instructions and data are assigned sequentially, and (2) Packed: instructions are packed using P2, data are assigned sequentially. The paging simulation makes no distinction between instruction and data references.

Fig. 5.4 shows the page fault rate under Seq. and Packed assignment for page sizes of 2048 bytes. Column (2) in Table 5.1 shows the difference in page fault rates (per 100 thousand references) between Seq. and Packed assignment for page sizes of 1024 bytes. Column (1) shows the same difference when only instructions are considered (data references ignored). In general, both columns show a decreasing difference as the available memory size increases. Column (2) has a higher reduction in page faults. For example, if a 10K-byte memory is occupied by only instructions, there is a reduction of 137 page faults per 100 thousand references; while if the same amount of space is given to both instructions and data, there is a reduction of 318 page faults per 100 thousand references. The same observation is also true between columns (3) and (4), and between columns (4) and (5) of Table 3.1 for page size of 2048 and 4096 bytes respectively.

Instead of comparing the magnitude of page fault rate
Table 5.1 Differences in page fault rates between sequential allocation and packed page assignment

<table>
<thead>
<tr>
<th>Memory Sizes</th>
<th>p = 1024 bytes</th>
<th>p = 2048 bytes</th>
<th>p = 4096 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-byte</td>
<td>I only</td>
<td>D seq.</td>
<td>I packed</td>
</tr>
<tr>
<td>80</td>
<td>0.5</td>
<td>-</td>
<td>0.6</td>
</tr>
<tr>
<td>72</td>
<td>3</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>64</td>
<td>5</td>
<td>-</td>
<td>24</td>
</tr>
<tr>
<td>60</td>
<td>8</td>
<td>-</td>
<td>35</td>
</tr>
<tr>
<td>56</td>
<td>19</td>
<td>-</td>
<td>46</td>
</tr>
<tr>
<td>52</td>
<td>40</td>
<td>-0.06</td>
<td>48</td>
</tr>
<tr>
<td>48</td>
<td>0.5</td>
<td>64</td>
<td>0.06</td>
</tr>
<tr>
<td>36</td>
<td>4</td>
<td>64</td>
<td>7</td>
</tr>
<tr>
<td>32</td>
<td>9</td>
<td>75</td>
<td>21</td>
</tr>
<tr>
<td>28</td>
<td>20</td>
<td>110</td>
<td>42</td>
</tr>
<tr>
<td>24</td>
<td>46</td>
<td>123</td>
<td>57</td>
</tr>
<tr>
<td>22</td>
<td>72</td>
<td>154</td>
<td>65</td>
</tr>
<tr>
<td>20</td>
<td>83</td>
<td>138</td>
<td>71</td>
</tr>
<tr>
<td>18</td>
<td>92</td>
<td>155</td>
<td>91</td>
</tr>
<tr>
<td>16</td>
<td>100</td>
<td>189</td>
<td>100</td>
</tr>
<tr>
<td>14</td>
<td>101</td>
<td>381</td>
<td>131</td>
</tr>
<tr>
<td>12</td>
<td>108</td>
<td>240</td>
<td>178</td>
</tr>
<tr>
<td>10</td>
<td>137</td>
<td>318</td>
<td>200</td>
</tr>
<tr>
<td>8</td>
<td>167</td>
<td>452</td>
<td>214</td>
</tr>
<tr>
<td>6</td>
<td>196</td>
<td>1445</td>
<td>553</td>
</tr>
<tr>
<td>4</td>
<td>200</td>
<td>1590</td>
<td>601</td>
</tr>
<tr>
<td>2</td>
<td>301</td>
<td>1661</td>
<td>2038</td>
</tr>
<tr>
<td>1</td>
<td>2040</td>
<td>1140</td>
<td>-</td>
</tr>
</tbody>
</table>

I = instructions  
D = data  
p = page size  
Differences = 1/100,000 references
Fig. 5.4 Page fault rate of the XPL compiler under different page assignments for page size of 2048 bytes (instructions and data combined).
Fig. 5.5 Relative paging performance of the XPL compiler under different page assignments and for different page sizes (instructions and data combined).
changes, we can also compare the ratio of changes shown in Fig.5.3(b) and Fig.5.5. The pattern of curves are similar, except that when data are considered (Fig.5.5), the ratio is lower and the peak is moved further to the right (larger memory size). The ratios are very low (between 1 and 15) when memory size is small because the reduction in page faults is only a small fraction of the total page faults.

In conclusion, the optimization of page assignment for the instruction part causes a small rate of improvement in over-all paging performance when memory size is extremely small or large, and a very high rate of improvement with moderate memory size (about 1/3 to 2/3 of the program size).
Chapter 6 Conclusions and Proposals for Future Research

In this thesis we have presented a way of describing the behavior of programs in terms of the page interreference distance density function, $f(x)$. At any reference in a reference string, the probability of the interreference distance being $x$ pages is $f(x)$. The probability of a page fault for any given reference in a reference string is the sum of $f(x)$ for $x > M$, given memory size of $M$ pages. If the peak of $f(x)$ is very close to the $x$-origin, then the program requires a small working set.

The goal of pagination is to assign sections of a program to pages in such a way that its $f(x)$ has a very high peak as close to the $x$-origin as possible. We have presented three pagination algorithms which assign pages based on loop analysis assuming that inner loops are executed more often than outer loops.

The effects of these algorithms on paging performance are tested under combinations of different memory sizes and page sizes. Given a page size, each of the three algorithms gives significant improvement of paging performance over sequential page assignments as long as the memory size is significantly smaller than the program size. When compared in absolute terms, the difference in the page fault
rate between packed and sequential assignment normally is a non-increasing function of memory size, so that when memory size approaches the size of the program, sequential assignment becomes better than any other algorithm. When compared in terms of ratios, the page fault rate of sequential assignment is between one and two times that of the packed assignment, when memory size is very small relative to the program size; the ratio is less than or equal to one when memory size approaches the program size; and the ratio is normally high in the middle range.

Given a memory size, the effectiveness of page assignment is normally an increasing function of page size. When a program is such that the page fault rate is already low for a given page size, then reducing the page size without repacking may not necessarily decrease the page fault rate.

This research could be extended to a study of the cost-effectiveness of pagination algorithms. The increased compilation time must be weighed against the reduction in execution time and the load on the computer system, so that a cut-off point may be determined.

In this thesis, the page assignment of the data part is not considered, and is left for further research. One solution may be designing a language which gives the user the ability to dynamically partition the multiply dimen-
sioned arrays into subarrays of size less than the page size so that the references may be in subarrays rather than rows or columns.
BIBLIOGRAPHY


