UNC Report No. TR84-010

PROCEDURE-LEVEL PROGRAM MODELING FOR VIRTUAL MEMORY PERFORMANCE IMPROVEMENT

by

Edward L. Jones

A dissertation submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

1984

PROCEDURE-LEVEL PROGRAM MODELING FOR VIRTUAL-MEMORY PERFORMANCE IMPROVEMENT

by

Edward L. Jones

A dissertation submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
1984

Approved by:

[illegible]

Advisor: Peter Calingaert

[illegible]

Reader: Bharadwaj Jayaraman

[illegible]

Reader: Frederick P. Brooks, Jr.

EDWARD I. JONES Procedure-Level Program Modeling for Virtual Memory Performance Improvement [Under the direction of DR. PETER CALINGAERT]

ABSTRACT

The page-fault overhead incurred by a program executing on a demand-paged virtual-memory computer system is a function of the program's module-to-module reference pattern and the program's layout -- the assignment of program modules to pages in the virtual name space. Defining the layout using program restructuring methods can greatly reduce this overhead, but the restructuring process is itself expensive when an execution trace of the program is required.

This research aims to reduce the computer and programmer time required to perform program restructuring. Only externally relocatable code modules (i.e., subroutines or procedures) are treated. A generative procedure-level program model (PAM) is defined and used to synthesize program executions that, when used in program restructuring instead of actual execution traces, are shown to produce acceptable layouts. Moreover, the PAM program modeling system can be fully automated.

DEDICATION

To my wife and children, the driving force behind this once in a lifetime venture.

ACKNOWLEDGEMENTS

I am the product of those people and events God in His wisdom has used to make me me. The list is long, and I feel unable to express adequately my thanks to everyone -- family, friends, teachers, encouragers -- or for everything -- the successes, the setbacks, the in-between times. So I simply thank God for enabling me to profit from the finite experiences of my life and, most of all, from the infinite reality of being one of His little ones.

A special honor is due my parents who gave me life, familial love and support. I am what they could have been in different times and under different circumstances.

Lil, the darling wife of my youth, has waited patiently, feeling along with me the ups and downs of graduate school. She is God's greatest gift to me. My children, Edward, Nicholas and Daroyce, explain why it took so long and why it was worth it. Their constant love keeps me reminded that there's more to life than school or work.

The list of friends is too long. Special thanks to Roy, Maris and Ricca Pargas, with whom we spent many fun times singing, playing guitars, eating and laughing. All the students who passed through -- Ann, Judy, Lee and Ava, Candy, ... were terribly important to my survival.

I am deeply indebted to the members of my Committee. Dr. Peter Calingaert, my advisor, carefully and (almost always) patiently directed the project, always having my best interests in mind, for which I hold him in high regard. In such an endeavor as this, one needs someone short on words, but long on encouragement. Dr. Donald Stanat was that man, and I thank him. Dr. Frederick P. Brooks, Jr., helped me in many ways, spiritually and professionally; I am glad I got to know him. I thank Dr. Bharadwaj Jayaraman for his willingness to serve as a reader, and Dr. Kishor Trivedi of Duke University, who provided much help and advice during the early stages of the research. I also acknowledge the contributions of Dr. Mehdi Jazayeri and Dr. David Parnas, who helped me get the research off the ground.

I was encouraged to develop my ideas after sharing them with currently active investigators: Dr. Alan Batson, of the University of Virginia, Dr. Domenico Ferrari, of Berkeley, and Dr. Edwan Synder of the University of Washington. Mike Padrick, of the UNC Computation Center helped me locate sample programs and data files that were used in the research.

I am grateful to those institutions that provided moral and financial support: The Ford Foundation, IBM, Wachovia Bank and Trust Company, and the UNC Board of Governors. Finally, I acknowledge the encouragement, confidence and patience shown by my current employer, Winston-Salem State University, primarily in the persons of Dr. V.K. Newell and Dr. Arnold Lockett.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

This chapter contains the motivation and claims of this research, along with a survey of related research. For those readers unfamiliar with the the general area of virtual memory, Denning's excellent tutorial [DENN70] is recommended. Ferrari's survey paper [FERR76a] gives a concise introduction to program restructuring techniques.

1.1 THE PROBLEM

The paging overhead (i.e., overhead incurred when a page fault occurs) experienced by a single program executing in a demand-paged virtual memory environment depends upon four major factors:

(1) the program's module-to-module reference pattern;

(2) the program's layout in its virtual name space;

(3) the system's memory management policy, primarily, the page replacement algorithm; and

(4) competition from other programs in the system for resources.

Given the program's reference pattern and the system's page size and replacement algorithm, a <u>program restructuring procedure</u> can produce a near-optimal layout, for which the paging overhead is near its minimum. Program restructuring procedures have been shown to reduce paging overhead significantly, but the inconvenience and high cost of collecting the required reference information discourage the widespread use of this effective performance-improvement technique. [FERR76a] Low-cost, easy-to-use, language-independent restructuring packages are sorely needed to encourage the acceptance of program restructuring as an integral part of the program development cycle for large programs.

The primary objective of this research is to reduce the cost of using conventional program restructuring techniques. The objective is <u>not</u> to define brand new restructuring procedures, but to reduce the cost of obtaining the required reference information. A secondary objective is to demonstrate that procedure-level program modeling is feasible, and that it yields insights into execution-time behavior. A fully-automatic restructuring system is designed and compared to conventional approaches in terms of ease of use, effectiveness and cost. It is hoped that, with such packages available, the use of program restructuring will become as commonplace as the use of optimizing compilers to reduce a program's execution-time cost.

1.2 A VIEW OF PAGED VIRTUAL MEMORY SYSTEMS

1.2.1 Notation and Terminology

Figure 1.1 shows an overview of a paged virtual memory system. The programmer writes a program using symbolic names; the set of names used is called the symbolic name space (SNS). When the program is translated to produce a load module, each symbolically named object is assigned a virtual address in the virtual name space (VNS). The virtual address identifies the page in which the first byte of the module is stored. When the program is loaded prior to execution, the virtual name space is stored in the storage hierarchy, usually in the backing storage. Backing storage and main storage (executable memory) define the physical name space (PNS) of physical addresses referenced during program execution.

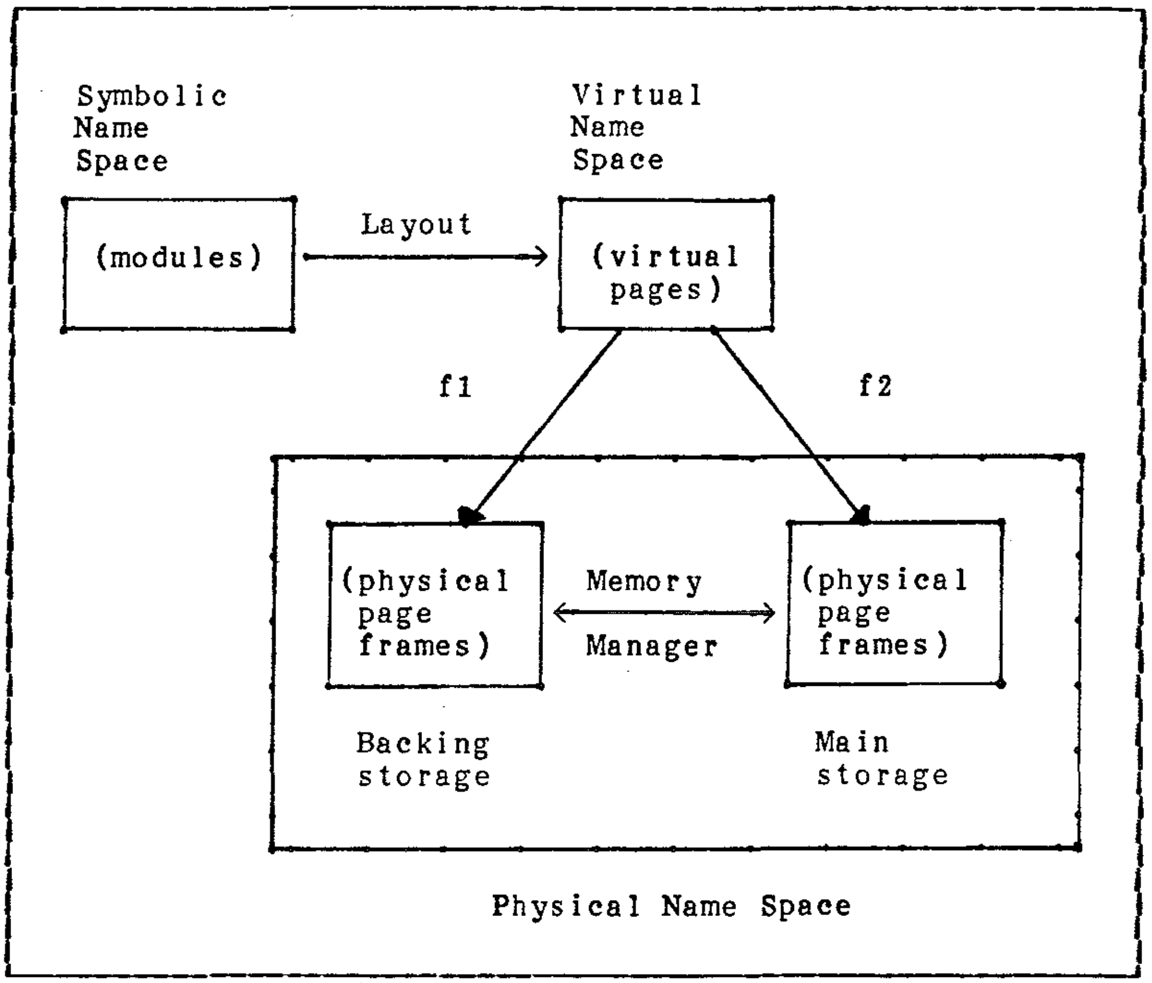

Figure 1.1
Name Spaces in a Paged Virtual Memory

The name spaces are related by way of maps that are defined at various times in the program's life cycle. The module-to-page map relates elements of the symbolic and virtual name spaces. This map is static, and is produced by a series of translation programs, including the link-editor. We call this map the layout, since it describes the layout of the program in its virtual name space. Consider the layouts of Figure 1.2 for a program consisting of modules A, B, C, D and E. Layout L0 is denoted L0=[(A),(B),(C),(D),(E)], where each module occupies a page by itself. Layout

L1=[(A,E), (B,D), (C)] assigns several modules to a single page, but allows no module to overlap more than one page. When no page overlap is permitted, the layout is a function L:SNS->VNS; otherwise, the layout defines a relation. For this research, we require layout L to be a function.

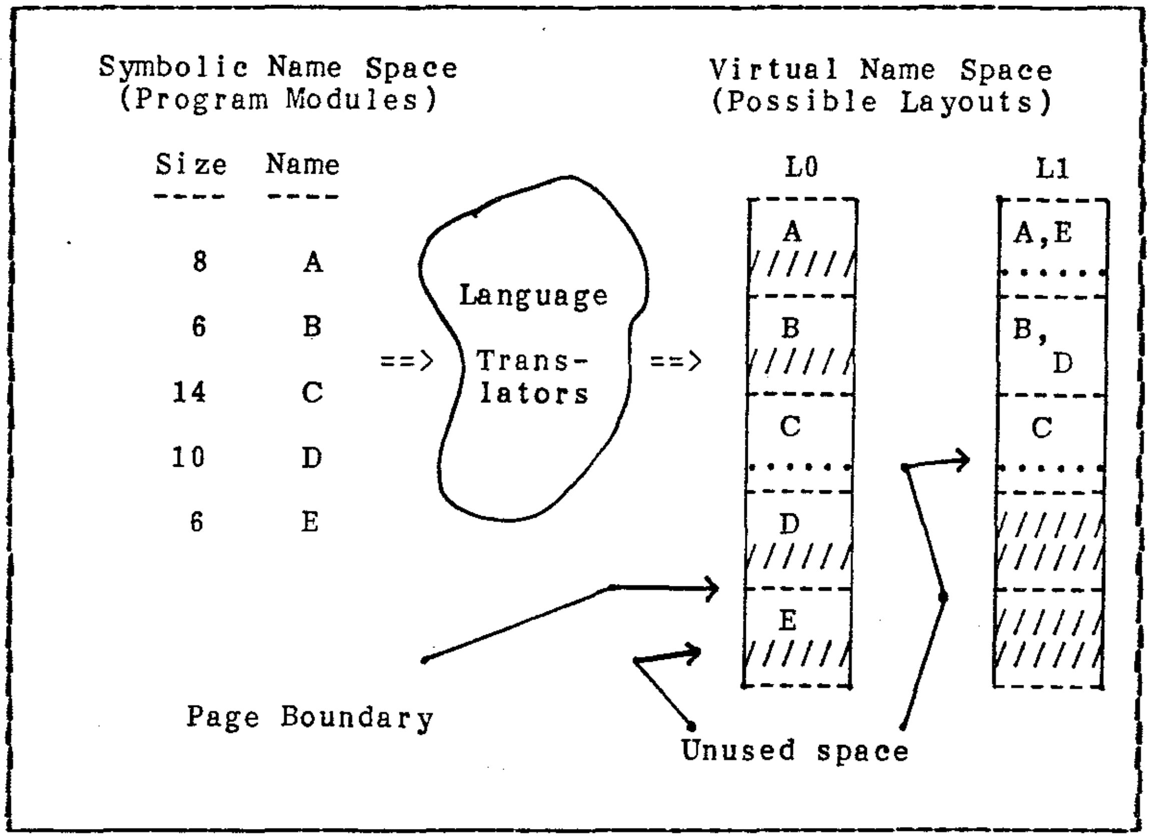

Figure 1.2
Layouts L0 and L1: Module-to-Page Mappings

The virtual and physical name spaces are related by way of two mappings. The first one, f1, maps a virtual page to a page slot in the backing store; the second, f2, maps a virtual page to a page frame in main storage. Both mappings are defined <u>dynamically</u>. Although f1 may change, it is always defined; f2 is defined only when the virtual page is in main storage. The set of virtual pages for which f2 is defined at time t is called the <u>resident set at time t</u>, and is denoted by R(t). A reference to virtual page vp1 causes a page fault when f2(vp1) is undefined. The page replacement algorithm fetches the virtual page from physical address f1(vp1) in the backing storage, stores that page in some page frame pf1 in main storage, and defines f1(vp1)<-pf1.

The dynamic mapping functions f1 and f2 represent the memory management component of a virtual memory operating system, and they reflect the system policies of allocation, replacement, and page fetching.

1.2.2 Two Important Observations

Two observations about the static and dynamic mapping functions defined above are important to this research. First, the static module-to-page map is constructed automatically, and therefore the process is amenable to improvement through more intelligent algorithms. This is the basis for applying program restructuring methods to this problem.

Second, the execution-time mapping functions, though dynamically defined, can use information gained during the definition of the static mapping to control the redefinition of the dynamic maps. The language translators can embed in the object code directives that suggest ways in which the replacement algorithm can alter its strategy of memory management. For example, the compiler can detect when an array spanning several pages will be accessed sequentially. It can then issue directives to the memory manager (replacement algorithm) suggesting that pages of the array be prefetched.

1.3 PROGRAM BEHAVIOR

1.3.1 Program Referencing Behavior

Page-level referencing behavior is captured in the page reference string x=x[1],x[2],...,x[K], where x[i] is the page that contains the i'th virtual address generated by the program. In the symbolic reference string w=w[1],w[2],...,w[M], each w[i] is a symbol in the symbolic name space. The two major behavioral characteristics present at the symbolic level are also present at the page level, namely, locality of reference and phase-transition behavior.

Locality of reference is the tendency of program references to favor a subset of the program's modules (pages) during some specific time interval. Moreover, the set of favored modules (pages), termed the locality set, tends to change membership slowly. Locality of reference is related to programming in that (1) programmers tend to concentrate on only a small collection of subproblems at any one time, and (2) programs tend to make extensive use of the looping control structure, which causes certain modules to be referenced repeatedly within short time intervals [DENN70,BATS76b]. Locality of reference does not hold throughout program execution, but during relatively long time intervals called phases, or regimes [FREI75]. Transitions between phases are marked by very poor locality of reference, resulting in a rapid increase in the rate of faults. Studies by Batson and Madison [BATS76b,MADI76], and by Denning and Kahn [DENN75] all came to the conclusion that

"... phases and transitions are of equal importance in program behavior -- long phases dominate virtual time, as anticipated by the earliest virtual memory engineers, and transitions, being unpredictable, account for a substantial part of the ... [page] faults ..."

Locality of reference explains why, for the most part, virtual memory works; the phase-transition behavior explains why page faults come in bursts, corresponding to transitions between phases.

At the symbolic level, locality of reference is caused by a group of modules being referenced close together in _time_. This type of locality is termed _temporal locality_. Storing these closely related modules within contiguous pages produces _spatial locality_, in which the next virtual address generated is likely to be numerically close to those generated in the immediate past. Spatial locality increases the density of references to a particular page (or group of pages), thereby increasing the time the page remains in main storage. For this reason, the module-to-page layout should achieve maximal spatial locality, given the symbolic-level temporal locality.

1.3.2 _Characterizing Locality of Reference_

What are some ways of characterizing or measuring locality? A single global description of a program's locality masks out the types of behavior observed within the different phases of execution. Decomposing an execution into a sequence of major phases, and studying each phase in detail is one way around this difficulty. Denning [DENN72] suggested that this decomposition might be a useful level at which to study locality. Madison and Batson's _bounded locality interval_ (BLI) method [MADI76,BATS76b] is the first widely-adopted approach to phase decomposition.

A more widely used measure of locality is the amount of faulting activity generated by a program execution, graphically displayed using a family of "fault curves." The independent variable is r, the average resident set size. The _swapping_ curve plots f(r), the number of faults when the average resident set size (allocation) is r. The _fault-rate_ curve $F(r) = f(r)/K$, where K is the number of references, plots the mean rate of faulting as a function of the allocated real storage. Finally, the _lifetime_ curve $l(r) = 1/F(r) = K/f(r)$, plots the mean number of references between faults, as a function of the allocated real storage. Consider Figures 1.3a-c, showing "fault" curves generated by two different programs P1 and P2. Notice that the fault-rate curve is a scaled version of the swapping curve. Observe that for low memory allocations, P1 performs better than P2; the relative performance switches for large allocations.

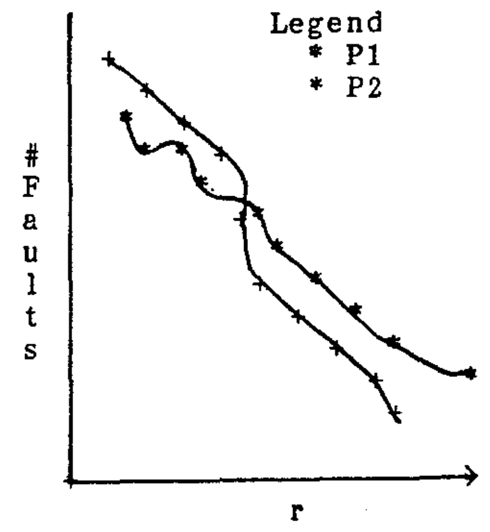

Fig 1.3a: Swapping Curves

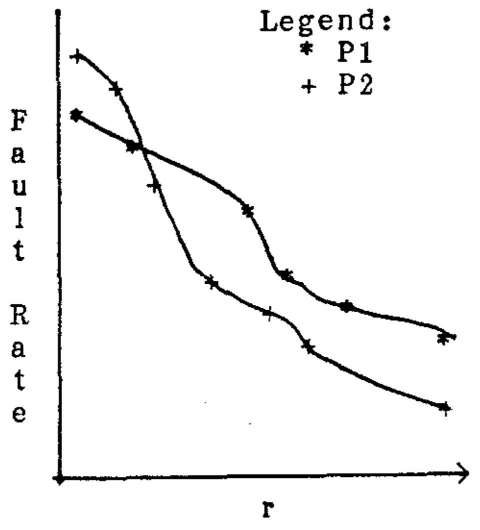

Fig 1.3b: Fault-rate Curves

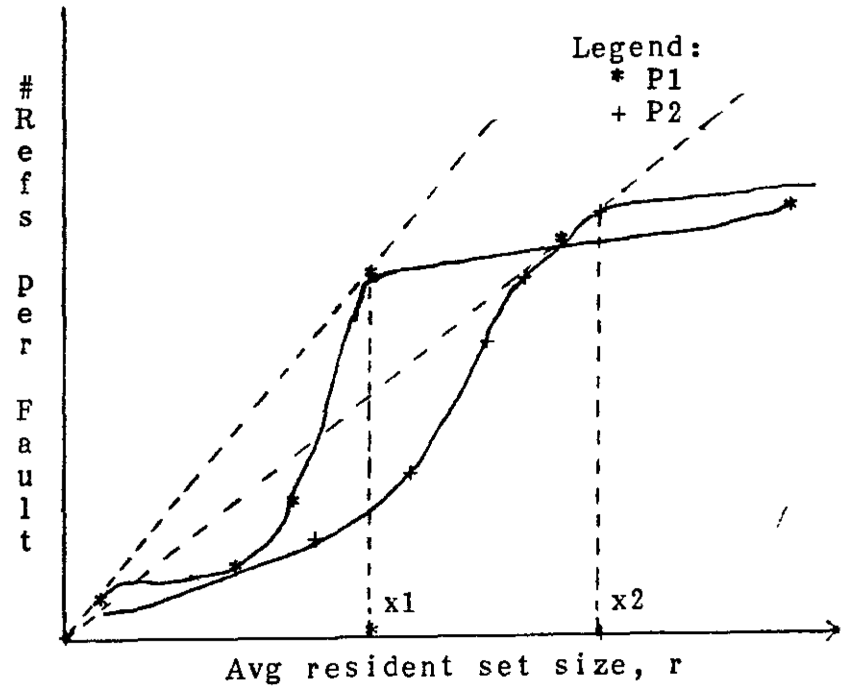

Fig 1.3c: Lifetime curves with primary knees x1 and x2

Figure 1.3

Family of Fault Curves

A _knee_ of a lifetime curve is a point (resident set size) beyond which the curve begins to flatten out. The _primary knee_ is defined geometrically as the point of tangency between the curve and a ray of maximum slope emanating from the origin. Figure 1.3c shows how the primary knee is determined graphically.

Lifetime curves have been studied extensively [BELA69,DENN76,LERO76,SPIR77], and are generally used to determine a region of memory allocation in which satisfactory paging performance is obtained. The replacement algorithm control parameter (e.g., the working set window) is then set to achieve an average resident set size within the knee region. That is, the memory allocation is set to match the locality properties of the program. One can look at locality improvement efforts as being directed at reducing a program's faulting activity at a given value of r.

One other barometer of a program's locality is its demand for space in main storage, as measured by the average resident set size, r. Under variable-allocation memory management, r reflects the over-all mean real-storage demand, constrained by the value of the control parameter. Denning's working set is the classic measure of memory demand. The working set curve (r versus WS window size T) is widely used to validate models of program behavior. Figure 1.4 shows working set curves for programs P1 and P2. The lower curve indicates smaller working set size for a given window setting, implying a greater degree of locality. Coffman and Denning [COFF73] have shown that the working set curve is a very compact description of program behavior that captures many aspects of a program's performance.

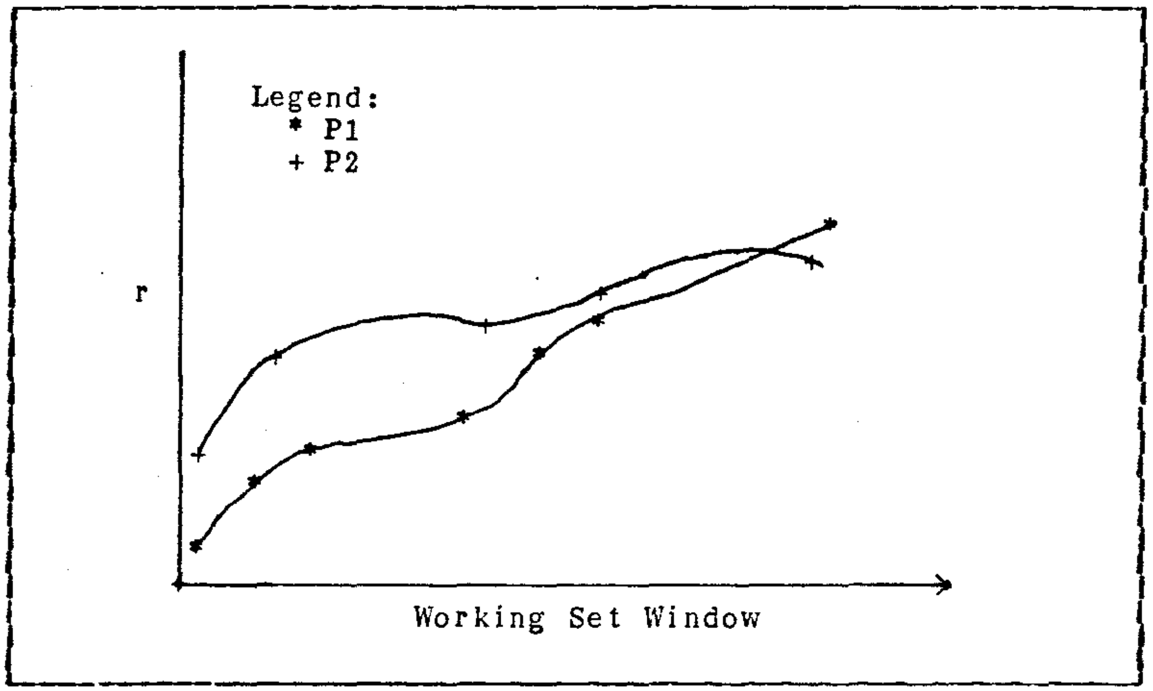

Figure 1.4
Working Set Curves

1.3.3 Phase-Transition Behavior

A reference string may be decomposed into phases and transitions, as shown below

|-X1-|---P1---|--X2--|-----P2-----|-X3-|... ,

where X's denote transitions, P's phases. The size of phase Pi is the amount of main storage required to keep its locality set Li resident. Since a phase is a period of relatively stable memory demand, loading the entire phase locality set Li into main storage at the start of Pi would bring the page fault rate down to zero for a long period of time. A reasonable goal of page replacement algorithms should be the prediction and detection of phases for the purpose of keeping resident the corresponding locality sets. Since the sizes of phases vary, the replacement algorithm must adjust to the size of the current phase to avoid excessive page faulting, on the one hand, and wasted space, on the other.

Phase-transition behavior is primarily a consequence of symbolic-level behavior: (1) traversing data structures; (2) modular design of the program, and (3) looping constructs. This suggests that language translators could, with some effort, lend a hand in the detection of phases and transitions. More specifically, translators could predict code phases, given the program's modular decomposition and a description of its use of internal control structures, such as loops and selection (conditional) constructs.

Until recently, it was believed that most paging activity was attributable to transitions between phases. Snyder [SNYD78a] has shown, however, that sequencing through large arrays spanning multiple pages can generate substantial paging activity, even within a phase of execution. This finding is important because it shows that page-level behavior <u>can not always</u> be deduced from symbolic-level behavior. This is because in the symbolic reference string, the size of a module is completely ignored, but at the page level, a reference to a module that is larger than the page size may generate references to one or more of the pages containing the module. Moreover, the identity of the referenced pages is not determinable from the symbolic reference string. Program restructuring can not be applied to such modules, but prefetching can be used to load the pages of a large module [SNYD78b,TRIV77,ABUS79].

1.4 SOME COMMON PAGE REPLACEMENT ALGORITHMS

We now present two of the most popular replacement algorithms, for the purpose of establishing notation and terminology, and to show how replacement strategies are affected by the phase-transition behavior of programs. Both algorithms are used in this research.

1.4.1 Least-Recently Used (LRU)

The LRU replacement algorithm belongs to the class of fixed-allocation algorithms, where a program is allowed a fixed maximum of m page frames in main storage at any given time. The variable m is termed the LRU control parameter, since it exerts control upon the replacement of pages. LRU(m) denotes the LRU replacement algorithm for an allocation of m page frames.

The resident set under LRU(m) at time t, R(LRU(m),t), is the set of the m most-recently referenced pages. If, at time t, a reference is made to page k, which is not in R(LRU(m),t-1), page k is placed in R(LRU(m),t), replacing the least-recently used page of R(LRU(m),t-1).

The major deficiency in fixed-allocation algorithms is that demand for pages varies according to the phase-transition behavior of the program. Within a phase, the set of pages referenced is approximately constant and the faulting rate is low, unless the size of the phase exceeds m pages. During transitions, the faulting rate is high, unless m is very large. Since the sizes of phases vary, finding the proper setting of m is not easily done using LRU.

1.4.2 Working Set (WS)

The WS replacement algorithm belongs to the class of variable-allocation algorithms, since the allocation of page frames to a program is allowed to vary during the course of program execution. WS, just like LRU, attempts to replace pages referenced furthest back in time. The WS control parameter, T, is called the WS (time) window. The resident set under WS(T) at time t, denoted W(t,T), is the set of pages referenced in the time interval [t-T,t].

During phases of execution, the working set size is stable, and may even shrink, depending upon the time between successive references to resident pages. During transitions, the working set expands rapidly, and shrinks again when the next phase is entered. This growing and shrinking, in response to program referencing patterns, is what replacement algorithms should do, instead of expecting a program to exhibit a constant demand for its pages.

1.5 VIRTUAL MEMORY PERFORMANCE

1.5.1 Program Performance Issues

Perhaps the most widely accepted cost function for measuring the performance of a program in a virtual memory environment is the *space-time product*, ST, which takes into account both the faulting behavior and the memory demand of a program. On most computer systems memory charge is assessed in "space-time" units, although the numerical value may not be that of the space-time product. Consider a program execution x=x[1],x[2],...,x[K]. (It is often convenient to associate an execution with its reference string.) Suppose x experiences N page faults at times t1,t2,...,tN, and has an average resident set size r. Then the space-time product is given by the formula

$$ST= T*r + D*\sum_{i=1}^{N} \left[|R(ti)| \right] .$$

In the formula, D is the mean (elapsed time) delay required to service a page fault, T is the total execution time, and |R(ti)| is the size of the resident set at the time of the i'th page fault. It has been shown [DENN76,GRAH76,GRAH77] that ST is close to its minimum when the average resident set size falls near the primary knee of the lifetime curve.

Notice that the two performance measures, r and N, affect ST. Both reflect the amount of locality of reference -- small values imply good locality. As we shall see later, these measures are related to each other indirectly through the module-to-page map. This suggests the potential for performance improvement through judicious definition of the module-to-page map.

1.5.2 System Performance Issues

In a multiprogrammed virtual memory system, each program contributes to the over-all performance of the system, measured by the amount of work that passes through the system per unit time. One way to achieve high throughput is to have the maximum number of simultaneous users at all times. But, since each user program's real storage requirements vary (as explained by the phase-transition behavior), simultaneous expansion of the working sets of two or more programs can lead to very high page fault rates. If too few real storage page frames are available to absorb this increased demand, thrashing results, and the system begins to spend most of its time paging, instead of completing user programs.

How, then, can near-optimum throughput be achieved, while, at the same time, staving off the onset of thrashing? Graham and Denning [GRAH77,DENN80] have found that operating each program in the region of the primary knee of its lifetime function minimizes system page fault rate, and optimizes system throughput. They have also shown [DENN80] how a working set dispatcher can be used to control the load (number of simultaneous users) in a multiprogrammed virtual memory system so that near-optimum throughput is achieved.

The foregoing discussion has an important application in this work: locality improvement techniques, such as program restructuring, should have as their goal the shifting of the location of the primary knee toward the origin. That is, restructuring should decrease the memory demand or the faulting activity of the restructured program, or both.

1.6 PERFORMANCE IMPROVEMENT TECHNIQUES

Much of the early research in virtual memory systems was directed at understanding the factors contributing to performance. Denning [DENN68a] suggested that

"... the troubles with paged memory systems arise not from any misconception about program behavior, but rather from a lack of understanding of a three-way relationship among program behavior, paging [replacement] algorithms, and the system hardware configuration ..."

Four approaches to improving virtual memory system performance have been used.

(1) Modification of parameters of the hardware configuration, such as page size or number of levels in the memory hierarchy.

(2) Modification of system policies and strategies such as scheduling and memory management algorithms.

(3) Modification of the program's reference pattern by strict adherence to programming style.

(4) Modification of the program's layout, <u>after</u> it has been translated into object form, using program restructuring.

1.6.1 Hardware Configuration

Special hardware devices and techniques are required to achieve acceptable virtual memory performance. Look-aside buffers and associative registers speed up the dynamic translation from virtual to physical addresses; faster paging drums and more intelligent drum scheduling algorithms have decreased the time to handle page faults; and, more recently, special hardware and firmware have been used to speed up memory management algorithms [COFF73].

Memory hierarchies received much attention during the early seventies [KUCK70,CHOW74]. Today, most virtual memory systems use multi-level hierarchies, although the number of levels is kept low: CPU cache, primary storage, drum or fixed-head disk primary paging device, and disk secondary paging device.

Much research was devoted to the selection of the optimal page size [HATF72,COFF73]. Small pages were found to be best for over-all main storage utilization, but slow transfer times required large pages. Large pages, on the other hand, are susceptible to "dynamic internal fragmentation" [MASU79]. What is needed is intelligent packing of pages, regardless of the page size.

1.6.2 System Resource Management

Studies of paging algorithms [BELA66,JOSE70,AHOA71] revealed several approaches to memory management: management on a local or global basis; memory allocations of a fixed or variable number of page frames; and paging on demand or in anticipation of demand (i.e., prefetching). Most algorithms can be classified according to these categories.

An interesting study by Sneeringer [SNEE75] of solutions to the performance improvement problem for time sharing systems points out a sensitive relationship between hardware and memory management software. Any solution should be based on a careful analysis of cost/performance tradeoffs affected by the cost/speed ratios of the (hardware and software) components of the solution system. A surprising conclusion she reached was that pure demand paging is not an optimal strategy for time sharing systems, unless the protection and sharing afforded by virtual memory were required.

The most recent progress in memory management is due to the elevation of the role of memory management to that of scheduling, as suggested by Kuehner and Randall [KUEH68]. A scheduler based on the working set model, and capable of controlling the load on a multiprogrammed system, has been implemented and shown to achieve near-optimal throughput [RODR73b,GRAH77,DENN80]. The area of load control through scheduling is currently quite active, with special interest in showing that proposed methods are not susceptible to unpredictable anomalies that can lead to performance degradation.

1.6.3 Programming Style

Brawn and Gustavson [BRAW68] showed that programming style affects the running time of programs in both uniprogrammed and multiprogrammed environments. They also found that, when programs are carefully written, virtual memory performance approaches the anticipated level.

Certain types of programs lend themselves to improvements through programming style. Many programs involving array operations are characterized by sequential traversal of rows or columns. The major order in which rows and columns are stored, and the way looping to access the arrays is performed can have a dramatic effect on the number of page faults generated [COFF68,MCKE69,BRAW70,MOLE72,ELSH74].

Morris [MORR73] and Rogers [ROGE75] give hints for writing high-level programs specifically for virtual memory environments. They show that knowing how a compiler builds object modules from the source program is essential to efficient program execution, and that the required modification in programming style does not appreciably increase programming costs.

One of the latest approaches to modifying the source program involves source-level transformations. Trivedi [TRIV77] and Abu-Sufah [ABUS79] transform array programs by rewriting loops to reduce the number of different arrays accessed inside each loop, thereby spacing out the expected time between successive page faults. A similar idea for optimizing array expression evaluation in APL was used by Abrams [ABRA70]. His design for an APL machine used a combination of deferred execution (which he termed "drag-along") and evaluation-sequence optimization (termed "beating"), in order to reduce the total memory requirement for expression evaluation.

1.6.4 Object Code Manipulation

Comeau [COME67] was among the first to demonstrate experimentally that program layout has a great influence upon program performance. His findings were substantiated by Tsao et al. [TSAO72], who showed, for fixed allocation replacement algorithms, that layout has a greater impact upon program performance than does the choice of replacement algorithm.

Early efforts to achieve good layouts involved modifying compilers to perform object code _pagination_ -- placement of modules within pages, avoiding page boundary overlaps. Much attention was given to ensuring that the bodies of nested loops, especially the most deeply nested ones, not overlap page boundaries. These attempts were characterized by compile-time analysis of the static source program and the use of boolean connectivity matrices as a model of program referencing [RAMA66,LOWE70,VERH71,BAER72].

These methods marked the beginning of program restructuring, which has as its goal the determination of an assignment of program modules to pages that will result in fewer execution-time page faults. The next section is devoted to reviewing program restructuring.

1.7 PROGRAM RESTRUCTURING

Program restructuring can be viewed as a so-called "optimization" such as that performed by an optimizing compiler. This section gives an overview of the goals, methods, issues and successes of program restructuring as a viable program performance improvement technique. A case is made for the adoption of automatic restructuring systems requiring minimal programmer involvement.

1.7.1 Overview

1.7.1.1 Goal of restructuring

The goal of program restructuring is the improvement of page-level locality of reference. This is accomplished by mapping (observed or predicted) module-level temporal locality into page-level spatial (within a single page, or within a cluster of pages) locality. Restructuring usually produces reductions in the number of page faults, in the average resident set size, or in both.

1.7.1.2 The basic procedure

The general procedure for program restructuring is outlined below.

Step 1: The program's symbolic name space is partitioned into relocatable blocks.

Step 2: A restructuring graph, represented by a square matrix C, is constructed. Each graph node represents a block; each edge represents a reference between two blocks. Node weights correspond to block sizes. Edge weight C[i,j] represents the closeness of blocks i and j, i.e., the savings in memory cost that is realized when blocks i and j are stored within the same page. The algorithm used to calculate edge weights is called the restructuring algorithm.

Step 3: A clustering algorithm takes the restructuring graph, and clusters blocks together into pages, attempting to maximize intra-cluster closeness, subject to the constraint that cluster size not exceed page size.

Step 4: The program blocks are relocated in the virtual name space. That is, the blocks are assigned to pages by some program such as the compiler or link-editor.

For procedure-level restructuring, the blocks in step 1 are external procedures, the restructuring graph contains (edges of) the program call-graph, and the relocation of program blocks is performed by the link-editor. The different restructuring methods differ primarily in the way in which the edge weights are assigned in step 2.

1.7.1.3 Classification of methods

Restructuring methods can be classified according to when the module-page assignments are made, what constitutes a block, and how the C[i,j] are defined.

When? Methods that are based on information obtained from one or more executions of the program are termed a posteriori methods [HATF71,FERR74a,RYDE74,MASU74,BAB077]. A priori methods are based on information derived from a static representation of the program, such as the program source code [RAMA66,LOWE70,VERH71,BAER72,SNYD78a].

What level? When the blocks of step 1 are groups of instructions or data within a procedure or data module, the method is termed an internal method. Otherwise, the method is termed an external method. External methods require no reprogramming or alteration of the object code produced by the compiler. Furthermore, the number of blocks is usually less for external methods, resulting in lower restructuring cost.

How are the C[i,j] defined? Methods that make assumptions about the page replacement algorithm under which the restructured program will executed are called *program tailoring* methods [FERR75]. Such methods have been shown to outperform non-tailoring methods consistently, because they take into account more of the factors influencing performance, namely, program behavior embodied in the reference string, and the replacement algorithm used by the system on which the program will be executed. In the next subsection we present examples of non-tailoring and tailoring approaches to defining the C[i,j].

1.7.2 *The Restructuring Phase*

The function of the restructuring phase is to define the restructuring graph, represented by the closeness matrix C. Matrix C defines a *closeness model* for the modules of the program. The closeness model is also called "inter-reference" or "affinity" model [JOHN75,MASU74,RYDE74]. Closeness models are generally symmetric, since "close to" suggests a *mutual* need for modules to be stored together within the same page.

Two issues are involved in defining the closeness model: (1) what constitutes a connection, i.e., when is C[i,j] nonzero?; and (2) how is the strength of connection defined, i.e., what values can the C[i,j] assume, and how are they assigned? In the simplest a priori closeness models, a connection is said to exist between modules i and j whenever i *can* reference j (or j *can* reference i), and the strength of connection is a constant, usually zero or unity. Such connectivity models require the least amount of information, but, not surprisingly, they yield the poorest results [HATF71,SNYD78a]. We now give examples of models that require more information and yield better results.

1.7.2.1 The Nearness method

The Nearness method of Hatfield and Gerald [HATF71] defines M[i,j] to be the number of times a reference to i is followed by a reference to j. For the symbolic reference string w="123232413121"

$$M = \begin{bmatrix} 0 & 2 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 2 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} .$$

For the Nearness method, C[i,j] is the number of times "ij" or "ji" appears in the reference string. Matrix C is defined by C = M + M', where M' is the matrix transpose of M.

$$C = \begin{bmatrix} 0 & 3 & 2 & 1 \\ 3 & 0 & 3 & 1 \\ 2 & 3 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix} .$$

1.7.2.2 Program tailoring: critical LRU

The next example of a closeness model belongs to the class of "critical-set" program tailoring methods [FERR75]. In general, a tailoring method is based upon some replacement algorithm A, having control parameter θ, denoted $A(\theta)$, and is applied to a symbolic reference string w=w[1],w[2],...,w[K]. Each symbol in w corresponds to a <u>block</u> in the symbolic name space. The behavior of $A(\theta)$ on w is simulated, assuming each block to occupy a single page. Block resident sets $R(A(\theta),t)$ are computed following each reference w[t], and are used to update matrix C, as shown in the following example.

The <u>Critical</u> <u>LRU</u> (CLRU) method assumes that LRU is the underlying replacement algorithm. The closeness measure is defined as follows. If a reference to j causes a fault, increment M[i,j] by unity for each i that belongs to the set of resident modules. Consider again the reference string "123232413121", assuming an allocation of two (block) frames. The sequence of resident sets is

{}, *{1}, *{1,2}, *{2,3}, {2,3}, {2,3}, {2,3},
*{2,4}, *{1,4}, *{1,3}, {1,3}, *{1,2}, {1,2}.

The asterisks identify resident sets formed as a result of (block) faults. The resulting closeness matrix C, denoted CLRU(2,w), is

$$\begin{bmatrix} 0 & 3 & 2 & 1 \\ 3 & 0 & 2 & 1 \\ 2 & 2 & 0 & 2 \\ 1 & 1 & 2 & 0 \end{bmatrix}$$

The "critical-set" tailoring algorithms have as their goal the reduction of the number of faults. In fact C[i,j] is the number of page faults that would go away if modules i and j are stored in the same page. Tailoring methods, in general, tend to be more expensive than non-tailoring methods, because of the simulation of the replacement algorithm, and the complexity of the code to increment the C[i,j].

1.7.2.3 Relation to program locality

Every page replacement algorithm is based on some implicit assumption about a program's pattern of page references; consequently, for programs exhibiting the assumed behavior, the replacement algorithm performs nearly optimally. When the underlying assumption is that the program exhibits locality of reference, the replacement algorithm performs the role of a locality estimator [DENN75], that is, it attempts to determine the identity of the set of favored pages in order to keep those pages resident during the time they are being favored. The goal of the restructuring phase is to determine the most likely (constrained) module locality sets, as approximated by the resident sets, and to define the closeness measure in such a way that the clustering phase will place in the same cluster (page) those modules that co-occur most frequently in the resident sets.

Program tailoring attempts to transform a module-level reference string into a page reference string whose reference pattern exhibits the locality properties assumed by the page replacement algorithm. Program tailoring methods succeed because they use replacement algorithms (primarily WS and LRU) that have been shown to be good estimators of locality. The Nearness method, which is actually a tailoring method based on the LRU(1) algorithm, is a poor estimator of locality because it has a very constrained locality set size -- one module. In light of the phase-transition view of program behavior, it is not surprising that performance gains obtained using the Nearness method are consistently less than those of other tailoring algorithms based on better locality estimators.

1.7.3 Trends in Program Restructuring

The earliest attempts at program restructuring were a priori methods applied at the internal level [RAMA66, LOWE70, VERH71, BAER72]. A priori approaches based solely on static boolean connectivity were limited in their effectiveness, mainly because boolean connectivity is a poor predictor of the dynamics of program execution [BABC77, SNYD78b]. What was needed was more information on which to base restructuring decisions. Monitoring actual program executions provided this information to a posteriori restructuring methods that achieved marked improvements over a priori approaches. The higher cost of the a posteriori approach, due to program monitoring overhead and the required analysis of the collected data, is compensated for by a higher level of performance gains achieved.

The Nearness method was among the first to make use of the dynamic behavior of the program, as embodied in the program trace. Other restructuring methods were proposed to improve upon the Nearness method [MASU74,FERR74a]. Johnson [JOHN75] and Ferrari [FERR73,FERR74a,FERR74b] were among the first to use program tailoring to achieve significant improvements over the Nearness method, but at higher analysis costs. The fully-automatic, adaptive and user-transparent system OPALE [BABO77,ACHA78], is a program tailoring approach that makes use of a program's history of faulting behavior to modify its layout periodically.

Snyder [SNYD78a,SNYD78b] has demonstrated that language-driven a priori restructuring based on static program analysis can produce layouts as good as those produced by a posteriori restructuring. He achieved this by taking into account the internal structure of each subroutine (procedure) and the semantics of parameter passing. Snyder's approach differed from the earlier a priori methods in that his layouts map externally-relocatable program components -- FORTRAN subroutines, arrays and COMMON blocks -- into pages. Snyder also proposed a method for using static structure information to perform prefetching for array processing programs, a performance improvement technique used successfully by others [JOSE70,TRIV77].

1.7.4 Summary

Program restructuring is an important technique for enhancing the performance of a program running on a paged virtual memory computer system. The effectiveness of restructuring depends upon (1) the quality of the information available about the program's symbolic-level behavior (an execution trace provides the best information), and (2) the degree to which the restructuring algorithm is based upon a good locality estimator.

A posteriori program restructuring does work, but the cost and effort required to obtain the execution trace make such restructuring economical only for often-used programs [HATF71,FERR76a] that have significant memory cost. Another factor in the widespread application of restructuring is the ease with which it can be applied. The OPALE system and the LOCALIZER [FERR73] are systems providing nearly fully-automatic restructuring capability -- steps in the right direction.

The critical-set tailoring algorithms of Ferrari represent the state of the art in a posteriori program restructuring, and thus provide an accepted base for evaluating new restructuring approaches.

1.8 OVERVIEW OF THIS RESEARCH

1.8.1 The Procedure-Activation Model: The Tool

The Procedure-Activation Model (PAM) attempts to use the way a program is written (decomposed into procedures and control structures) to predict its procedure-level reference behavior. The major advantages of such an approach are simplicity, naturalness and suitability for automation. After all, procedures and control structures are the units of program composition closest to the conceptual solution of a problem.

PAM is used to generate synthetic procedure reference strings to approximate execution trace strings. The subject (modeled) program is modeled procedure by procedure. Each procedure description has two components.

- Static component. A context-free grammar, the call-sequence grammar, describes the placement of calls and control structures (loops, conditionals and gotos) within the procedure body.

- Dynamic component. Numerical vectors, called parameters, describe the flow of control through the procedure, observed during one or more executions of the procedure.

Parameters are obtained simply by counting the execution frequency of selected program statements, an idea proposed and strongly recommended by Knuth [KNUT69], who called the counts profiles. He suggested that profiles be used in program testing to detect sections of untested code. Profiles can further be used to improve the efficiency of programs (by optimization of selected program sections) and of language translators (by identifying and optimizing the translation of language constructs used most often at a particular installation). In Chapter 2, the cost of parameter estimation is shown to be low.

There are three major steps in program modeling.

- Model construction. The call-sequence grammar is constructed, and the subject program is instrumented to produce parameters when it is executed.

- Parameter estimation. Each execution of the instrumented subject program produces a set of parameters. These can be combined with parameters from other executions to form parameters which describe a set of program executions.

- Synthetic string generation. The static and dynamic components of the model are combined, and used to generate synthetic procedure reference strings.

Each modeling step can be fully automated. The major thrust of this research is an investigation of PAM modeling techniques and the effectiveness of synthetic reference strings when used in program restructuring. A secondary concern is the cost of constructing and using the automatic modeling system.

1.8.2 The Scope of This Research

This research is basically a simulation study of the use of synthetic reference strings in program restructuring. Existing restructuring algorithms — Ferrari's CLRU and CWS -- are used, providing accepted bases for evaluating the results we obtain. A paged virtual memory machine is simulated to produce performance data.

The automatic modeling system described herein has not been implemented, but guidelines for its construction are given, and clearly show how the system can be implemented. Several PL/I programs, of medium size and complexity, are modeled and the performance of their synthetic strings in program restructuring is evaluated.

The study of model properties is limited to the use of the model in program restructuring; other applications and investigations are suggested as topics for further research.

1.9 MAJOR THESES AND CLAIMS OF THIS RESEARCH

This research proposes an automatic system for program model construction, parameter estimation and synthetic reference string generation, as a front end to a posteriori program restructuring. The theses of this research are presented in decreasing order of contribution to the state of the art.

- Program restructuring using synthetic reference strings produces layouts whose performance does not differ significantly from the performance of layouts obtained from program restructuring using execution trace strings.
- Extensive modeling effort is not required to obtain good restructuring results using synthetic reference strings.
- The synthetic-reference-string program restructuring system can be fully automated.
- The cost of constructing and using the automatic modeling system does not exceed the cost of the execution-tracing approach.

The following claims, although they do not constitute theses, do point out some of the more promising features of the model developed in this study. The PAM model has possible application beyond that attempted in this research.

- PAM is a useful conceptual model that gives insights into the relationship between program structure and program referencing behavior.
- PAM instrumentation and parameter estimation provide a low-cost way of monitoring program execution, and can be used in automatic program testing.

1.10 ORGANIZATION OF THE DISSERTATION

This chapter has presented a brief overview of the research, along with sufficient background material for reading the remaining chapters.

The Procedure-Activation Model (PAM) is defined in Chapter 2. A completely automatic modeling system is designed, and shown to require low overhead. The model is shown to be flexible enough to accommodate different approaches to modeling, specifically in the areas of parameter estimation and string generation.

Chapter 3 describes the subject programs used in the research, reviews the issues of program restructuring relevant to the research, and defines the layout comparison methodology. Finally, results are presented that show the degree to which the execution of the subject programs can be improved through the use of restructuring.

Chapters 4, 5 and 6 present empirical results from program restructuring experiments that used the synthetic reference strings generated from instances of PAM. Chapter 4 contains results from modeling using the simplest model versions and parameter estimation approaches. In Chapter 5, more sophisticated model versions and parameter estimation were used in an attempt to improve model accuracy. Chapter 6 presents a case study, where the techniques of Chapters 3-5 were applied to a final subject program.

Chapter 7 contains conclusions and recommendations for further investigation.

A glossary of acronymns is provided for easy reference to the many acronymns used throughout the dissertation.

Chapter 2

THE PROCEDURE-ACTIVATION MODEL

The Procedure-activation Model (PAM) is defined in this chapter. We begin by defining SSPL, a simple structured programming language that will be used as the high level source language in which program examples are written, and as the language in which the model construction algorithms are expressed. Several methods of estimating model parameters are given, followed by a discussion of synthetic string generation techniques. We conclude the chapter with an over-all analysis of the cost of using the model, and with the definition of model variants. We show that the entire modeling process can be fully automated.

2.1 PRELIMINARIES

2.1.1 A Simple Structured Programming Language

The programming language SSPL contains the basic sequencing primitives that allow looping, selection, procedure calls and escapes (restricted branching).

All procedures are external. The syntax of the procedure statement is

```
proc <procname> <parameter list>

    <declarations>

    <stmt>

endproc
```

The main procedure is distinguished by the occurrence of the keyword main in the parameter list.

2.1.1.1 Sequencing primitives

Construct	Syntax	Example

looping	repeat(control clause)
	  statement
	endrepeat;
		repeat(for X:=1 to 10)
		  S:=S+X;
		endrepeat;

selection	select
	  if(cond1) stmt1
	      ...
	  else stmt
	endselect;
		select
		  if(X>Y) X:=X+Y;
		  if(X<Y) Z:=X;
		  else Z:=X-Y;
		endselect;

Escape	escape<arith expr>;	escape 3;
Call	call <procname>;	call A;
Return	return;	return;

The statement "escape n " causes an exit from n levels of control structure nesting. When n≤0, no branch is executed; when n is greater than the nesting level, the effect of the statement is that of the "return " statement.

2.1.1.2 Data types

Three attributes characterize variables:

- scope -- local or global
- structure -- scalar or array
- type -- integer or real or string.

2.1.1.3 Example program P1

We conclude the introduction of SSPL with an example. Program P1, shown in Figures 2.1(a-c), will be used throughout this chapter. Its function is of no particular importance; we merely need to study its structure, i.e., the sequence and nesting of control structures.

```
proc A(main);
    global integer array F[1:50];
    global integer scalar A1,A2,A3;
    local integer scalar I,J,N,X,S;
    read(N,X);
    repeat(for I:=1 to N)
        read(F[I]);
    endrepeat;
    select
        if (N>25) call B(F,25);
        else
            call C(N,X);
            S:=0;
            repeat(for I:=1 to N)
                S:=S+F[I];
                call D(I,F[I],S);
                call B(F,N);
            endrepeat;
    endselect;
    S:=0;
    repeat(for I:=1 to J while S<X)
        call E(F[I]);
        S:=S+F[I];
    endrepeat;
endproc A;
```

Figure 2-1 (a)
Procedure A of Sample Program P1

```
proc B(X,N);
  integer scalar N;
  integer array X;
  global integer array F[1:50];
  local integer scalar I,J,U;
  U:=X[1];
  repeat(for J:=1 to N)
      call C(N,U);
      select
          if (U<F[N]) call D(I,F[J],U);
          if (U>F[N])
              call C(N,F[N]);
              return;
          else escape 2;
      endselect;
      U:=X[I+1];
  endrepeat;
  call E(I);
endproc B;
```

Figure _2.1_ (_b_)
Procedure B of Sample Program P1

```
proc C(U,V);
  integer scalar U,V;
  local integer scalar S;
  S:=U+V;
  select
      if (U=V) S:=0;
      if (U>V) call E(V,U);
      else call D(U,V,S);
  endselect;
endproc C;

proc D(L,M,N);
  integer scalar L,M,N;
  call E(L);
  call E(M);
  call E(N);
endproc D;

proc E(X);
  integer scalar X;
  print(X);
endproc E;
```

Figure 2.1 (c)
Procedures C-E of Sample Program P1

2.1.2 Model Assumptions

2.1.2.1 No data references

Data references are not modeled, for three reasons. The first and most important reason is that no efficient software tools besides execution interpretation are available for monitoring data references. Tracing data references slows program execution [HATF71]. Second, passing parameters by reference makes it impossible to model data references using a context-free grammar, since the identity of the data item referenced inside the called procedure depends upon the site of the call, i.e., the context of the caller. Any attempt to solve this problem would require a grammar more powerful than context-free. The third reason is that, even if it could be done, the grammar used to describe all possible reference sequences would be so large that synthetic string generation would be very inefficient.

2.1.2.2 Predictable flow of control

Flow of control must be predictable from the source code, execution must always begin at the first statement of a code module, and a procedure must always return control to its most recent caller. Therefore, coroutines and interrupt routines are not modeled.

2.1.2.3 Call-path independence

A procedure's behavior is assumed to be independent of the program's call history. That is, a procedure is assumed to behave the same way each time it is called, and therefore can not be influenced to behave differently for different callers. This assumption is required because the call-sequence grammar is context-free. Procedures violating this assumption can nonetheless be modeled, as we shall see in a later section.

2.1.3 The Automatic Modeling System

In Chapter 1 we outlined the major phases of PAM modeling:

(1) call-sequence grammar construction and program instrumentation;
(2) program execution and parameter estimation; and
(3) synthetic string generation.

These steps correspond to instrumentation, observation and generation. The process is described in Figure 2.2. In the figure, rectangles indicate steps in the process , and arrows indicate data flow into and out of each step.

The program to be modeled is called the <u>subject</u> program. Model construction produces three outputs: the call-sequence grammar (CSG); instrumented object code for the subject program (ICBJ); and the parameter descriptor database (PDDB), which describes the parameters required to model each procedure. Parameter estimation involves executing IOBJ a number of times. Each execution produces execution <u>coefficients</u>, counts of loop and selection construct usage, which are stored in the coefficient database (CDB) following program execution. Coefficients must be converted into the parameters for use during actual synthetic string generation. Parameters are stored in the parameter database (PDB). Synthetic string generation requires the grammar (CSG) and parameters (PDB) as inputs, and produces one or more synthetic strings (SYN) as output.

The call-sequence grammar and the parameters used in synthetic string generation constitute an <u>instance of PAM</u> for the subject program. A model instance for program P is denoted PAM(P,W) = <CSG(P),PARM(P,W)>. PARM(P,W) denotes the model parameters derived from a set W of executions of the instrumented object code (IOBJ). When no ambiguity can result, the model instance can be denoted PAM(W) = <CSG,PARM(W)>.

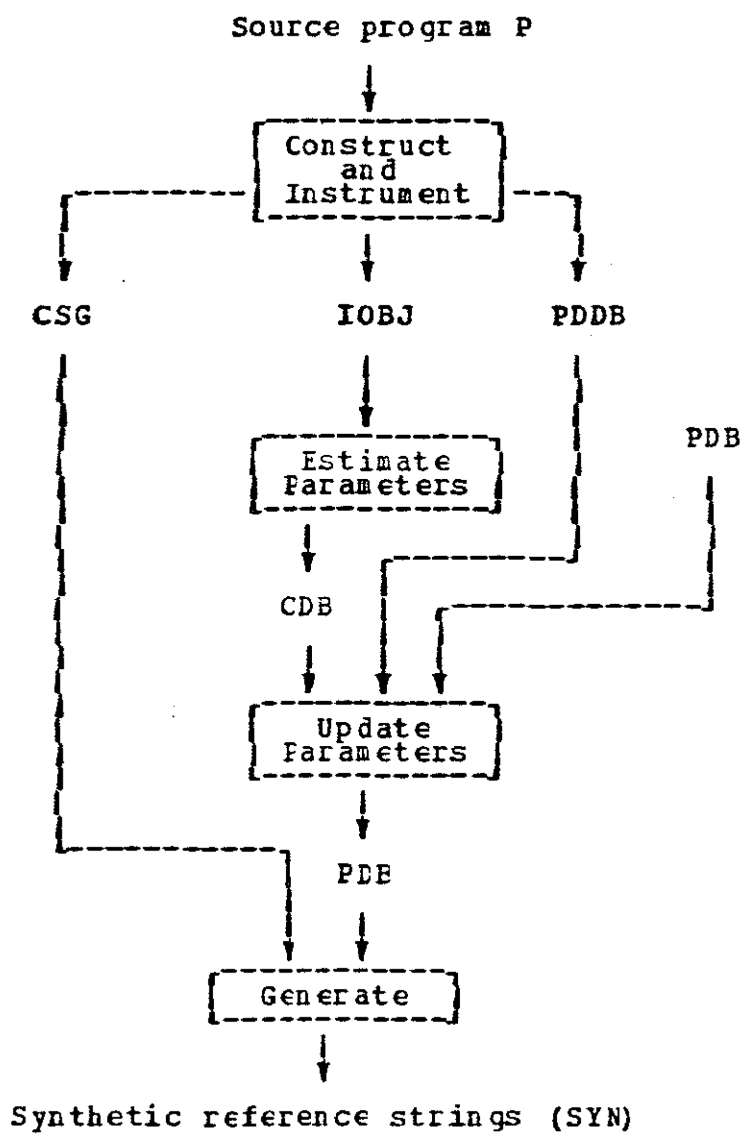

Figure 2.2
The Automatic Modeling System

The next three sections of this chapter discuss the phases of PAM modeling in terms of inputs, data structures, algorithms, output and cost. Particular attention is given to showing that the automatic modeling system can easily be integrated into existing system software to provide an additional program "optimization" akin to that provided by optimizing compilers.

2.2 MODEL CONSTRUCTION

The construction phase of PAM modeling is depicted in Figure 2.3. The source text of a subject program module is the input; the outputs are its PAM call-sequence grammar, an instrumented version of its object code (IOBJ) and a database record describing the parameters required by the CSG. Each output will be described in the succeeding sections.

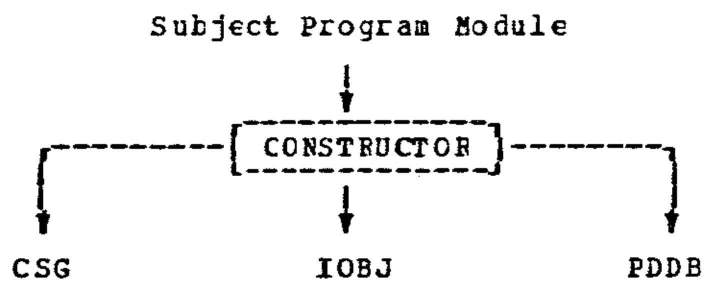

Figure 2.3
Model Construction

We should point out here that the assumptions and algorithms we are about to present probably do not represent the most efficient way to construct automatically the model of the subject program. Our intention is rather to present straightforward algorithms and analyses that give insights into the nature of the model construction process.

2.2.1 The Call Sequence Grammar

The call-sequence grammar (CSG) for program P1 is given in Figure 2.4. The CSG gives a linear representation of the structure of each procedure, i.e., the placement of *significant control structures* within the procedure. A significant control structure is defined recursively as

(1) a call or return statement;

(2) a loop or selection construct that contains a significant control structure;

(3) an escape from a significant control structure.

Only significant control structures affect the sequence of execution-time procedure calls.

```
A = ( B | C < D B > ) < E >;
B = < C ( D | C! | -2 ) > E;
C = (    | E | D );
D = E E E;
E = ;
```

Figure 2.4
Call-Sequence Grammar for Sample Program P1

Consider the CSG production for procedure A, which shows that either A calls B, followed by repeated calls to E, or A calls C, followed by repeated calls to D and B, followed by repeated calls to E. In the CSG production for B, the selection (or conditional) construct is nested inside the loop construct. Whenever the second alternative is selected, C is called and B immediately returns to its caller; whenever the third alternative is selected, an exit is made from two levels of control structure nesting, resulting in the call to E. The production for procedure C has as its first alternative a null call, which corresponds to an execution of the selection construct that does not lead to a procedure call. All alternatives of a significant selection construct must be modeled.

A summary of the CSG operators and the corresponding SSPL keywords is presented below.

<u>Operator(s)</u>	<u>SSPL Keyword</u>	<u>Explanation</u>
"<","> "	<u>repeat</u>,<u>endrepeat</u>	loop delimiters.
"(",")"	<u>select</u>,<u>endselect</u>	selection construct delimiters
"|"	<u>if</u>,<u>else</u>	alternative separators.
"–"	<u>escape</u>	exit control structure.
"!"	<u>return</u>	exit from procedure call.
"="	<u>proc</u>	start of CSG production definition. Left-hand side is name of procedure; right-hand side contains operators and procedure names.
";"	<u>procend</u>	production terminator.

2.2.2 The CONSTRUCTOR

The table of CSG operators and SSPL keywords reveals that CSG construction can be keyword driven and, as such, it can be incorporated into either the lexical or syntactic analysis phase of a compiler. We now present one such (not necessarily optimal) algorithm for CSG construction in order to show the amount of work that is required. This algorithm, called the CONSTRUCTOR, takes as input the SSPL source code for an arbitrary procedure P, and produces as output the CSG production for P. The CONSTRUCTOR may also be used to produce other outputs that will be used in later phases of the modeling process, such as the instrumented source program (source level or object level) and a descriptor for the parameters required for that production. In the next section we discuss program instrumentaton; here we will show how the grammar and parameter descriptor are constructed.

2.2.2.1 Required data structures

Active Construct Stack (ACS). A construct is active if its initial delimiter has been scanned, but not its terminal delimiter. Each stack element has the following format.

| construct | count | size | #params |, where

construct encodes the type of control structure:

-n -- the n'th loop;

0 -- the start of procedure;

+n -- the n'th selection construct.

count is the number of significant control structures within the construct.

size is the number of CSG symbols generated.

#params is the number of parameters required.

Output buffer (OB) contains CSG symbols generated during model construction.

Construct Descriptor Table (CDT). Each construct in the final CSG production has a table entry of the form

construct	#params

.

The *Parameter Descriptor* (PD) record for each production has the format shown in Figure 2.5. It describes the number and types of constructs contained in the production, along with the number of parameters required for each construct, and for the production as a whole. As an example, procedure A has the production parameter descriptor

A	6	2	1	2

.

Procedure A requires six parameters: two each for its two loops, and two for its one two-way selection construct. The ui in Figure 2.5 are referred to collectively as the *selection descriptor list* (SDL).

```
| procname | L | l | q | u1 | u2 | ... | uq |
```

procname is the name of the procedure;

L is the total number of parameters required;

l is the number of loops;

q is the number of selection constructs;

ui is the number of alternatives in the
    i'th selection construct.

<u>Figure 2.5</u>
Parameter Descriptor Data Base Record Format

2.2.2.2 Description of the algorithm*

The output buffer OB is filled one symbol at a time. When a construct initiator symbol (a "<u>proc</u>", "<u>repeat</u>" or "<u>select</u>") is scanned, an ACS record is pushed onto the stack, and the corresponding CSG output symbol is moved to OB. The top element of ACS is the <u>current construct</u>; its count field is incremented each time "<u>call</u>" is scanned; its size field is incremented each time a CSG symbol is moved to OB; and its #params field is initialized to two for loops, and is incremented by one each time "<u>if</u>" or "<u>else</u>" is scanned. When the current construct terminator (an "<u>endproc</u>", "<u>endrepeat</u>" or "<u>endselect</u>") is scanned, its CSG output symbol is moved to OB. If, when the end of the current con-

*This algorithm does not handle escapes. A significant escape from a construct makes that construct significant, and vice-versa. Determination of significance requires look-ahead -- to the end of a (possibly nested) construct. Algorithm modifications required to handle escapes are similar to the use of branch-ahead tables in assemblers and compilers. We will not give further details.

struct is reached, its count field is zero (i.e., no calls were made from within the construct), OB is purged of all output symbols generated by that construct. The size field contains the number of symbols to be deleted. If the count field is nonzero, a CDT entry is made for the construct, the element is popped from ACS, and the count field of the new current construct is incremented by the count field of the old current construct (this reflects the nesting of control constructs).

The PD is constructed from the CDT after the source program has been scanned. The Parameter Descriptor records for the procedures of program P1 are given in the following table.

name	#params	#loops	#selects	SDL
A	6	2	1	2
B	5	1	1	3
C	3	0	1	3
D	0	-	-	-
E	0	-	-	-

2.2.2.3 Discussion

This algorithm requires only one pass over the source program, and can be incorporated into the lexical analysis phase of the compiler if the source language contains the equivalents of SSPL constructs. Otherwise, for languages such as FORTRAN, some constructs must be simulated using the sequencing primitives available, which may require the CSG construction to be incorporated into the syntactic analysis phase of the compiler.

2.2.3 Instrumenting the Subject Program

The second aspect of model construction is the instrumentation of the subject program so that it will produce parameter estimates when it is executed. Instrumentation involves

-allocation of instrumentation variables (ivars);

-insertion of instrumentation code (icode) to cause ivars to be incremented;

-insertion of code to write the values of ivars to the coefficient database.

The values of the ivars from a single execution of the subject program are called coefficients, and are written to the coefficient database (CDB). Coefficients from one or more executions are combined to form the parameter database (PDB).

We now show how the CONSTRUCTOR can be extended to instrument the subject program. Again, we will discuss the required modifications to the data structures. The instrumentation algorithm we now present is decomposed into two passes to enhance clarity, although an actual implementation may use only a single pass.

2.2.3.1 Allocation of instrumentation variables

The number of ivars required for a procedure depends upon its CSG production. Each loop construct requires two, one to count the number of times the loop is entered, the other to count the total number of times the body is executed. Each selection construct requires one ivar per alternative. Instrumentation variables are assigned to loops first, in the left-to-right order in which the loops appear in the CSG production. The first loop in the production is assigned ivar[1] and ivar[2]. Next, ivars are assigned to selection constructs in a left-to-right order of the constructs; within a construct, the ivars are assigned consecutively. Consider the CSG production right-hand side below.

$$\begin{tikzcd}
< \arrow[d] & A & (\arrow[d] & B & | \arrow[d] & D & (\arrow[d] & B & | \arrow[d] &) & | \arrow[d] & < \arrow[d] & C & > &) & > & ; \\
1,2 & & 5 & & 6 & & 8 & & 9 & & 7 & 3,4 & & & & &
\end{tikzcd}$$

The indices of the ivars assigned to each construct appear underneath the respective CSG construct symbols.

Instrumentation can be viewed as a source-language transformation that inserts statements within the source program. Allocation of ivars amounts to declaring an array to contain the ivars, namely the SSPL statement

global integer array <procname>ivar[1:#params]

2.2.3.2 Insertion of instrumentation code

Instrumentation code takes the form "incr(n)", which causes ivar[n] to be incremented by unity. Insertion of icode requires two items of information: the ivar index and the insertion point.

The location of the insertion point is based on syntax, as shown below. Insertion points are indicated by asterisks.

```
Loop Construct: *1 repeat(   ) *2 stmt endrepeat
Selection construct:
        select if (cond)  *  stmt1
              . . .
              else  *  stmtn
        endselect
```

At the loop instrumentation point *1, the loop entrance frequency -- the number of times the loop is entered from the top -- is determined. The loop repetition frequency -- the number of times the loop body is executed -- is computed at *2.

Insertion points can be determined during pass one, as the CSG is being constructed. The ACS and CDT are extended to include a field that points to a list of construct insertion points. An insertion point list (IPL) has nodes of the form

ip	ivar	next

,

where ivar is the index of the ivar to be incremented, ip is the position in the source (or intermediate) code after which the icode is to be inserted, and next is the list pointer. For simplicity, ip is expressed as the serial number of the source program symbol following which the insertion is to be made.

At the end of pass one, the ivar field of each IPL node is assigned an ivar index. During pass two, each time an insertion point is reached, the icode to increment the corresponding ivar is inserted. The instrumented version of procedure B of program P1 is given in Figure 2.6.

```
proc B(X,N);
    integer scalar N;
    integer array X;
    local integer scalar I,J,U;
    global integer array Bivar[1:5];
    U:=X[1];
    incr(1);
    repeat(for J:=1 to N)
        incr(2);
        call C(N,U);
        select
            if (U<A[N])
                incr(3);
                call D(I,A[I],U);
            if (U>A[N])
                incr(4);
                call C(N,A[N]);
                return;
            else
                incr(5);
                escape 2;
        endselect;
        U:=X[I+1];
    endrepeat;
    call E(I);
endproc B;
```

Figure 2.6
Instrumented Procedure B of Program P1

2.2.3.3 Extraction of execution coefficients

The execution coefficients must be written to the coefficient database after the termination of the execution of the subject program. A straightforward instrumentation to cause this to happen requires two new procedures, WRITECR and DUMP. WRITECR writes the coefficients belonging to one subject program procedure. It takes as parameters the name of the procedure, the number of parameters (ivars) and the ivar array for that procedure, and writes a CDB record having the format

$$\boxed{\text{procname} \mid \#\text{params} \mid \text{ivar}[1{:}\#\text{params}]}\,.$$

Procedure DUMP, which contains (global) ivar declarations from each procedure, passes to WRITECR the ivars from the executions of each procedure. DUMP can be generated from the information in the parameter descriptor database (PDDB). A call to DUMP must be the last statement executed in the main procedure. Figures 2.7 and 2.8 show the DUMP and WRITECR instrumentation routines for program P1.

This instrumentation produces output that must be manipulated further to make it suitable for generating synthetic strings. The parameter database is constructed from the coefficient databases produced by multiple program executions, and may contain detailed historical and statistical data such as extrema, averages and variances of coefficients of selected constructs. For example, the generation algorithm may require that loop parameters be expressed as a single scalar representing the average loop repetition frequency, or as a range [n1,n2], where n1 and n2 are, respectively, the minimum and maximum loop repetition frequencies, or as a pair of descriptive statistics (e.g., mean and variance of repetition frequency).

```
    proc DUMP;
        global integer array Aivar[1:6];
        global integer array Bivar[1:5];
        global integer array Civar[1:3];
        call WRITECR("A",6,Aivar);
        call WRITECR("B",5,Bivar);
        call WRITECR("C",3,Civar);
    endproc DUMP;
```

Figure 2.7

Instrumentation Routine DUMP

```
proc WRITECR(PROCNAME,#PARAMS2,IVAR);
    scalar string PROCNAME;
    scalar integer #PARAMS2;
    integer array IVAR;
    write (PROCNAME,#PARAMS2,IVAR) to file(CDB);
endproc WRITECR;
```

Figure 2.8

Instrumentation Routine WRITECR

2.3 PARAMETER ESTIMATION

Parameter estimation can begin during the testing and debugging of individual modules, provided that the tests use representative input. Such an early start can provide insights into the nature of the program long before it has been completed. In fact, PAM parameter estimates provide a characterization of the data used in program testing, as follows. If exhaustive program unit (modules and control structures) testing is desired, a set of test data is assembled, the program is executed using the test data to produce execution statistics from which can be determined which modules and module units have been exercised. Additional test data are created until all the desired program units have been exercised. PAM parameters record precisely the information of interest, provided that the program has been instrumented to produce execution coefficients.

Parameter estimation is depicted in Figure 2.9, which shows the required databases and processing.

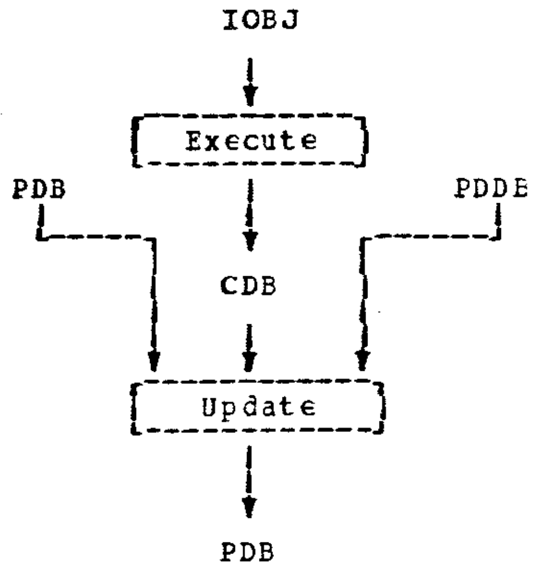

Figure 2.9

Parameter Estimation

2.3.1 Maintaining the Parameter Database

The various databases, Parameter Descriptor (PDDB), Coefficient (CDB) and Parameter (PDB), which are used during parameter estimation, are described in Figures 2.5, 2.10 and 2.11, respectively.

During the PDB update, the procname field is used to match records from each of the three databases, PDDB, CDB and PDB. The CDB ui values for alternative constructs are added tc the corresponding PDB cjk fields, where the PDDB record is used to determine the correspondence between the ui and cjk fields. The first 2*l ivars are allocated to loops. The average repetition count for loop j is the quotient u[2j]/u[2j-1]. For loop i, mi, Mi, ai and ni are updated using u[2i-1] and u[2i].

```
[ procname | K | u1 | u2 | ... | uK ]
```

K is the number of ivars;

ui is the value of the i'th ivar.

Figure 2.10
Coefficient Database Record Format

```
procname|N|m1|M1|n1|a1|v1|...|c11|c12|...|c[q,nq]
```

N is the number of numeric subfields in the record;

mi is the average minimum loop i repetition count;

Mi is the average maximum loop i repetition count;

ni is the total number of loop i executions;

ai is the average repetition frequency for loop i.

vi is the variance of repetition frequencies for loop i.

cjk is the cumulative execution count for the k'th alternative of the j'th selection unit.

<u>Figure 2.11</u>

A Parameter Database Record Format

2.3.2 Approaches to Execution Sampling

Estimating PAM parameters involves *observing*, or *sampling*, one or more executions of the instrumented subject program. One would expect that as the number of observations increases, so would the confidence one could place in the quality of the estimates. We now consider three approaches, and show that the PAM modeling system database contains (or can be made to contain) the information necessary to support any of these approaches. PAM parameter estimation produces a database which characterizes the observed executions.

2.3.2.1 One-time observation

A single execution of the subject program is used to collect parameters. Although such an approach may seem unacceptable, there are cases when it will provide a very good characterization of the subject program -- when the program is *data-insensitive*. Some programs have been found to exhibit this property. But unless the subject program is known with certainty to have this property, a single observation should not be used.

The pursuit of a representative sample is common to any sampling endeavor, and the intuition that the larger the sample, the more representative it is, gives a good rule of thumb for sampling. What is needed is a way of characterizing a sample, so that uncertainty can be dealt with using statistical methods. PAM parameters, which, as we shall see later, can be determined at a very low cost, provide such a characterization.

2.3.2.2 Predetermined number of observations

For most programs, execution time behavior depends upon the inputs used to drive the program. Although the set of all possible input values is infinite, the set can be partitioned into a small number of groups Gi, such that within a particular group, program behavior is (approximately) constant. For such programs a reasonable stratified sampling method is to choose representative inputs from each data group Gi, to observe the program's execution with these inputs, and to form cumulative parameters from these observations. This approach requires that the programmer or user of the program know the expected range of inputs and the effect of each group of inputs upon the program's control flow.

Another multiple-observation approach is random sampling. That is, during the in-production lifetime of the program, executions to be sampled are chosen at random, without regard to the type of inputs used for the execution. This approach may require a larger number of observations than stratified sampling, but it does appeal to those desiring some statistical basis for the parameter estimation approach. Moreover, such an estimation can be undertaken without depending upon (possibly inaccurate) information from programmers or users.

In both the stratified and random approaches, the number of observations is predetermined. Factors contributing to the number of observations include the sensitivity of the subject program to its inputs, deadlines for completing the observations, and the amount of programmer involvement required. Of the two approaches, random sampling involves no programmer involvement.

2.3.2.3 Statistically controlled observations

The first two approaches to execution sampling do not attempt an explicit statistical characterization of the sample. Under these conditions, one may be concerned with the possibility of failing to model the program's intrinsic behavior. Granted, the possibility exists that a small sample size will give an inaccurate view of the subject program, but because most programs tend to be quite data-insensitive [HATF71,FERR76a],eger array_ Bivar[1:5]; we feel that the likelihood of this happening to the detriment of restructuring results is not great.

Although we do not think it necessary to control the sampling process by statistical analysis, whenever such an approach is required, PAM lends itself to statistical parameter analysis. The parameter database can be expanded to keep track of the distribution of <u>each</u> parameter. Based upon parameter distribution measures (means, variances) and assumptions (such as normality), statistical inference methods can be used to decide when the sampling process can be terminated. Notice that a parameter-by-parameter statistical testing procedure involves considerable computation. This testing process can be simplified by focusing attention on critical constructs that are felt to contribute most to the characterization of the program's execution, e.g., construct parameters for top-level modules.

2.4 SYNTHETIC REFERENCE STRING GENERATION

In this section we describe the algorithm for string generation. As in previous sections, we begin with a definition of data structures, and proceed to describe the algorithms in terms of manipulations of the data structures. After we present the generator, we discuss approaches to determining when to terminate the generation process.

2.4.1 The GENERATOR

The major data structures required by the GENERATOR are the parameterized CSG (PCSG), the production descriptor table (PDT), the generator stack (GS) and the loop stack (LS). The PCSG and PDT are constructed prior to actual generation; GS and LS represent the state of the generation process. We now describe each data structure.

2.4.1.1 Data structures

Parameterized CSG (PCSG). The CSG and the parameter database are combined to form an internal representation of each production. PCSG is a one-dimensional array which contains a parameterized production for each procedure. Within PCSG, productions are stored contiguously. Parameters are inserted between grammar elements as follows:

loops*: < body > becomes < n1 n2 body >, where n1 and n2 specify the range of repetition counts for the loop; and

---

*We use this form of loop parameter primarily for illustration. In a later section we suggest other representations.

selection constructs:

(case1 | case2 | ... | casem) becomes
(T k1 case1 | k2 case2 | ... | km casem),
where T= k1 + k2 + ... + km, and each ki is proportional to observed frequency of selecting alternative i.

Production Descriptor Table (PDT). This table is used to look up the start and end of each production in PCSG. Each table entry is of the form

$\boxed{\text{first} \mid \text{last}}$, where

first(P) is the index into PCSG of the start of the parameterized production for procedure P; last(P) is the index of the final production symbol.

Loop Stack (LS). This global stack is used to keep track of generate-time looping. Conceptually, each production has its own stack of elements of the form

$\boxed{\text{n} \mid \text{start}}$, where

n is the the number of remaining repetitions;

start is the cursor position (in PCSG) of the start of the loop body.

Upon exit from a production (procedure), any loops active within that production are automatically popped from the loop stack.

Generator Stack (GS), with stack pointer GSP. This stack keeps track of the generate-time state. Each element has the format

prodn	cursor

, where

prodn is the production number;

cursor indicates the current position in the production (i.e., an index into PCSG).

Output Buffer (OB) contains output symbols.

2.4.1.2 Data manipulation primitives

The following data manipulation primitives are used in the description of the algorithm.

push(stackname,[datalist]) pushes data onto specified stack.

pop(stackname,[resultlist]) pops specified stack and extracts information into result variables.

output(symbol) places symbol in OB.

uniform(p,q) generates a random number uniformly distributed between p and q.

select(n,cursor) chooses an index of a selection construct alternative, given random number n, and the current cursor position.

findend(C,m) determines the cursor position of the end of the m'th enclosing construct, where the current cursor position is C.

loopcount(C,x) determines the amount by which the loop nesting level will decrease when moving from cursor position C to position x.

2.4.1.3 Notation

During generation, activity is centered around the generator stack, GS. To facilitate describing the generation algorithm, we will use the following symbols to refer to the data on the top of the GS stack. (Recall that GSP is the stack-top pointer for GS.)

P = GS(GSP).prodn, the current production;

C = GS(GSP).cursor, the current cursor position within the current production;

S = PCSG(C), the current CSG symbol within the current production.

The symbol N represents a PCSG nonterminal (i.e., the name of a program module), as distinguished from the PCSG operators and parameters.

Suppose that the main procedure of program P is MAIN. The first production in PCSG is "$ = MAIN ‡", where "$" is the start symbol, representing the operating system function of job initiation, and "‡", the special terminator symbol, represents job termination. The <u>initial</u> <u>state</u> of the GENERATOR is

$$\boxed{\$ \mid 3} .$$

2.4.1.4 The algorithm

```
When S is      Do the following

‡              terminate generation

N              C <- C+1
               push(GS,[N,first(N)+1])

"="            output(P)
               C <- C+1

";"            pop(GS,[-,-])
               output(P)

"!"            repeat (for I=1 to loopcount(C,last(P))
                pop(LS,[-,-])
               endrepeat
               C <- last(P)

"-"            E <- findend(C,PCSG(C+1))
               repeat (for I=1 to  loopcount(C,E))
                  pop(LS,[-,-]);
               endrepeat
               C <- E+1

"("            r <- uniform(1,PCSG(C+1))
               i <- select(r,C+1)
               C <- position of i'th alternative

"|"            C <- findend(C,1)

")"            C <- C+1

"<"            r <- uniform(PCSG(C+1),PCSG(C+2))
               push(LS,[r,C+3])
               C <- findend(C,1)

">"            pop(LS,[n,start])
               if n>0 then
                  push(LS,[n-1,start])
                  C <- start
               else C <- C+1
```

2.4.1.5 An Example

Consider the following CSG, with average loop repetition (for procedure A) of 1, and selection execution frequencies 2 and 3.

```
$ = A ǂ
A = B < C > ;
B = ( C | D ) ;
C = ;
D = ;
```

The parameterized CSG is

```
[$ = A ǂ A = B < 1 1 C > ; B = ( 5 2 C | 3 D ) ; C = ; D = ; ].

                   1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3
 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
```

The Production Descriptor Table (PDT) is

prodn	first	last
$	1	4
A	5	13
B	14	24
C	25	27
D	28	30

Snapshots of the GENERATOR as it generates the string "ABDBACA" follow. The rightmost stack element is at the top.

<u>Step</u>	<u>S</u>	<u>Generate Stack</u>	<u>IStack</u>	<u>Output</u>	<u>Note</u>
1	A	($,3)	-		
2	=	($,4) (A,6)	-	A	
3	B	($,4) (A,7)	-		
4	=	($,4) (A,8) (B,15)	-	B	
5	(	($,4) (A,8) (B,16)	-		
6	D	($,4) (A,8) (B,22)	-		r=3,i=2
7	=	($,4) (A,8) (B,23) (D,29)	-	D	
8	;	($,4) (A,8) (B,23) (D,30)	-	B	
9	)	($,4) (A,8) (B,23)	-		
10	;	($,4) (A,8) (B,24)	-	A	
11	<	($,4) (A,8)	[1,11]		r=1,n=1
12	>	($,4) (A,12)	[0,11]		
13	C	($,4) (A,11)	[0,11]		
14	=	($,4) (A,12) (C,26)	[0,11]	C	
15	;	($,4) (A,12) (C,27)	[0,11]	A	
16	;	($,4) (A,12)	-		Loop exhausted
17	;	($,4) (A,13)	-	$	
18	ǂ	($,4)	-		Terminate

2.4.2 Terminating Generation

Just as in the case of parameter estimation, where we had to consider the number of observations required to obtain a certain confidence in the parameters obtained, we must decide how many synthetic strings should be generated. Three alternatives exist:

-generate one string;

-generate a predetermined number of strings;

-generate strings until some statistical test is satisfied.

The GENERATOR can be made to terminate using any of these approaches.

Single string generation is not recommended, since it defeats the purpose of the model, namely, replacing a small number of trace strings by a potentially infinite number of synthetic strings. A single execution trace string is preferred over a single synthetic string.

A predetermined number of synthetic strings is a more appealing way to terminate string generation. As in any multiple string generation, there is concern for the independence of individual strings. This will improve the chance of "true" random sampling from the set of all possible synthetic strings. In the implementation of the GENERATOR used in this research, each string is based on a different random number seed computed from the time-of-day clock.

The determination of the number of strings is not easy, because without any statistical characterization of the generation process, it is difficult to know when the finite set of generated strings has covered (uniformly sampled from) the space of all synthetic strings generable from the model

2.4.3 <u>Generation Environments</u>

The GENERATOR can be used either as an off-line program to produce an output file of synthetic strings, or as an on-line coroutine to produce symbols of the synthetic strings on demand. Since the storage of the entire string is not required, the on-line approach is ideal for efficient program restructuring. The off-line approach can be used when the synthetic strings will be subjected to further analyses.

2.5 ANALYSIS OF COSTS

2.5.1 <u>Assumptions</u>

In this section we assume that the subject program has the following characteristics:

- 100 procedures;
- each procedure contains 2 significant loops;
- each procedure contains 2 significant selection constructs, each containing 3 alternatives;

Further, we assume that

- each procedure name is 8 characters long;
- each ivar is two 8-bit bytes long;
- the icode for <u>incr</u> requires one 4-byte instruction.

2.5.2 Compiler modifications

Considering the inherent complexity of good compilers, the modifications required by the CONSTRUCTOR should have a negligible effect on the size and speed of the compiler. The description of the CONSTRUCTOR's data structures and algorithms indicate that only a small amount of code need be added to the compiler, and that the data structures ACS, CDT and IPI will not be large.

2.5.3 Instrumented Object Code Size

Program instrumentation changes the subject program by introducing ivars and icode. According to our assumptions, each procedure requires 10 ivars (20 bytes) and 10 increment instructions (40 bytes). The over-all increase in space requirements for the entire program is 6000 bytes (60 bytes/procedure * 100 procedures). Instrumentation routine DUMP contains 100 call statements, but since it is invoked only at the end of program execution, it can reuse space occupied by program procedures no longer required in main storage. DUMP does, however, increase the size of the program's virtual name space by an amount proportional to the number of bytes of object code required to effect a procedure call (to WRITECR) with three parameters. We estimate that 20 bytes are required per call; hence 2000 bytes are required for DUMP. Thus the over-all increase in the program's object code will be around 8000 bytes.

2.5.4 Instrumented Object Code Speed

Although the additional space requirements can be quantified, it is more difficult to characterize the execution slowdown experienced by the instrumented program. The slowdown is proportional to the number of execution-time additions introduced by the instrumentation.

The reader may recall from the section on model construction that only significant control structures are instrumented. Thus, the number of instrumentation additions is proportional to the number of procedure calls made. The constant of proportionality is unity or less, as the following analysis shows. Let us assume that the majority of the procedure calls will occur from within loops, and that each (significant) loop contains at least one (unconditional) procedure call. Now consider the two loops, < A > and < A B >, both having repetition frequency n. For the first loop, n calls and n additions are made; for the second, 2n calls and n additions. The respective ratios of calls to additions are 1.0 and 0.5. We will use the higher figure of one addition per procedure call, indeed an extremely low, and practically negligible overhead.

2.5.5 Managing the Databases

We now show that the space overhead for maintaining and updating the required databases is 15,000 to 20,000 bytes. Only the coefficient database need be on-line during parameter estimation; PDB updating and synthetic string generation can be relegated to times of low demand for the system.

The storage space required to hold the production parameter descriptor, coefficient and parameter databases is quite small. The typical PPD record looks like

procname	4	2	2	3	3

and requires 13 bytes of storage. The CDB record looks like

procname	10	n1	a1	n2	a2	c11	c12	c13	c21	c22	c23

,

which occupies 30 bytes. (Record field definitions use the same symbols used to describe the PDB.)

The size of the PDB depends upon the amount of detail maintained. Assume that the PDB record has the format given in Figure 2.11. Then each record requires 38 bytes.

The CSG database can contain either the non-compressed (symbolic) or compressed CSG productions. Assume that a production contains an average of seven procedure calls and ten CSG operators. Then each non-compressed production requires 7*8 + 10 = 66 bytes.

The total space requirements for all these databases is 6600 (CSG) + 3000 (CDB) + 3800 (PDB) + 1300 (PDDB), or 14700 bytes. A more sophisticated PDB can increase this space requirement, as well as the time required to perform parameter estimation.

2.5.6 String Generation Costs

The speed of the GENERATOR depends upon the CSG, the language in which the GENERATOR is written, and the efficiency of the GENERATOR code. The most important GENERATOR performance index is the rate of synthetic string production, which should be several orders of magnitude faster than that of execution tracing.

The GENERATOR used in the research was implemented as a PL/C program running on an IBM 3081 (or 370/158) under the MVS operating system, and was in no wise optimized for speed of generation. Generation speed ranged from 750 to 3000 references per CPU second, depending upon the complexity of the CSG productions. (Selection constructs slow down the generation process more than do loops.) The performance of the GENERATOR can easily be improved by an order of magnitude and, indeed, should be were the GENERATOR to be implemented as a production program. The inefficiency of the PL/C GENERATOR notwithstanding, it seems clear that synthetic strings can be generated at an acceptable rate and cost to justify use of the model.

As a final word, we should note that the ease and convenience of using the model, from the standpoint of the program developer, is more than adequate compensation for any generation inefficiencies.

2.6 VARIANTS OF THE BASIC MODEL

In this section we relax some of the assumptions and restrictions of the PAM model. We begin by showing how to model data-sensitive programs. Next, we simplify the CSG syntax to a normal form. Before proceeding, we need to give a name to the version of PAM we have been discussing up to now. We call it the generative PAM, or GPAM, since it involves synthetic string generation.

PAM is actually a family of models. In this section we start with the GPAM and move in two directions -- generalization, to remove the requirement of the call-path independence assumption -- and simplification, to reduce the complexity and size of the CSG. PAM attempts to capture two aspects of program structure: static control-structure nesting and dynamic procedure-call nesting. Model variants differ in the extent to which they capture these aspects of program structure. Each model version can be thought of as being either generative (when used to produce synthetic strings) or descriptive (when used to describe or characterize observed program executions).

2.6.1 A Normal-Form CSG

GPAM attempts to reproduce procedure level referencing behavior in terms of sequencing and distribution of calls. Sequencing is controlled by the grammar operators -- loops, alternation and branching -- and by the synthetic string generator. By distribution of calls, we mean the relative frequencies of calls to potential targets. The distribution of calls is governed at the level of the individual procedure by the procedure's parameters. If situations exist in which the distribution of calls is more important than their sequence, a simpler form of PAM CSG productions is possible. We say that a production of the form

X = < (x1 | x2 | ... | xm) >;   , where xi is either a call or a return (!), is in *distribution-normal form* (DNF). This form of production guarantees (in a statistical sense) to preserve the distribution of procedure calls to the possible call targets, provided that the GENERATOR uses a random number generator that produces true uniform random deviates. The DNF CSG for program P1 is shown in Figure 2.12.

```
A = < ( B | C | D | E ) > ;
B = < ( ! | C | D ) > ;
C = < ( ! | D | E ) > ;
D = < E > ;
E =  ;
```

Figure 2.12
DNF Call-Sequence Grammar for Program P1

We briefly compare DNF-PAM and GPAM. The DNF grammar is compact because of the simpler productions. The required CONSTRUCTOR algorithm and data structures are simpler since no analysis of the structure of the subject program is required. Instrumentation is simpler: ivars simply count the number of times one procedure calls another, completely disregarding the control-structure context of the calls. The number of ivars is equal to the number of different procedures called (plus one if the procedure contains multiple exit points); the amount of icode is proportional to the number of call statements in the procedure.

DNF-PAM characterizes a program's execution at the level of inter-procedural references, as opposed to the intra-procedural characterization GPAM produces. DNF-PAM is the simplest version of PAM, and it used in this research primarily as a generative model.

From a restructuring standpoint, DNF parameters are equivalent to the Nearness matrix. DNF paramters measure pair-wise call frequencies, from which can be derived the pairwise adjacency frequencies that constitute the entries in the Nearness matrix. That is, for very low instrumentation overhead, enough information is obtained to achieve very acceptable performance gains from program restructuring.

2.6.2 Relaxing the Call-Path Independence Assumption

Consider the following CSG.

```
P = A B C;
A = D;
B = D;
C = D;
D = ( X | Y | Z );
```

Suppose that D always calls X, Y or Z, accordingly as D is called by A, B or C, respectively. Then, since D's behavior depends upon its caller, D violates the independence of call path assumption. The variation of D's behavior according to its caller requires an examination of some data, either one of its parameters, or some global variable. Such a procedure is said to be data-sensitive.

Procedure D can still be modeled by creating aliases DA, DB and DC, whose CSG productions are

```
DA = X;   DB = Y;   and DC = Z;  .
```

Productions for A, B and C become

```
A = DA;   B = DB;   and C = DC;  .
```

We now present a general aliasing technique that introduces no unnecessary aliases.

1. Determine non-aliasable productions, those that do not contain any significant control structures. (Non-aliasable productions do not require any parameters.)

2. Assign serial number unity to each aliasable production.

3. Copy each production. For each nonterminal on the right-hand side whose production is aliasable, append to the nonterminal the serial number of its corresponding production, and increment the serial number by one.

4. Write new productions for each aliased production according to step 3. When there are no more unexpanded productions, stop.

The importance of this transformation is that it allows modeling of programs that contain data-sensitive modules. Aliasing has two side effects: an increase in the number of productions; and an increase in the amount of work done by the CONSTRUCTOR. Since each alias requires its own set of execution coefficients, the compiler must allocate and properly reference a stack of instrumentation variables for each aliased procedure. During execution of the instrumented program, calls to aliased procedures must include a parameter value for properly indexing the ivar stack. The value of this index must be determined during instrumentation. It should be clear that aliasing increases instrumentation complexity. Fortunately, however, it is probably the case that the aliasing transform would be applied rarely.

Aliasing provides a way of injecting global call-path context information into the model. (Non-aliased productions contain no information about the static chain of procedure calls by which entry is made into the corresponding procedure. The only information provided is the return point for any calls made by that procedure.) The serial number assigned to an aliased procedure encodes the call path to the procedure from the start of the program. Applying the aliasing transform to GPAM produces aliased GPAM, or AGPAM. AGPAM contains the maximum amount of call context and control-structure context information possible in a context-free call-sequence grammar used by the PAM family. When aliasing is applied to every production in a DNF CSG, the resulting model version is called the descriptive PAM, or DPAM.

2.6.3 The Descriptive PAM

Procedure-level program execution is captured in the sequence of active procedure call chains. The descriptive PAM (DPAM) defines program state to be a procedure call chain, and a state-transition to be a procedure call or return.

A static representation of an arbitrary program P is the DPAM state-diagram, denoted DPAM(P). Figure 2.13 shows DPAM(P1), the state-diagram for sample program P1 of Figure 2.1. The state-diagram is an undirected tree, which we term a call tree. Each node of the tree corresponds to a DPAM state (i.e., a call chain). Nodes are identified by unique integers assigned according to a preorder traversal of the tree. For example, in DPAM(P1), state 7 represents call chain [A,B,D]. State 7 has label "D", the most-recently activated procedure in call chain [A,B,D].

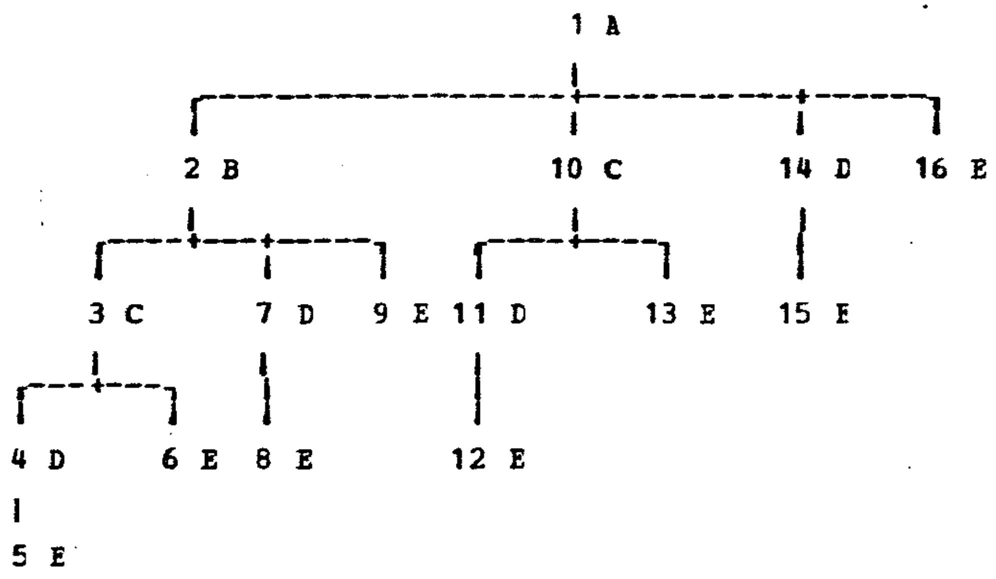

Figure 2.13
DPAM State-Diagram for Program P1

Given procedure reference string w=w[1]w[2]...w[K], there is a corresponding DPAM state-sequence string z=z[1]z[2]...z[K], where symbol w[i] causes a DPAM state transition into state z[i]. (The DPAM initial state is z[0]=0.) The program execution, w, can be described (characterized) in terms of DPAM(P). We now list several characterizations.

(1) cfreq(s) = the number of times state s is entered via a call transition.

(2) totfreq(s) = the total number of times state s is entered.

(3) totdur(s) = the total number of state transitions (references) made to states belonging to the subtree of DPAM(P) having state s as the root.

(4) avgdur(s) = the average number of state transitions made between subsequent entries and exits from the subtree of DPAM(P) having state s as the root. (avgdur(s)=totdur(s)/cfreq(s))

We call these characterizations execution profiles, which are analogous to GPAM execution coefficients. Figure 2.14 contains values for these profiles derived from reference string "AECECEDEDEDEDBCECBCECBEBAEAEA" of length 29.

state	proc	cfreq	totfreq	totdur	avgdur
1	A	1	4	29	29
2	B	1	6	24	24
3	C	3	6	9	3
4	D	0	0	0	0
5	E	3	3	3	1
6	E	3	3	3	1
7	D	1	1	7	7
8	E	3	3	3	1
9	E	1	1	1	1
10	C	0	0	0	0
11	D	0	0	0	0
12	E	0	0	0	0
13	E	0	0	0	0
14	D	0	0	0	0
15	E	0	0	0	0
16	E	2	2	2	1

Figure 2.14
Sample DPAM Execution Profiles

We now propose ways in which the profiles just introduced can be used in program restructuring, to suggest one or more areas for further investigation.

(1) DPAM(P) with node weights cfreq provides enough restructuring information to duplicate the Nearness method.

(2) Profiles totdur and avgdur measure locality since they capture the reference density to a subset of procedures during an interval of program execution.

It should be noted that DPAM and many of its profiles are directly derivable from GPAM CSG and parameters. Development of restructuring algorithms based on DPAM is suggested as a topic for further research.

2.6.4 A Comparison of the PAM Variants

Table 2.1 summarizes the different versions of the PAM model. A level-i model version (i=1,2,3) is derivable from level-j versions (j>i). DNF-PAM is the simplest, modeling only the local call-context information implicit in the "calls" relation between procedures. AGPAM, which models global call-context and local control-structure information, is the most general.

Level	Version	Context Information Represented
4	AGPAM	global call and local control structure.
3	GPAM	local call and control structure.
2	DPAM	global call.
1	DNF-PAM	local call.

Table 2.1
The Procedure-Activation Model Family

The nature of the PAM family is perhaps most clearly evident from DPAM, where the notion of model states and state-transition probabilities are explicit. This basic structure is common to all model versions, as is the capacity to generate (or recognize) a sequence of properly formed procedure references.

PAM differs from many stochastic program models primarily in the definition of program state. The PAM variants differ from each other in the way the state-transition probabilities are derived. For DNF-PAM and DPAM, they are based on direct measurements of inter-procedural referencing. For GPAM and AGPAM, the determination of these probabilities has both a static and a dynamic component. The static component consists of the model parameters -- statistics for loops, and probabilities for selection constructs. During synthetic string generation, the choice of the next reference to generate is based upon a dynamically-determined probability, whose value is some complex function of program structure and model parameters.

2.7 SUMMARY

In this chapter, we have defined the elements of the Procedure-Activation modeling system. We have shown that modeling can be automated, and that the model lends itself to various choices of CSG grammar form, parameter estimation techniques, and generation environments. The cost of model construction and parameter estimation were shown to be low.

We have shown that PAM is a family of models, having in common a constrained (by internal control-structures or call chain or both) probabilistic view of the program execution phenomenon. In this research, the primary investigations treat only GPAM and DNF-PAM. Although DPAM will not be investigated thoroughly in this research, it shows promise of yielding insights into program locality, and of forming the basis of new restructuring algorithms.

Chapter 3

RESTRUCTURABILITY OF THE SUBJECT PROGRAMS

This research is divided into four major studies.

(1) Restructurability. For each subject program, we determine the extent to which program restructuring can reduce execution-time memory cost.

(2) Elementary modeling. These approaches to modeling are attempted: the simples model version, DNF-PAM; and the more general model version, GPAM, for which point estimates for model parameters are used.

(3) Parameter estimation. The execution-sampling phase of modeling is studied statistically: the effect of random sample size upon model accuracy, and the underlying distribution of model (loop) parameters.

(4) Advanced modeling. More modeling effort is expended to achieve better accuracy: point estimates of loop parameters are replaced by interval estimates; model version DNF-PAM is replaced by GPAM.

This chapter presents the design and findings of the restructurability study. Elementary modeling is the subject of Chapter 4; the parameter estimation studies and advanced modeling results are presented in Chapter 5. We begin this chapter with a review of the relevant issues of a posteriori program restructuring using actual program traces, and define the method of comparing layouts. Next we describe the subject programs, noting in particular their static program structure -- nesting of procedure calls and control-structure constructs. Finally, we describe and present results from the restructurability study: the feasibility of applying restructuring to subject programs is determined; and the parameters of the restructuring process are assigned values for use in subsequent experiments.

3.1 A POSTERIORI PROGRAM RESTRUCTURING ISSUES

In this research we do not study restructuring for its own sake, but as the tool by which PAM will be validated. To avoid making bad choices in applying the restructuring procedure, we decided to deal systematically with the variables of restructuring to determine values of these variables that least obscure the benefits derivable from using PAM. We now examine the parameters of the restructuring process, with the primary concern for avoiding values that consistently yield poor restructuring gains.

3.1.1 Overview of the Restructuring Process

Figure 3.1 illustrates the steps in the program restructuring process, and the input variables for each step. The program characterization phase involves execution sampling and the representation of the collected trace data, usually in the form of a set W of reference strings. The set W, restructuring algorithm A and control variable Θ (the pair is denoted $A(\Theta)$), are inputs to the restructuring phase, during which the restructuring matrix C is constructed. Matrix C contains module-module affinity weights that suggest clusters. The clustering phase takes the affinity weights and module sizes, and builds clusters, subject to the page size (p) constraint. The extent to which the clustering suggestions contained in C can be carried out depends upon p and the module sizes. The resulting layout is denoted $L(W,A(\Theta),p)$, which identifies the parameters of the restructuring process.

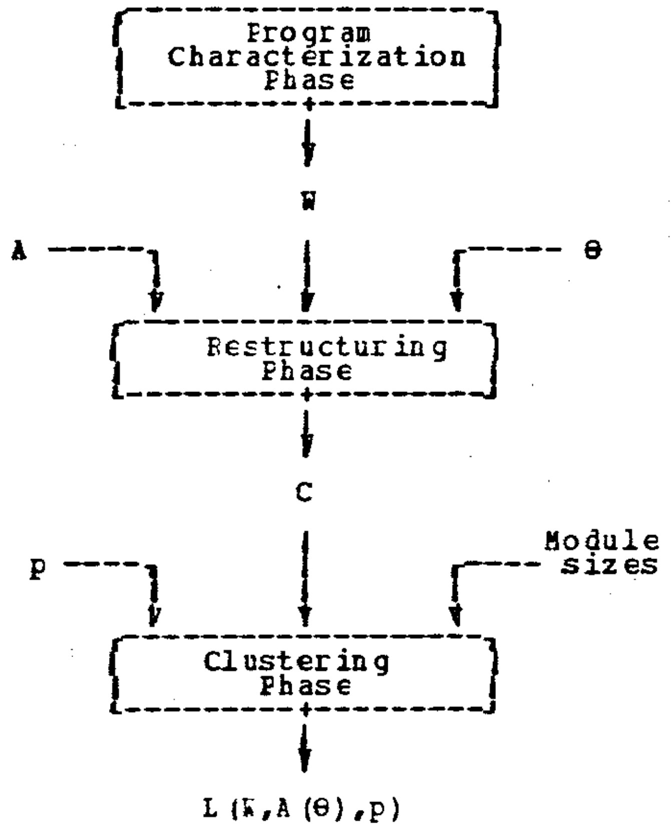

Figure 3.1
Schematic of the Restructuring Process

The following sections discuss, for each phase in the restructuring process, options for implementing that phase, the effect of choices at one phase on subsequent phases, and the rationale for some of the choices we made.

3.1.2 The Program Characterization Phase

3.1.2.1 Choosing the executions to sample

One of the first steps in the standard a posteriori program restructuring process is to decide which executions of the subject program to use as inputs to the restructuring phase. In order to achieve the greatest benefit from restructuring for the majority of the runs of the program, these executions should be representative of the "average" or typical behavior of the program. Fortunately, empirical studies [HATF71,FERR76a] have demonstrated that programs tend to be quite insensitive to their input data. This result supports the common practice of carefully choosing a small number of executions (sometimes just one), taking into account the function and structure of the program, and typical input values.

3.1.2.2 Reference string representation

The reference string captures information about the sequence and, possibly, times of program references. When restructuring makes use of the LRU replacement algorithm, which is driven by sequence alone, reference times are not required. Time information must, however, be provided for restructuring based on the WS replacement algorithm. Direct measurement of reference times can require significant overhead, since calls to the operating system clock routines are required. Furthermore, on multiprogrammed systems not providing each concurrent process with its own virtual clock, controlling measurement errors is difficult because of the sharing of the system clock. For these reasons, most studies resort to estimating reference times.

We now examine a widely-used estimation technique, which we term the counting method. Given procedure reference string w=w1,w2,...,wK, let the times of each of the K references be given by the time string T=t1,t2,...,tK. When the counting method is used for machine-level tracing, the time of the k+1'st reference is given by t[k+1]=t[k]+Ni*A, where Ni is the number of machine instructions separating w[k] and w[k+1], and A is the average instruction execution time. This method is based upon the assumption (which is quite reasonable at the level of machine-instruction execution) that the time interval between the executions of successive instructions is a constant.

We have used a similar method, based upon the following assumptions.

(1) the cost of a call linkage is one time unit, chargeable to the caller.
(2) the cost of a return linkage is one time unit, chargeable to the returner.
(3) the cost of executing the non-call portion of the body of a procedure is _zero_.

These assumptions lead to the simple time string T=0,1,2,...,K-1. Notice that this is actually a counting method based on the assumption of a _constant rate of linkage events_. Under these assumptions, the execution time chargeable to a procedure is one plus the number of calls it makes. An example of the time string computed for a short reference string is shown in Figure 3.2, where "$" represents the operating system job initiator routine. A call linkage has type "C", a return type "R".

Reference:	A	B	C	B	A	D	A	D	A
Type:	C	C	C	R	R	C	R	C	R
Charged to:	$	A	B	C	B	A	D	A	D
Time:	0	1	2	3	4	5	6	7	8

Figure 3.2
Estimation of Reference Times

Batson and Brundage [BATS77a] studied the distribution of the mean inter-procedural reference times for Algol programs. They found the mean to be 2-10 times the median, with coefficient of variation between 2 and 10. Their findings suggest that (1) the median interval between most linkages is much shorter than the mean, and (2) most of the variability in times (high coefficent of variation) is caused by the small percentage of very long intervals between linkages, contrasted against a very large percentage of short intervals. We are satisfied that the assumption of (nearly) constant time between linkages holds for the major portion of the program's execution-time references.

Under the time-dependent replacement algorithm WS, the working set will be overestimated, for a given window size, during one of the few long intervals between linkages, resulting in an underestimation of the number of faults for that window. But this situation will rarely occur, since long intervals are few. Underestimation of the working set for very short intervals can occur in like manner, but since the variability is low (when long intervals are excluded) the amount of estimation error is small.

Despite the possibility of errors in estimating the working set, we did not feel that such errors will invalidate WS restructuring results, for the following reasons. First, and perhaps most important, is the fact that this study is really a simulation study in which we felt that sequence was the most important program property to model. Second, if indeed a synthetic string reproduces the reference sequence of a subject trace string, when it is used in restructuring, it will produce the same systematic error as was produced by restructuring using the subject trace string. That is, the two strings being compared both have roughly the same amount of measurement error, and the differences in behavior cannot be attributed to that systematic measurement error.

3.1.3 The Restructuring Phase

The restructuring phase takes as input a set W={w1,w2,...,wM} of reference strings. One restructuring matrix, Ck, is constructed per string wk in W, and the collective restructuring information embodied in set W is captured in the matrix C=C1+C2+...+CM. Recall that C[i,j] is the number of faults that would be eliminated should modules i and j occupy the same page. Stated another way, C[i,j] measures the competition between i and j for membership in the resident set under replacement algorithm A(θ). Competition, hence memory cost, is reduced when modules i and j are stored in the same page.

Matrix C suggests a clustering of modules based on mutual competition for residency under A(θ). For C[i,j]>0, modules i and j belong to the same natural cluster. Any module k having affinity for a module belonging to a natural cluster is itself a member of that same natural cluster. Natural clusters are of unconstrained size, and are usually too large to be stored entirely in one page. We term the l-m pair having the largest weight C[l,m] the critical pair. The natural cluster containing the critical pair is termed the critical cluster.

Output from the restructuring phase is a list of module pairs and affinity weights, in decreasing order of affinity weights. The list suggests module pairings directly, and natural clusters indirectly. Consider the list of weighted affinity pairs shown in Figure 3.3. Critical pair (1,3), having affinity weight 29, suggests the natural cluster <1,3,8>, which is critical because it contains the critical pair (1,3). The function of the restructuring phase, then, is to make clustering suggestions to the clustering phase; it is up to the clustering phase to carry out the suggestions.

Weight	i	j	Natural Cluster
29	1	3	A
14	2	4	B
14	5	6	C
12	1	8	A
5	6	7	C

Figure 3.3
Example of Output from the Restructuring Phase

The phase-transition view of program behavior suggests an interesting interpretation of the i-j affinities. Three types of competition occur.

a) Intra-phase -- both i and j belong to the same phase of execution.

b) Inter-phase -- i and j belong to different phases. Competition is observed during transitions between phases.

c) Intra-transition -- both i and j belong to a transition between the same two phases, but not to either phase.

The restructuring phase has as its primary goal the definition of natural clusters that correspond to intra-phase competition. That is, intra-phase affinities should be greater than the other types of affinities.

3.1.3.1 Choice of algorithm

From the outset, we intended not to introduce any new restructuring methods, but to concentrate on finding sources of reference strings other than execution traces. Critical-set program tailoring methods of Ferrari were chosen because they have been shown to perform well [FERR76a], and are simple and relatively inexpensive to use. LRU and WS were chosen to be the underlying replacement algorithms because they are the most widely used algorithms for fixed and variable allocation policies, respectively.

The obvious question is whether the choice of algorithm affects restructuring effectiveness? The many papers written on the subject give evidence that the choice of algorithm is important. In the restructurability study, however, we are concerned only with showing that both the CWS and CLRU restructuring algorithms are affected in the same way by changes in the other parameters of the restructuring process.

3.1.3.2 Choice of control parameter

What should the <u>restructuring window</u>, θ (the WS window size or the LRU allocation), be to achieve good results from restructuring? Can some values of θ lead to poor clustering suggestions? As far as we have been able to determine from the literature, there is no widely accepted set of control parameter values used for restructuring. We now present a few guidelines.

Care must be taken in choosing θ. Bad choices do exist. Consider the reference pattern "ABABABAB...ABA", where "AB" is repeated k times. CLRU restructuring with $\theta=1$ determines A-B affinity to be 2k. When $\theta=2$, the affinity is 1. If program execution is dominated by this cycle, $\theta=1$ is the optimal setting. So we see that referencing patterns, typically cyclic patterns produced by looping, have a bearing on the proper choice of θ. Having θ too large "swallows up" the dominant phase.

What is too large a value of θ? The window θ is a function of the number of modules referenced during a phase, and the mean time t between successive references to modules. When $t > \epsilon$, there will be turnover in (hence, competition for) resident set membership throughout the phase. The value t is related to the cyclic reference patterns generated by the program.

A method for selecting the restructuring window θ is to compute the memory cost curve (s versus r), assuming each module occupies exactly one page. The r value for which the cost drops dramatically identifies the average number of modules resident during the dominant phase. Since the dominant phase should not be "swallowed up" by too large a restructuring window, θ should not be set much larger than this r value. Such a θ will be sufficiently small to ensure that competition for the resident set is indeed observed during restructuring. Furthermore, improvements obtained for small values of θ are preserved in operating environments whose control parameters exceed θ [FERR76a].

3.1.4 The Clustering Phase

The clustering suggestions received from the restructuring phase are used to construct page-sized clusters. The extent to which the suggestions can be followed depends upon the page size constraint. As the page size becomes smaller, relative to the average module size, more and more information contained in clustering suggestions is discarded. Large natural clusters must be broken into smaller clusters in order to satisfy the page size constraint.

3.1.4.1 Estimating module sizes

Knowledge of module sizes is required to perform restructuring. Since, in this study, the restructuring experiments are basically simulations, the number of source language statements in a procedure was used to approximate the size of each procedure module.

3.1.4.2 Page size

Program restructuring works best when, on the average, at least three modules are packed to a page [HATF71]. Since this research is a simulation study -- the execution of the restructured program on a paged virtual memory computer is simulated -- we are free to choose page size to be at least three times the average module size m. Since the modules correspond to procedures from structured programs, it is reasonable to assume further that no module is larger than the page size. The minimum page size p is therefore chosen to be max(3*m,M), where M is the size of the largest module. Given the restructuring matrix C, clustering is performed for page sizes p, 1.5p and 2p.

The natural clusters suggested by the restructuring phase should be stored in a single page in order to realize the maximum benefit from restructuring. Hence, the optimal page size is a function of the program's static (module sizes) and dynamic (looping) properties. The existence of critical clusters explains why large page sizes are preferred. From a paging overhead perspective, optimum performance occurs when the minimum number of faults is generated, that is, when the page size is the size of the program's name space. Such a large page size is, of course, impractical. Moreover, a large natural cluster often requires a page size that is impractically large. Smaller natural clusters are formed when small θ values are used during the restructuring phase. These small clusters are less likely to have to be broken up to fit small pages and, when the page size is large, several may be stored in the same page.

3.1.4.3 Clustering algorithm

The clustering algorithm used in this research is essentially the one described by Ferrari [FERR73]. The first phase of clustering is based on pair-wise affinity weights contained in the restructuring matrix. The second phase is a clean-up pass during which unclustered modules are assigned to partially filled pages according to a first-fit strategy, subject, of course, to the page size constraint. The second phase totally ignores the connectivity of modules, and is concerned mainly with reducing the number of pages spanned by the layout.

Although this particular clustering procedure is quite simple, it produces good layouts cheaply. More sophisticated clustering methods produce marginally better layouts, but at a much higher cost [FERR76a].

3.2 LAYOUT PERFORMANCE EVALUATION

3.2.1 *Overview*

The general layout evaluation procedure is depicted in Figure 3.4. The simulation of the replacement algorithm $A(\theta)$ on the execution y, assuming layout L, produces some cost measure, which we now define.

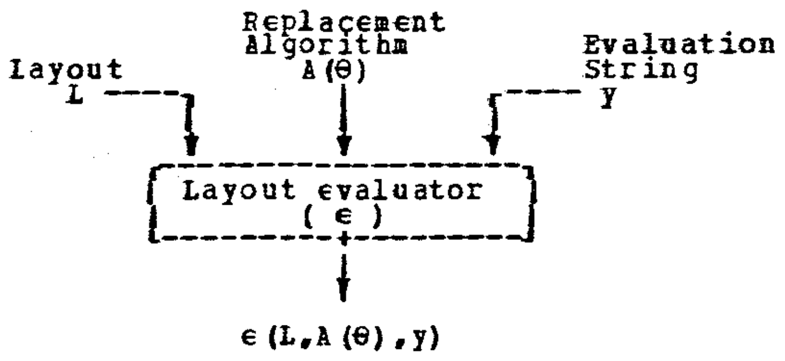

Figure 3.4
Layout Evaluation Schematic

Definition: $\epsilon(L,A(\theta),y)$ is the *cost* of executing execution y under replacement algorithm $A(\theta)$, using layout L. Layout L is said to be *evaluated under A(θ) against evaluation string y*.

When no ambiguity as to the layout L and algorithm A can occur, $\epsilon(L,A(\theta),y)$ is written $\epsilon(\theta)$.

A layout can also be evaluated against an *ordered set* of strings, $Y=[y1,y2,\ldots,yM]$, termed an *evaluation string set*. In this case, the replacement algorithm input string is $\text{CONCAT}(Y)=y1+y2+\ldots+yM$, the concatenation of the members of Y. As a further generalization, layout evaluation can be extended over some arbitrary *evaluation interval*, $Q=[\theta i,\theta j]$. We term such an evaluation of layout L a [Q,Y]-*evaluation*.

<u>Definition</u>: E(L,A,Q,y) is the <u>extension</u> of the cost function e <u>over interval</u> Q. E is given by

$$E(L,A,Q,y) = \frac{\theta i * e(\theta i) + \ldots + \theta j * e(\theta j)}{\theta i + \ldots + \theta j}.$$

This weighting scheme penalizes a large cost at high memory allocations. In order to ensure a fair comparison of two or more layouts, all evaluations must use identical θ sample points within the interval Q.

Consider the following example, where the cost measure is the number of faults, f, and the interval Q=[2,4].

	Cost			Extended Cost	
Layout	f(2)	f(3)	f(4)	E[2,3]	E[2,4]
L1	100	80	20	88	58
L2	80	70	40	74	59

Layout L1 experiences more faults than L2 at θ=2 and θ=3, but its lower cost at θ=4 (20 versus 40) reduces its cost over [2,4] to below that of L2. Over [2,3], the cost of L1 is higher. So we see that the extended cost function performs a smoothing, so that poor performance at low allocations can be compensated for by good performance at higher allocations.

Perhaps the major issue in layout evaluation is the choice of the evaluation interval Q. Using a narrow interval can give either overly optimistic or overly pessimistic results. Using a wide interval gives conservative results since, given sufficient allocation of memory, layouts tend to perform about the same.

3.2.2 Performance Measures

Cost functions, which apply to executions, are also used to measure the performance of a layout. In the cost function $e(L,A(\theta),y)$, holding y fixed while varying L from $L1$ to $L2$ measures the relative goodness of layouts L1 and L2.

3.2.2.1 Scalars

We use three cost functions to measure memory cost and layout performance.

(1) $r(L,A(\theta),y)$ is the mean resident set size at a fixed value of θ.

(2) $f(L,A(\theta),y)$ is the number of page faults generated for a fixed θ.

(3) $s(L,A(\theta),y) = r(L,A(\theta),y)*f(L,A(\theta),y)$ is the space-fault product, which is the space-time cost of handling page faults, for a fixed θ.

When no ambiguity can result, we refer to these functions as the scalar variables r, f and s, respectively.

3.2.2.2 Curves

Performance data are either presented graphically, in the form of a curve of some type, or they are presented in summary form as a scalar. Curves give a visual summary of the effects of varying the values of variables under consideration, and often suggest places where further quantitative analyses are needed. Scalars provide a quantitative measure for comparing performance.

We use three performance curves.

(1) The working set curve plots r versus θ, when the replacement algorithm is WS. This curve shows the packing efficiency of layouts.

(2) The swapping curve plots f as a function of r. It shows the effect of memory allocation upon the number of faults.

(3) The space-fault, or memory cost curve plots the product r*f versus r. It shows the effect of memory allocation upon memory cost.

3.2.3 Comparing Layouts

We can compare the performance of layouts L1 and L2 by comparing e(L1,A(Θ),y) and e(L2,A(Θ),y), for arbitrary cost function e. If the values of e versus the appropriate independent variable are plotted for both layouts on the same graph, the relative performances of L1 and L2 can be seen. Such curves summarize relative performance over a wide range of values of the independent variable.

At times one is concerned with a quantification of relative performance. We use a single measure of comparison for two layouts L1 and L2.

Definition: R(Ls,L,E,A,Q,y) is the cost-reduction of L over Ls under [Q,y]-evaluation, with respect to cost function e. R is given by

$$R(Ls,L,E,A,Q,y) = 1 - \frac{E(L,A,Q,y)}{E(Ls,A,Q,y)} \text{, where}$$

layout Ls is the comparison standard.

When no ambiguity can result, we write R(Ls,L). This measure of comparison has two uses: (1) comparing the closeness of two layouts, and (2) determining the superior, or inferior, layout. We find it more convenient to express R as a percentage. Negative values indicate performance degradation; positive values indicate improvement. From this point on, we will use the terms "improvement", "degradation" and "closeness" in comparing layouts.

Suppose we have two layouts, $L1=L(w1,A(\theta),p)$ and $L2=L(w2,A(\theta),p)$, formed using the same restructuring algorithm and clustering page size. We say that "w1 produces layout L1" and "w2 produces L2" since the input strings, w1 and w2, distinguish the two applications of the restructuring process that produced the layouts L1 and L2. Comparing layouts L1 and L2 is tantamount to comparing the restructuring effectiveness of the strings w1 and w2; the one that leads to the better layout can be thought of as being better in terms of restructurability. We will often indirectly compare strings by comparing directly the layouts they produce. That is, saying "strings w1 and w2 are within 10% of each other" means that the performances of layouts L1 and L2 are within 10% of each other.

Type-2 programs provide benchmarks for GPAM modeling: GPAM must capture and reproduce the restructuring signature of observed executions of the subject program. We expect that the signature is so distinctive that type-2 programs can also be modeled using DNF-PAM to produce good quality layouts.

Modeling type-3 programs using GPAM should prove to be challenging. We expect this type of program to require careful parameter representation and estimation.

Even if the program can not be successfully modeled, GPAM provides a useful vehicle for describing and characterizing the subject program. Being able to predict the extent to which restructuring is viable is a useful capability that GPAM might provide.

3.3.2 Static Program Descriptions

Three of the four subject programs that were studied are described in this chapter. (The fourth one is studied separately in Chapter 6.) Subject programs were selected on the bases of availability, size (in number of statements or procedures), program structure (structured programming style was preferred), and programming language (PL/I). These criteria were necessary to facilitate modeling. Three of the selected programs were written by the author and used as a part of the modeling and restructuring system used in this research. No conscious attempt was made to code these programs in a manner that would in any way bias the results of this research. As a further guard against inadvertent bias, we chose as the fourth program one written for a use that is totally unrelated to the requirements of this research.

Table 3.1 summarizes the collection of subject programs. Figures 3.5- 3.7 give the procedure-level static structure of the subject programs GENREF, RESTRUCT and CLUSTER. A relatively long period during which a program performs a distinct logical function is called a <u>logical</u> <u>phase</u> of execution, which corresponds coarsely to the phase-transition behavior discussed in Chapter 1. For each subject program, logical phase boundaries were determined from a knowledge of the over-all function of the program, and from an examination of its source code.

The static procedure nesting level and sharing index (ratio of the number of call graph edges to the number of nodes) reflect the inter-procedural complexity. Over-all construct nesting level, and specific construct nesting levels -- selection and loop constructs -- indicate the intra-procedural control structure complexity. A procedure that calls other procedures is termed a caller procedure.

All the subject programs have about the same static characteristics. In all, the percentage of caller procedures is 30-40%. None have very complex intra-procedural structure, as evidenced by the low levels of construct nesting. All appear to have the call-graph structure one would expect when structured programming is used, in that procedures are used instead of deeply nested constructs.

Program GENREF performs synthetic string generation, given a PAM model instance for a subject program. There are two large (involving more than eight procedures) logical phases: (1) the model internalization phase, during which the parameterized call-sequence grammar is loaded and stored internally; and (2) the string generation phase. These two phases are essentially disjoint, the driver being the only procedure common to both phases. The first phase consists of two smaller phases corresponding to CSG loading and CSG parameter loading, respectively.

	Subject Program		
	GENREF	RESTRUCT	CLUSTER
#Statements	769	640	590
#Procedures	32	21	26
#Caller procedures	14	7	8
#Logical phases	2	3	3
Share-index	1.4	1.3	1.1
Avg procedure nesting level	2.6	1.7	2.5
Max procedure nesting level	4	3	4
#Loops	16	7	7
Avg loop nesting level	0.3	0.6	0.3
Max loop nesting level	2	1	1
#Selection constructs	18	15	10
Avg selection nesting level	0.3	0.5	0.4
Max selection nesting level	2	2	2
#Constructs	34	22	17
Avg construct nesting level	1.0	1.4	0.9
Max construct nesting level	4	4	4

Table 3.1
Static Characteristics of Subject Programs

```
                 ┌--DUMPCSG ====DOUTPUT
                 |
                 |                                             ┌--FINDLEV
                 |                           ┌--CNTLESC      =+--OUTPUT
                 |                           |                 └--PCFLOOP
                 |                           |--FINDLEV
                 |                           |--LOOPEND
                 |--GENSTR   ===[STRGEN] =+  |                 ┌--FINDLEV
                 |                           |--LPENTRY      =+
                 |                           |                 └--OUTPUT
                 |                           |--OUTPUT
                 |                           |                 ┌--FINDALT
                 |                           └--SEL_ALT      =+
                 |--INITCSG                                    └--FINDLEV
                 |
                 |            ┌--GETSYMB
                 |            |              ┌--GETSYMB
                 |            |--LOADNT    =+--INSFIND
                 |            |              └--NEWCARD
GENREF         =+-[LOADCSG]=+
                 |            |              ┌--DUPLICATE
                 |            |              |--FIXJUMP
                 |            |              |--GETSYMB
                 |            └--LOADPDN   =+--INSFIND
                 |                           |--NEWCARD
                 |                           └--SETDUP    ====GETSYMB
                 |
                 |            ┌--GETSYMB
                 |            |              ┌--FIND_IP
                 |-[LOADPRM]=+--LDCCND    =+--GETSYMB
                 |            |              └--INSPARM
                 |            └--LDLOOP    ====GETSYMB
                 |--NUMSTR
                 |--OUTPUTL
                 └--PGMSIZE
```

Figure 3.5
Call Tree for Subject Program GENREF

One distinctive feature of GENREF is its non-deterministic behavior -- two executions of GENREF using the same inputs will produce two different outputs (and hence, different execution trace strings). This is due to random sampling from GPAM parameter distributions during the generation phase. On each execution of GENREF, the initial random number seeds are changed, which makes this non-determinism possible. The procedure-level static structure of GENREF is shown in Figure 3.5.

```
                        ┌--CLEARF
                        |--SETCFLD
          ┌--BLDLBL   =+
          |             |--SETFFLD
          |             └--SETIFLD
          |             ┌--FETFLD
          |             |--SETCFLD
          |--BLDCLBL  =+
          |             |--SETFFLD
          |             └--SETIFLD
          |--FETDATA  ====READATA
          |             ┌--POSFILE
RESTRUCT=+--FETINIT   =+
          |             └--RFD
          |--RMDUMP   ====SETIFLD
          |  ┌-----┐      ┌--FETDATA ====READATA
          |-[SCIRU]    =+
          |  └-----┘      └--UPDATE
          |  ┌----┐       ┌--FETDATA ====READATA
          |-[SCWS]     =+
          |  └----┘       └--WSUPDATE
          |--SETFILE
          └--SETMAP
```

Figure 3.6
Call Tree for Subject Program RESTRUCT

Program RESTRUCT constructs a restructuring matrix C, given as inputs a reference string and replacement algorithm specification (i.e., the algorithm and a value of its control parameter). RESTRUCT's longest phase (containing three procedures) occurs during construction of the restructuring matrix. The identity of the modules referenced within the phase depends upon the replacement algorithm. Its other two phases, both short-lived, occur during program initiation and termination. The only aspect of RESTRUCT's behavior subject to change appreciably across executions is the duration of the longest phase, which is proportional to the length of the input string. Figure 3.6 shows the call tree for RESTRUCT. Notice that the height of the tree is less than GENREF's, which indicates that GENREF has a more complex (static) call structure.

The restructuring matrix C produced by RESTRUCT is an input to program CLUSTER, which is essentially a list-processing algorithm whose behavior is governed by the number of non-zero elements in C. Its first phase occurs during the loading of C; the second phase, the longest, occurs during actual clustering; the third and last phase occurs during the assignment of clusters to pages.

Logical phases of execution can be described in terms of the call trees. A phase of execution includes all references made from the time of entry to a subtree, to the time of the succeeding exit, provided that the time (or number of references) between entry and exit is sufficiently long. In Figures 3.5-&Fignum(&ALLIXDPAM)., modules whose execution leads to phases are surrounded by boxes. The subtrees defined by these modules contain the modules referenced during these phases; the frequency with which they are referenced, and the over-all length of the phase, depend upon the static program structure.

Observe that phases for GENREF can involve a larger number of different modules than RESTRUCT or CLUSTER. Since phases represent periods of locality of reference, modules referenced during the same phase should, if at all possible, be stored in the same page. The restructuring process ultimately decides which of these modules will occupy the same page.

```
                                 +--CLEANUP
                                 |--CONSTCL
                                 |--DUMPCL
                                 |--DUMPEL
                                 |--FETIFLD
                 +--CLUSTR   =+--INITCD
                 |               |                   +--DUMPEL
                 |               |--[MERGE]  =+--INSERT
                 |               |                   +--REMOVE
                 |               |
                 |               |                   +--DUMPMAP
  CLUSTER  =+               |--PRNTMAP =+--PAGENUM
                 |               |                   +--SETIFLD
                 |               |
                 |               |--SETCFLD
                 |               |--SETIFLD
                 |               +--SETPTRS
                 |
                 |               +--FETIFLD
                 |               |                   +--FETDATA  ====READATA
                 +--[LOADCP]=+               |
                                 |                   |--FETINIT  =+--POSFILE
                                 +--READRMT  =+                  +--RFD
                                                     |--SETFILE
                                                     +--SETMAP
```

Figure 3.7
Call Tree for Subject Program CLUSTER

3.3.3 The Testbed of Execution Traces

For each subject program, a testbed of up to twenty execution traces was collected. The testbed is used to determine the intrinsic referencing characteristics of the subject program, based on observed executions. It is also used to validate the synthetic strings generated from the FAM model. Each execution trace is associated with the input data the subject program ran against to produce the trace. The lengths of the execution traces ranged from 500 to 50,000 procedure references, with a median length of around 5,000. Execution traces are referred to by subject program name (GENREF, RESTRUCT or CLUSTER) and testbed index (e.g., TB-3, TB-11B).

Table 3.2 gives the names of the testbed string sets used in the restructurability experiment. As seen in the table, single strings from the testbed are named using lower case "y", e.g., ye. Sets of multiple strings (e.g., Y1 and Y2) are actually ordered sets of strings. For instance, Y1 represents the string formed by the concatenation (TB-2)+(TB-3)+(TB-5), in the case of subject program GENREF. The set TBED contains all the strings in the testbed.

Subject Program	Testbed Trace String Sets		
	ye	Y1	Y2
GENREF	TB-6	TB-2+3+5	TB-3A+4+6
RESTRUCT	TB-4	TB-2+4+5	TB-0+1+3
CLUSTER	TB-12	TB-2+3+5	TB-1+4+6

Table 3.2
Trace String Testbed

3.4 THE RESTRUCTURABILITY EXPERIMENT

In this phase of the research, we wanted to verify whether program restructuring does indeed work for the subject programs used in this research, and whether the same relative performance improvements are achieved from restructuring regardless of the restructuring algorithm. In order to reduce the volume of data generated by the succeeding experiments, we selected values of the restructuring algorithm A, control variable θ and page size p for which restructuring yields consistent performance improvements when applied to execution trace strings. We also wanted to investigate the robustness of restructuring as a function of the set of evaluation strings. Finally, we wanted to see the extent to which performance improvements differ when the trace string used as input to the restructuring process is changed.

The variables, or parameters, of the restructuring process are the restructuring algorithm A, its control parameter θ, and the clustering page size p. The values of these variables used in this experiment are presented in Table 3.3.

Subject Program	Restructuring Variable				
	Algorithm	Theta (θ)	Page size		
			p1	p2	p3
GENREF	CLRU	1,2,3,4	80	120	160
	CWS	1,3,5,7			
RESTRUCT	CLRU	1,2,3	60	90	120
	CWS	1,3,5			
CLUSTER	CLRU	1,2,3,4	72	108	144
	CWS	1,3,5,7			

Table 3.3
Variables of the Restructuring Experiment

3.4.1 Purpose of the Experiment

In order to determine the extent to which restructuring could be expected to improve the performance of the subject programs, we performed restructuring using actual trace strings. The purpose of the experiment was four-fold.

(1) To demonstrate that the choice of layout does indeed affect the performance of the subject programs.

(2) To demonstrate the feasibility of applying program restructuring to the subject programs.

(3) To select values of the restructuring variables, A, θ and p, for use in the remainder of the research.

(4) To determine how sensitive the amount of improvement is to the choice of trace string used as input to the restructuring process, and to determine the extent to which the amount of improvement is preserved across different executions.

3.4.2 Intrinsic Program Characteristics

In this section we display and interpret performance curves that exhibit the distinctive referencing behavior of each subject program. Further, we decide upon a region of memory allocation over which a significant reduction in memory cost is observed. This will be the evaluation interval over which performance comparisons will be made. The intrinsic characterizations presented describe a carefully chosen set of execution traces. (Using other executions leads to the same general results as those shown.)

Intrinsic locality properties, such as paging activity, memory cost and memory demand, can be determined by simulating a page replacement algorithm on a symbolic reference string, assuming a one-to-one mapping of modules into pages. The normalized cost curves in Figures 3.8 and 3.10 plot memory cost (relative to the maximum observed cost) versus working set size. The working set curves in Figures 3.9 and 3.11 show the comparative growth rate of working set sizes for the subject programs.

The intrinsic behavior of the single execution ye is shown in the curves in Figures 3.8 and 3.9. The execution ye for subject programs GENREF, RESTRUCT and CLUSTER has respective lengths 27K, 7K and 14K references. RESTRUCT and CLUSTER both have phases that are shorter than those of GENREF. A further similarity is that they cycle through their entire code body, once for each value in the input stream. GENREF reenters only its dominant (the generation) phase. The effect of cycling through the program body is that the working set grows with the WS window, as the window spans more than one moderate length phase. For programs whose dominant phase is longer than the maximum window (such as GENREF), the growth is much slower.

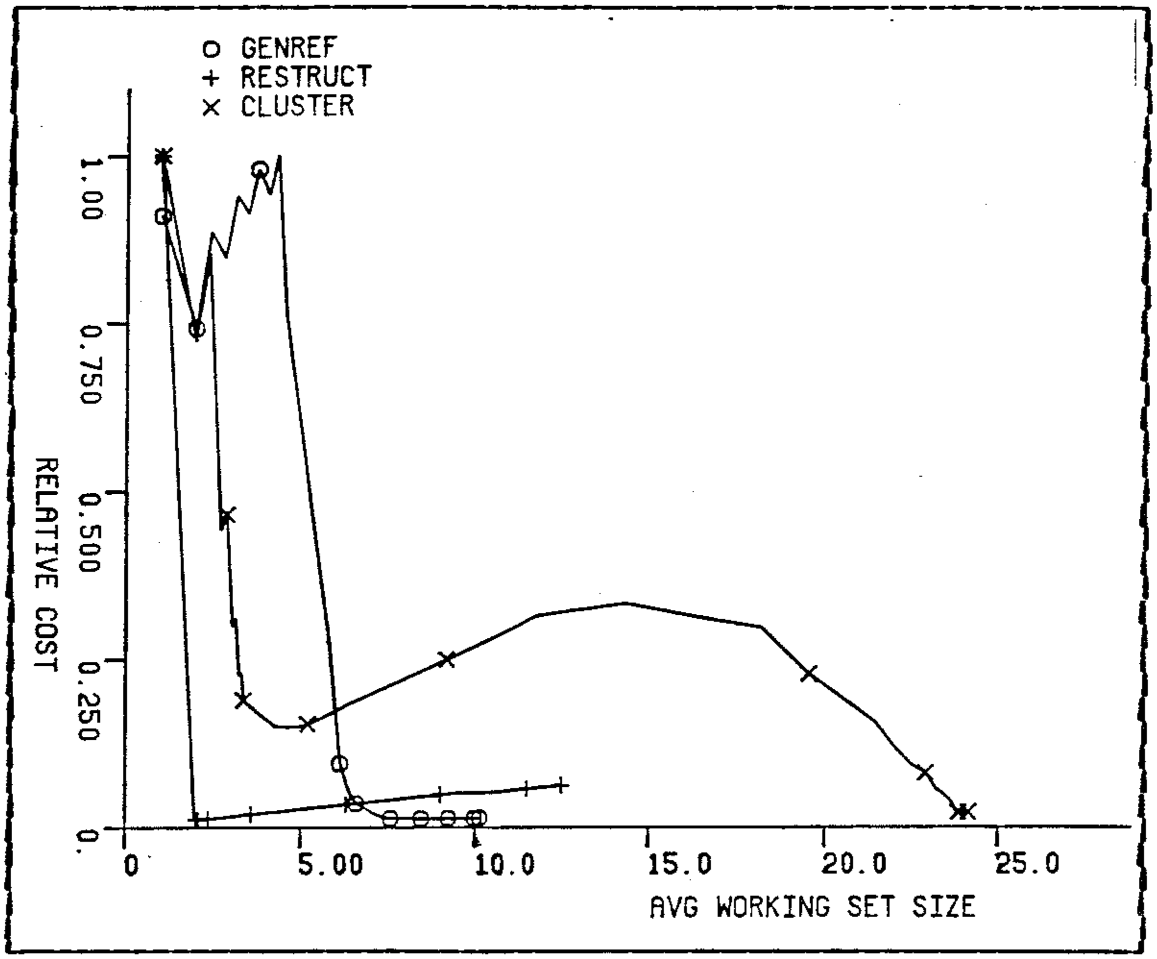

Figure 3.8
Intrinsic ye Cost Curves

In the normalized memory cost curves of Figure 3.8 we see a rapid drop in memory cost when the average working set size approaches a certain value: GENREF near 6, RESTRUCT near 2 and CLUSTER near 4, suggesting that the dominant phases require 6, 2 and 4 modules, respectively. From our discussion in section 3.1.3, it would appear that using restructuring windows $\theta=5$, $\theta=1$ and $\theta=3$ should yield good CWS restructuring results for GENREF, RESTRUCT and CLUSTER, respectively.

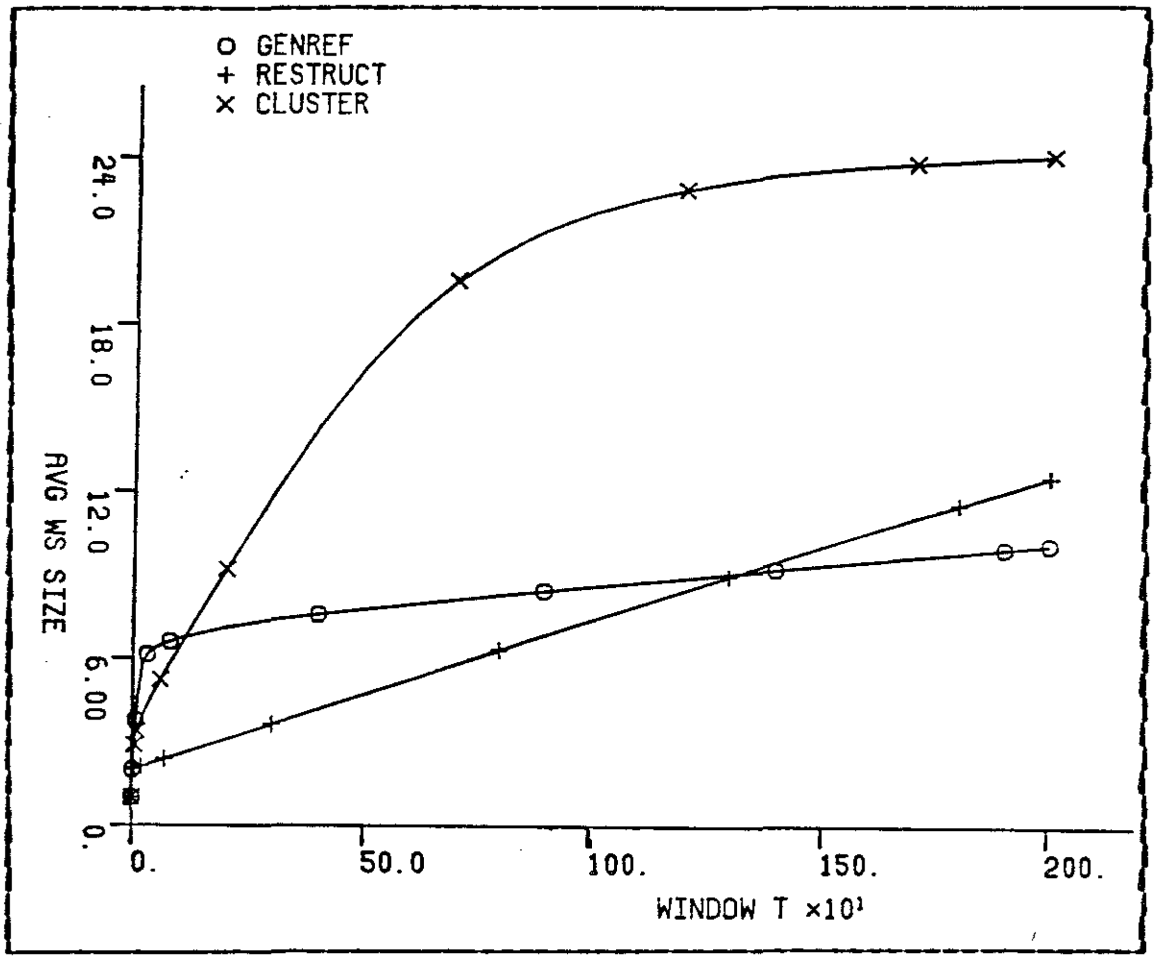

Figure 3.9
Intrinsic ye Working Set Curves

The intrinsic behavior of an ordered set (Y1) of three execution traces was determined. The set Y1 for the subject programs were shorter than the ye executions, ranging in length from 4.4K to 13K references. The executions were carefully chosen to represent a diversity of subject program input types. Y1 represents three back-to-back executions of the subject program. That is, Y1 cycles through the entire program code three times. For programs such as RESTRUCT and CLUSTER, the behavior of Y1 is not significantly different from that of ye; for GENREF, though, the induced program cycling should change the behavior.

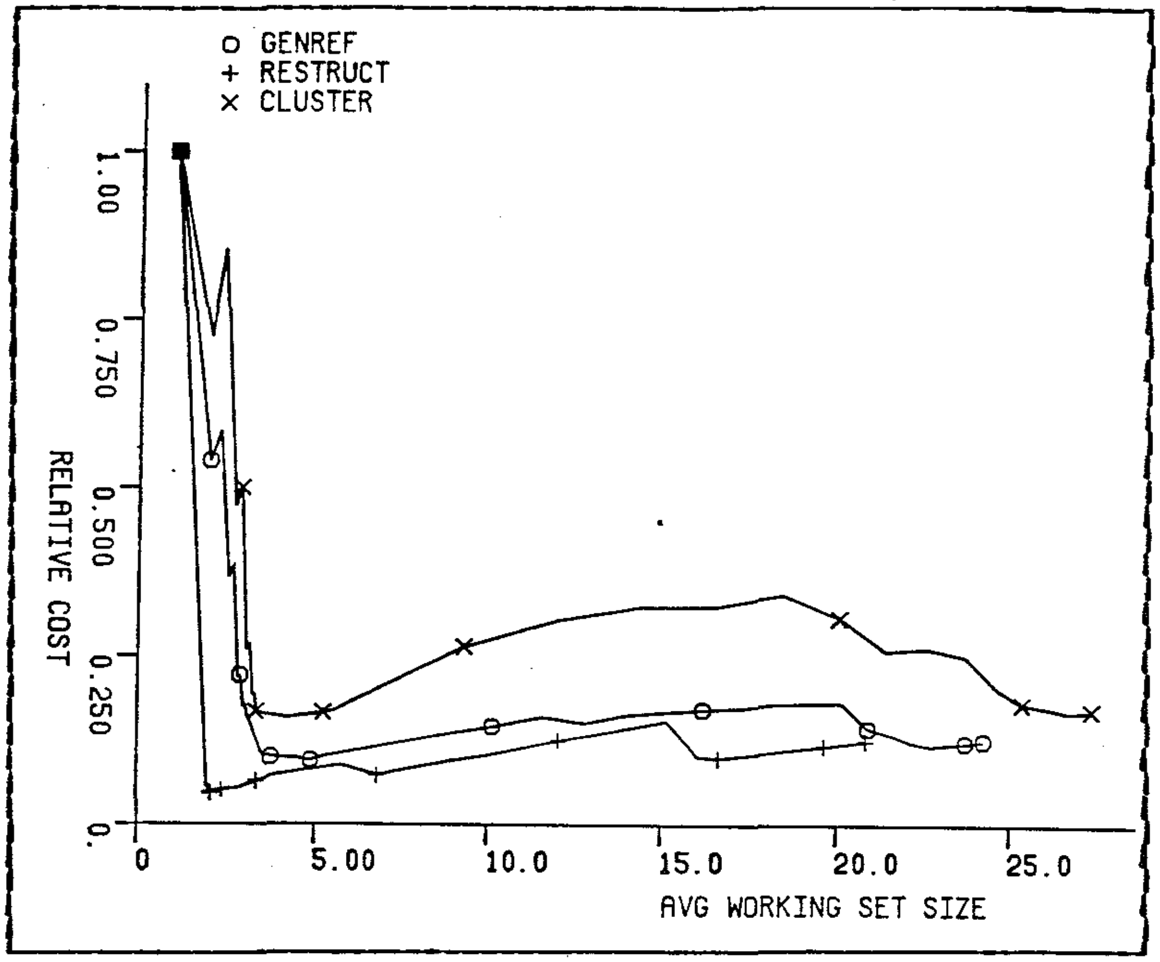

Figure 3.10
Intrinsic Y1 Cost Curves

Compared to single execution ye, for execution set Y1, the maximum working set sizes (at window 2000) are larger for all programs: by 25% for CLUSTER, 60% for RESTRUCT and 150% for GENREF. The cycling for GENREF causes the window to encompass parts of terminal (generation) phase and the initial phase, increasing the over-all working set size. The effect upon the memory-cost drop-off point is most marked for GENREF, changing from 6 to 3; they are unchanged for RESTRUCT and CLUSTER, since ye contains the same cycling pattern as does Y1. The decrease in drop-off point for

GENREF is caused by two factors. First, in each of the three short executions in Y1, the generation phase was not the dominant one, whereas in ye, which is four times longer than the length of Y1, the the generation phase dominates virtual time. Second, the cycling in Y1 causes the non-generation phases to dominate, by virtue of their increased frequency.

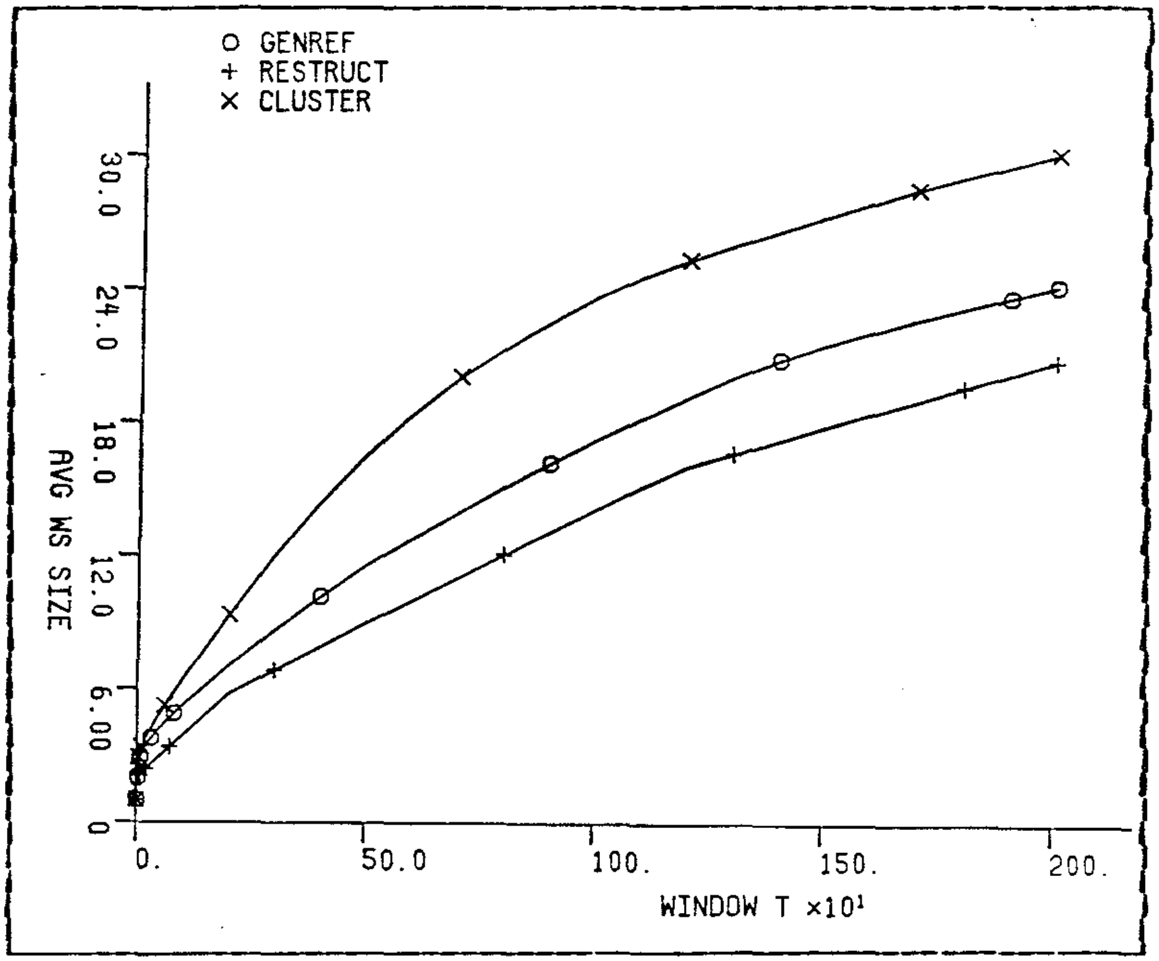

Figure 3. 11

Intrinsic Y1 Working Set Curves

The intrinsic curves can also help to identify an evaluation interval, Q=[θ1,θ2], of θ values over which different layouts should be compared. We use the following guidelines.

(1) The total cost reduction over interval Q should be at least 50%.

(2) Over interval Q, the WS size should reach 50-70% of its maximum size. (This represents the normal operating region of memory allocation on paged systems.)

Such a choice of Q covers both a region of tight memory constraint, and one of lesser constraint. For the LRU replacement algorithm, the interval Q=[1,number of pages] was chosen; for WS evaluation, Q=[1,100]. A wide evaluation interval will result in more conservative comparisons of layout performance, because of the smoothing produced by the weighted sum used to compute cost over an interval. For these choices of evaluation intervals, the subject programs meet guidelines (1) and (2).

3.4.3 Performance of Standard Layouts

Standard layouts represent typical ways of assigning modules to pages without using restructuring techniques. We used four.

(1) L.ALPHA -- the modules are presented to the linker in alphabetical order;

(2) L.RANDOM -- the modules are presented to the linker in random order;

(3) L.TEXTUAL -- the modules are presented to the linker in the order in which they appear in the source text;

(4) L.WORST -- the linker is instructed to store one module per page.

The layouts are formed by a first-fit placement of modules taken from the input list. Except for L.WORST, page breaks occur when an attempt to store the next input module would result in a page overflow.

Figure 3.12 shows typical cost curves comparing the standard layouts against a common evaluation string set Y1. In the figure, ordinate values are scaled (down) by the largest factor of ten such that the smallest scaled value is less than ten, and the common logarithm of the scaled value is plotted. (The scaling merely improves the appearance of the curve.) For small allocations, L.TEXT is clearly superior, but for increased allocations the differences among L.TEXT, L.ALPHA and L.RANDOM diminish.

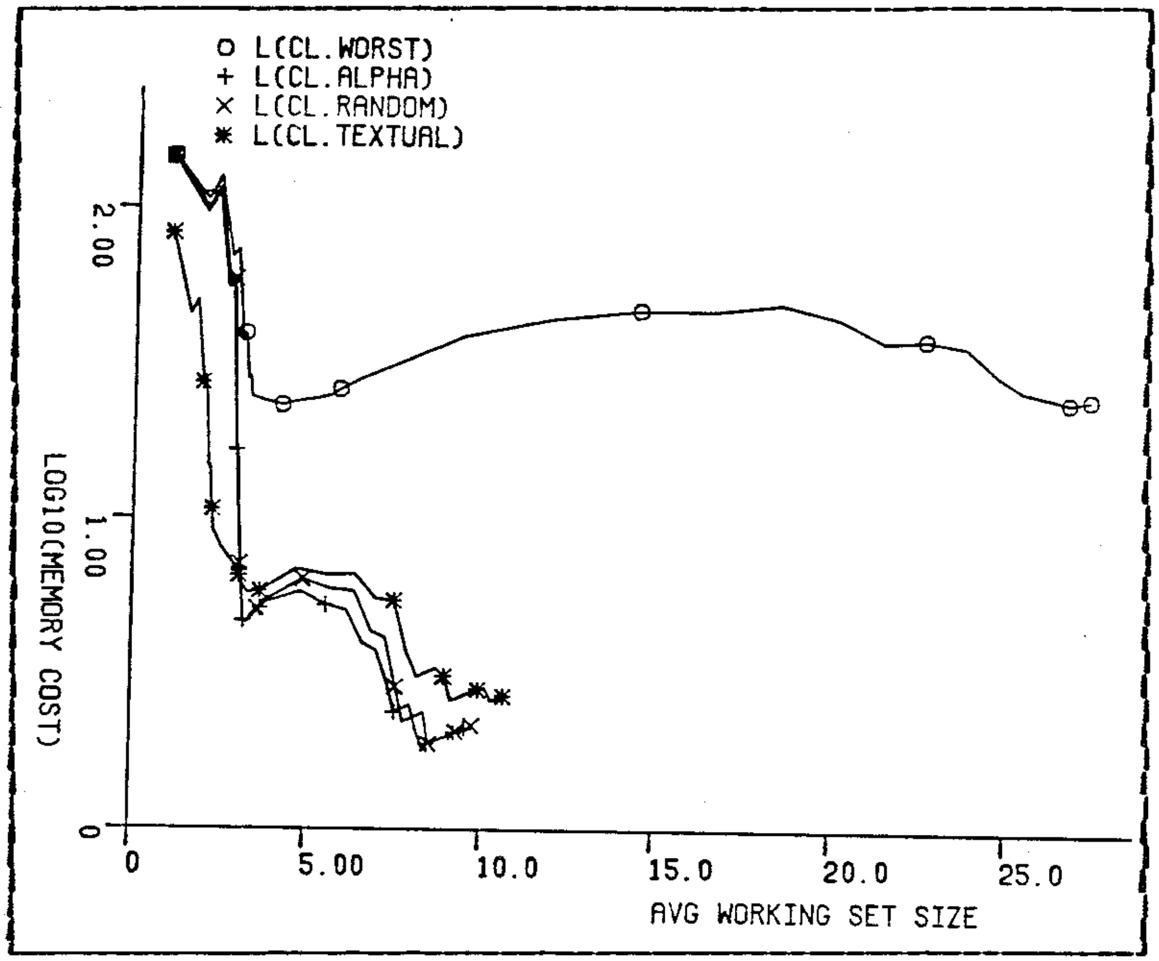

Figure 3.12
CLUSTER STD Layout Y1 WS Cost Curves

We compared the performance of L.ALPHA, L.RANDOM and L.TEXTUAL relative to L.WORST by evaluating them against Y1 over the selected evaluation intervals, and for each page size. Tables 3.4 and 3.5 show the performance improvements over the WORST layout. In particular, we observed the following trends.

(1) The benefits of restructuring increase with page size. The increase is generally monotonic.

Subject Program	Eval Str Set	STD Layout	Page Size p1	p2	p3	Avg
GENREP	Y1	ALPHA	32	54	62	49
		RANDOM	32	54	60	49
		TEXTUAL	44	55	63	54
		Avg	37	54	62	51
RESTRUCT	Y1	ALPHA	45	59	75	60
		RANDOM	39	61	71	57
		TEXTUAL	45	82	86	71
		Avg	43	67	77	63
CLUSTER	Y1	ALPHA	49	73	88	70
		RANDOM	50	73	91	72
		TEXTUAL	56	74	89	73
		Avg	52	73	89	71

<u>Table 3.4</u>

STD Layout WS Cost Reductions over L.WORST

(2) Except for the subject program RESTRUCT, where the TEXTUAL layout is vastly superior to L.ALPHA and L.RANDOM, there appears to be no substantial difference in the standard layouts. For RESTRUCT, the two tightly-bound modules that account for the dominant phase appear adjacent to each other in the source text. Since L.ALPHA and L.RANDOM ignore this order, they fail to store these modules in the same page.

(3) The best standard layout across all subject programs was L.TEXTUAL. Henceforth, we denote by L.BEST the best standard layout for a given subject program.

Subject Program	Eval Str Set	STD Layout	Page Size p1	p2	p3	Avg
GENREF	Y1	ALPHA	55	63	60	59
		RANDOM	53	60	55	56
		TEXTUAL	58	63	64	62
		Avg	55	62	60	59
RESTRUCT	Y1	ALPHA	52	28	53	44
		RANDOM	47	27	24	33
		TEXTUAL	47	92	94	78
		Avg	49	49	57	52
CLUSTER	Y1	ALPHA	58	57	74	63
		RANDOM	54	56	62	57
		TEXTUAL	64	78	80	74
		Avg	59	64	72	65

Table 3.5
STD Layout LRU Cost Reductions over L.WORST

Now we would like to see the extent to which further improvements are possible from the use of program restructuring.

3.4.4 Performance of Computed Layouts

Each computed layout is identified by the restructuring algorithm (CWS or CLRU) used to produce it. For each subject program we computed layouts, L(Y1,A(θ),p), using different values of A, θ and p. We then evaluated these layouts against the evaluation string set Y1. Their comparative performances, relative to L.BEST, the best standard layout for a given replacement algorithm, are presented in Tables 3.6 and 3.7.

Subject Program	Eval Str Set	Computed Layout	Page Size p1	p2	p3	Avg
GENREF	Y1	CWS(Y1,1)	-7	44	53	30
		CWS(Y1,3)	42	60	67	56
		CWS(Y1,5)	45	58	72	58
		CWS(Y1,7)	36	55	71	54
		Avg	29	54	66	50
RESTRUCT	Y1	CWS(Y1,1)	3	15	11	10
		CWS(Y1,3)	2	-77	-86	-54
		CWS(Y1,5)	7	-40	-42	-25
		Avg	4	-34	-39	-23
CLUSTER	Y1	CWS(Y1,1)	6	38	25	23
		CWS(Y1,3)	34	56	42	44
		CWS(Y1,5)	31	46	31	36
		CWS(Y1,7)	32	56	48	45
		Avg	26	49	37	47

Table 3.6

CWS Layout WS Cost Reductions over L.BEST

Subject Program	Eval Str Set	Computed Layout	Page Size p1	p2	p3	Avg
GENREF	Y1	CLRU(Y1,1)	5	53	72	43
		CLRU(Y1,2)	13	39	76	43
		CLRU(Y1,3)	23	53	62	46
		CLRU(Y1,4)	34	51	69	51
		Avg	19	49	70	45
RESTRUCT	Y1	CLRU(Y1,1)	6	22	38	22
		CLRU(Y1,2)	-7	-537	-583	-376
		CLRU(Y1,3)	10	-341	-690	-340
		Avg	3	-285	-445	-231
CLUSTER	Y1	CLRU(Y1,1)	14	53	64	44
		CLRU(Y1,2)	21	59	72	51
		CLRU(Y1,3)	28	17	-2	14
		CLRU(Y1,4)	41	13	54	36
		Avg	26	36	47	36

<u>Table 3.7</u>

CLRU Layout LRU Cost Reductions over L.BEST

These results suggest the following.

(1) For RESTRUCT, the Nearness method (CWS(1) or CLRU(1)) is clearly superior to any other restructuring algorithm. This is no surprise, however, since the Nearness method is based on adjacency of reference, a one-one relation between modules. Negative table entries for RESTRUCT's CLRU and CWS layouts indicate that other layouts actually perform <u>worse</u> than the TEXTUAL layout. Notice also that very small improvements occur for the small page size (p1) because it is too small to contain the dominant phase consisting of two modules. In this case we see that a good restructuring algorithm can be thwarted during the clustering phase.

(2) For CLUSTER, the CWS(3) and CLRU(2) restructuring algorithms result in the best improvements over L.BEST. The

effectiveness of the Nearness method is nearly nil for the small page, but increases with page size, owing largely to the capacity of the clustering algorithm to pack several small clusters within the larger page. This trend is also present in program GENREF.

(3) The effectiveness of using the Nearness method varies with the replacement algorithm used during layout evaluation.* Across all page sizes, Nearness layouts reduced LRU memory costs by up to twice as much as they reduced WS costs.

(4) A slight anomaly in the relationship between page size and restructuring effectiveness was observed for program CLUSTER. As shown in Table 3.6, an increase from page size p2 to p3 resulted in a reduction in restructuring effectiveness, relative to the L.TEXTUAL layout. A possible explanation is the the L.TEXTUAL layout was better able to use the larger page size, resulting in less margin for improvement through restructuring. In Table 3.4 we see that L.TEXTUAL improved layout L.WORST by up to 90%, which is much higher than the 60-70% improvements recorded for programs GENREF and RESTRUCT.

Program restructuring is indeed beneficial for the subject programs, the amount of benefit depending upon the values of the restructuring variables used.

3.4.5 <u>Setting Restructuring Parameters</u>

We have already seen in Tables 3.4-3.7 that, for a given restructuring algorithm, the amount of improvement is affected by the values of θ and p. As expected, increasing page size increases improvement. Excluding RESTRUCT, cost reductions of more than 50% are achieved for the intermediate page size. Although more impressive improvements are realized for the large page size, improvements realized for the intermediate sized page represent conservative estimates of the benefits of applying program restructuring to the subject programs. Table 3.8 summarizes the θ of choice for each restructuring algorithm, using the intermediate sized page.

Program	Algorithm	Theta	Page Size	Memory-Cost Reduction
GENREF	CWS	5	120	54
	CLRU	1	120	53
RESTRUCT	CWS	1	90	15
	CLRU	1	90	22
CLUSTER	CWS	3	108	64
	CLRU	2	108	59

<u>Table 3.8</u>
Restructuring Parameter Settings

It appears that the differences between the CWS and CLRU layouts (for WS and LRU replacement, respectively) are not substantial: both show the same relative effects from changes in θ and p, and both produce the same relave performance improvements. We will henceforth use CWS exclusively.

3.4.6 Stability of Restructuring Improvements

In order to identify executions that lead to good layouts, we restructured using each string in a set of selected traces and compared the memory costs of the layouts they produced. The strings (i.e., layouts) were ranked according to their performance against a common trace string (1) to give us an idea of what the chances were of making a good (or bad) choice of execution upon which to base restructuring, and (2) to identify the best choice of evaluation string for subsequent layout comparisons. We found that, except for TB-2, RESTRUCT is very insensitive to the values of its inputs, as shown in Table 3.9. Even short executions of RESTRUCT provide the same restructuring information as much longer executions. Programs GENREF and CLUSTER show more sensitivity. For each subject program, the execution with the highest restructurability rank (indicated by the asterisk) was used as the standard layout evaluation string, ye, for the subsequent phases of the research.

In Table 3.9 we see that GENREF executions TB-6, TB-4, TB-5 and TB-3A all produce layouts whose performances differ by less than 10%. CLUSTER's executions TB-11, TB-12 and TB-5 are within 1% of each other, and TB-1, TB-3 and TB-2 are within 15% of the best string (layout). We also see that, when restructuring is based on multiple executions, a smoothing effect occurs. For example, the worst execution (to use in restructuring) for GENREF was TB-2, but when it was used together with TB-3 (rank 7) and TB-5 (rank 3), the resulting layout was only 15% inferior to the best layout, as opposed to the 125% inferiority of TB-2 when used alone. Similar results were observed for the other subject programs. We draw the conclusion that restructuring is stablest when it is based on multiple execution trace strings.

		Subject Program					
		GENREF		RESTRUCT		CLUSTER	
		ID	%DIFF	ID	%DIFF	ID	%DIFF
SINGLE EXECUTIONS		*TB-6	0	*TB-4	0	*TB-12	0
		TB-4	-3	TB-1	0	TB-5	0
		TB-5	-4	TB-3	0	TB-11	0
		TB-3A	-7	TB-5	0	TB-1	-11
		TB-1	-16	TB-0	-8	TB-2	-11
		TB-2A	-24	TB-2	-77	TB-3	-11
		TB-3	-33			TB-7	-37
		TB-2	-124			TB-6	-83
						TB-4	-122
MULTIPLE	Y1=	(TB-2 TB-3 TB-5)	-15	(TB-0 TB-1 TB-3)	-8	(TB-2 TB-3 TB-5)	0
	Y2=	(TB-3A TB-4 TB-6)	-1	(TB-2 TB-4 TB-5)	0	(TB-1 TB-4 TB-6)	0
	Y3=	(TB-1 TB-2A TB-5)	-1			(TB-7 TB-11 TB-12)	-22
	YT=	(Y1 Y2 Y3)	0	(Y1 Y2)	-8	(Y1 Y2 Y3)	0

<u>Table 3.9</u>

CWS Restructurability of Testbed Traces

(Improvement over Layout L(ye))

Once specific values of the restructuring variables, A, θ and p, were determined, we wanted to verify that the comparable performance improvements could also be achieved using a different set of strings as inputs to the restructuring phase. We chose this set, Y2, to be three strings representing different classes of subject program inputs than those representing Y1. The results from evaluating layouts L(Y1,A(Є),p) and L(Y2,A(Θ),p) against evaluation string sets Y1, Y2 and ye are shown in Table 3.10. This table shows CWS improvements over the best standard layout. Over-all, there is little variation in the performances of layouts L(Y1), L(Y2) and L(ye) across executions Y1, Y2 and ye.

Subject Program	Layout	Evaluation String Set Y1	Y2	ye	Avg
GENREF	L(Y1)	58	66	65	63
	L(Y2)	59	71	69	66
	L(ye)	59	71	70	67
RESTRUCT	L(Y1)	15	14	19	16
	L(Y2)	13	11	12	12
	L(ye)	15	14	19	16
CLUSTER	L(Y1)	56	56	78	63
	L(Y2)	56	56	78	63
	L(ye)	56	56	78	63

Table 3.10
Stability of CWS Cost Reductions over L.BEST

3.4.7 Conclusions

Program restructuring is feasible for the subject programs. We found that performance improvements are preserved across a range of different executions. Of all the subject programs, RESTRUCT is by far the easiest one to restructure, since very short executions provide enough information to achieve an optimal layout. The set of restructuring variables we will use during the remainder of the study are such that intermediate improvements are achieved from restructuring.

In terms of classifying the subject programs based on restructurability, RESTRUCT is a type-2 program, whereas GENREF and CLUSTER appear to be type-3 programs. We expect RESTRUCT to be easier to model than GENREF and CLUSTER. (In general, one can only make an intelligent guess as to the restructurability of a program, given only its static source text and some knowledge of its function.)

3.5 SUMMARY

Program restructuring works for the subject programs. We have chosen values of the restructuring variables, A, θ and p, and the evaluation interval, $Ç$, such that program restructuring produces performance improvements of 20-75% over the best standard layouts. Restructuring works best when it is based on more than one execution trace.

Chapter 4

ELEMENTARY MODELING

This chapter describes the first level of PAM modeling in which the simplest model versions and parameter distribution assumptions are used. We begin with a review of the issues of modeling. Next we describe the two modeling experiments that constitute this major study of the research. In the next chapter, we introduce the next higher level of modeling in which more sophisticated parameter estimation techniques are used.

Both model versions DNF-PAM and GPAM were found to reproduce actual trace strings successfully. The subject programs appear to have a range of modeling difficulty which is related to the restructurability of the modeled executions.

4.1 GENERAL MODELING ISSUES

4.1.1 Overview of Issues

The general PAM modeling procedure involves the four steps shown in Figure 4.1. At each step, certain issues relating to modeling must be treated before proceeding to the next step. The issues of PAM modeling fall into four categories.

(1) Model version. Which model version, GPAM or DNF-PAM, produces better synthetic strings for use in program restructuring?

(2) Parameter representation. What statistics should be used to characterize loop and selection constructs? How much detail about the statistical distribution of construct parameters is required to produce acceptable restructuring results?

(3) Parameter estimation. Which program executions should be sampled to provide the raw data for parameter estimation? How should the raw data be combined to produce a single characterization of the sampled executions?

(4) Model validation. How accurate is a model instance? How is accuracy determined?

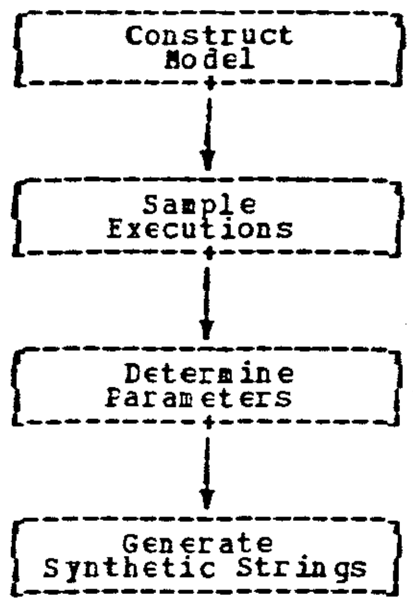

Figure 4.1
Overview of PAM Modeling

4.1.2 Model Version

Of the four versions of PAM introduced in Chapter 2, we studied extensively only DNF-PAM and GPAM. Our discussion of parameter estimation treats GPAM almost exclusively. GPAM is more general than DNF-PAM and, since it uses more information about the program's static and dynamic characteristics, one would expect it to have a better chance of accurately modeling the subject program.

In essence, PAM is a simulation model. PAM synthetic string generation simulates the execution of the subject program. The simulation is driven by model parameters derived from observing actual program executions driven by input data. As with any simulation model, the quality of the model depends upon its underlying structure and upon the quality of parameter estimates.

The structure of GPAM is directly related to the control structures used in the subject program. GPAM recognizes loops, selection, escapes and procedure entry and exit as the only significant program execution activity. No distinction is made among the three types of loops (for, while and until) found in structured programming languages. All loops are modeled as for loops. GPAM models all conditional-execution statements as selection constructs in which exactly one of many alternatives is selected for execution upon entry to the construct. The minimum number of alternatives is two; hence, the "if-then" statement is indistinguishable from the "if-then-else" statement. The only program constructs that require parameterization are loops and selection constructs.

Recall that the DNF-PAM call-sequence grammar takes the form

$$A = < (x1 \mid x2 \mid \dots \mid xm) >.$$

DNF-PAM parameters have slightly different interpretations, all independent of the structure of the source program. The DNF-PAM loop parameter represents the number of procedure calls made per entry to the procedure; selection construct parameters are derived from the number of calls made by the procedure. DNF-PAM parameters can be derived from GPAM parameters, or they can be gathered by program instrumentation.

4.1.3 GPAM Parameter Representation

A program is characterized in terms of the way it was observed (during one or more executions) to execute its loop and selection control structures. The usage of a loop is characterized by a random variable that represents the number of repetitions, i.e., the number of times the body is executed per entry to the loop. An m-vector of selection probabilities characterizes the execution of an m-way selection construct.

The statistics used to characterize a loop or selection construct can vary -- extrema, averages, variances, modes -- depending upon the modeler's discretion. The choice of statistic affects slightly the instrumentation overhead and, to a greater extent, the cost of calculating parameters from the execution coefficients. The process of parameterizing program constructs involves two major decisions.

(1) The statistic to be used to characterize the execution of a construct.

(2) The amount of detail required about the distribution. Three approaches are typically taken:

a) a point estimate of the distribution;

b) an interval estimate of the distribution (which may require an assumption of the class of the distribution and several of its moments); and

c) the identification of the form of the distribution together with its characterizing variables (e.g., mean and variance). For example, a loop repetition parameter may have a normal distribution with mean 18 and variance 7.

Regardless of how these issues are resolved, we assume that all constructs of a given type have the same form of underlying distribution, but different values for the moments.

4.1.3.1 Loops

Loops have been shown to be the primary cause of locality of reference [DENN76,BATS77b,SNYD78a] because they cause the modules inside the loop to be referenced repeatedly. Such modules tend to have a high clustering affinity for each other. The number of loop repetitions contains restructuring information. Low values discourage clustering of modules referenced inside the loop; high values encourage clustering.

The execution of a given loop can be characterized by random variables r and R, where r represents the mean loop repetition frequency, and R is the maximum loop repetition frequency. Each program execution produces a list of frequencies from which r and R are estimated.

4.1.3.2 Selection constructs

Suppose we have an m-way selection construct. One program execution produces a vector of coefficients, S=(s1,s2,...,sm), where the i'th alternative was executed si times. The vector-valued random variable Q=(q1,q2,...,qm) is estimated using S. The probability of selecting the i'th alternative upon entry to the selection construct is estimated by qi=si/(s1+s2+...+sm), the relative frequency of selecting the i'th alternative during the observed execution.

4.1.3.3 Relative importance of constructs

Since program restructuring is a locality-improvement technique, it should be influenced by the same factors affecting locality, of which looping is a major contributor. GPAM must, then, capture the looping characteristics of the subject program during parameterization, and reproduce those properties during synthetic string generation. It was our feeling that modeling loops would be crucial to successful GPAM modeling.

This is not to say that selection constructs are unimportant. The contribution of a selection construct depends upon the global nesting level of the procedure containing the construct. At the top level, a selection construct can alter a major portion of a program execution. In such "transaction-type" programs, the selection construct is dominant in that it explains a great deal of the variability among executions. Even then, it is looping at some level that accounts for the length of the execution, and for the clustering of the modules invoked during the processing of a particular transaction.

Henceforth, we focus our major attention on the estimation of loop parameters.

4.1.4 Parameter Estimation

The parameter estimation phase of modeling involves combining coefficients from one or more executions to approximate the underlying distributions of the characterizing statistics. Two approaches to combining coefficients from multiple executions are considered.

(1) The composite-execution approach. Given a sample of multiple executions, W={w1,w2,...,wk}, statistics are computed as if the actual execution were w1+w2+...+wk, where '+' indicates string concatenation. Such an approach leads to a single value of the desired statistic which will, in the text that follow, be identified by the superscript ^{1}.

(2) The individual-execution approach. Each execution contributes one data point, and statistics are computed using these data points. For example, each execution produces an average loop repetition frequency, and the average for the entire sample is computed as the mean of the individual execution averages. Parameters computed in this manner are identified by the superscript ^{2}.

In the composite-execution approach, long executions tend to contribute more to the statistic value than do shorter ones. Each execution contributes equally in the individual-execution approach.

A second aspect of parameter estimation is the selection of the program executions to observe. The various approaches were discussed in Section 2.3.2, where the adaptability of GPAM to a variety of parameter representation and estimation approaches was demonstrated.

4.1.4.1 Loop and selection parameter estimators

We now present estimators for the mean loop repetition frequency r, and for the maximum loop repetition frequency R. Given K executions of the program, we define estimators r^1 and r^2 for r, and R^1 and R^2 for R.

r^1 = (total #loop repetitions)/(total #loop entrances);

$r^2 = (r1+r2+...+rK)/K$;

$R^1 = \max(R1,R2,...,RK)$;

$R^2 = (R1+R2+...+RK)/K$.

Rj and rj represent, respectively, the maximum and mean loop repetition frequencies during the j'th observed execution. R^2 is the mean maximum loop repetition frequency over the set of observed executions. Both r^2 and R^2 lend themselves to calculation of variances and to standard statistical estimation techniques.

For an arbitrary m-way selection construct in the subject program, K program executions produce the composite-execution coefficient vector (f1,f2,...,fm), where fi is the total number of times (during K program executions) that alternative i was executed. Each execution also produces its own estimate of Q. Let Qi=(qi[1],qi[2],...,qi[m]) be the random variable of selection probabilities derived from the i'th observed execution. We now have two estimators for Q, Q^1 and Q^2, whose j'th components are given by

$Q^1[j] = fj/(f1+f2+...+fm)$, and

$Q^2[j] = (q1[j]+q2[j]+...+qK[j])/K$, respectively.

Q^1 is easily computed, but does not lend itself to statistical analysis of the individual selection probabilities, as does Q^2.

4.1.4.2 Instrumentation requirements

GPAM subject program instrumentation can easily be extended to provide raw data on repetitions from which the desired statistics can be computed. Collecting raw (versus condensed) data does, however, increase the volume of data produced by the instrumented program. In most cases the condensed data should be just as informative, but they might hide the nature of the underlying distribution of parameter values. Until the form of the distribution has been ascertained (or until some assumption about it is accepted), collecting raw data can yield insights for making parameterization decisions. During preliminary stages of modeling, raw data can pinpoint constructs that have high variability in their usage. These should be modeled carefully.

4.1.4.3 Statistics and estimators used

We have introduced a number of statistics and estimators that can be used in parameter estimation. Each row in Table 4.1 describes a set of parameter representation/estimation approaches. For example, for approach A, loop parameters are estimated from average repetition frequencies computed using the composite-execution approach, and selection parameters are estimated using the composite-execution approach. The approaches investigated in this chapter are indicated by an asterisk in the "Approach" column.

In order to control the amount of experimental data generated during multiple-execution modeling, we had to settle upon a small number of intuitively appealing approaches that showed promise in preliminary studies. For loops, we investigated the use of two statistics, mean and maximum repetition frequencies, and their respective estimators, r^1 and R^1. Of particular interest was whether it is better to use a large, extreme value of the loop repetition parameter, or a more typical value, such as the mean. The values given by estimators r^1 and B^1 should differ from each other enough to account for any observable difference in the synthetic strings produced from the respective model instances.

Construct Estimators

	Loops				Select	
Modeling Approach	r^1	B^1	r^2	R^2	Q^1	Q^2
*A	X				X	
B	X					X
*C		X			X	
D		X				X
E			X		X	
F			X			X
G				X	X	
H				X		X

Table 4.1

Variables of the GPAM Point-Estimation Study

In this chapter, we deal exclusively with point estimates of the underlying parameter distributions. Stratified sampling is used to select program executions from which model parameters are derived. These executions are driven by different classes of input data, present in proportion to their expected occurrence in the population of all executions of the subject program. In the next chapter, we will deal in more detail with the underlying distributions, but only for those subject programs that appear sensitive to the choice of statistic, or that are unusually difficult to model. Random sampling will be used to obtain model parameters.

4.1.5 Model Validation

4.1.5.1 Experiment overview

Each modeling experiment consisted of the following steps.

(1) Select the set OBS of subject executions, from which parameters PARM(OBS) are determined.

(2) Restructure using OBS as input to the restructuring phase to produce layout L(OBS).

(3) Generate set SYN of synthetic reference strings from model instance PAM<CSG,PARM(OBS)>.

(4) Restructure using SYN as input to the restructuring phase to produce layout L(SYN).

(5) Evaluate layouts L(OBS) and L(SYN) against the standard evaluation string set ye to yield performance indices eOBS and eSYN, respectively.

4.1.5.2 Terminology

The quality of the model instance PAM<CSG,PARM(OBS)> is defined in terms of the relative performances of layouts L(OBS) and L(SYN), and given by

$$\text{Quality} = 100 * \frac{eOBS - eSYN}{eOBS},$$

the percentage performance improvement of L(SYN) over L(OBS). Quality measures the closeness of SYN to OBS. A negative Quality value indicates that SYN is inferior (for restructuring) to OBS; a positive value indicates that SYN is superior; and a zero value indicates that SYN is equivalent to CBS in restructurability. When Quality=0, we also say that the model instance reproduces (the restructuring signature of) the subject executions OBS.

At this point we feel it would be helpful to discuss the usage of terms by which model quality is expressed. The quality of a model instance measures the closeness of the synthetic strings it produces to the subject executions (traces) from which the model instance was constructed. Model accuracy is a statement of the over-all quality of instances of the model constructed for different subject programs. Accuracy is somewhat subjective. We arbitrarily define four levels of accuracy:

(1) poor, when model quality is less than -20%;
(2) fair, when model quality is between -10% and -20%;
(3) good, when model quality is between -5% and -10%;
(4) excellent, when model quality is above -5%.

The effort required to obtain a good quality model instance is a function of the execution-sampling approach, the model version used, and the number of synthetic strings generated from the model instance. We expect that as the model version becomes more general, as the number of observed subject executions increases, and as the number of synthetic strings generated from the model instance increases, so will the quality of the model instance. A subject program is said to be difficult to model when quality does not improve with modeling effort.

Since there are approaches to model validation [SPIR77] other than the one we used, our conclusions about the quality of instances of PAM may not be consistent with those obtained when other validation techniques are used. How well results from our validation procedure correlate with those obtained using other techniques (such as WS curve analyses or phase decomposition) is perhaps best treated in a separate study.

4.2 THE POINT-ESTIMATION MODELING APPROACH

4.2.1 Introduction

From this point on, the emphasis will be on modeling program loops. The simplest parameter-estimation and synthetic string-generation approach is to use point estimates of the loop repetition frequencies to characterize loop execution. Under such assumptions, generate-time simulation of an arbitrary program loop always produces the same number of repetitions -- the value given by the parameter.

The first concern about such an approach is the quality of the approximation to the actual trace string obtainable using such simplifying assumptions.

This phase of the research has the following objectives.

(1) To determine whether such a simple approach produces any useful results at all.

(2) To characterize instances when this approach produces acceptable results.

(3) To determine the relative benefits of using DNF-PAM versus GPAM based on point estimates of model parameters.

4.2.2 Overview of the Study

Each execution of the subject program produced execution coefficients which were stored in the coefficient database (CDB). The CDB for each subject program contained coefficients from 80-100 executions. The CDB was used to compute model parameters, and to perform statistical analyses of model parameters -- the topic of the next chapter. A test-bed of 8-15 execution trace strings was collected for each subject program. For the modeling effort of this chapter, the test bed enabled us to validate model instances against "answer" strings.

Two experiments were conducted per model version: single-execution modeling and multiple-execution modeling. Single-execution modeling was used to determine the basic accuracy of the model as a function of (1) the subject program, (2) the restructurability of the subject execution, and (3) the statistic used to characterize model parameters. One to three single executions were modeled. We selected executions that spanned a range of restructurability.

Since in Chapter 3 we saw that restructuring using multiple executions was superior to that using single executions, we concluded that it was important to show that multiple executions could be modeled accurately. For each subject program, two sets of multiple executions, Y1 and YT, were chosen, again based on restructurability. Y1 contains three carefully selected executions; YT contains a major portion of the testbed. Y1 represents a small-sample approach to modeling; YT represents the expenditure of more effort in execution-sampling phase.

It should be understood, at this point, that there is a fundamental difference between modeling a single execution and modeling multiple executions. To illustrate, let vectors X1,X2,...,Xm be the model parameters from m executions of the subject program. When single-execution modeling is used, the model instances PAM<CSG,PARM(Xi)> are based on actual executions. On the other hand, the parameter vector X used in multiple-execution modeling is some function X=f(X1,X2,...,Xm) of parameters from actual executions. X describes the "typical" execution which, for all practical purposes, does not really exist.

Vector X is likely to be close to an actual execution parameter vector Xi, when Xi represents the <u>dominant</u> execution in the set of m executions. The dominant execution contributes most to the layout computed directly from the set of executions. The use of statistics that measure extrema (such as maximum) increases the likelihood of having a dominant execution, as does modeling a small number of executions. For larger samples, or when statistics that smooth (such as the mean) are used, the "typical" execution parameter vector X is more likely not to describe an actual execution.

The foregoing discussion points out that results from multiple-execution modeling should not be used so much to measure model accuracy, as to measure how well a set of executions can be characterized using PAM parameter estimation techniques. In particular, multiple-execution modeling enables us to evaluate the use of various statistics to define model statistics, and to study the effect of the sample size upon the characterization. Although the term "quality of the model instance" will still be used when discussing all modeling results, for multiple-execution modeling it is the process whereby parameters are obtained that is being judged.

4.2.3 Hypotheses

We now present some verifiable hypotheses representing claims that can be shown true or false on the basis of the results presented in this chapter.

- Good model instances can be constructed from a small number of subject executions.

- The simplest model version, DNF-PAM, comes to within 10% of the more sophisticated GPAM version in quality.

- The quality of the model instance increases with the number of synthetic strings generated from a model instance. That is, a smoothing effect occurs in which the composition of synthetic strings better represents the subject executions than does a single synthetic string.

- Obtaining a good quality multiple-execution model instance requires more modeling effort than is required to achieve a single-execution model instance of comparable quality.

- Choice of statistic and estimator affects model accuracy.

- Ease of modeling is related to the restructurability of the subject executions.

4.3 MODEL ACCURACY

4.3.1 DNF-PAM

From Table 4.2, we see that this simple model version did a good job of modeling GENREF. In the best case, the synthetic strings were within 1% of TB-4; in the worst case, they were only 4% inferior. Subject program CLUSTER was the hardest to model. Synthetic strings for TB-3 (which ranks second in restructurability) was just 1% inferior in the best case, and 12% in the worst case. For TB-5 (rank one), the best synthetic string set was 11% inferior to the actual execution trace.

			Synthetic Set	
Subject Program	Trace String	Rank	Size	Quality
GENREF	TB-4	2	3	-4
			3	-1
			6	-4
RESTRUCT	TB-3	1	3	-8
			3	-182
			3	-8
			6	-8
			9	-8
CLUSTER	TB-3	2	3	-11
			3	-1
			6	-12
	TB-5	1	3	-11
			3	-11
			6	-24

Table 4.2
Model Quality: DNF-PAM Single-Execution Modeling

RESTRUCT, the subject program thought to be the easiest to restructure, exhibited a disturbing, anomalous behavior in which it produced a synthetic string set that was nearly 200% inferior to the subject execution. Further investigation disclosed that the major phase of execution corresponding to CLRU restructuring was not present in the synthetic strings, although it was present in the subject execution. The layout constructed from the synthetic strings was not optimized with respect to the CLRU phase. The unusually poor performance of this layout occurred because the evaluation string TB-4 contained an instance of the CLRU phase. When this anomalous synthetic string set is treated as a data outlier, the remaining synthetic strings come to within 8% of the subject execution trace string TB-3.

A counter-intuitive relationship between model accuracy and the number of synthetic strings was observed in the case of CLUSTER. One would expect that increasing the number of synthetic strings would provide a better composite picture of the program that would lead to a better layout. That this does not happen, we conjecture, is because DNF-PAM distorts the sequence of program references (since the model does not contain intra-procedural sequence information). As a result, a synthetic string produces a restructuring matrix whose entries suggest different module affinities than trace strings could possibly suggest. Since matrices from single synthetic strings are <u>added</u> to obtain the restructuring matrix for multiple synthetic strings, these aberrations are magnified.

Synthetic string generation using DNF-PAM can fail to be accurate because of loss of sequence information. When data outliers are excluded, synthetic strings from DNF-PAM model instances were 0-24% inferior to their subject executions. Model instances for CLUSTER were the least accurate; those for GENREF were the most accurate.

4.3.2 GPAM

Results in Table 4.3 show that subject program RESTRUCT was easy to model using either statistic, r or R. For the maximum loop-repetition frequency statistic, R, synthetic string sets for all subject programs were as good as their subject executions -- in the best case. Synthetic strings for GENREF were actually 3% superior to execution TB-5. Except for CLUSTER, using the mean loop-repetition frequency statistic, r, produced synthetic strings that were as good as those obtained using R. CLUSTER was the most difficult to model using r. Model instances for TB-3 and TB-5 were 22% inferior in the worst case, and 10% inferior in the best case.

			Loop Parameter Statistic			
			Mean		Maximum	
Subject Program	Trace String	Rank	#str	Quality	#str	Quality
GENREF	TB-4	2	3	0	3	-7
					2	-1
					5	-7
	TB-5	3	3	-1	3	0
					3	-1
					6	+3
RESTRUCT	TB-3	1	3	0	3	0
CLUSTER	TB-3	2	3	-11	3	-10
			3	-10	6	-10
			6	-10	9	0
	TB-5	1	3	-22	3	-22
			3	-22	3	-22
			6	-22	6	0

Table 4.3
Model Quality: GPAM Single-Execution Modeling

For GENREF and RESTRUCT, the choice of statistic appeared to have little or no effect upon model quality. For GENREF, using the statistic r was superior for TB-4, but R was slightly superior for TB-5. For CLUSTER, however, using the maximum statistic was clearly superior. Of further interest is that with statistic R, increasing the number of synthetic strings improved model quality, a phenomenon absent from DNF-PAM modeling.

4.3.3 Conclusions

GPAM is accurate. For each subject program, there was a choice of statistic (r or R) for which the model instance reproduced the subject execution. Poor choices of statistic exist for some subject programs, such as CLUSTER. DNF-PAM, which does not model control structures, is not as accurate, although the model can generate synthetic strings that are within 11% of the subject execution. Table 4.4 compares the (test-case) accuracy of model versions GPAM and DNF-PAM.

Subject Execution		Rank	DNF	GPAM Estimator	
				Mean	Max
GENREF	TB-4	2	-1	0	-1
RESTRUCT	TB-3	1	-7	0	0
CLUSTER	TB-3	2	-1	-10	0
	TB-5	1	-11	-22	0

Table 4.4
Model Quality: DNF-PAM versus GPAM
(Best results used)

We see from the table that model quality is a function of the restructurability rank of the subject executions, and that the choice of loop parameter statistic makes a significant difference in model quality, for some subject programs, notably CLUSTER. GPAM was most accurate for programs GENREF and RESTRUCT. DNF-PAM was most accurate for GENREF(TB-4) and CLUSTER(TB-3), the subject executions of restructurability rank two.

4.4 MULTIPLE-EXECUTION MODELING

At this point we wanted to see whether multiple executions could be modeled as accurately as single executions. We also wanted to see whether some sets of executions were more difficult to model than others. We modeled subject execution sets Y1 and YT which, for RESTRUCT and CLUSTER, had the same restructurability rank (i.e., Y1 and YT produced equivalent layouts). For GENREF, YT had restructurability rank one, Y1 had rank five.

4.4.1 DNF-PAM

The following discussion refers to the results shown in Table 4.5. It appears that model accuracy is a function of the restructurability of the subject executions. As the restructurability of the subject executions increases, so does the modeling difficulty. It appears that, conversely, multiple-execution modeling can lead to improvements over subject executions that have fair restructurability properties (such as Y1 model instances for GENREF and RESTRUCT). For RESTRUCT and CLUSTER, sets Y1 and YT, which had the same restructurability, were modeled to the same level of accuracy.

Subject Program	Synthetic String Set			
	Y1		YT	
	#str	Quality	#str	Quality
GENREF	3	+5	2	-18
	3	+9	2	-4
	6	+9	4	-5
RESTRUCT	3	-8	3	-8
	3	0	3	+8
	6	-8	6	0
CLUSTER	3	-12	3	-11
	3	-13	3	-11
	6	-23	6	-11

Table 4.5
Model Quality: Multiple-Execution DNF-PAM Modeling

The anomalous behavior that RESTRUCT experienced when a single execution was modeled did not appear in the modeling of multiple executions. This result lends support to the use of multiple subject executions instead of a single subject execution.

Certain similarities between modeling single and multiple executions can be seen. CLUSTER continues to be difficult to model, especially for the shorter string set, Y1, having a best-case accuracy of -11%. Modeling RESTRUCT for multiple executions is about same as for single executions. For GENREF, the influence of restructurability on model accuracy is most evident.

4.4.2 GPAM

The relationship, seen in DNF-PAM modeling, between model accuracy and the restructurability of the subject execution was also seen in GPAM modeling. In Table 4.6 we see that for GENREF, YT was more difficult than Y1 to model, whereas for RESTRUCT and CLUSTER there was a smaller difference in the difficulty of modeling Y1 and YT. Compared to DNF-PAM, GPAM was slightly less effective in modeling GENREF, but was much more effective in modeling CLUSTER. For CLUSTER, at least, GPAM was able to capture and reproduce more of the restructurability signature of the subject executions than could DNF-PAM -- GPAM was 12% better than DNF-PAM when statistic R was used.

		Set of Subject Executions			
		Y1		YT	
Subject Program	Stat	#str	Quality	#str	Quality
GENREF	r	3	-7	5	-8
	R	1	+5	1	-14
RESTRUCT	r	3	-8	5	0
	R	2	-8	1	+8
CLUSTER	r	3	-6	5	0
	R	2	0	1	-1

Table 4.6
Model Quality: Multiple-Execution GPAM Modeling

Model quality seems to be influenced by the size of the subject execution set, or its length, or both. The quality of model instances based on short executions seems to be improved when the maximum statistic (R) is used, as if statistic R compensates for short executions. As the number of subject executions increases, the mean (r) becomes the statistic of choice.

4.4.3 Conclusions

For small samples, DNF-PAM showed potential for modeling as accurately as GPAM. This is explained in part by the notion of dominant execution: in a small set of executions, the characterization of the entire set may be very close to the characterization of one of the executions, the dominant one. For example, in Table 4.7, we see that using the maximum loop-repetition statistic, which increases the likelihood of having a dominant execution, was more effective than using the mean loop-repetition statistic. (Loop parameter statistic is not a factor in DNF-PAM modeling, since control structures are not modeled by DNF-PAM.)

			G P A M Loop Parameter Statistic	
Subject Program	Rank	DNF	Mean	Max
GENREF	5	+9	-7	+5
RESTRUCT	1	+3	-8	-8
CLUSTER	1	-12	-6	0

Table 4.7
Y1 Model Quality: DNF-PAM versus GPAM
(Best results used)

When modeling was based on a larger set of executions, differences in the model versions and in the subject programs were more apparent. GENREF was the most difficult to model; RESTRUCT was the easiest. Compared to DNF-PAM, GPAM was 10% more accurate for CLUSTER, 8% more accurate for RESTRUCT, and 5-10% less accurate for GENREF. (See Table 4.8.) As the number of subject executions increases, GPAM seems to be better able to absorb and reproduce the increased amount of information.

Of the sets, Y1 and YT, of subject executions, YT is more representative of multiple-execution sampling, in terms of sample size and restructurability (restructuring should be based on executions that yield good layouts). We see that for CLUSTER, DNF-PAM could get no closer to YT than 10-15%, whereas there was a GPAM model instance that could reproduce YT.

Subject Program	Rank	DNF	Loop Statistic Mean	Loop Statistic Max
GENREF	1	-4	-8	-14
RESTRUCT	2	0	0	+7
CLUSTER	1	-11	0	-1

Table 4.8
YT Model Quality: DNF-PAM versus GPAM
(Best results)

4.5 SUMMARY

Results presented in this chapter show the effectiveness of modeling using simple model versions and point-estimates of model parameters. In the next chapter, we move towards a more realistic representation of the distribution of model parameters. The progression towards model sophistication is expected to result in improved modeling accuracy.

The results of elementary modeling investigations are presented in Table 4.9. Since the term accuracy relates to the over-all quality of a range of model instances, perhaps we should consider the worst, best and average cases. It is safe to say, based on elementary modeling, that GPAM and even DNF-PAM are accurate, although there are situations in which they need improvement.

	Single-Execution Modeling			Multiple-Execution Modeling		
		GPAM Stat			GPAM Stat	
Accuracy Level	DNF	Mean	Max	DNF	Mean	Max
Best	-1	0	+3	+9	0	+8
Average	-9	-10	-6	-6	-5	-2
Worst	-24	-22	-22	-23	-8	-14

Table 4.9

Summary of Model Accuracy

4.5.1 Discussion

Over-all performance of DNF-PAM and GPAM. Some synthetic strings were superior to their subject trace strings. In the worst case, synthetic strings were 24% inferior, but in the best case, 9% superior. Surprisingly, DNF-PAM does a good job of approximating execution trace strings.

Effect of subject program. RESTRUCT is by far the easiest subject program to model. CLUSTER was most difficult to model using DNF-PAM, GENREF was most difficult using GPAM. The success with which DNF-PAM and GPAM model RESTRUCT confirms that these model versions work for programs whose locality results from tight looping during which a small number of modules are referenced. The lack of accuracy with which GPAM modeled executions YT of GENREF indicates the need for more accurate modeling techniques. Henceforth, we will no longer investigate RESTRUCT, but will study GENREF and CLUSTER, since they represent more challenging subject program types.

Effect of subject execution set. Modeling difficulty increased with the restructurability of the subject executions. Furthermore, model instances constructed from subject executions having the same level of restructurability had nearly the same level of accuracy. Thus some executions are inherently more difficult to model.

Effect of model version. The simplest model version, DNF-PAM, was fairly accurate, but as the number of subject executions increased, DNF-PAM lost accuracy. GPAM showed the ability to represent and reproduce referencing information contained in multiple executions of the subject program, whereas DNF-PAM, by virtue of its simpler model structure, was unable to capture the significant referencing characteristics of program execution.

Effect of loop statistic used in GPAM modeling. There was no consistent difference in results obtained using loop parameter statistics r and R, except that r worked better for large samples of long executions, whereas R worked better for small samples of short executions. We did not find this result very surprising. Our conjecture is that each loop parameter has a threshold value, below which locality information is suppressed and lost, and above which locality information is distorted. The distortion results in phases (in the synthetic strings) whose lifetimes are extended and whose sizes (in number of different modules referenced) are changed. These distortions in the synthetic string produce a memory-demand signature that is different from that of the subject execution. When this distorted string is used as input to the restructuring phase, the distortion carries over into the restructuring matrix that is produced, and into the layout.

Effect of the number of synthetic strings. Increasing the number of synthetic strings used as input to the restructuring phase did not consistently increase model accuracy. (Our limited computer budget made it infeasible to generate a very large number of synthetic strings.) These findings, although a little disturbing at first glance, give an important insight into the role of selection constructs in GPAM modeling. When point estimates of the underlying model parameter distributions are used, the only variation among the individual strings belonging to the same synthetic string set is due entirely to selection constructs. That little variation is observed suggests that selection constructs do not have a major impact on the locality signature of the subject program, and that to achieve the variation that is observed in actual trace strings requires the introduction of randomness in the values of loop parameters used during synthetic string generation. This is done in the next chapter.

4.5.2 Evaluation of the Hypotheses

Results presented in this chapter generally support the hypotheses of Section 4.2.3, with one notable exception: the quality of the model instance does not improve with the number of synthetic strings generated from that instance. This suggests that the parameter estimation phase is more crucial to model accuracy than is the generation phase. That no hypothesis was overwhelmingly supported by the experimental data suggests that the model is sensitive to a large number of factors, and that as yet, we do not understand the relative importance of these factors.

Chapter 5

ADVANCED MODELING STUDIES

5.1 INTRODUCTION

In the last chapter we saw that modeling based on point-estimation of the distributions of model parameters works. The accuracy of the simplest model version, DNF-PAM, was improved by changing to a more detailed model version, GPAM. In this chapter we continue the quest for more accurate model instances by moving along two different fronts: (1) increasing the number of execution observations used to construct point-estimate model instances, and (2) increasing the amount of information about parameter distributions that is retained in the model instance.

The results of this chapter show that the point-estimation approach does not require a large number of sampled executions--small random samples of 10-20 executions lead to good model instances. Attempts to improve upon the point-estimation approach by increasing parameter distribution information in the model leads to marginal improvements in some instances, and to significant improvement in others. Good point-estimate model instances were not improved upon by increasing parameter distribution information.

5.1.1 Overview

The models in Chapter 4 were deterministic with respect to loops, since point estimates of loop repetition distributions were used. In this chapter we concentrate on large-sample modeling where loops are modeled probabilistically. In the ideal case, the type of probability distribution and a few of its moments are known. In the more likely case, when the probability distribution is not known, or is not the same for all loops in the program, the distribution must be approximated by some "safe" distribution which captures the essence of the observed distribution. As a first approximation, we consider the uniform distribution over some interval because it is the simplest, and leads to the most efficient generator. Given a large number of subject executions, the normal distribution is a reasonable choice, since programs, in the long run, tend to have a "typical" behavior, and only a low percentage of executions differ significantly from this behavior.

The first investigation looks again at point-estimation of the distribution of loop repetitions, this time using the mean (r) as the statistic. From the previous chapter we saw that r was a good statistic for modeling large samples. The purpose of this study is twofold: (1) to study the effect of sample size upon the mean-valued characterization of samples, and (2) to determine whether the choice of statistic estimator (r^1 or r^2) affects the mean-valued characterization. We want to determine the lower bound on sample size in the event that mean-valued (MV) modeling is used.

The second investigation focuses on the distribution of loop repetitions to see if there is a standard probability distribution that describes loop repetitions, given the parameters for an arbitrary program loop, such as mean and variance. If one is found, it should describe a majority of the program loops, to make parameter estimation and string generation as streamlined and efficient as possible.

Finally, we construct model instances that use more information about the distribution of loop parameters. Three approaches are used: (1) approximate the distribution by means, relying on large sample size to give accurate values for the means; (2) approximate the distribution by uniformly sampling from some interval that covers a major portion of the observed distribution; (3) Use a standard distribution function, where the parameters have been determined from a large sample of program executions.

5.1.2 Hypotheses

- A small random sample provides adequate information for effective modeling.

- Model quality improves with an increase in the amount of parameter distribution information contained in the model instance.

5.2 THE GPAM LOOP PARAMETER DISTRIBUTION STUDY

In the first part of this study, we investigate the use of the mean (statistic r) to characterize sampled executions. In particular, we compare the characterizations derived from the two estimators for the mean, r^1 and r^2, which represent different approaches to computing model parameters from multiple executions. In the second part of the study, we study the underlying loop repetition distributions in order to identify the probability distribution function to be used in generating loop repetitions during synthetic string generation.

5.2.1 Introduction

We are interested in answering the following questions.

(1) How well does a sample of a given size reflect, in terms of mean loop repetition frequencies, the population of all executions?

(2) Does the choice of statistic estimator, r^1 or r^2, affect the mean-valued characterization of a sample?

(3) What is the typical underlying distribution of loop repetition frequencies for a given loop?

Each execution of the subject program produces an execution coefficient record that is stored in the Coefficient Database (CDB). Each CDB record contributes up to one value for the characterizing statistic, r. For subject programs GENREF and CLUSTER, the first eighty CDB records were chosen to represent the finite population (POP) of all subject executions. The testbed of trace strings introduced in the previous chapter was used as a stratified sample from POP.

Random samples S10, S20, S30 and S50, of sizes ten, twenty, thirty and fifty, respectively, were selected from POP, for each of which mean loop repetitions were computed using estimators r^1 and r^2.

For selected loops, histograms were constructed to display the distribution of values, and were used to identify the type of underlying distribution. No analytic fitting of the distribution was attempted, nor was a goodness-of-fit test used, because histograms constructed during preliminary stages showed no particular pattern of distribution from which could be formulated a single hypothesis of distribution type.

5.2.2 Mean-Value Characterization of Samples

5.2.2.1 Comparison of means across samples

We computed r^1 and r^2 means for samples TBED, S10-S50 and POP. Sample means for S10-S50 were compared to the population means, and the coefficient of variation among the S10-S50 sample means was computed. Tables 5.1- 5.2 show the r^1 and r^2 means for subject programs GENREF and CLUSTER. In the tables, the columns labeled "MEAN" and "CVAR" refer to statistics computed from the S10-S50 sample means. The coefficient of variation measures the variability in means across samples.

For subject program GENREF, the means of the S10-S50 sample means (computed using either r^1 or r^2) differ from their respective population means by at most one repetition. (Loop L13 of program GENREF is the only exception to this observation.) The same holds true for all loops of program CLUSTER, except for loops L3 and L5. This suggests that accurate means can be computed using small samples.

		Means for Sample							
Loop	Est	TBED	S10	S20	S30	S50	POP	MEAN	CVAR
L1	r^1	1	3	4	3	3	3	3	0.07
	r^2	1	3	4	3	3	3	3	0.07
L2	r^1	2	2	2	2	2	2	2	0.00
	r^2	2	2	2	2	2	2	2	0.00
L3	r^1	25	26	25	26	25	25	26	0.01
	r^2	25	26	26	26	25	26	26	0.01
L4	r^1	5	6	6	5	6	6	6	0.05
	r^2	5	5	6	5	6	6	6	0.13
L5	r^1	58	47	67	52	61	58	57	0.16
	r^2	58	47	57	52	61	58	57	0.16
L6	r^1	11	9	8	10	9	9	9	0.06
	r^2	11	9	8	10	9	9	9	0.06
L7	r^1	2	3	3	2	3	3	3	0.13
	r^2	1	3	2	2	3	2	3	0.12
L8	r^1	10	10	11	10	10	10	10	0.05
	r^2	10	10	11	10	10	10	10	0.05
L9	r^1	13	13	12	12	13	12	12	0.03
	r^2	12	12	12	11	12	12	12	0.03
L10	r^1	49	50	50	51	50	51	50	0.01
	r^2	49	50	50	51	50	51	50	0.01
L11	r^1	3	3	3	3	3	3	3	0.02
	r^2	2	3	3	3	3	3	3	0.03
L12	r^1	2	2	2	2	2	2	2	0.04
	r^2	1	2	2	2	2	2	2	0.05
L13	r^1	440	3568	5579	9269	8235	6987	6663	0.39
	r^2	295	4172	4645	9378	8008	5601	6551	0.39
L14	r^1	2	2	2	2	2	2	2	0.06
	r^2	2	2	2	2	2	2	2	0.08
L15	r^1	1	1	1	1	1	1	1	0.32
	r^2	1	1	1	1	1	1	1	0.15
L16	r^1	1	1	1	1	1	1	1	0.03
	r^2	1	1	1	1	1	1	1	0.12

<u>Table 5.1</u>

Mean Loop Repetitions for GENREF Samples

There is no appreciable difference in the means computed by r^1 and r^2 for low-repetition loops (those with fewer than ten repetitions), where the maximum difference between r^1 and r^2, across all samples but one, is less than one repetition. For high-repetition loops, differences of up to 35% (r^2 gave the higher value) were observed for program CLUSTER, but no significant differences for GENREF. GENREF has five (out of sixteen) high-repetition loops; CLUSTER has two (out of seven).

		Means for Sample							
Loop	Est	TBED	S10	S20	S30	S50	POP	MEAN	CVAR
L1	r^1	4	4	7	8	6	7	6	0.30
	r^2	4	4	7	8	7	7	6	0.30
L2	r^1	2	1	1	1	2	1	1	0.11
	r^2	2	2	1	1	2	2	2	0.19
L3	r^1	79	58	88	76	76	73	75	0.17
	r^2	79	76	92	87	81	77	84	0.08
L4	r^1	3	1	2	1	1	1	1	0.15
	r^2	3	1	2	1	2	2	2	0.13
L5	r^1	39	27	52	53	47	47	45	0.27
	r^2	34	39	57	60	54	50	52	0.18
L6	r^1	5	6	5	4	5	5	5	0.11
	r^2	4	4	5	4	5	4	5	0.05
L7	r^1	6	6	7	7	7	7	7	0.10
	r^2	7	6	7	7	7	6	7	0.12

Table 5.2
Mean Loop Repetitions for CLUSTER Samples

5.2.2.2 Confidence intervals for the mean of r

The discussion from here on deals with the sample of When the MV model instance is already accurate, up to n values of r, the mean loop repetition frequency, extracted from n executions of the subject program, and computed using r^2. (An execution for which a loop is never executed does not produce an estimate of r.) The mean and standard deviation were computed for each sample, and used to compute 90% confidence intervals for the mean of r. The standard deviation of the (infinite) population was assumed to be that of PCP. Assuming further that each sample was large, confidence intervals were computed using the normal distribution. (The Student's t-test, which requires no assumptions, yielded tighter confidence intervals. At any rate, the intervals we will be using are conservatively wide.)

From Figures 5.1- 5.2 we see that for large samples, confidence intervals narrow, degenerating into point estimates. Even for small samples, the intervals for low-repetition loops narrow to a width less than one repetition, suggesting that these loops can be modeled using means. High-repetition loops that have non-trivial confidence intervals appear to have underlying distributions that cannot be characterized by means alone.

```
Sample
-------
TESTBED               |-------------------------------------------|
S10                       |-------------------------------------|
S20                                   |----------------|
S30                                       |---------|
S50                                     |-------|
POP                                        |----|

                   |-----------------------------------------------|
                   1               2               2               3
                   9               3               7               1

                                  I n t e r v a l
```

Figure 5.1
90% Confidence Intervals for Loop 1 of DUMPCSG

When the t-test is used to compute confidence intervals, the median widths of confidence intervals for sample S20 are 0.73 for CLUSTER and 0.63 for GENREF; for sample S50, respective widths are 0.51 and 0.13. For the high-repetition loops (the one with the highest repetition excluded) of GENREF, the average interval width is 77 for S20, and and 14 for S50. The respective widths for CLUSTER are 265 and 68.

```
Sample
-------
TESTBED                       |---------------|
S10                  |-------------|
S20                                        |-------|
S30                                  |---|
S50                                   |---|
POP                                |-|

                |---------------------------------------|
                4         5         6         7         8

                              I n t e r v a l
```

Figure 5.2
90% Confidence Intervals for Loop 2 of MERGE

We conclude that, when means of the underlying distribution are used, the sample size need not be very large; in fact, samples of size twenty or less are adequate. This indicates that there is a sample size beyond which additional sampling will have little or no effect upon mean-valued loop parameter values. Of course, using the mean as the sole approximation to the distribution for high-repetition loops results in loss of information about the distribution, such as its variability and range.

5.2.3 The Distribution of Loop Repetitions

In the discussion that follows in this section, the random variable is not r (the mean loop repetition frequency), but, let us say, x = loop repetition frequency. Random variable r is estimated from the collection of x values observed during one or more program executions.

Because of the potential for a large volume of data, raw loop repetition frequencies were gathered sparingly. The data gathered were used to determine the shapes of the distributions and to see whether there is a commonality of distribution types for loop repetition frequencies.

We constructed histograms showing the relative frequencies for (raw) loop repetition frequencies for all loops, since the volume of data was small. We have included only a few of the more typical histograms. Histogram shapes fell into four categories.

(1) Decaying. The probabilities decreased (approximately monotonically) with increasing repetition frequency. (See Figure 5.3.)

(2) Bell-shaped. The probabilities increased, then decreased. (See Figure 5.4.)

(3) Hybrid. Probabilities are nearly constant over an interval of small values (i.e., the distribution is uniform over this interval), and beyond this interval, the curve is type-1 or type-2. (See Figure 5.5.)

(4) No particular shape. (See Figure 5.6.)

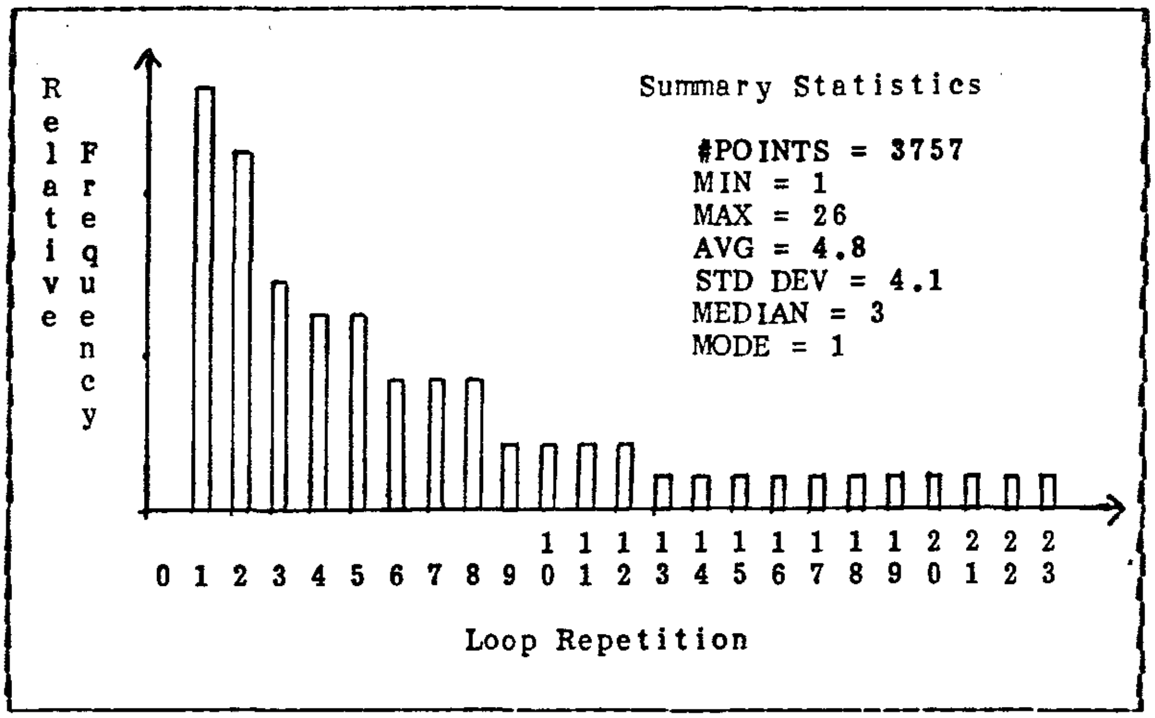

Figure 5.3
Decaying Distribution for Loop 1 of MERGE

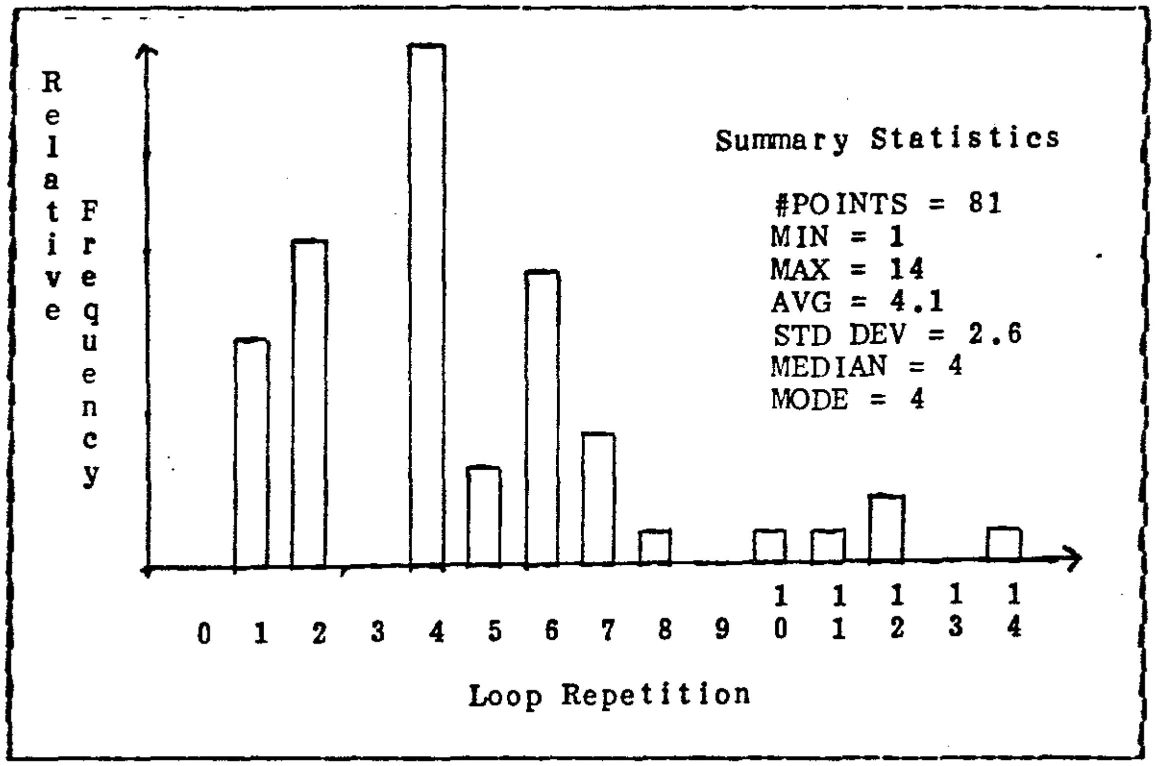

Figure 5.4
Bell-Shaped Distribution for Loop 1 of MERGE

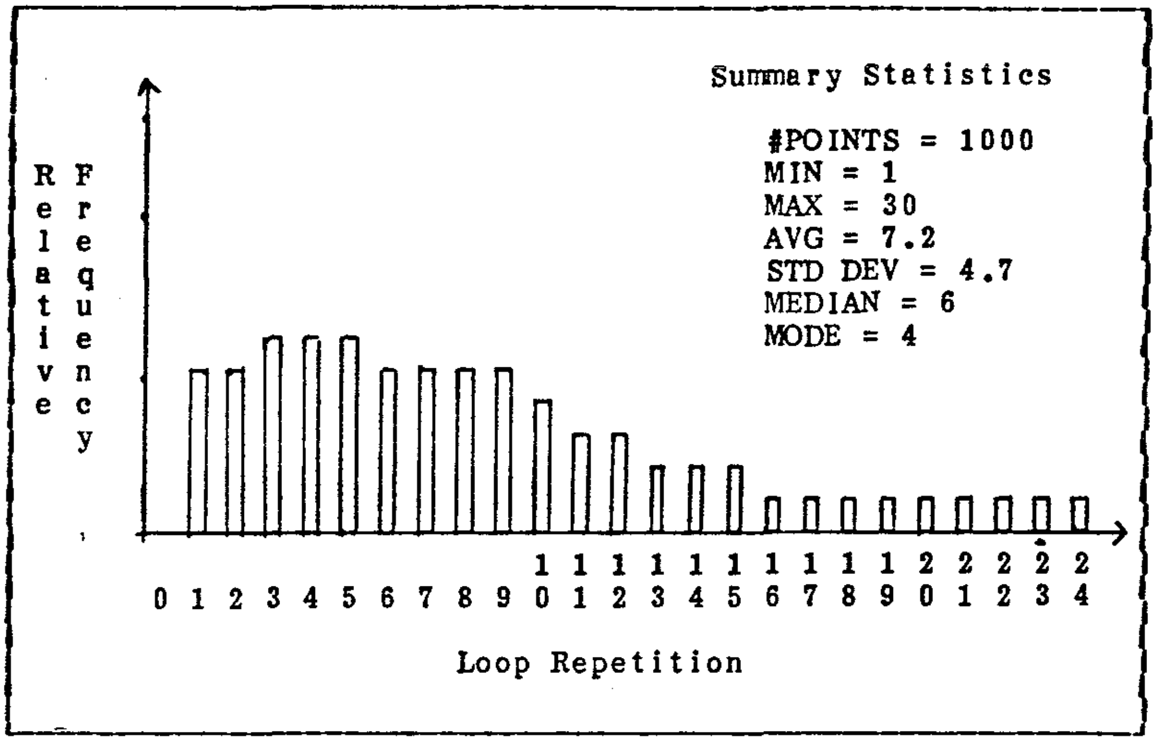

Figure 5.5
Hybrid Distribution for Loop 2 of MERGE

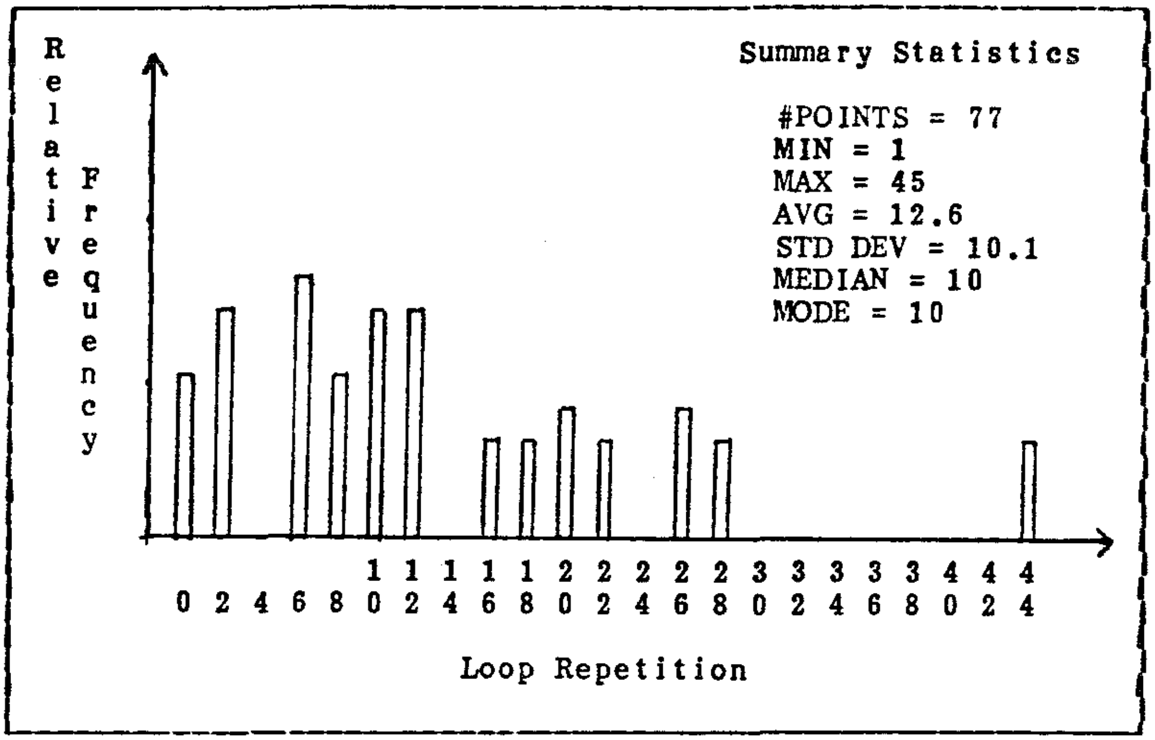

Figure 5.6
Untyped Distribution for Loop 2 of LOADPDN

An interesting finding was that the shape of the distribution depended upon the executions sampled. For example, Figures 5.3 and 5.4 have different shapes, but describe two different sets of observations of the same loop. Notice that, although the shapes differ, the mean repetition frequencies are quite close.

The findings are summarized below.

(1) No single loop repetition distribution describes all loops.

(2) For a given loop construct, the type of distribution may vary from sample to sample.

(3) Although the type of the distribution may vary from sample to sample, for a given loop, the means of the distribution appear to be rather stable from sample to sample.

5.2.4 Conclusions

We have shown that the means of the loop repetition frequencies are quite stable across random samples of different sizes, even though the means may contain very little information about the underlying distribution. Across samples, variation in the sample means is so low that when the sample means are used as point-estimates of the underlying distributions, the differences in the resulting model instances (one per sample) are slight.

Finally, we saw that there is no general distribution function that generates repetition frequencies for all loops. This strongly suggests that one must resort to approximating the distribution by some interval that preserves the mean, yet contains a major portion of the distribution of repetition frequencies.

5.3 GRAM MODELING REVISITED

During synthetic string generation, some approximation to the underlying loop repetition distribution is required. Results from the previous section suggest that the uniform distribution, defined over an appropriate interval of repetitions, may be the only viable representation of the distribution, since the type of distribution varies from loop to loop, and from sample to sample.

In this section we investigate three modeling approaches that use different representations of the distribution of loop repetitions. Approaches (2) and (3) are intended to produce better model instances than approach (1).

(1) Mean-valued (MV) modeling, where the distribution is approximated by the sample mean.

(2) Interval-based modeling, where the distribution is approximated by an interval from which repetitions are sampled uniformly.

  a) Average-Maximum (AM) interval, defined by [mean repetition,maximum repetition].

  b) Extended interval, defined by [r-kd,r+kd], where d is the sample standard deviation, and k is either one (E1) or two (E2).

(3) Distribution-based modeling, where a standard probability distribution is used.

  a) Normal distribution (ND). Negative values are clipped, and replaced replaced by a zero repetition.

  b) Poisson distribution (PD).

In Tables 5.3- 5.5 that follow, model instances are identified by sample identifier and modeling approach. For example, the mean-valued model for sample S20 is written "S20-MV"; its Poisson instance by "S20-PD".

5.3.1 Mean-valued Modeling

We have seen that MV modeling works well, but now we would like to see if its accuracy can be improved by investing more effort in the sampling process. We computed model instances for each sample, TBED, S10-S50 and POP, and compared them to TBED-MV to determine whether increased sample size leads to improved model accuracy.

Percent Improvement over TBED-MV

Model Instance	Subject Program	
	GENREF	CLUSTER
S10-MV	-4	-11
S20-MV	-4	0
S30-MV	0	-37
S50-MV	+2	0
POP-MV	+2	0

Table 5.3
Mean-Valued Modeling: Effect of Sample Size

Table 5.3 shows the percentage improvement of MV model instances over model instance TBED-MV. For GENREF, POP-MV, constructed from the largest sample, was only 2% better than TBED-MV, and the over-all maximum difference in the quality of the model instances was 6%, which is quite low. Three

CLUSTER MV model instances matched TBED-MV, but S10-MV was was 11% inferior, and S30-MV was 37% inferior. (Our immediate reaction to such a poor S30 sample was that it represents the bad "luck of the draw." Indeed, a second random sample of size thirty matched TBED-MV, as did the other samples of size greater than ten. Further examination of S30 revealed that it contained debugging executions that contained calls to debugging routines that were rarely (if ever) called during "production" executions.) We conclude that for MV modeling, there appears to be a sample-size threshold, beyond which further sampling does not lead to improved model accuracy. Furthermore, "chance" sampling notwithstanding, this threshold is between ten and twenty executions.

The similarities in the model instances, and the lack of marked superiority of models constructed from large samples, are explained in part by the statistical characterization of the samples. The sample means used in mean-valued modeling were close numerically, which resulted in model instances that differed in only a few loop parameters. We also found that selection construct parameters for all the model instances were nearly identical across all samples. When executions are characterized using means, there does appear to be a "typical" behavior, which can be surmised from observing a small number of executions.

5.3.2 Interval-Based Modeling

We have seen that when MV modeling is used, there is little need for extensive sampling, since small samples tend to produce the same means as large samples. We also saw that confidence intervals, which capture more of the underlying distribution, degenerate into point estimates when the sample size is large or the confidence level is low. We now use sample statistics to construct intervals that capture some of the range of values in the underlying distributions.

5.3.2.1 Uniform sampling from [mean,maximum]

Estimator R^2 computes the maximum loop repetition random variable R by averaging the maxima from the sampled executions. This produces a smoothed value which negates the effects of very large outliers. Thus the interval $[r, R^2]$ deletes from the distribution very high values and values less than the mean. The rationale for such an interval is the conjecture that high repetitions have a more significant effect upon program locality than low repetitions.

The GENREF model instance S10-AM was 4% better than its MV counterpart, and matched TBED-MV, the best MV model instance. For CLUSTER, on the other hand, S10-AM was 10% worse than S10-MV, and 22% worse than TBED-MV.

GENREF Model Instance	PERCENT IMPROVEMENT over TBED-MV	S10-MV	S20-MV	TBED
TBED-MV	NA	+4	+4	-8
TBED-ND	-7	-2	-2	-15
TBED-PD	-1	+3	+3	-10
S10-MV	-4	NA	0	-13
S10-AM	0	+4	+4	-8
S20-MV	-4	0	NA	-13
S20-E1	+3	+6	+6	-6
S20-E2	+3	+7	+7	-5
S20-ND	-3	+1	+2	-11
S20-PD	-1	+3	+3	-9

<u>Table 5.4</u>

Improvements for GENREF Model Instances

CLUSTER Model Instance	PERCENT IMPROVEMENT over TBED-MV	S10-MV	S30-MV	TBED
TBED-MV	NA	+9	+27	0
TBED-ND	-1	+9	+26	-1
TBED-PD	-1	+9	+26	-1
S10-MV	-11	NA	+19	-11
S10-AM	-22	-10	+11	-22
S30-MV	-37	-24	NA	-37
S30-AM	-23	-11	+10	-23
S30-E1	-50	-36	-10	-50
S30-E2	-79	-61	-30	-79
S30-ND	-1	+10	+27	-1
S30-PD	-1	+9	+26	-1

Table 5.5
Improvements for CLUSTER Model Instances

5.3.2.2 Uniform sampling from [r-kd,r+kd]

The mean-maximum interval captures only the high end of the distribution; all repetitions less than the mean are lost. We would like an interval that represents the distribution of values about the mean, and we would like to form the interval so that the there is an associated measure of the proportion of the underlying distribution that the interval covers. Symmetric extension of the interval about the mean by k standard deviations (d), produces interval [r-kd,r+kd], for which, by the Chebyshev inequality, we know that no more than $1/k^2$ of the distribution falls outside this interval. For k=2, interval [r-2d,r+2d] covers at least 75% of the distribution.

Since negative numbers are not allowed, negative left-endpoints produced by interval extension are replaced by zero. In such cases, the center of the interval is no longer the sample mean, so sampling from this truncated interval will produce values whose mean is higher than the (true) sample mean. In fact, the maximum of the extended interval may be larger than the observed maximum, which may result in synthetic strings that are much longer than the subject string.

From Table 5.4 we see that subject program GENBEF again realized improvements when E1 and E2 intervals were used. Although S20-MV was the worst of the mean-valued model instances, S20-E1 and S20-E2 were 4% better than TBED-MV, and 6% better than S20-MV. The model instances S20-E1 and S20-E2 produce synthetic strings that are within 6% of the subject string set TBED.

Neither CLUSTER model instance, S30-E1 nor S30-E2, was an improvement over S30-MV. In fact, they were 10-30% worse. Could the reason be that S30 was a poor sample, and therefore any attempt to improve a model instance based on it will fail? Close examination of the samples TBED and S30 reveal a few differences, which are related to the function of the CLUSTER program and the nature of its inputs. For TBED, three layouts were produced for each restructuring graph, one for each of three different page sizes; S30 produced only one layout per restructuring graph. S30 processes eight graphs, TBED four, so that S30 executions are longer. S30 also worked on more complex graphs than TBED, requiring more (50 versus 40) list merges during the formation of layouts. Even these differences do not indicate that TBED and S30 are as radically different as their model instances are. The erratic behavior of the CLUSTER AM, E1 and E2 model instances may best be explained as the "luck of the draw."

5.3.3 Distribution-Based Modeling

For large samples, the normal distribution is generally accepted as a good approximation to the underlying distribution, whatever its true form. Since loop repetitions are non-negative integers, the normal distribution must be truncated at the origin, to disallow negative values. This clipping actually skews the normal approximation towards values larger than the mean. What is needed is a discrete, non-negative distribution that has properties similar to the normal distribution. We considered two, the geometric and the Poisson.

The geometric distribution was appealing since its random number generator is efficient, and its parameter p has a natural interpretation relative to loop execution: p is the probability that the loop repeats its body. The drawback is that the distribution is monotone decreasing, and skewed toward values less than the mean. The fact that smaller values have higher probability than larger ones makes the distribution unrealistic for loops with very high repetitions. Furthermore, very few loops in sample POP satisfied the theoretical requirement that the standard deviation and the mean be nearly the same. (Interestingly, the high-repetition loops of CLUSTER come close to this geometric distribution requirement; those of GENREF conform more closely to the Poisson distribution requirement.)

Next, we considered the Poisson distribution, which, for large mean looks like the normal distribution. Its rate parameter represents the number of loop repetitions occurring between loop initiation and termination. The Poisson random number generator is iterative, and requires considerable overhead, relative to uniform and geometric generators.

We investigated the use of the normal and Poisson distributions. Over-all, representing the underlying distribution by a standard function was superior to all other modeling approaches. As usual, there was one exception, S20-E1 and S20-E2 for GENREF. As to which distribution is superior, they seem equally good. The Poisson distribution has the advantage of restricting the range of repetitions generated for loops with high variances, whereas the normal distribution will generate a wider (sometimes unrealistic) range of values. Despite the attractive features of the Poisson distribution, the normal distribution is probably the one of choice, since it is more general and its generator is much more efficient.

5.3.4 Conclusions

The objective of this investigation was to produce a model instance for a sample (e.g., S20) that was better than the MV model instance for that sample. The approach used was to increase the amount of information about the underlying distribution of loop repetitions. We met this objective for both subject programs, GENREF and CLUSTER, but the amount of improvement, and the effort required to obtain that improvement were different. For GENREF, the worst MV model instance differed from the best one by only 4%, and this difference was made up by representing the distribution as the interval between the mean and maximum repetitions observed in the sample. Further improvement was obtained when the interval was extended on both sides of the mean. In the case of program CLUSTER, attempts to construct a model instance for the sample S30, the worst MV instance, were successful only when the normal (or Poisson) distribution was used. The best model instance for that sample was as good as any MV model instance for any sample.

None of the model instances were able to produce synthetic strings that were superior to actual trace strings, although synthetic strings generated from the best CLUSTER model instances reproduced (in terms of restructuring) the strings in the testbed. The best GENREF synthetic strings were within 5% of the testbed strings.

5.4 SUMMARY

We have shown, on the basis of modeling results, that for MV modeling, the observation phase of GPAM modeling need not be very extensive. A small random sample of 10-20 executions provides an adequate characterization of the subject program. MV modeling is susceptible to sampling "chance", which is best minimized by sampling only production-type executions. The hypothesis that a small random sample yields a good model instance is supported by the results of this chapter.

The distributions of loop repetition frequencies tend to be of no predictable type, and the type (form) may change depending upon the sample. Despite these obstacles to modeling, we found that the means of the execution samples' loop repetition frequencies are rather stable, and that using the means to formulate the model instance yields acceptable results.

Increasing the amount of information about the underlying loop repetition distribution improves the quality of a poor MV model instance. Although the extended-interval approach, which uses uniform sampling from an interval defined by loop-repetition statistics, works in some instances, in others it produces poorer model instances. We feel that using a standard distribution (normal or Poisson) is the safer approach, since neither one was shown to result in dramatically worse model instances, as was the case with the extended-interval approach.

When the MV model instance is already accurate, increasing the amount of distribution information does not further improve model accuracy, as was seen with TBED-MV, and with the accurate MV instances constructed from samples of size greater than ten. Even when increased information leads to an improvement, that improvement is at best marginal (10% or less, discounting CLUSTER sample S30). The second hypothesis, that model quality improves with an increase in the amount of parameter distribution information contained in the model instance, is not strongly supported by the results of this chapter.

Chapter 6

A CASE STUDY

6.1 INTRODUCTION

Up to this point, we have conducted an investigation of the parameters of the restructuring process and PAM modeling techniques, when applied to subject programs GENREF, RESTRUCT and CLUSTER. The reader may recall that these programs were written by the author as a part of the modeling-restructuring system used in this research. Under such circumstances, a legitimate concern might be whether these programs have some property that favorably bias the results reported in Chapters 3-5. The case study presented in this chapter applies modeling and restructuring to an arbitrary program.

The fourth subject program, ADDIX, was not written by the author, neither does it solve a problem related to program restructuring or modeling. When it was selected, its referencing properties (and to some extent, its function) were unknown. We found that, even though ADDIX was larger than any of the other subject program, it had very similar static structure, dynamic referencing properties, and restructurability properties. We also found that, for synthetic-string restructuring, simplest PAM model version, DNF-PAM, was superior to all other model versions.

In the next section we present some practical problems associated with an automatic modeling-restructuring system, and propose some solutions that are suggested by the results of Chapters 3-5. In section 6.3 the static, dynamic and restructurability characteristics of subject program ADDIX are presented and compared with those of the other subject programs.

6.2 PRACTICAL ISSUES OF AUTOMATIC RESTRUCTURING

The subject programs GENREF, RESTRUCT and CLUSTER were studied in far more detail than would be practical for the type of "automatic" modeling-restructuring system we propose. In particular, the careful tuning of the parameters of the restructuring process (A, Θ and p) that was done with the subject programs, would not be feasible for an automatic system. The last thing that the automatic system should do is to force the programmer to become a performance analyst. Nonetheless, these parameters must be determined. We now discuss some possible approaches.

6.2.1 Determination of Program Restructurability

There seems to be no predictor of a program's inherent potential for improvement through restructuring. The benefits must be measured by monitoring the restructured program's memory cost for some period of time. Nonetheless, our findings, based on a small sample of programs, along with earlier results [FERR76a,HATF71], suggest that it is safe to assume that a program can be improved by restructuring.

More than likely, the decision to restructure a program will be based on some external factors: program size, running time and frequency of use. For programs with short running times, or for small programs, the savings realizable from restructuring is low.

6.2.2 Setting Restructuring Parameters

Critical-set restructuring algorithms, such as CWS, require a knowledge of A and p, the paging algorithm and page size used by the system on which the restructured program will execute. On a given system A and p will be fixed. The restructuring window, θ, is the one parameter that must be set at the time restructuring is applied. We have observed that bad choices of θ do exist, but that once they have been avoided, there appears to be no significant difference in the layouts obtained.

We have seen that θ is a function of the behavior of the subject program, especially during its dominant phase, which can be related to the static, procedure-level structure of the program. An analysis of the PAM parameters can identify procedures that make a large number of calls each time entered; such procedures either participate in or form the nucleus of phases of execution. The distribution of calls from this ²nucleus² procedure can be used to determine the size of the phase, which gives a reasonable choice for ϵ. That is, ϵ can be chosen from an analysis of model parameters, at the time the model instance is constructed.

6.2.3 Measuring Restructuring Effectiveness

In an automatic modeling-restructuring system, there will be the storage of neither the execution trace strings nor layouts computed from trace strings. Consequently, direct comparison of synthetic-string layouts with trace-string layouts will not be practicable. In such an environment, the operating system should, at minimum, make a post-execution report of memory cost. The effectiveness of the restructuring can be determined from the comparative _before and after_ costs, monitored over a reasonable period of time, or from head-to-head comparisons of the restructured and non-restructured versions of the same program, executing against identical input data.

6.2.4 Selecting the Modeling Approach

Our limited experience gained from the study of the three small subject programs GENREF, RESTRUCT and CLUSTER, shows that synthetic strings are acceptable approximations to trace strings. In fact, we have seen that model version selection, DNF-PAM or GPAM, is an issue which is secondary to that of observing a sufficient number and variety of executions upon which to base a model instance. Moving from DFN-PAM to GPAM does improve accuracy, but requires more expenditure of effort in order to achieve slight improvements. In some instances, the tradeoffs do not favor using GPAM over DNF-PAM.

We do not have sufficient experience to be able to predict the relative quality of DNF-PAM and GPAM model instances, nor can we characterize when one model version is preferable. We have found that model instances, whatever version is used, should be based on multiple executions of the subject program.

6.3 CHARACTERISTICS OF SUBJECT PROGRAM ADDIX

6.3.1 Static characteristics

The subject program ADDIX was written in well-structured PL/I and is maintained by the University of North Carolina Computation Center. ADDIX automatically scans a Script text file for terms that are to be included in an index. When a match occurs, Script ².ix² commands are inserted at the appropriate places in the text file.

ADDIX has two major logical phases. The first phase occurs during the construction of the finite-state machine used to match index terms. Its duration depends upon the number of index terms. The second and dominant phase involves scanning the text file and processing matches. The character of this phase depends upon the processing mode (batch or interactive, under TSO), the size of the text file, whether or not the text file is line numbered and the number of matches found.

There are a few noteworthy differences among the subject programs. Table 6.1 compares ADDIX to an equally-weighted, composite profile of the other subject programs, GENREF, RESTRUCT and CLUSTER. ADDIX is the largest program, has the most complex call graph (average procedure nesting level of 3.5 and share-index of 1.6), but the lowest average construct nesting. RESTRUCT has the highest average construct nesting. Programs GENREF and CLUSTER are quite similar, but have no distinguishing static characteristic.

	SUBJECT PROGRAM	
	ADDIX	The Others
#Statements	1561	667
#Procedures	33	26
#Caller procedures	12	10
#Logical phases	2	3
Share-index	1.6	1.3
Avg procedure nesting level	3.5	2.3
Max procedure nesting level	6	4
#Loops	11	10
Avg loop nesting level	0.3	0.4
Max loop nesting level	2	2
#Selection constructs	31	14
Avg selection nesting level	0.3	0.4
Max selection nesting level	2	2
#Constructs	42	24
Avg construct nesting level	0.8	1.1
Max construct nesting level	4	4

<u>Table 6.1</u>

Comparative Static Characteristics of ADDIX

The procedure call-tree for ADDIX is shown in Figure 6.1. In the figure, ²AD#*² and ²LIMIT*² indicate that the subtrees emanating from AD# and LIMIT have not been expanded (because of lack of space).

```
                                   r---EXTRACT=+r--CONVTCT
                                   |           L---ERRMSG
                                   |
                                   |--FCHART
                   r--[BUILD]=+
                   |              |--FFAIL
                   |              |
                   |              L---FGOTO
                   |
                   |              r--ERRMSG
                   |--CHKFILS=+--FILEORG
                   |              |
                   |              L--STATDMP
                   |
                   |--DATE9
                   |
                   |--IFTSO
                   |              r--ERRMSG
                   |              |
                   |              |--KEYWORD====ERRMSG
ADDIX == MAIN =+==PARMPAR=+
                   |              |--PARMDMP
                   |              |
                   |              L--PARZ
                   |                           r--AD#    ======PUTNUM
                   |                           |
                   |              r--GETNUM    |               r--AD#*
                   |              |            |               |
                   |              |            |               |--ERRMSG
                   |              |            |--ADCYCLE=+
                   |              |-[IXWRIT]=+                 |--LIMIT*
                   |              |            |               |
                   |--[PROCESS]=+--MATCH       |               |--STIKUM
                   |              |            |               |
                   |              |            |               |--TGET
                   |              |            |--GETNUM       |
                   |              |            |               L--TPUT
                   |              |--NOBLANK   |
                   |              |            |
                   |              |--NOSCRIP   |               r--ERRMSG
                   |              |            |               |
                   |              |            |               |
                   |              |            |--LIMIT     =+--TGET
                   |              L--PUTNUM    |               |
                   |                           |--PUTNUM       L--TPUT
                   |--STATDMP                  |
                   |                           |--STIKUM
                   |--TERMINI                  |
                   |                           |--TGET
                   L--TIME8                    |
                                               L--TPUT
```

Figure 6.1
Call Tree for Program ADDIX

6.3.2 The Testbed of Execution Traces

The testbed of execution traces for ADDIX consisted of twenty executions, of length in the range 600-25,000 procedure references. Three single strings, TB-1, TB-8 and TB-13, were chosen as representatives of the diverse types of ADDIX executions. TB-1 represents a short, batch-mode execution; TB-8 is a long interactive execution; and TB-13 is a moderate-length batch execution. The following sets of executions were chosen. ye=TB-13, Y1=TB-8+9+11, Y2=TB-16+17+18 and YT=Y1+Y2. The respective lengths are 19K, 25K, 42K and 67K references. Y1 is a mix of batch and interactive executions of moderate length; Y2 contains long batch executions, the predominant way ADDIX is used. To achieve larger subsets, we combined Y1 and Y2 to form YT=Y1+Y2, which represents a cross-section of ADDIX executions. Finally, set TBED contains seventeen executions from the testbed. TBED represents a large random sample of ADDIX executions, and will be used as the best possible sample of ADDIX executions.

6.3.3 Dynamic characteristics

Figures 6.2 and 6.3 display the intrinsic execution-time characteristics of ADDIX across different executions. Figures 6.4 and 6.5 show the locality properties of ADDIX in relation to the other subject programs.

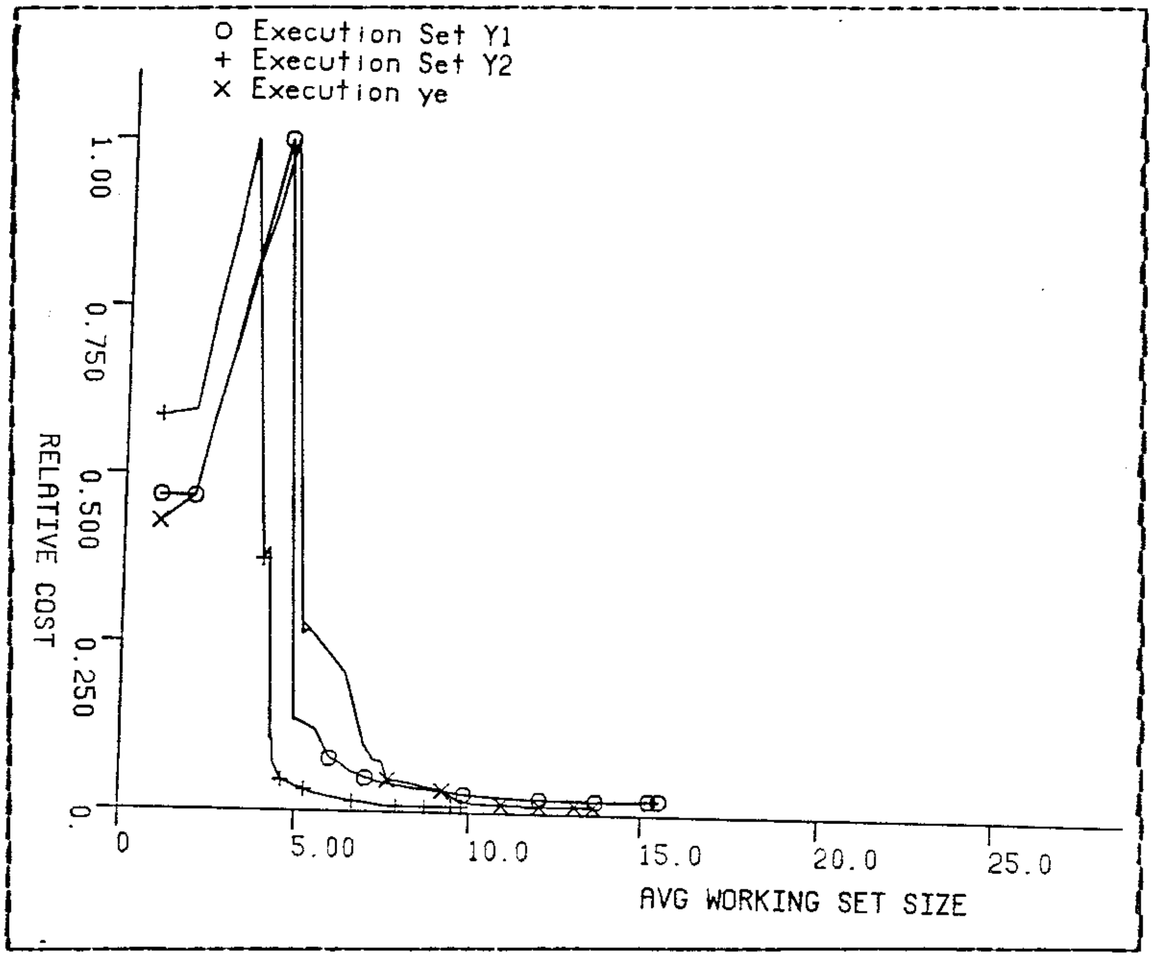

Figure 6.2

ADDIX Intrinsic Memory Cost Curves

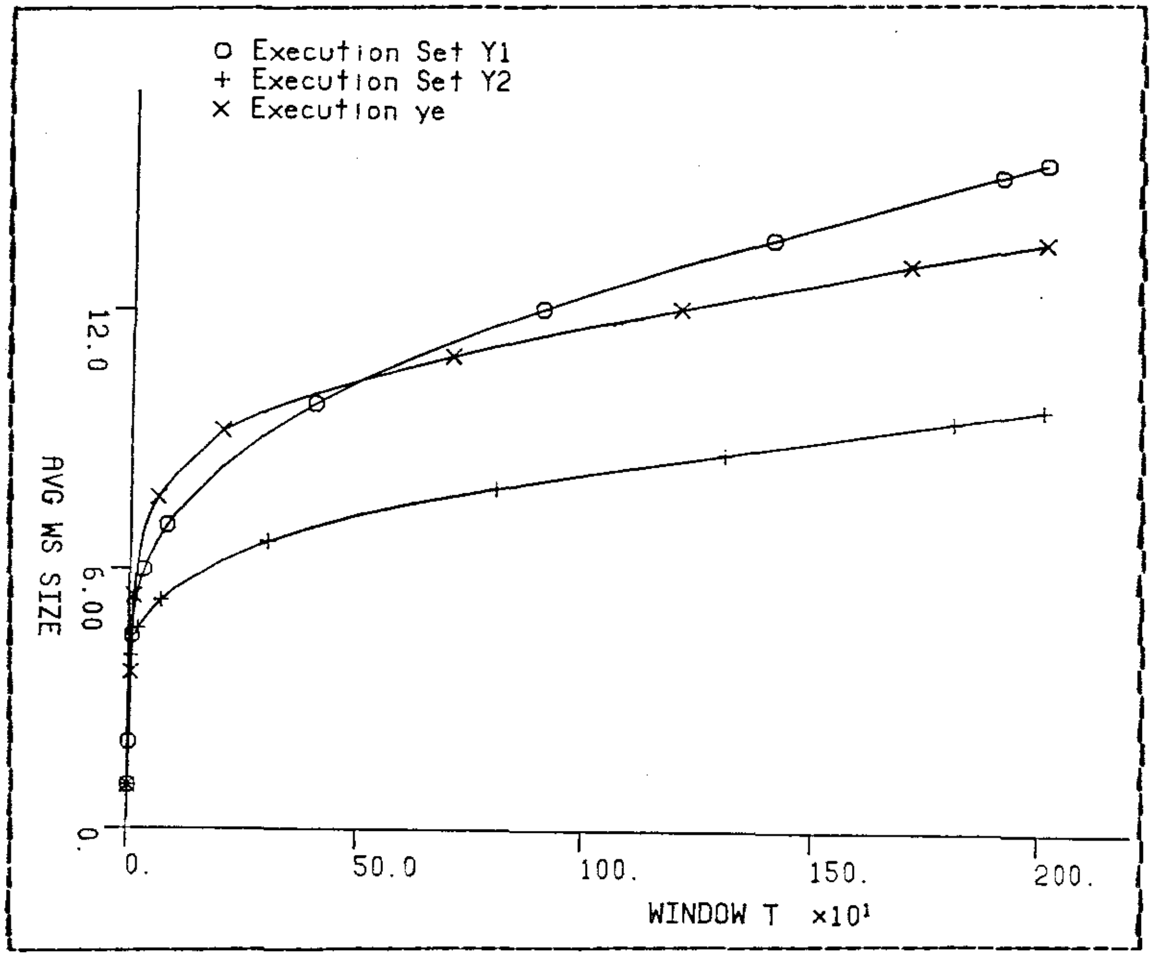

Figure 6.3
ADDIX Intrinsic Working Set Curves

For ADDIX, the drop-off in memory cost occurs when the average working set size is five modules, regardless of the execution set. Y2, the set consisting of the longest executions, has a much greater degree of locality, owing to its very long dominant phases. Not surprisingly, Y1, which has the shortest executions, also has the poorest locality, evidenced by higher memory cost and larger working sets. ADDIX is similar to CLUSTER and RESTRUCT, in that the location of the memory-cost drop-off point is not affected significantly by the choice of execution. In comparing the ye executions of all subject programs, ADDIX appears closer to GENREF in behavior. Both have memory-cost drop-off points near 5 (Figure 6.4), and both have working sets that grow very little for working set windows over 100 references. Of all the programs, the ratio of the average working set size at window 2000 to the number of modules is lowest for GENREF (0.38) and ADDIX(0.43).

Subject programs GENREF and ADDIX illustrate quite well the principle of locality: programs can execute with only a small portion of the program resident. Such behavior is explained by the existence of dominant phases of execution.

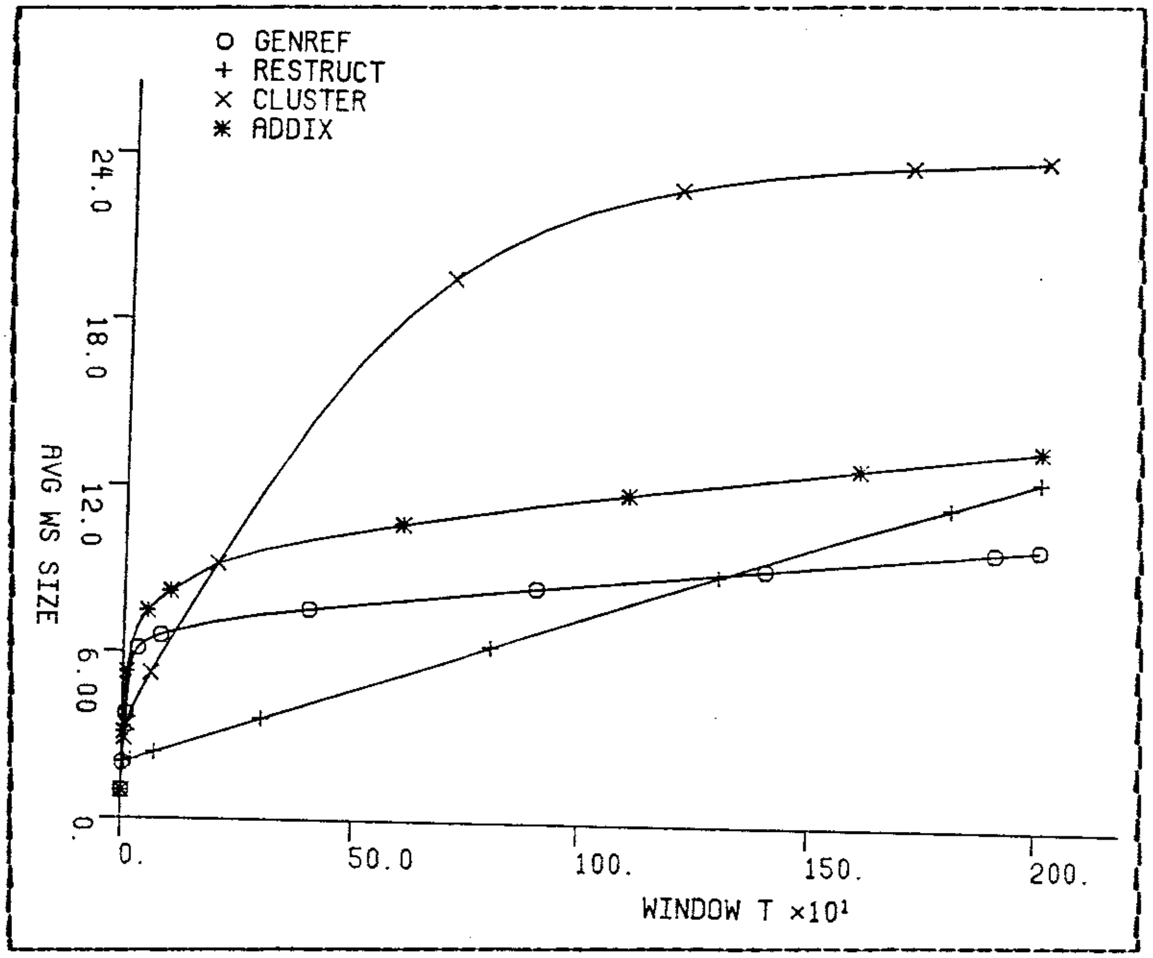

Figure 6.5

Working Set Curves for All Programs

6.3.4 Restructurability Characteristics

ADDIX showed the same potential for improvement through restructuring as the other subject programs did. The standard (STD) layouts were nearly 60% better than the worst layout, and computed (CWS) layouts were 50% better than the best standard layout. (See Table 6.2.) As was the case for the other subject programs, L.TEXTUAL was the best standard layout, which suggests that the order in which modules definitions appear in the source program gives some indication of execution-time locality.

STD Layout	Page Size p1	p2	p3	Avg
ALPHA	32	34	65	44
RANDOM	50	66	68	61
TEXTUAL	53	66	78	66
Avg	45	55	70	57

Table 6.2
Cost Reductions of STD Layouts for ADDIX
(Over L.WORST Layout)

Based on results for the other subject programs, we decided to use the intermediate page size. Our choice of restructuring window θ was based on intuition, on experience with GENREF, the program most like ADDIX in terms of size and phases, and on a trial and error experiment to see which values worked best. We chose θ=5, which, as seen in Table 6.3, leads to the best layout for the intermediate sized page. (Our choice of θ was made before our analysis of the intrinsic cost curve, as discussed in section 6.3.2.)

Computed Layout	Page Size p1	p2	p3	Avg
CWS(Y1,1)	39	67	17	41
CWS(Y1,3)	46	68	71	62
CWS(Y1,5)	54	82	66	67
CWS(Y1,7)	58	81	84	73
CWS(Y1,9)	34	77	48	53
Avg	46	75	57	59

Table 6.3
Cost Reductions of CWS Layouts for ADDIX
(Over L.BEST Layouts)

Before proceeding to modeling, we wanted to verify that the same level of restructuring improvement occurs, even when different trace strings were used as input to the restructuring process. We also wanted to see if improvements were observed over a range of program executions. We found the improvements obtained from restructuring using the intermediate page size and window θ to be stable across other execution strings, and across layouts based on different executions. (See Table 6.4).

	Evaluation String Set			
Layout	Y1	Y2	ye	Avg
L(Y1)	81	67	70	73
L(Y2)	66	66	70	67
L(ye)	60	67	72	66

<u>Table 6.4</u>

Stability of CWS Layouts for ADDIX

(Improvement over L.BEST)

6.4 APPLYING THE MODEL TO ADDIX

6.4.1 Mean-Valued Modeling for Small Samples

The reader should recall that the basic accuracy of a model instance is best measured by its ability to reproduce the restructurability signature of a single subject execution. Since, for a sample of size one, there is not enough information to decide upon a distribution of loop repetitions, the mean-valued approach is reasonable in this case. In Table 6.5 we see that the model is accurate to within 5%, in the worst case. For execution TB-8 (which ranked last in restructurability), the GPAM produced synthetic strings that were 25% better than TB-8. Execution TB-1, which had the best restructurability properties, was modeled to within 5%. We see again the relationship between subject execution restructurability and MV model accuracy that was seen in the other subject programs. DNF-PAM model instances were also accurate.

Subject Trace String	Model Instance	
	DNF-PAM	GPAM
TB-1	-1	-5
TB-8	+3	+26
TB-13	-2	0

Table 6.5
Single-Execution Model Quality
(Improvement over Subject Trace Strings)

We were interested in the effects of modeling based on a small number of carefully selected executions. Table 6.6 shows that for a small number of observed executions, the resulting model instances for ADDIX were excellent for DNF-PAM. The GPAM synthetic strings for Y2 were almost 40% inferior to the subject string set Y2, while the synthetic strings for Y1 reproduced Y1. The major differences between Y1 and Y2 are restructurability (Y2 is slightly better) and lengths of their respective component executions (for Y2 they average 14K references, compared to 8K for Y1). Without a doubt, the inaccuracy of Y2 gives one reason to be wary of modeling based on a small number of executions, regardless of the quality of those executions.

Subject Trace String Set	Model Instance	
	DNF-PAM	GPAM
Y1=TB-8+9+11	+3	0
Y2=TB-16+17+18	+4	-37

Table 6.6

Multiple-Execution Model Quality

(Improvement over the Subject String Set)

6.4.2 Modeling Larger Samples

When large samples of ADDIX executions were modeled, we observed the same pattern of progressive increase in accuracy with an increase in distribution information, with one remarkable exception. In Table 6.7 we see that DNF model instances are superior to all others[1] The very model that contains the least amount of sequence information does the best job of reproducing the restructurability signature of its subject executions. What is more remarkable is that YT-DNF models TBED better than any other TBED model instance, even though TBED is vastly superior to YT. (The best YT model instance, which comes to within 1% of YT, is still 30% worse than TBED.)

Model Instance	PERCENT IMPROVEMENT over YT-MV	YT	TBED-MV	TBED
YT-DNF	+20	+3	+3	-3
YT-MV	NA	-20	-20	-28
YT-ND	-1	-22	-21	-30
YT-PD	-1	-22	-21	-30
TBED-DNF	+19	+3	+3	-3
TBED-MV	+17	-1	NA	-7
TBED-E2	+17	0	+1	-6
TBED-ND	+16	-1	-1	-7
TBED-PD	+16	-1	-1	-7

Table 6.7
Summary of Large-Sample Model Quality

GPAM modeling results for ADDIX show the wisdom of basing the model instance on large samples. Sample TBED contains seventeen strings, YT only six. All GPAM model instances for YT are poor, all those for TBED are good, coming to within 7% of TBED. It is reasonable to assume that YT inherits much of the behavior seen in Y2, which also led to poor model instances.

6.5 CONCLUSIONS

Applying a posteriori program restructuring techniques to reference strings generated from instances of the procedure-activation model works surprisingly well for the program ADDIX. Program ADDIX has many of the same characteristics found the the earlier subject programs, and has similar modelability characteristics as well. The extent to which ADDIX represents the typical program running in a virtual memory environment can be determined only by further investigation.

That DNF-PAM is superior to GPAM is surprising. If this is true for a large class of programs, then the automatic modeling-restructuring system proposed in this research can indeed be implemented at a very low cost.

Chapter 7

CONCLUSIONS AND IDEAS FOR FURTHER RESEARCH

7.1 FINDINGS AND CONCLUSIONS

7.1.1 Review of the Research Environment

7.1.1.1 Model assumptions

The model versions used in the research contained a set of simplifying assumptions that probably are not satisfied by actual programs.

- Call-path independence. This assumption is present at the level of the model version: it is present in DNF-PAM and GPAM, but not in DPAM and AGPAM. Under this assumption, procedures execute in a "memory-less" state, in which the caller can not exert any distinguishing influence on the execution of the called procedure. AGPAM and DPAM contain a memory of the call path (from the driver module), in that each procedure has multiple sets of parameter values -- one per unique call path to that procedure.

- Statistical independence of construct parameters. This assumption is present at the level of parameter representation and synthetic string generation. It requires that all construct parameters be independently distributed, i.e., that there are no correlations among parameters. This assumption, when present, nullifies the effects of control variables defined globally or passed to procedures as parameters. This assumption is probably more unrealistic than the call-path independence assumption.

7.1.1.2 Goals

The goal of the research was to investigate the use of PAM as an integral part of a low-cost, automatic program restructuring system.

- Programmer-free automatic restructuring system. Such a system requires automation at all phases: model construction, parameter estimation, synthetic string generation and restructuring.

- Model accuracy. The automatic modeling system should produce good layouts that are competitive in performance with layouts obtained through standard restructuring methods.

- Low cost. It was crucial that extensive parameter estimation and generation not be required to achieve model instances of good quality, since these phases of modeling were far more expensive than model construction.

7.1.1.3 The experimental approach

Four PL/I programs were modeled. We investigated the differences in model accuracy as a function of subject program, model version, parameter representation (statistic and detail about the underlying parameter distributions) and parameter-estimation approach. The general modeling approach is outlined below.

1. Model version selection.
2. Model construction and program instrumentation by the compiler.
3. Parameter estimation: selection of executions to observe, program execution and computation of parameters.
4. Synthetic string generation from model instance.
5. Program restructuring using synthetic strings.
6. Model validation by layout comparison.

7.1.2 Automatic Model Construction

We showed that all model versions can be constructed by a modified compiler requiring very little additional complexity or execution resource. We further showed that execution of the subject program to estimate model parameters increases the program's execution time by a negligible amount. Algorithms for model construction and subject program instrumentation were given only for model version GPAM.

7.1.3 Major Results

- Model-based restructuring works. In Chapters 4-6 we saw that synthetic-string layouts had memory costs 5-20% higher than trace-string layouts. In other words, 80-95% of the maximum performance improvement achievable through restructuring was realized when synthetic strings were used.

- Mean-Valued model versions are adequate. Model instances based in which the distributions of loop parameters are approximated by the mean repetition frequencies are accurate, especially when the means are estimated from data collected from a random sample of 10-20 program executions.

- Extensive sampling is not required for mean-valued modeling. Sampling more than twenty executions does not result in significantly better model instances. Even when improvements are realized, they are marginal (around 5%).

- Complex models are not cost-effective. Increasing the amount of parameter distribution information represented in a model instance does not, in general, result in cost-effective improvements in model accuracy. In some cases only marginal improvements are achieved, in others model accuracy decreases with increased complexity.

- Programs have similar properties. Differences in programs, both in static structure and in dynamic characteristics, are not as great as one might think. Perhaps the unifying thread is the concept of program locality. Even though the differences are not that great, we were not able to develop any approach to characterizing a program's restructurability, or modelability, apart from actual experience with the program. Fortunately, we have growing evidence that programs are both restructurable and modelable, using PAM.

7.1.4 Evaluation of Major Theses

Of the four theses set forth in Section 1.9, three have been supported by the finding of the research. The fourth thesis -- that the cost of using the modeling system does not exceed the cost of restructuring using execution traces -- is not supported by the results to date. The major cost of using the modeling system accrues during the string generation and a posteriori restructuring phases. Even when a small number of synthetic strings is generated, the cost of producing them, plus the cost of model construction and parameter estimation, probably exceed the cost of tracing a small number of program executions. A more efficient generator and the use of modeling shortcuts may lower the cost of using the model, but the total elimination of the need to generate synthetic strings is sure to lower the cost to a fraction of the cost of a posteriori restructuring applied to actual trace strings. Algorithms for computing layouts directly from a model instance are needed.

7.2 RESEARCH NEEDS FOR AUTOMATIC RESTRUCTURING

That PAM works at all demonstrates its potential; that its accuracy is not predictable suggests that more study is required. The case where model accuracy was substantially improved by moving to a more realistic model instance shows that there are situations that require the most general member of the PAM family, together with a careful parameter estimation effort. In other cases the simplest model version is adequate.

Subsequent research in automatic program restructuring using PAM should address the following unresolved areas.

- _Correlations among loop parameters_. What is a procedure for measuring dependencies, and at what point (e.g., correlation coefficient) are the dependencies significant enough to affect model accuracy?

- _Characterization of modelable programs_. How can the modelability of a program be characterized using PAM parameters or program structure? When is the most general model version required? When will the simplest suffice?

- _Automatic termination of string generation_. The basic generation termination procedure outlined in Chapter 2 should be implemented.

7.3 AREAS FOR FURTHER RESEARCH

The studies suggested in the previous section can be thought of as prerequisites for the studies suggested here. These fall into two categories: (1) expansion of the automatic restructuring system and its algorithms in order to enhance modeling effectiveness and efficiency; and (2) application of modeling to problems in other areas of current research, such as program behavior and program testing.

7.3.1 Modeling and Restructuring

- Modeling low-level source languages. Apply the model to assembler language programs, or to non-structured programming languages. Many large programs written in such languages exist, but have never been optimized for execution in a virtual memory environment.

- Implementation of other model versions. Implement the complete modeling system.

- Data referencing. Devise approaches to referencing data. Determine when such modeling is feasible. The approach should group data into large blocks to reduce the size of the model grammar, and, ultimately, to ensure the efficiency of synthetic string generation.

- Selective modeling of critical constructs. Which procedures contain constructs that are crucial to the successful modeling of the subject program? Which constructs are insignificant, and can therefore be ignored in modeling?

- Algorithms based on PAM parameters. Devise non-generative algorithms that use the parameter database. We feel that the elimination of the generation and a posteriori restructuring phases will reduce the cost of using the system by an order of magnitude.

- <u>Use of page size in tailoring algorithms</u>. Current program tailoring algorithms ignore page size until the clustering phase. Define "space-critical" tailoring algorithms that allocate during restructuring a fixed-size region in which the block resident-sets are stored. What are the advantages and disadvantages of this approach?

- <u>Filtered restructuring</u>. Eliminate from the clustering phase module pairs whose affinities fall below some threshold, k. That is, define the restructuring matrix $C' = C - k*C$. It is possible, by proper choice of k and the restructuring window θ, to filter out faults caused by transitions between phases, so the restructuring matrix records only the competition amongst members of the phase.

- <u>Code-duplication clustering</u>. Since code blocks tend to be small, high-demand blocks can be duplicated without significantly increasing the size of the program's virtual name space. One possible method would allow the clustering phase to have a threshold parameter. A high-demand block can be assigned to any natural cluster having an affinity for it exceeding the threshold value. DPAM may suggest where code duplication can be beneficial.

7.3.2 Other Applications

- Other approaches to model validation. In this research, we used program restructuring as the vehicle for model validation. As a validation tool, restructuring is expensive, compared to approaches such as those presented in [SPIR77]. These should be compared with restructuring, to see whether the properties they measure are related to program restructurability, and to the modelability of the subject program.

- Program testing. PAM instrumentation and execution-coefficient extraction provide a measurement tool for program execution testing. Since only significant control structures are instrumented, and since these may be the most crucial parts of a module to test, the volume of data is greatly reduced.

- Automatic coding standards checker. The compiler can be made to monitor program source code to ensure conformity to organizational standards of program structure.

- Locality studies. Batson's bounded locality intervals (BLI) provide a way to decompose an execution into its phases, and to characterize each phase [MADI76,BATS76b,BATS77b]. DPAM appears to suggest a more natural way of expressing locality [JONE80].

7.4 CONCLUSION

This dissertation has shown that program modeling to achieve performance improvement through program restructuring is feasible. There remain interesting questions that can be answered only by further experimentation with a greater number of larger programs than those we studied. It is our hope that this work has brought us a step closer to compiler-assisted "virtual-memory" program optimization.

BIBLIOGRAPHY

ABRA70 Abrams P., "An APL Machine," Stanford Linear Accelerator Center Report SLAC-114 (February 1970).

ABUS79 Abu-Sufah W., Kuck D. and Lawrie D., "Automatic program transformations for virtual memory computers," *AFIPS Conference Proceedings 49* (1979), pp. 969-974.

ACHA78 Achard M.S., Babonneau J.Y., Carpentier M., Morriset G., Mounajjed M.E., "The clustering algorithms in the OPALE restructuring system," in *Performance of Computer Installations*, (D. Ferrari, Editor), North-Holland Publishing Company (1978).

AHOA71 Aho A.V. and Denning P.J., "Principles of optimal page replacement," *Journal of the ACM 18*,1 (January 1971), pp. 80-93.

ALEX75 Alexander W.G. and Wortman D.B., "Static and dynamic characteristics of XPL programs," IEEE *Computer 8*,11 (November 1975), pp. 41-46.

ALJA79 Al-Jarrah, M.M. and Torsun I.S., "An empirical analysis of COBOL programs," *Software-Practice and Experience 9*,5 (May 1979), pp. 341-359.

ALLE80 Allen F.E., private communication (May 1980).

BABO77 Babonneau J.Y., Achard M.S., Morisset G., Mounajjed M.E., "Automatic and general solution to the adaptation of programs in a paging environment," *Proceedings of the 6'th ACM Symposium on Operating Systems Principles* (November 1977), pp. 109-116.

BAER72 Baer J.L. and Caughey R., "Segmentation and optimization of programs from cyclic structure analysis," AFIPS Conference Proceedings 37 (1972 SJCC), pp. 23-36.

BAER76 Baer J.L. and Sager G.R., "Dynamic improvement of locality in virtual memory systems," IEEE Transactions on Software Engineering SE-2,1 (January 1976), pp. 54-62.

BARD73 Bard Y., "Characterization of program paging in a time-sharing environment," IBM Journal of Research and Development 17,5 (September 1973), pp. 387-393.

BARD75a Bard Y., "Performance analysis of virtual memory time-sharing systems," IBM Systems Journal 17,5 (1975), pp. 366-384.

BARD75b Bard Y., "Application of the page survival index (PSI) to virtual memory system performance, IBM Journal of Research and Development 19,3 (May 1975), pp. 212-220.

BARR79 Barrese A.L. and Shapiro S.D., "Structuring programs for efficient operation in virtual memory systems," IEEE Transactions on Software Engineering SE-5,6 (November 1979), pp. 643-652.

BATS70 Batson A.P., Ju S.M. and Wood D., "Measurements of segment size," Communications of the ACM 13,3 (March 1970), pp. 155-159.

BATS76a Batson A.P., "Program behavior at the symbolic level," IEEE Computer 9,11 (November 1976), pp. 21-26.

BATS76b Batson A.P. and Madison A.W., "Measurements of major locality phases in symbolic reference strings," Proceedings of the International Symposium on Computer Performance Modeling, Measurement and Evaluation, Cambridge, Mass. (1976), pp. 75-84.

BATS77a Batson A.P. and Brundage R.E., "Segment sizes and lifetimes in Algol 60 programs," Communications of the ACM 20,1 (January 1977), pp. 36-44.

BATS77b Batson A.P., Blatt D.W.E. and Kearns J.P., "Structure within locality intervals," in Measuring, Modeling and Evaluating Computer Systems, (H. Beilner and E. Gelenbe, Editors), North-Holland Publishing Company (1977).

BELA66 Belady L.A., "A study of replacement algorithms for virtual storage computers," IBM Systems Journal 5,2 (1966), pp. 789-101.

BELA69 Belady L.A. and Kuehner C.J., "Dynamic space sharing in computer systems," Communications of the ACM 12,5 (May 1969), pp. 282-288.

BOGO75 Bogott R.P. and Franklin M.A., "Evaluation of Markov program models in virtual memory systems," Software-Practice and Experience 5,4 (October-December 1975), pp. 334-346.

BRAW68 Brawn B. and Gustavson F.G., "Program behavior in a paging environment," AFIPS Conference Proceedings 33 (1968 FJCC), pp. 1019-1032.

BRAW70 Brawn B., Gustavson F.G. and Mankin E., "Sorting in a paged environment," Communications of the ACM 13,8 (August 1970), pp. 483-494.

BROW79 Brown P.J., "Software methods for virtual storage of executable code," Computer Journal 22,1 (February 1979), pp. 50-52.

BRYA75 Bryant P., "Predicting working set sizes," IBM Journal of Research and Development 19,3, (May 1975), pp. 221-229.

CHEV78 Chevance R.J. and Heidet T., "Static profiles and dynamic behavior of COBOL programs," ACM SIGPLAN Notices 13,4 (April 1978), pp. 44-57.

CHOW74 Chow C.K., "On optimization of storage hierarchies," IBM Journal of Research and Development 18,3 (May 1974), pp. 194-203.

COFF68 Coffman E.G. and Varian L.C., "Further experimental data on the behavior of programs in a paging environment," Communications of the ACM 11,7 (July 1968), pp. 471-474.

COFF73 Coffman E.G. and Denning P.J., Operating System Theory, Prentice-Hall (1973).

COHE74 Cohen J. and Zuckerman C., "Two languages for estimating program efficiency," Communications of the ACM 17,6 (June 1974), pp. 301-308.

COME67 Comeau L.W., "A study of the effect of user program optimization in a paging system," Proceedings of the ACM Symposium on Operating System Principles (October 1967).

COUR76 Courtois P.J., "A decomposable model of program behavior," Acta Informatica 6,3 (1976), pp. 256-275.

DEAR64 Dearnly F.H. and Newell G.B., "Automatic segmentation of programs for two-level store computers," Computer Journal 7,3 (October 1964), pp. 185-187.

DEFR78 De Freitas S.L. and Lavelle P.J., "A method for the time analysis of programs," IBM Systems Journal 17,1 (1978), pp. 26-38.

DENN65 Dennis J.B., "Segmentation and the design of multiprogrammed computer systems," Journal of the ACM 12,4 (October 1965), pp. 589-602.

DENN68a Denning P.J., "Thrashing: its causes and prevention," AFIPS Conference Proceedings 33 (1968 FJCC), pp. 915-922.

DENN68b Denning P.J., "The working set model for program behavior," Communications of the ACM 11,5 (May 1968), pp. 323-333.

DENN70 Denning P.J., "Virtual Memory," Computing Surveys 2,3 (September 1970), pp. 153-189.

DENN72 Denning P.J., "On modeling program behavior," AFIPS Conference Proceedings 40 (1972 SJCC), pp. 937-945.

DENN75 Denning P.J. and Graham G.S., "Multiprogrammed memory management," IEEE Proceedings 63, (June 1975), PP. 924-939.

DENN76 Denning P.J., Kahn K.C., Leroudier J., Potier D. and Suri R., "Optimal multiprogramming," Acta Informatica 7,2 (1976), pp. 197-216.

DENN80 Denning P.J., "Working sets past and present," IEEE Transactions on Software Engineering SE-6,1 (January 1980), pp. 64-84.

DITZ80 Ditzel D.R., "Program measurements on a high-level language computer," IEEE Computer 13,8 (August 1980), pp. 62-72.

ELSH74 Elshoff J.L., "Some programming techniques for processing multi-dimensional matrices in a paging environment," AFIPS Conference Proceedings 43 (1974), pp. 185-192.

ELSH76a Elshoff J.L., "An analysis of some commercial PL/I programs," IEEE <u>Transactions on Software Engineering SE-2</u>,2 (1976), pp. 113-120.

ELSH76b Elshoff J.L., "A numerical profile of commercial PL/I programs," <u>Software-Practice and Experience 6</u>,4 (October-December 1976), pp. 505-525.

FERR73 Ferrari D., "A tool for automatic program restructuring," <u>ACM 1973 National Conference Proceedings</u> (1973), pp. 228-231.

FERR74a Ferrari D., "Improving program locality by strategy-oriented restructuring," <u>IFIP Congress 74 Proceedings</u> (1974), North-Holland Publishing, Amsterdam, pp. 266-270.

FERR74b Ferrari D., "Improving locality by critical working sets," <u>Communications of the ACM 17</u>,11 (November 1974), pp. 614-620.

FERR75 Ferrari D., "Tailoring programs to models of program behavior," <u>IBM Journal of Research and Development 19</u>,3 (May 1975), pp. 244-251.

FERR76a Ferrari D., "The improvement of program behavior," IEEE <u>Computer 9</u>,11 (November 1976), pp. 39-47.

FERR76b Ferrari D. and Lau E., "An experiment in program restructuring for performance enhancement," <u>Proceedings of the 2nd International Conference on Software Engineering</u> (1976), pp. 203-207.

FERR77 Ferrari D. and Kobayashi M., "Program restructuring algorithms for global LRU environments," in <u>International Computing Symposium 1977</u>, (E. Morlet and D. Ribbens, Editors), North-Holland Publishing Company (1977).

FILI77 Filipski A., "Call by restricted memory reference," ACM SIGPLAN Notices 12,10 (October 1977), pp. 75-77.

FINE66 Fine G.H., Jackson C.W and McIssac, P.V., "Dynamic program behavior under paging," Proceedings of the ACM 21st National Conference (1966) pp 223-228.

FRAN74 Franklin M.A. and Gupta R.K., "Computation of page fault probabilities from program transition diagrams," Communications of the ACM 17,4 (April 1974), pp. 186-191.

FREI75 Frieberger W.F., Grenander U. and Sampson P.D., "Patterns in program references," IBM Journal of Research and Development 19,3 (May 1975), pp. 230-243.

GENT77 Gentleman W.M. and Munro J.I., "Designing overlay structures," Software-Practice and Experience 7,4 (July-August 1977), pp. 493-500.

GRAH76 Graham G.S., "A study of program and memory policy behavior," Ph.D. Dissertation, Department of Computer Science, Purdue University, W. Lafayette, Indiana (December 1976).

GRAH77 Graham G.S. and Denning P.J., "On the relative controllability of memory policies," in Proceedings of the International Symposium on Computer Performance Modeling, Measurement and Evaluation, (K.M. Chandy and M. Reiser, Editors), North-Holland Publishing Company (1977).

HATF71 Hatfield D.J. and Gerald J., "Program restructuring for virtual memory," IBM Systems Journal 10,3 (1971), pp. 168-192.

HATF72 Hatfield D.J., "Experiments on page size, program access patterns, and virtual memory performance," IBM Journal of Research and Development 16,1 (January 1972), pp. 58-66.

INNE76 Innes D.R. and Tsur S., "Interval analysis, pagination and program locality," Information Processing Letters 5,4 (October 1976), pp. 91-96.

JOHN75 Johnson J.W., "Program restructuring for virtual memory systems," Project MAC Technical Report TR-148 (March 1975).

JONE80 Jones E.L., "Procedure-level computer program modeling: detecting major phases of execution," Compendium: Region II Student Symposium of the National Technical Association, NASA Langley, Hampton, Virginia (March 1980).

JOSE70 Joseph M., "An analysis of paging and program behavior," Computer Journal 13,1 (February 1970), pp. 48-54.

KERN71 Kernighan B.W., "Optimal sequential partitions of graphs," Journal of the ACM 18,1 (January 1971), pp. 34-40.

KILB62 Kilburn T., Edwards D.B.G., Lanigan M.J. and Sumner F.H., "One-level storage system," IRE Transactions on Electronic Computers EC-11,2 (February 1962), pp. 223-235.

KNUT69 Knuth D.E., "An empirical study of Fortran programs," Software-Practice and Experience 1,2 (April-June 1971), pp. 105-133.

KNUT73 Knuth D.E. and Stevenson F.R., "Optimal measurement points for program frequency counts," BIT 13,3 (1973), pp. 313-322.

KOBA77 Kobayashi M., "Strategy-independent restructuring algorithms," Software-Practice and Experience 7,5 (September-October 1977), pp. 585-594.

KUCK70 Kuck D.J. and Lawrie D.H., "The use and performance of memory hierarchies -- A survey," in Software Engineering Vol. I (J.T. Tou, Editor), Academic Press, New York (1970), pp. 45-78.

KUEH68 Kuehner C.J. and Randall B., "Demand paging in perspective," AFIPS Conference Proceedings 33 (1968), pp. 1011-1017.

LEBO76 Lenfant J. and Burgevin P., "Characteristics and models of program behavior," ACM Annual Conference Proceedings 19 (1976), pp. 344-350.

LOWE70 Lowe T.C., "Automatic segmentation of cyclic program structures based on connectivity and processor timing," Communications of the ACM 13,1 (January 1970), pp. 3-9.

MADI76 Madison A.W. and Batson A.P., "Characteristics of program localities," Communications of the ACM 19,5 (May 1976), pp. 285-294.

MASU74 Masuda T., Shiota H., Noguchi K. and Ohki T., "Optimization of program organization by cluster analysis," 1974 IFIP Congress Proceedings (1974), pp. 261-265.

MASU79 Masuda T., "Methods for the measurement of memory utilization and the improvement of program locality," IEEE Transactions on Software Engineering SE-5,6 (November 1979), pp. 618-631.

MCKE69 McKellar A.C. and Coffman E.G., "Organizing matrices and matrix operations for paged memory systems," Communications of the ACM 12,3 (March 1969), pp. 153-164.

MOLE72 Moler C.B., "Matrix computation with Fortran and paging," Communications of the ACM 15,4 (April 1972), pp.268-270.

MORR72 Morris J., "Demand paging through the use of working sets on the MANIAC II," Communications of the ACM 15,10 (October 1972), pp. 867-872.

MORR73 Morrison J.E., "User program performance in virtual storage systems," IBM Systems Journal 12,3 (1973), pp. 216-237.

OLIV74 Oliver N.A., "Experimental data on page replacement algorithms," AFIPS Conference Proceedings 43 (1974), pp. 179-184.

OPDE74 Opderbeck H., "Performance of page-fault frequency replacement algorithms in a multiprogramming environment," IFIP Congress 74 Proceedings (1974), North-Holland Publishing Company, Amsterdam, pp. 235-241.

RAMA66 Ramamoorthy C.V., "The analytic design of a dynamic look-ahead and program segmenting scheme for multiprogrammed computers," Proceedings ACM 21st National Conference (1966), pp. 229-239.

RAND69 Randell B., "A note on storage fragmentation and program segmentation," Communications of the ACM 12,7 (July 1969), pp. 365-369.

ROBI76 Robinson S.K. and Torsun I.S., "An empirical analysis of FORTRAN programs," Computer Journal 19,1 (January 1976), pp. 56-62.

ROGE75 Rogers J.G., "Structured programming for virtual storage systems," IBM Systems Journal 14,4 (1975), pp. 385-406.

RODR73a Rodriguez-Rosell J. and Dupuy J., "The design, implementation and evaluation of a working set dispatcher," Communications of the ACM 16,4 (April 1973), pp. 247-253.

RODR73b Rodriguez-Rosell J., "Empirical working set behavior," Communications of the ACM 16,9 (September 1973), pp. 556-560.

RUSS69 Russell E.C. and Estrin G., "Measurement based automatic analysis of FORTRAN programs," AFIPS Conference Proceedings 34 (SJCC 1969), pp. 723-732.

RYDE74 Ryder K.D., "Optimizing program placement in virtual systems," IBM Systems Journal 13,4 (1974), pp. 292-306.

SAYR69 Sayre D., "Is automatic folding of programs efficient enough to replace manual?" Communications of the ACM 13,12 (December 1969), pp. 656-660.

SNEE75 Sneeringer C.C., "Models of memory management techniques for time-sharing," Ph.D. Dissertation, University of North Carolina at Chapel Hill, Chapel Hill, N.C. (1975).

SNYD78a Snyder R., "On a priori program restructuring for virtual memor systems," Proceedings 2nd International Symposium on Operating Systems, IRIA, Le Chesney, France (October 1978).

SNYD78b Snyder R., "On the application of a priori knowledge of program structure to the performance of virtual memory computer systems," Ph.D. Dissertation, University of Washington, Seattle, WA (1978).

SPIR72 Spirn J.R. and Denning P.J., "Experiments with program locality," AFIPS Conference Proceedings 41 (FJCC 1972), pp. 611-621.

SPIR76 Spirn J.R., "Distance string models for program behavior," IEEE Computer 9,11 (November 1976), pp. 611-621.

SPIR77 Spirn J.R., Program Behavior: Models and Measurements, Elsevier North-Holland (1977).

TRIV77 Trivedi K.S., "An analysis of prepaging," Report CS-1977-7, Computer Science Dept., Duke University, Durham, NC (August 1977).

TSAC72 Tsao R.F., Comeau L.W. and Margolin B.H., "A multi-factor paging experiment: I. The experiment and the conclusions," in Statistical Computer Performance Evaluation, (W. Freiberger, Editor), Academic Press (1972), pp. 103-134.

URSC75 Urschler G., "Automatic structuring of programs," IBM Journal of Research and Development 19,2 (March 1975), pp. 181-194.

VERH71 Ver Hoef E.W., "Automatic program segmentation based on Boolean connectivity," AFIPS Conference Proceedings 38 (1971 SJCC), pp. 491-495.

WILK73 Wilkes M.V., "The dynamics of paging," Computer Journal 16,1 (February 1973), pp. 4-9.

ACRONYMN GLOSSARY/INDEX

ACS	Active-Construct Stack. See Section 2.2.2, "model construction."
AGPAM	Aliased GPAM model version. See Section 2.6.2.
AM	Modeling approach where loop-parameter distributions are estimated using the interval [mean,maximum] of loop-repetition frequencies. See Section 5.3.
BLI	Bounded Locality-Interval. See Section 1.3.
CDB	Coefficient Database. See Section 2.3.1.
CLRU	Critical LRU restructuring algorithm. See Section 1.7.2.2.
CSG	Call-Sequence Grammar. See Section 2.2.
CWS	Critical WS restructuring algorithm. See Chapter 3.
DNF	Distributive Normal-Form version of PAM. See Section 2.6.
DPAM	Descriptive version of PAM. See Section 2.6.3.
E1, E2	Modeling approach where loop-parameter distributions are estimated using the interval [r-kd,r+kd], for k=1,2. See section 5.3.
GPAM	Generative version of PAM. See Section 2.6.
GS	Generator Stack, with pointer GSP. See Section 2.4.
IOBJ	Instrumented Object Code. See Section 2.2.
IPL	Insertion-Point List. See Section 2.2.2.
LS	Loop Stack, with pointer LSP. See Section 2.4.
L.BEST	The best of the four standard layouts. See Section 3.4.

L.WORST	Standard layout formed by assigning one module per page. See Section 3.4.3 for definition of other standard layouts -- L.ALPHA, L.RANDOM and L.WORST.
MV	Modeling approach where loop-parameter distributions are estimated using mean loop-repetition frequencies. See Chapter 5.
ND	Modeling approach where loop-parameter distributions are assumed to be Normal. See Section 5.3.
OB	Output buffer during CSG construction. See Section 2.2.2.
OBS	General set of execution trace strings. See Section 4.1.5.
PAM	Procedure-Activation Model. See Section 1.8, Chapter 2.
PARM	Set of parameters for model instance. See Section 2.1.
PCSG	Parameterized CSG. See Section 2.4.
PD	Modeling approach where loop-parameter distributions are assumed to be Poisson. See Section 5.3.
PDB	Parameter Database. See Section 2.3.1.
PDDB	Parameter-Descriptor Database. See Section 2.2.
PDT	Production Descriptor Table. See Section 2.4.
POP	The set of all execution coefficients. See Section 5.2.
SDL	Selection-Construct Descriptor List. See Section 2.2.2.
SSPL	A Simple Structured Programming Language. See Section 2.1.1.
ST	Space-Time Product. See Section 1.5.

SYN	Set of synthetic strings generated from model. See Chapter 2.
TBED	The testbed of actual trace strings. See Section 3.3.3.