A STUDY OF INFORMATION FLOW IN MULTIPLE-COMPUTER AND MULTIPLE-CONSOLE DATA PROCESSING SYSTEMS

K. B. Irani
J. W. Boyse
Don M. Coleman
et al
The University of Michigan

FOREWORD

This annual technical report was prepared by Messrs. K. B. Irani, J. W. Boyse, D. M. Coleman, D. L. Hinshaw, G. A. McClain, K. M. Whitney, and A. M. Woolf of The University of Michigan, Systems Engineering Laboratory, Ann Arbor, Michigan, under Contract F30602-69-C-0214, Project 5581. The Rome Air Development Center project engineer was Mr. Rocco F. Iuorno (EMBIH).

The period of time covered in this report is April 1969 to March 1970. Contractor's identification is Annual Report No. 1

Distribution of this document is limited as the report may project the state of the art in multiprogramming computer designs which are of specific interest to USAF.

This technical report has been reviewed and is approved.

Rocco F. Iuorno

Approved: ROCCO F. IUORNO
Information Transfer Sciences Section
Information Processing Branch

[illegible]

Approved: CARLO P. CROCETTI
Chief, Info Prcs Branch
Info Sciences Division

ABSTRACT

This report documents the achievements from April 1969 to March 1970 of continuing research into the development and application of mathematical techniques for the analysis and optimization of multiple-computer, multiple-user systems.

TABLE OF CONTENTS

Section — Page

1. INTRODUCTION 1

1.1 Contract Objective 1
1.2 Contract Requirements 1
1.3 Progress Toward Contract Objectives 3

2. INFORMATION STORAGE AND COMMUNICATION 6

2.1 Design of Message Processing and Communication Systems 6
2.1.1 Terminal Systems 9
2.1.2 Processing Systems 9
2.1.3 File Systems 12
2.1.4 Communication Systems 13
2.1.5 MPC System Problems and Solutions 17
2.2 Storage Hierarchies 20
2.2.1 The Currently Active Part of a Process 22
2.2.2 The Two Level Hierarchy 24
2.2.3 The Hardware and System Software 31
2.2.4 The N-Level Hierarchy 31
2.2.5 Storage Allocation 33
2.2.6 Traffic Flow 34
2.2.7 A List of System Variables and Functions 35

3. MULTIPROGRAMMED SYSTEMS 38

3.1 Optimum Program Pagination 38
3.1.1 Background 38
3.1.1.1 Paging 39
3.1.2 Model 39
3.1.3 Solution Techniques 40
3.1.3.1 Optimal Solutions 40
3.1.3.2 Known Optimal Solutions 41
3.1.3.3 Heuristic Procedures 42
3.2 Optimum Task Scheduling 45
3.2.1 Multiprogramming and Reentrant Procedure 46
3.2.2 Modeling Computer Systems 48
3.2.3 Scheduling Model 51

TABLE OF CONTENTS

Section		Page
4.	CENTRAL PROCESSOR OPTIMIZATION	56
4.1	Data Path Optimization	56
4.1.1	Example	57
4.1.2	Model for the Study	61
4.1.2.1	Architecture Specification	63
4.1.2.2	Hardware Unit and Algorithm Catalogues	64
4.1.2.3	Data Path and Flow Charts	64
4.1.2.4	Criteria for Optimization	65
4.1.3	Status of Research	65
4.2	Micro-program Control	65
4.2.1	What is Micro-programming?	66
4.2.2	Advantages and Disadvantages of Micro-programmed Computer Systems	70
4.2.3	Problem Statement	76
4.2.4	A Mathematical Model of Micro-program Control	78

LIST OF FIGURES

Figure		Page
2-1	Typical MPC System Configuration	7
2-2	Cost/Capacity Curve for Typical Processing System	11
2-3	Leased Private Line Communication Systems	14
2-4	Typical Leased Private Line Costs	15
2-5	Typical Data Set (Modem) Costs	16
2-6	Definition of $W(t, \tau)$	23
2-7	$f(s)$ for Random Page Accesses	26
2-8	Fraction of Accesses Causing Page Faults	28
2-9(a)	Effect of Non-random Page Access on $f(s)$	29
2-9(b)	Effect of Non-random Page Access on $\frac{1}{f(s)}$	30
2-10	A Four Level Hierarchy	32
3-1	System Configuration	50
3-2	Scheduling Model	53
4-1	CPU Data Path I	58
4-2	CPU Data Path II	59
4-3	CPU Data Path III	60
4-4	Block Diagram of Model	62
4-5	Wilkes Microprogram Control	67

1. INTRODUCTION

1.1 Contract Objective

This effort is for applied research in the area of mathematical techniques for analyzing multiple computer, multiple console, real time on-line data processing systems, and for analytical techniques and hypotheses to assist system designers and users in determining the optimum configuration, most complete utilization, and most efficient scheduling of this type of system.

The main objective of this work is to make it possible, through development and application of new mathematical techniques, to more optimally design and control computer systems. A computer system consists of a collection of electronic data processing machines, data transmission channels, and multiple user terminals, organized to efficiently service the computational needs of a geographically or functionally diverse population of users. Such systems permit: remote communication and manipulation of shared data bases; cooperative operations between user and computer (symbiosis); and immediate access to a high-capability facility for problem solving and data manipulation.

1.2 Contract Requirements

1) Fxploration of Queueing Theory to enable the analysis of more general models of computer utilities and their subsystems. Emphasis shall be concentrated on numerical techniques, and

shall include extension of earlier work on quasi-birth-and-death (QBD) models and the Recursive Queue Analyzer.

2) Collection and analysis of statistical data from existing systems to determine the validity of the mathematical models developed, and to isolate problem areas in need of attention. The techniques of computer data collection shall be studied.

3) Application of new mathematical techniques in conjunction with those previously available, to the analysis and optimization of hardware and software configurations of general purpose computers. These techniques shall be applied to typical systems in order to test the analytical methods and provide specific analyses/recommendations concerning the effectiveness of these systems.

4) Continued development of general design guidelines for time-shared computer systems with distributed processing capabilities. Distributed processing has become economically feasible because of rapidly decreasing small computer costs. This task shall extend previous investigations of remote display terminal structures.

5) Continued development of mathematical models for the optimal structuring of communication networks associated with computing systems.

6) Continued development of optimal design of storage systems and data base structures.

7) Continued exploration of Discrete Optimization Theory and Graph Theory in relation to applications concerning the scheduling of programs in multi-processor systems. Also to investigate the use of these theories in relation to problems of program organization.

8) Development of new conclusions and rules which can be used by persons performing initial designs of real-time computer systems having a large number of user consoles. Such rules shall allow system designers to more rapidly choose the type of hardware/software system needed to fit a particular organization or problem.

9) Application of statistical analysis to data collected from various computing systems in order to gain an understanding of user demand structures and their effects on systems performance. A search for other theoretical approaches to the analysis of multiple computer systems shall be pursued.

1.3 Progress Toward Contract Objectives

The following sections detail the progress made during the first year of this contract. Section 2 reports progress in the area of design of storage systems, data base structures, and communication networks. In Section 3 we report research into the application of optimization theory to the problems of program organization and program scheduling in multi-programmed computer systems. Finally, Section 4 reports on efforts to apply mathematical techniques to the analysis and optimization of the hardware configuration of the central processor.

During this past year a report entitled, "Selected Aspects of Large-Scale Computer System Design, No. 1" [1] was published. Work on this report was done under the previous contract. Also papers appeared in the IEEE Computer Groups News [2] and in Proceedings Third Hawaii International Conference on System Sciences [3]. These papers present some results which were originally published as Systems Engineering Laboratory Technical Reports Nos. 34 [4] and 36 [5]. Two other papers based on the results in these reports will soon be presented [6 7].

References for Section 1

1. "Selected Aspects of Large-Scale Computer System Design No. 1 " K. B. Irani and A. W. Naylor (Eds.), Rome Air Development Center, RADC-TR-69-132, May, 1969 (Two Volumes).

2. J. D. Foley, "Evaluation of Small Computers and Display Controls for Computer Graphics," IEEE Computer Group News, Vol. 3, No. 1, January/February 1970.

3. D. R. Doll, "The Optimum Assignment of Subchannel Capacity in Multiplexed Data Communication Networks," Proceedings Third Hawaii International Conference on System Sciences, 1970.

4. J. D. Foley, "Optimum Design of Computer Driven Display Systems," SEL Technical Report No. 34, The University of Michigan, Ann Arbor, March 1969.

5. D. R. Doll, "The Efficient Utilization of Resources in Centralized Computer-Communication Network Design," SEL Technical Report No. 36, The University of Michigan, Ann Arbor, June 1969.

6. J. D. Foley, "Optimum Systems Design of Computer Driven Graphics Terminals," to be presented at Computer Graphics '70, Brunel University, Middlesex, England, 14-16 April 1970.

7. D. R. Doll, "Topology and Transmission Rate Considerations in the Design of Centralized Computer-Communication Networks," to be presented at IEEE International Conference on Communications, San Francisco, June 1970.

2. INFORMATION STORAGE AND COMMUNICATION

In this section we first (2.1) report research into the design of communication nets to link terminals and computational facilities over a wide geographical area. A related problem which is being attacked in this work is that of assignment of data bases to each of the computational facilities. In 2.2 we discuss the design of storage systems as a function of the type of computational facility making use of the storage system.

2.1 Design of Message Processing and Communication Systems

In recent years the use of on-line time-shared, real-time computer systems has increased rapidly. Barkleys Bank LTD is installing a 2500 terminal Burroughs 8500 system to bring all of its 4, 500, 000 customer accounts on line [1]. The American Airlines SABRE system reserves seats on several hundred flights each day from over 1000 agent sets in the United States and abroad [2]. Ninety percent of all transactions must be processed in three seconds. General Electric Corporation operates an on-line nationwide order system processing over 26, 000, 280 character messages per day [3]. Messages from 136 terminals are concentrated at six message switching centers and directed to a computer data base in Atlanta.

Each of these sample systems has a basic configuration like that shown in Figure 2-1. In each system, the user stations are connected by a large telecommunication network to one or more processing facilities. Each processing facility may have its own data files or request the

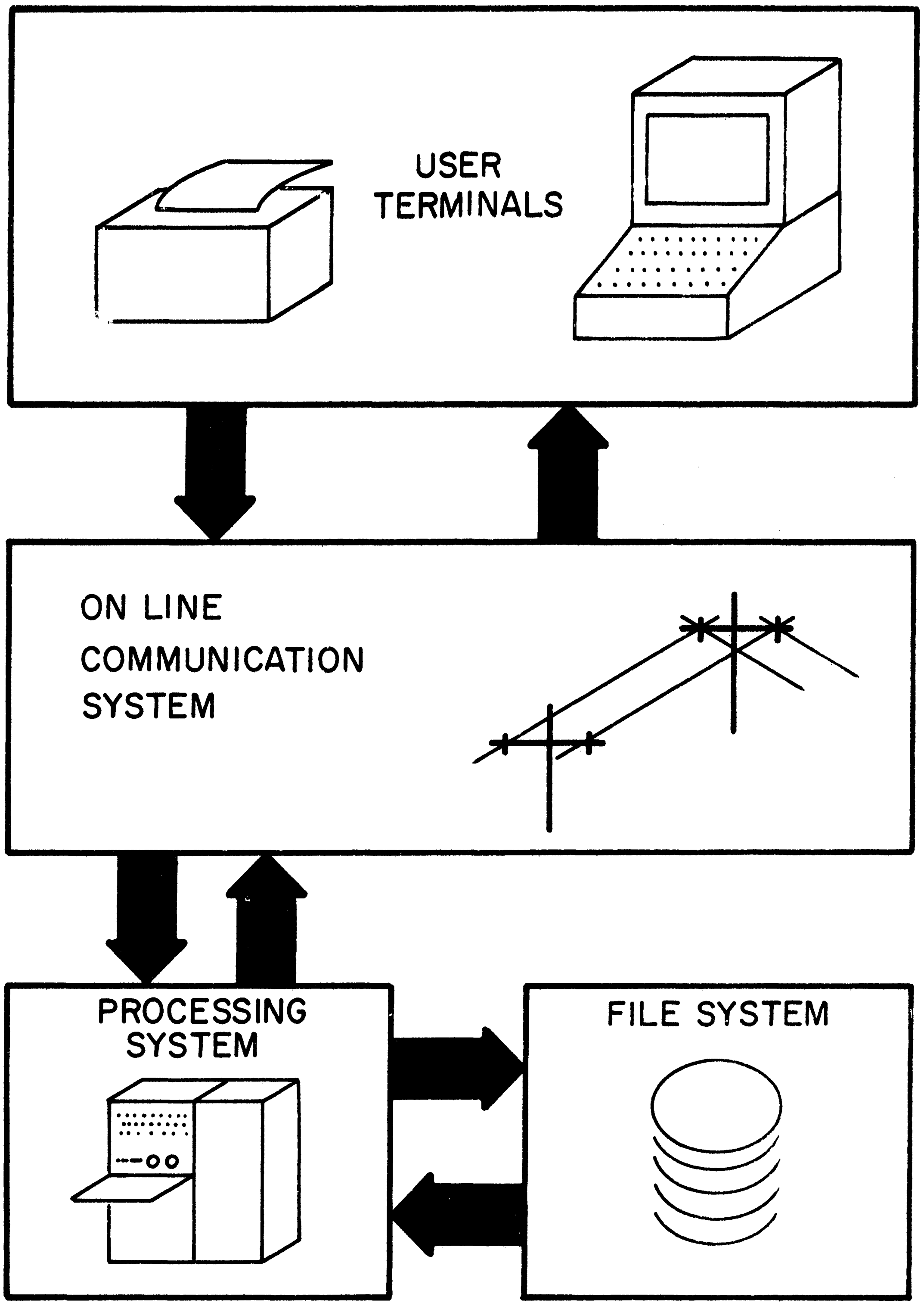


Figure 2-1. Typical MPC System Configuration

necessary data from other processing facilities through the telecommunication network. Certain properties are common to all of these systems:

- Many users at widely separated geographic locations requesting records of a sizeable data base.
- A large data base consisting of files of homogeneous records.
- One or more processing facilities which select, process, and answer the user requests for information.
- An on-line real-time communication system linking the users to the processing facilities.
- Quantitative measures of system performance such as average message delay time, average line utilization, or maximum hourly throughput.
- Performance constraints on the system, such as the requirement that the average message delay not exceed 3 seconds.

The goal of this work is to formulate a realistic comprehensive mathematical model for such message processing and communication (MPC) systems and to study this model seeking methods for the optimal design of these systems. In this report we will discuss in more depth the model of these systems being studied, the design problems which may be formulated and solved with the model, and the nature of the solutions obtained.

A Message Processing and Communication System has four distinct subsystems, a terminal (user) system, a message processing system, a file system, and a communication system. Each of these subsystems will be explained in the following paragraphs.

2.1.1 Terminal Systems

The terminals of a MPC system are not single user stations, but homogeneous sets of such users. A terminal may represent 30 or more individual users. For example, in an on-line banking system, the user stations would be individual teller's cages for a branch office or for all the offices in an entire city. The problem of gathering the users into terminal systems is an interesting and difficult one. In this report, we will not consider that problem, but will assume the terminals and user sets are inputs to the analysis. Normally, a terminal system will include a moderately large group of user stations, all of which request records of the data base files in a similar manner.

A terminal (system) i will be specified by its location (X_i, Y_i) and a measure h_i of the relative cost of locating a processing facility at that terminal. The terminals of a system serve not only as the origins of message traffic, but also as the possible sites for the files of the data base. Hence all messages in the system operation travel between the terminals via the communication system.

2.1.2 Processing Systems

In the message processing and communication systems for which the model of this work is applicable, the processing of each file inquiry is very simple. A request for an airline seat reservation is checked to see if a seat is available, and a simple confirmation or rejection message is returned to the inquiring user station. An order is checked for validity and a simple confirmation is given. Although the bulk of the processing sub-system is a fast access storage system, this does not mean that the cost of the central processors is negligibly smaller than the cost of the drums, discs, and controllers of the file storage. The processor may serve as a terminal's message switching center as well as an access device for the files.

The system cost and performance are affected by the speed of the processing sub-system. As the cost increases, the mean processing time normally decreases. However, since the processing time is usually small compared to message communication time, we will ignore its effect on system performance. The processing system will be specified to have a certain fixed maximum processing time per message. For that particular constraint, a function can be defined giving the cost of a processing system with any desired storage capacity. This function will be assumed to include the cost of the appropriate central processor as well as the storage media. Studies of the design of such storage systems have been made by Woolf (Section 2.2 of this report), Chandy and Ramamoorthy [4], Gecsei and Mattson [5], and others. Hence this topic will not be discussed further.

An example of such a processing system is an IBM 1130 Computer system with 2311 and 2314 disc storages. The monthly rental cost as a function of capacity in bytes is given in Figure 2-2. For capacities of less

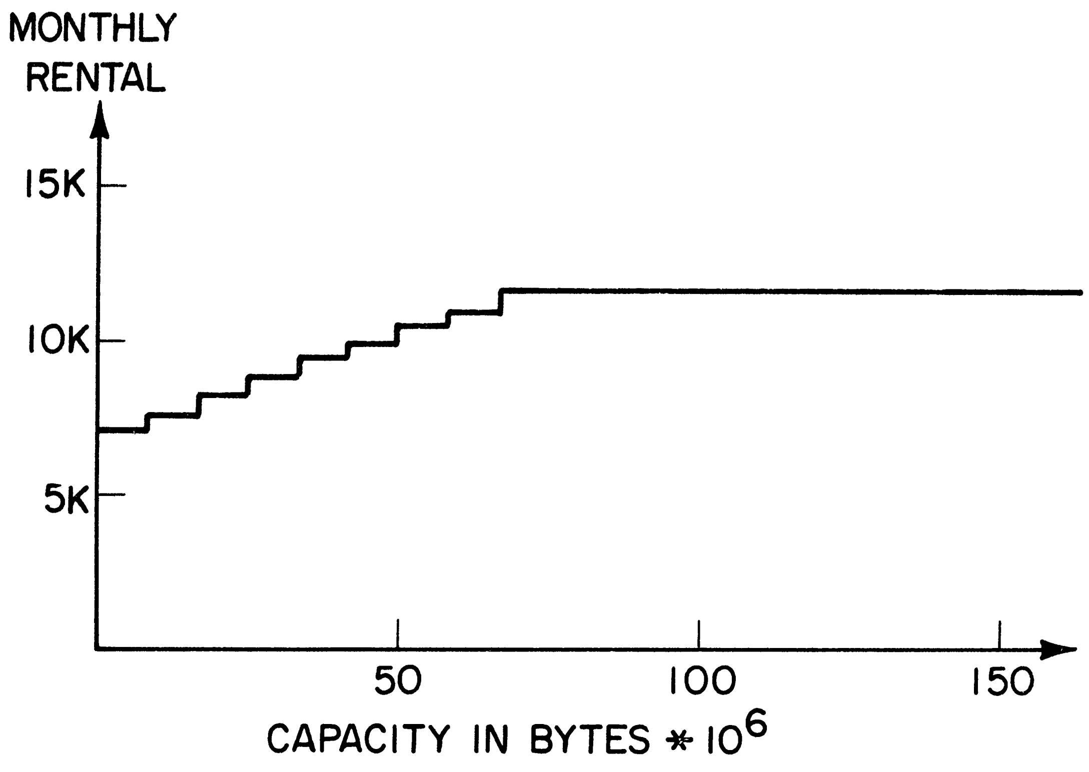


Figure 2-2. Cost/Capacity Curve for Typical Processing System.

than 63 $\times$ 10^6 bytes, 2311 storage units are used; for larger capacities 2314 units are used.

2.1.3 File Systems

A file system or data base is a large collection of records organized in several homogeneous files, together with as much information as possible about the usage of the records in each file.

As an example of a file system, consider the American Airlines SABRE reservation system's data base. A file is the collection of reservation records for a single route. Each record is a single seat on a flight for that route. The data base is the collection of all the flight files. Whenever a ticket agent wants to reserve, release, or inquire about the status of a seat on some flight, it is necessary to fetch a record from the data base, display or alter the desired information, and return a confirmation to the agent.

A file is a collection of records, each of which has the same values of the following basic parameters:

$E(k)$	Size of record in bytes of file k.
$R(k)$	Request rate for records of file k.
$P_i(k)$	Probability that a request for a record of file k came from terminal i.
μ	Mean message length .
$Q(k)$	Length of reply to each request for records of file k.

For the reservation system example, the storage size of a record might be 256 bytes per seat. In the file would be 125 records, one for each seat. The probabilities of origin of record requests would be high for cities along the route and low for others.

2.1.4 Communication Systems

For this research effort a complete survey of currently available communication system types was made. Those systems best adapted to Message Processing and Communication Systems are the leased private line systems. Figure 2-3 is a chart of the basic variables available of this system type. The communication system is divided into system segments which provide message transfer capabilities among groups of terminals. The cost of each segment is composed of two parts. There is a charge for the length of the line of the minimal spanning tree joining all the terminals of that segment and independent of the traffic on the segment. This cost is illustrated in Figure 2-4. There is also a charge for data sets (modems) and multiplexers at each terminal to perform the digital-to-analog-to-digital conversions necessary for message transmission. This charge, a linear function of the capacity (maximum transmission rate) of the modems, is illustrated in Figure 2-5. In general, the cost of the communication system will be represented by an expression of the form:

$$C = \sum_{w \in W} \{CL(\ell_w) + CM(\lambda_w)\}$$

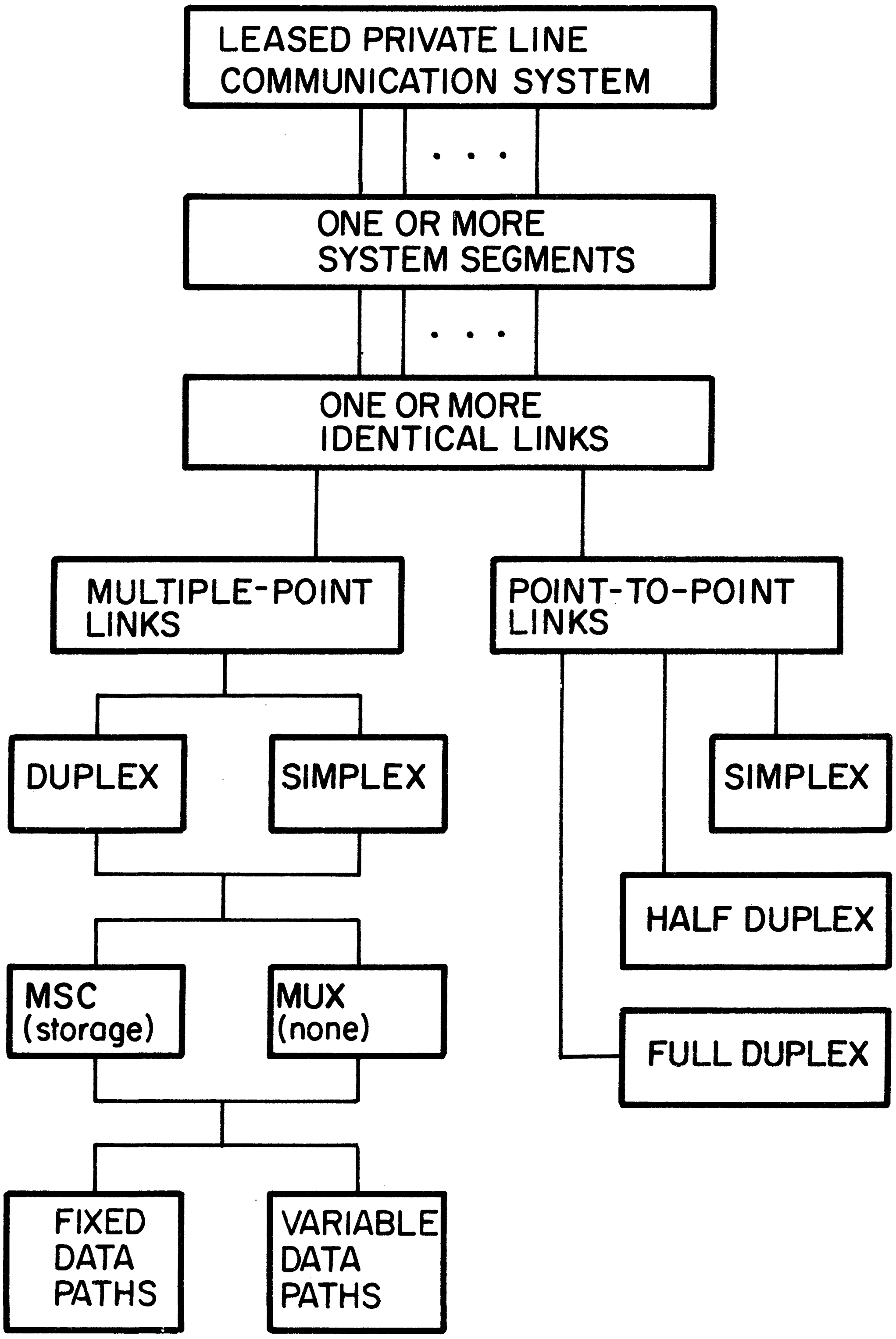


Figure 2-3. Leased Private Line Communication Systems.

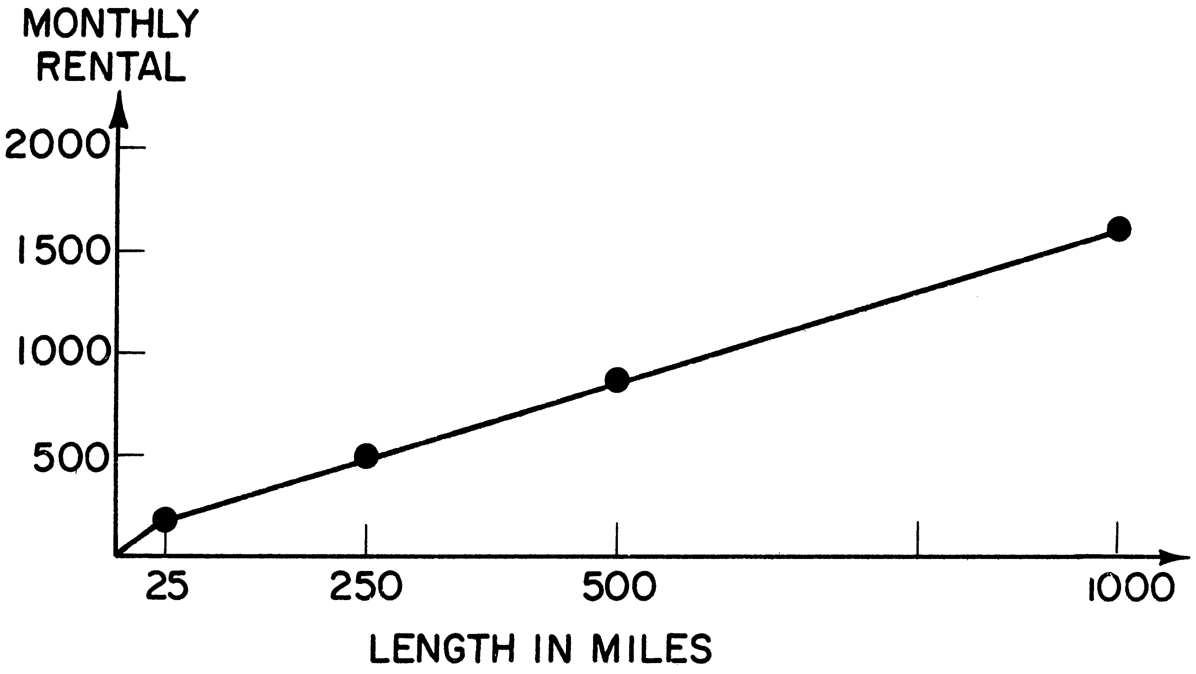


Figure 2-4. Typical Leased Private Line Costs.

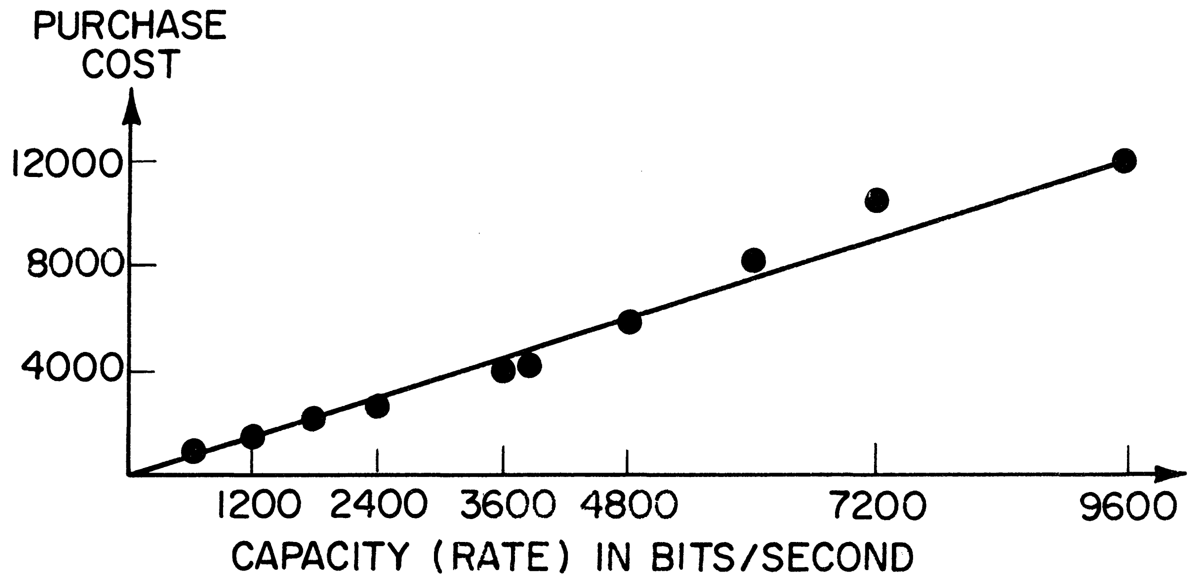


Figure 2-5. Typical Data Set (Modem) Costs.

where CL is the line cost function and CM is the modem cost function. Great effort has been undertaken to design optimally cost-effective communication networks. The problems attacked and some of the solutions will be discussed in the next section.

2.1.5 MPC System Problems and Solutions

In the preceding sections we introduced terminals, files, and communication networks. When the files of the data base are stored at some of the terminals, traffic between source terminals and sink terminals is induced. Location of the files at the "proper" sites will result in minimizing this traffic. To formalize the notion of "proper" location, the following definitions are necessary.

Let K be the set of files, T the set of terminals, $S \subseteq T$ the set of sites where files may be stored. Then an assignment of the data base files is any function with domain K and range S.

$$f: K \to S$$

A file assignment will induce message traffic between the terminals as follows. Let γ_{ij} be the mean number of bits per second of message originating at terminal i toward terminal j. This value includes requests from i for files stored at j as well as replies from files at i to requests from j.

$$\gamma_{ij} = \sum_{k \in f^{-1}(j)} \{R(k)P_i(k)\mu_k\} + \sum_{k \in f^{-1}(i)} \{R(k)P_j(k)Q(k)\}$$

An assignment f also induces storage costs for the files. Let F_i denote the bytes of file storage required at node i by the assignment $f: k \rightarrow S$, i.e.,

$$F_i = \sum_{k \in f^{-1}(i)} \{E(k)\}$$

The variables γ_{ij} and F_i are the basic parameters in our system design study. The main problem attacked in this work is to select that assignment of a data base to a set of terminals which results in minimal system cost:

$$C = CC(\gamma_{11}, \ldots, \gamma_{tt}, T_{res}) + CP(F_1, \ldots, F_t)$$

for a class of communication systems represented by the cost function CC and a class of processing systems represented by the cost function CP.

This general problem is specialized by a choice of the types of communication, terminal, processing, and file systems to be used in the system design. These more specific problems are then solved. In the design of message processing and communication systems with this model three classes of problems are studied.

One major problem area studied is the determination of the optimal number and locations of sites for the system files. Several basic properties of file site assignment on linear graphs are demonstrated and used in efficient procedures for file assignment. When the topology of the communication network is a tree, some special properties of file assignment allow especially efficient algorithms. Furthermore, on tree structured graphs, the optimal assignments of files with respect to linear cost functions, are also optimal with respect to a wide class of other functions.

After the file locations have been specified, the communication requirements of the system are known. A second major problem area may then be considered, the optimal design of communication channels and networks of these channels. A thorough study of stochastic message transmission channels has been undertaken. The effects of channel capacity with one or more channels, message length distributions, message retransmission order, and departures from the model assumptions on message delay are studied. Both analytic and simulation methods are used. Several channel and network performance measures are defined and compared. The problem of optimally allocating channel capacities in the channels of a network is solved for several important system models. A complete set of guidelines is given to aid the designer of communication systems and channels.

The third major problem area studied is the design of communication networks. This includes the layout of the network line topology, the optimal selection of the channel capacities (transmission rates) and the selection of appropriate modems, multiplexers, and other terminal equipment. Throughout the study, a major effort has been made to model realistically communication systems which might actually be constructed rather than to design hypothetical networks which are entirely impractical. Several previously available techniques for the design of centralized networks are carefully evaluated. New techniques for the design of centralized and non-centralized networks are presented. These new procedures have been extensively tested and compared to other design procedures and found to yield solutions of higher performance and greater generality than those procedures heretofore available.

Work is continuing on the improved solution of all these problem types. The solution methods in use include analytic, heuristic, and simulation techniques, whichever are appropriate. In addition to developing improved design procedures for each of these problem areas, research continues into the combination of these partial solutions into complete and comprehensive design system tools for the effective design of entire systems rather than just the subsystems.

2.2 Storage Hierarchies

In general, storage hierarchies are found in almost all computing systems of any size. They are indicated by the presence of more than one media of storage such as thin film, core, disk, data cell or tapes. The particular kind of system which presents the largest problems are those which are automatically managed. This includes such systems as the cache-core relationship in the IBM 360/85, virtual memory systems such as the IBM 360/67 and GE 645, and online file systems ordinarily found in time shared systems and management information systems. It is the design of this general class of automatically managed hierarchies which is the focus of this work.

Every mathematical model of a physical system is a tradeoff between reality and complexity. The goal here as elsewhere is to devise a model which is as useful as possible in one sense or another. In order to achieve a truly useful and realistic model the dominant factors which influence the performance of the system must be represented. Here we have chosen to include:

1. Program Characteristics
    a. Size
    b. Paging characteristics
    c. Number of programs
2. Hardware Characteristics
    a. Access times including for example the seek, latency and transfer time of a disk.
    b. Data paths
    c. Storage capacity
3. System Characteristics
    a. Queueing for channels
    b. Queueing for devices
    c. Queueing for the processor
    d. Choice of routing
    e. Record size

In order to provide a qualitative understanding of the model we must consider each of the various aspects of the model in more detail.

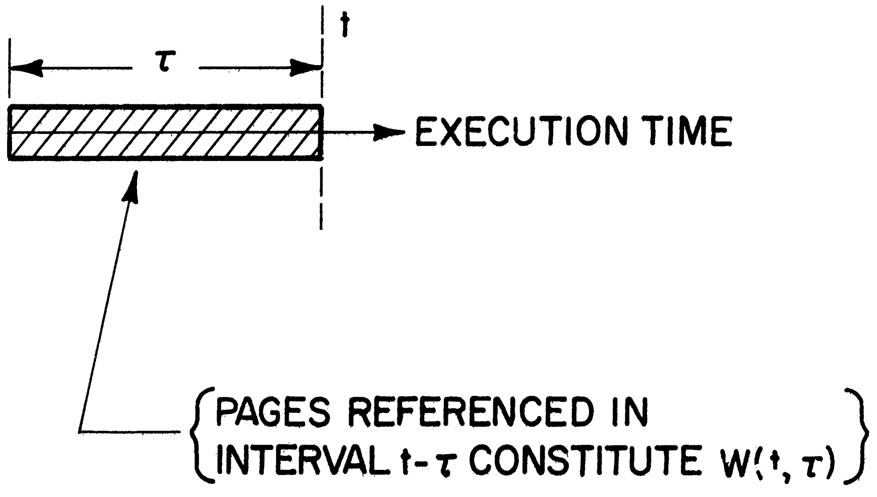
We will begin by considering how programs and program characteristics are modeled. Some of the basic concepts which were used as a foundation for this model were provided by L. A. Belady, C. J. Kuehner, and P. J. Denning. Let us begin by examining some of these concepts.

The lifetime function as such was first proposed by L. A. Belady and C. J. Kuehner [6] in an effort to identify and describe the characteristics of a computer program or process which affect the behavior and efficiency of a paged storage system. Belady and Kuehner were not alone in attempting to model this aspect of a process. Very similar constructs have been considered by Peter J. Denning [7]. These efforts at least in part attempt to determine what part of a progrm or process must be located in core to avoid an excessive number of transfers between core and drum.

2.2.1 The Currently Active Part of a Process

It is generally agreed that most processes have a non-random storage access behavior and that it is meaningful to discuss the currently active part of a process. The concept of the currently active part of a process has been expressed in several different and useful ways. This idea is basic to the question of what part of a process should be in core.

P. J. Denning [7] refers to this currently active part as the working set. Denning defines the working set as "...the set of information $W(t,\tau)$ of a process at time t to be the collection of information referenced by the process during the process at time interval $(t-\tau-t)$." See Figure 2.6. There are many other ways to define a set of currently active pages. For instance we may



τ
t
EXECUTION TIME
PAGES REFERENCED IN INTERVAL t-τ CONSTITUTE W(t, τ)

Figure 2-6. Definition of W(t, τ).

modify the working set by considering t and τ of $W(t,\tau)$ to be real time instead of process time. We might consider a different parameterization such as the set $W'(t, n)$ defined as the n most recently used pages as a function of process time t. This is very similar in concept to the work by L. A. Belady and C. J. Kuehner [6]. Belady and Kuehner define the _locality_ of storage references as a basic program property. "_Locality_ is defined as the total range of storage references during a given execution interval." The remainder of the paper [6] would indicate that Belady's notion of _locality_ is very similar to $W'(t, n)$.

2.2.2 The Two Level Hierarchy

The concept of a lifetime function was developed by Belady and Kuehner [6] in the context of a paging system consisting of a core and a drum. Here we will extend this concept in several ways but first we will consider it in a form close to its original form as developed for two levels of storage.

When a program begins execution following a page fault some of its pages will be in core and others generally will not. The precise information contained in core will determine the number of executive cycles required to generate another page fault. Since we are considering a paged system the precise information contained in core is determined by three factors:

1. The method of selecting pages to be paged in and out.
2. The number of pages remaining in core.
3. The page size.

If we consider 1 and 3 to be fixed we may express the average number of execution cycles required to produce a page fault as a function of 2. Being more precise about units we will define f(s) to be the average number of storage accesses necessary to produce a page fault as a function of s where

$$s = \frac{\text{Number of Pages in Core}}{\text{Total Number of Pages}} \qquad (2.1)$$

Notice that f(s) is a property of a program. This "lifetime function" is very similar to Belady's with some exceptions.

Let us consider some of the properties of this lifetime function. As we have just pointed out if there are no pages in core, $s = 0$, the number of storage accesses required to produce a page fault will be 1 thus $f(0) = 1$. At the other extreme, all pages in core, $s = 1$, the number of accesses required to produce a page fault is without bound thus $f(1) = \infty$.

In order to get a feel for the general shape of f(s) let us consider the simple case of completely random page accesses. In this case f(s) can be shown to be

$$f(s) = \frac{1}{1-s} = 1 + s + s^2 + s^3 + \ldots \qquad (2.2)$$

This function is clearly convex and a curve of this form is shown in Figure 2-7.

Another important measure is simply $\frac{1}{f(s)}$ the fraction of all storage access which causes a page fault. In the random page access case this simply becomes

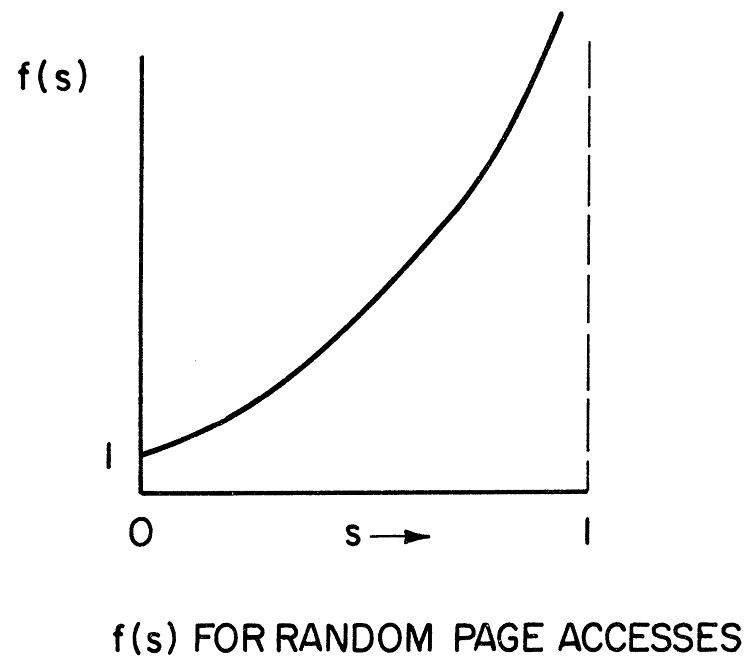


Figure 2-7.

f(s) for Random Page Accesses

$$\frac{1}{f(s)} = 1 - s. \tag{2-3}$$

Graphed, this appears as shown in Figure 2-8 and agrees with intuition.

As discussed earlier such random behavior is rare and in most cases we can rely on some clustering of accesses. This will generally result in curves of the form shown in Figure 2-9, (a) and (b).

Here we have shown the random behavior with the solid line and the clustered accesses with the dashed line.

As is illustrated in the above graphs a clustering of accesses affects f(s) so that for a given s = s',f(s') becomes larger. This simply means that for a given storage allocation the program with clustered accesses will require more accesses to produce a page fault. Likewise for a fixed storage allocation the program with clustered accesses will produce fewer page faults.

We of course would like to be more specific about the shape of f(s). Belady [6] has proposed an approximation for f(s) which when adjusted to fit the definition of f(s) given here becomes:

$$f(s) \approx 1 + a(s\,Q)^k \tag{2.4}$$

where a, k, and Q are constants for a given program. Of the three quantities, Q, the total size (bits) of the program, is the easiest to determine. a and k must be adjusted to fit the behavior of a given program. Belady indicates that k is generally in the vicinity of 2. It should be pointed out that this approximation has only two reasons for its existence. First it seems

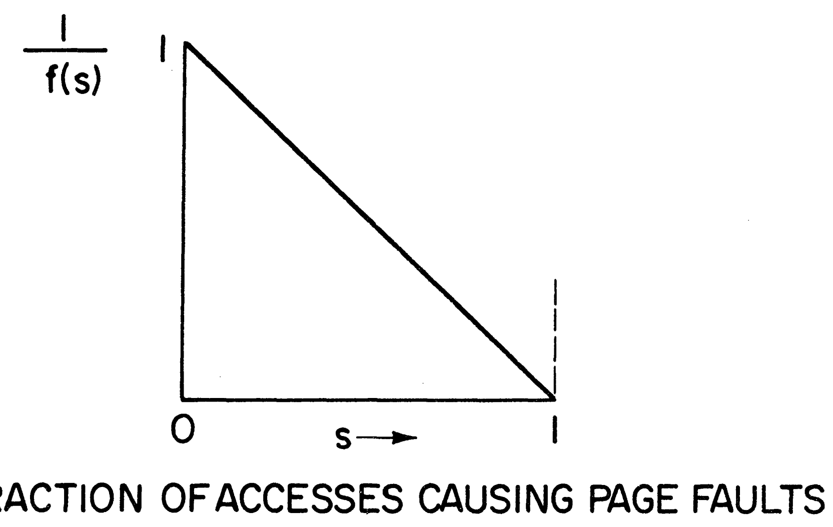


Figure 2-8. Fraction of Accesses Causing Page Faults

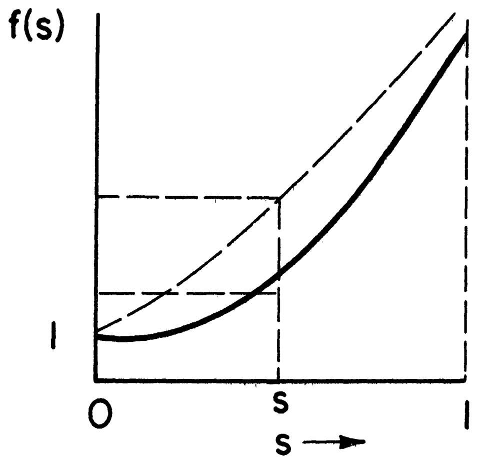


Figure 2-9 (a).

Effect of Non-Random Page Access on f(s)

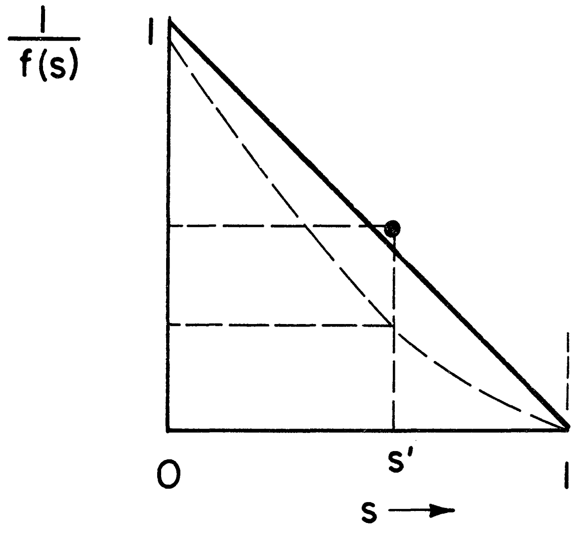


Figure 2-9 (b).

Effect of Non-Random Page Access on $\frac{1}{f(s)}$

to be possible to fit it to some real programs reasonably well and second it is simple.

A considerable amount of work has gone into extending the basic lifetime function presented thus far. The following extensions have been made.

1. Extension for use in an N-level hierarchy in contrast to a simple 2 level hierarchy such as a drum and core.
2. Inclusion of page size as well as allocation as a parameter.
3. The development of a lifetime function model based on program characteristics which includes the effects of page size.

The lifetime function is only part of the system model. We will now turn our attention to hardware and systems aspects of the model.

2.2.3 The Hardware and System Software

The hardware and system software are being considered together because the hardware constrains the design choices available in system software design. We will begin by considering what we mean by an N-level hierarchy.

2.2.4 The N-Level Hierarchy

The model which is under development is for the general case of N levels. For discussion here, however, we will choose $N = 4$. The choice of $N = 4$ is attractive in that it is large enough to demonstrate all the complexities of hierarchies where $N > 4$ and yet it is small enough to be convenient for discussion. Let us consider the 4-level hierarchy shown in Figure 2-10.

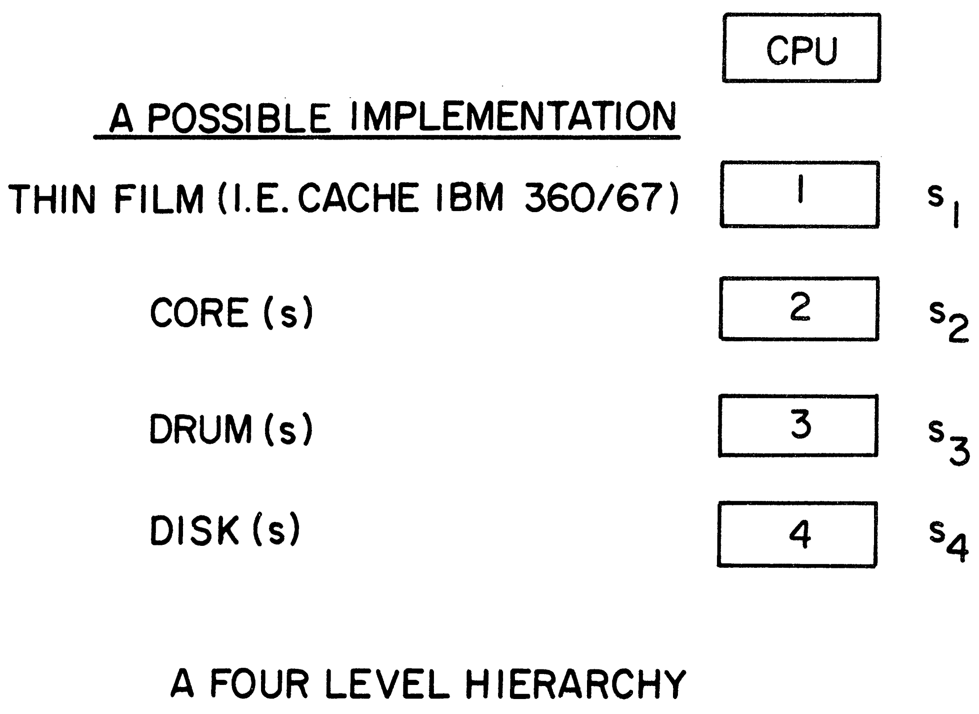


Figure 2-10. A Four Level Hierarchy

Here we have shown a CPU and 4-levels of storage and on the left a feasible set of devices is given. It should be clear that program information will migrate up and down in these 4 levels in much the same manner as a 2-level system.

2.2.5 Storage Allocation

When considering a system with $N > 2$ certain new questions arise concerning the allocation of storage space. First we must allocate storage at several levels rather than just core, and second we must be more precise in stating exactly what is stored where.

We may resolve the first question as it relates to the model by simply supplying the variable s with a subscript. Thus we will define s_i as the storage allocated to a process at level i. Figure 2-10 shows this vector as s_i's associated with the 4-level hierarchy.

The second question is a bit more complex. Exactly what is stored where? We will follow the following convention: If the current copy of some process information is resident at level k then a specific space must be reserved for this information at all lower levels (i.e., levels i where $i > k$). This space may or may not contain a current copy of the process information.

As a direct result of this convention we may state:

$$s_i \leq s_{i+1} \tag{2.5}$$

and

$$s_n = 1 \tag{2.6}$$

Notice that s_i represents not only the storage allocation at level i but also the amount of unique current information stored at and above level i.

2.2.6 Traffic Flow

Further complications arise regarding how traffic flows through the system. Specifically, we need to know what is transferred, where it is transferred and how much is transferred as the result of a program access to some given level. In addition the resultant delay in execution incurred as a result of these transfers is a result of the ordering and content of the transfers. For instance, consider a case in which a user program accesses the disk (level 4 in the example Figure 2-10) and as a result a block of information is moved from disk to core (level 2) and then part of that block is moved down in the hierarchy to the drum (level 3). Execution is certain to be delayed during the disk to core transfer but not during the core to drum transfers.

These detailed considerations of data transfers and their effect on the system are represented in the model by a number of matrices and vectors.

The performance of the system is also clearly dependent on the hardware at each level. We not only consider the choice of hardware at each level but its performance under loads imposed by the data transfers. The average total waiting necessary for a read or write at each level is considered to be a function of the hardware at that level including the number of channel paths and the average traffic of that level.

The complete model relates the characteristics of programs being executed, data transfer decisions, and hardware characteristics to system performance. Using this model we may vary the parameters of the model to study the system behavior and to optimize the cost effectiveness of the system.

It is important to note that the model describes a multiprogrammed system. We have ignored this here in order to simplify some of the discussion. Notice that in the following list of variables each process is t treated distinctly. For instance there is a specific allocation $s_{i,h}$ for each process h at each level i.

2.2.7 A List of System Variables and Functions

This is a list of the variables and functions used to describe the program characteristics, hardware characteristics and the system characteristics which together are used to describe the total system.

Q_h	size of process h (bits).
$f_h(\cdot)$	a function used to describe the paging characteristics of process h, the lifetime function.
w_h	fraction of write accesses by process h.
M	number of processes being multiprogrammed.
N	number of levels in the hierarchy.
q_i	record size at level i.
$\bar{S}$	allocation matrix whose elements $s_{i,h}$ = storage allocation at level i for process h.

$\bar{T}$ a description of the data paths (non-zero elements) and data transfer patterns. The elements of $\bar{T}$

$t_{i,j,k,\ell}$ = the number of bits transferred from level i to level j as a result of a primary read (ℓ=0) or write (ℓ=1) to level k.

$\bar{G}$ a description of the critical transfer paths in the system. Elements of $\bar{G}$

$g_{j,k}$ = weighting factor for the elemental read/write time at level k when expressing a primary access time to level j.

μ' a description of the normal or base execution rate. The mean time between storage accesses by the CPU when executing a process contained entirely in level 1 of the hierarchy with no other traffic in or out of level 1.

$\beta_i(C_i)$ a function used to describe the hardware at level i, the average waiting time at level i as a function of the average traffic at that level C_i.

References for Section 2

1. "Look Ahead," Datamation, April 1968, p. 53.

2. James Martin, Design of Real-Time Computer Systems, Prentice-Hall, Inc., Englewood Cliffs, N. J., 1967, p. 629.

3. C. DeGabrielle, "Design Criteria for an On-Line Nationwide Order Processing System," pp. 71-75 in Disk File Applications, American Data Processing, Inc., 1964, Detroit, Michigan.

4. K. M. Chandy and C. V. Ramamoorthy, "Optimization of Storage Systems," Information and Control, Vol. 13, No. 6, December 1968.

5. J. Gecsei and R. L. Mattson, Structural Optimization of Memory Systems, IBM Technical Note TN 02, pp. 525-585, June 1968.

6. L. A. Belady and C. J. Kuehner, "Dynamic Space-Sharing in Computer Systems," Communications of the ACM, Vol. 12, No. 4, May 1969, pp. 282-288.

7. P. J. Denning, "The Working Set Model for Program Behavior," Communications of the ACM, Vol. 11, No. 4, May 1968, pp. 323-333.

3. MULTIPROGRAMMED SYSTEMS

In this section we discuss research into two problems related to the operational optimization of multiprogrammed, paged computer systems. The first problem is that of breaking up programs into pages in such a way as to minimize the number of page faults when that program is run. The second problem is that of scheduling the set of programs requesting service in such a way as to maximize the throughput of the computer system.

3.1 Optimum Program Pagination

3.1.1 Background

The objective of this research is to explore methods of improving the operating efficiency of paged multiprogrammed systems by optimization of the paging process.

Multiprogramming, as the name implies, means having several programs occupy high speed memory simultaneously. Problems of effectively using storage in a multiprogrammed mode are appropriately called problems of storage allocation. When we consider these problems in the context of the early days of computing, they were relatively straightforward.

In early batch systems where programs were run one at a time, each program had the entire high speed memory (core) available. Problems arose when a program was larger than the available core. In these cases the programmer had to improvise by "segmenting" his program (instructions and data), and controlling the "overlaying" of segments. Segmenting, of course, referred to dividing the program into parts (segments), and overlaying was the process of bringing the unused parts of a program into core from auxiliary storage as they were needed during execution.

When the problem of segmenting a program was given to the operating system instead of the programmer, we had what was called an automatic segmenting system; the problem of segmenting a program in an optimum manner, e.g., so as to minimize the number of overlays, became known as the classical overlay problem or the problem of program segmentation. In some systems, excessive overlaying could cause a serious decrease in the system's operating efficiency.

In multiprogrammed systems overlaying or segmenting is intended to increase the size of effective memory. A summary of techniques for overlaying can be found in [1].

3.1.1.1 Paging

When high speed memory is divided into fixed size blocks called page frames, and programs and data divided into similar fixed sized blocks called logical pages, the process of mapping the logical pages to page frames is called paging. It is the determination of efficient algorithms for the automatic production of logical pages of a program which is the focus of this research. We call such algorithms "pagination" algorithms.

3.1.2 Model

A rather natural model for study of program pagination is a linear graph. Ramamoorthy [2] was the first to formulate the problem of program paginations as a problem of partitioning of a linear graph into certain specified sets of vertices.

Basically, the model for study of optimum construction of program pages is as follows. Each vertex of a graph represents a block of instructions or data; the directed arcs between blocks represents transfers of control between blocks of instructions. Usually, undirected arcs are used to indicate a block of instructions referencing data. Since the arc is undirected, control is to remain in the instruction block.

The abstract statement of the problem in graph theoretic terms is, given a graph G with weighted arcs partition the graph into disjoint subsets of vertices such that no subset of vertices is larger than some maximum size and the sum of the weights of the arcs between the subsets is minimized.

There are many practical applications, other than program pagination, which makes use of this same abstract model. For example, the problem of assignment of logic gates to modules such that the intermodule delay is minimized is a problem with the same abstract model.

3.1.3 Solution Techniques

3.1.3.1 Optimal Solutions

It is possible to formulate the graphical partitioning problem in terms of an integer linear program with a large number of constraint equations to insure the feasibility of the subsets of vertices selected. However, for problems of practical significance, i.e., number of vertices greater than

25, methods of integer linear programming techniques are not sufficiently powerful to give solutions in a reasonable amount of time. For example, the number of ways in which a set of 25 objects can be partitioned into 10 subsets is 1, 203, 163, 392, 175, 387, 500. Now given that many partitions will be infeasible due to size constraints on the vertices, there will remain an inordinate number of cases even after those infeasible partitions are deleted.

3.1.3.2 Known Optimal Solutions

There has been reported in the literature one algorithm for the exact solution for the graphical partitioning problem [3]. This algorithm uses an enumerative technique, backtrack programming; hence it is impractical for large graphs, i.e., on the order of 100 vertices. Therefore this technique was not pursued in this research.

Another algorithm which gives an exact solution for a special case of the partitioning problem was reported in [4]. However, the class of graphs for which the methods are applicable is rather restrictive, i.e., the graph must be a tree. The methods are efficient enough for graphs with hundreds of vertices. The same algorithm was extended for acyclic graphs (directed graphs with no cycles) but in this case vertices must be replicated; this would correspond to placing certain blocks of code on more than one page.

3.1.3.3 Heuristic Procedures

In view of the fact that most practical problems involve graphs with numbers of vertices too large for any exact method, we have been evaluating and developing heuristics for practical applications.

One previously reported heuristic is the so-called unit merge algorithm [3]. Here a partition is formed as follows. Let c_{ij} represent the weight of arc between vertex i and vertex j; we examine the matrix $[c_{ij}]$ for maximum c_{ij}, if the vertices i and j are compatible, i.e., the sum of the sizes do not exceed a given constant, then vertices i and j are combined (merged) to form a single vertex; otherwise matrix $[c_{ij}]$ is scanned for the next largest value c'_{ij} until finally a merge can occur. After merging the graph is updated by appropriately changing the weights of arc which **previously** were connected to the merged vertices and updating the size of the merged vertex (the new size is the sum of the sizes of the two vertices which were merged). The process is continued until it is no longer possible, due to size constraints, to make any other merges.

This technique is quite fast since each vertex, once it is merged, is never removed from a merging. However, we have found that one can obtain some very poor partitions using this technique. And this technique does not work at all as a general partitioning procedure when the matrix $[c_{ij}]$ is Boolean.

Another procedure has been reported recently in [5]. This technique is based upon a heuristic which was used to solve the traveling salesman problem. The procedure is based upon finding the optimum partitioning into two blocks of a set of vertices. Basically, the procedure is as follows for optimum two-way partitionings. Given an arbitrary partition $\{A, B\}$ of a set of vertices V such that $|A| = |B| = n$, hence $|V| = 2n$, an optimum partition $\{A^*, B^*\}$ is obtained as follows. For every $a \in A$ compute an external cost E_a, by

$$E_a = \sum_{y \in B} c_{ay}$$

and an internal cost I_a by

$$I_a = \sum_{y \in A} c_{ay}$$

Similarly, define E_b, I_b, for each $b \in B$. For each $v \in V$ we compute $D_v = E_v - I_v$. Now the gain from interchanging $a \in A$ and $b \in B$ is

$$g = D_a + D_b - 2c_{ab}$$

The algorithm goes as follows. First, compute D for each $v \in V$. Next find $a_i \in A$, $b_j \in B$ such that

$$g_1 = D_{a_i} + D_{b_j} - 2c_{a_i b_j} \quad \text{is maximum.}$$

Then remove a_{i_1} from A and b_{j_1} from B. Now the D values are recomputed for the sets $A - \{a_{i_1}\}$ and $B - \{b_{j_1}\}$. Then g_2 is computed, as was done for g_1, for some $a_{i_2} \in A - \{a_i\}$ and $b_{j_2} \in B - \{b_{j_1}\}$. This is continued until all pairs of nodes have been exhausted. The gain for interchanging pairs (a_{i_1}, b_{j_1}), $(a_{i_2}, b_{j_2}), \ldots, (a_{i_n}, b_{i_n})$ is $g_1 + g_2 + \ldots + g_n$. (Note that

$\sum_{i=1}^{n} g_i = 0$). Choose k such that the partial sum $G = \sum_{i=1}^{k} g_i$ is maximum. If $G > 0$, then a reduction in cost has been realized and the procedure is repeated starting with the new partition just obtained. Ways of extending this algorithm to the general problem are given in [5].

A procedure which we have found to be effective is to sequentially select sets of a partition by solving a relatively simple quadratic programming problem. Our procedure assumes that one can select a set of vertices which is in a sense optimal by solving the following problem.

(1) maximize $f = \sum_{i,j} c_{ij}(kx_ix_j - x_i - x_j + 1)$

subject to

(2) $\sum a_ix_i \leq \mu$

(3) $x_i \in \{0,1\}$, where .

$x_i = 1$, vertex v_i selected for optimal block

0, otherwise.

a_i-size vertex v_i.

The cost function was motivated by the following consideration. We would like to select a set of vertices for a block of the partitions which are as tightly connected as possible. This we could do by using

$$f = \sum_{i,j} c_{ij}x_ix_j$$

as a cost function. But a cost function of this form has been seen to leave the remaining graph poorly connected. Therefore we've found that a cost function such as

$$f = k \sum_{i,j} c_{ij}x_ix_j + \sum_{i,j} c_{ij}(1-x_i)(1-x_j), \; k \geq 1$$

takes into consideration the connectivity of the graph G-V where V is the set of vertices removed. The first term measures the connectivity of V, the block of vertices which are removed; the second term measures the connectivity of the graph G-V, the graph from which the remaining blocks of the partitions are chosen.

We solve (1), (2) and (3) for the first block of vertices of the partition. Then we resolve (1), (2) and (3) for the new graph minus those vertices which were selected by the previous solution. The procedure is continued until the size of the remaining set of vertices is feasible; now at each step we are solving a smaller problem. We had some fairly effective ways to solve (1), (2) and (3) which we are now in the beginning stages of programming.

We intend to compare all the given algorithms of graphical partitioning to see which ones are the most efficient in terms of practical applications.

3.2 Optimum Task Scheduling

This section is a discussion of methods and models for determining optimal task scheduling rules in multiprogrammed computer systems. In this work an optimal scheduling rule is one which maximizes the throughput of the computer system under consideration. This effort will lead to expressions for optimal or near-optimal scheduling rules where these expressions will be functions of such system parameters as the current status of tasks using the system and the arrival rates of incoming tasks. In Section 3. 2. 3 we give a description of the mathematical model to be used for determining optimal scheduling policies, and in Section 3. 2. 2 useful modeling and optimization procedures are discussed. Section 3. 2. 1 is devoted to a review of material necessary to an understanding of the problem and the approach to be taken to it.

3. 2. 1 Multiprogramming and Reentrant Procedure

Multiprogramming or the simultaneous residence of more than one program in main (core) memory is widely used in today's large computer systems in order to make the best possible use of system resources. The wide disparity between speeds in the central processing unit (CPU) and input/output (I/O) devices means that in a mono-programming mode the CPU is likely to be idle a considerable amount of time while waiting for I/O operations to complete. This is extremely undesirable because of the high cost of having the CPU stand idle. Multiprogramming allows the CPU to switch to another program when one program is held up by an I/O operation and thus may considerably increase the CPU utilization. On the other hand, the amount of main memory required for successful multiprogramming is considerably greater than that required when only a single program is in core. Since core memory is expensive it is desirable to use it as efficiently as possible.

The procedure in a computer program is made up of instructions in that program as opposed to the data on which those instructions operate. Reentrant or pure procedure is procedure which does not modify itself in the course of being executed. Thus more than one user may execute the same copy of a reentrant procedure and this is the definition of sharing as it is used in this work. Three advantages accrue from using reentrant procedure.

1. When several tasks use a single copy of a procedure simultaneously, a considerable space saving in core memory may result.
2. Since this procedure must be brought into core memory via a channel, a reduction in required channel capacity may be achieved by sharing.
3. Because the procedure is never altered it need never be removed from core (it is simply overwritten and a copy is maintained external to main memory).

In the model of Section 3.2.3 the procedure for each task is assumed to be reentrant.

The sorts of routines which might be shared are those routines which are heavily used such as translators, input-output conversion routines, and library routines. The advantage gained by sharing a routine is dependent on its frequency of use and the required response to users. There is not much point in allowing a routine to be shared which is used only once a week when the required response time to users is one-half day. The addressing structure and hardware features [6, 7] in many modern large computer systems [8] make possible the sharing of procedure among users without introduction of very much additional supervisory overhead.

These same machines also include the ability to maintain in core only those parts of programs currently in use. This allows a greater number of programs to share core simultaneously than would otherwise be possible. Programs are broken into fixed-size units called pages and these units are brought into core when needed and removed from core when they are no longer being used. A common way of handling the paging in is to bring a page into core only when it is referenced by a program (called page-on-demand). At this point a program interrupt occurs and the program is ineligible to execute until the referenced page is available in core.

Next we turn to a discussion of possible techniques for modeling computer systems having the features just outlined.

3.2.2 Modeling Computer Systems

For purposes of obtaining a model which allows an investigation of the effects on throughput of the scheduling rule used in the computer system, we may look at the computer as a service system. A task requiring execution arrives at the computer and passes through particular queues and servers dependent on the task's computational requirements. Control of these queues is carried out by the scheduler through an allocation of the servers to tasks requiring the use of these servers.

Two ways to model a system like this are by use of queueing theory [9, 10, 11] to develop a formal queueing model and by use of Monte Carlo simulation. Simulation models [12, 13] have been used with success in modeling computer systems although the expense of running them usually precludes their use for investigating more than a small number of system parameter values. Markovian queueing models which have yielded closed form solution [13, 14, 15, 16] have given some insight into computer system behavior, but the ability of a model of this type to portray the complexities of computer system behavior is severely limited. An intermediate approach is to model the system with a large Markovian queueing system which defines closed form analysis but which yields useful results by application of numerical technique [17]. Models of this type have been quite successful [18, 19, 20, 21] by allowing a complexity approaching that in many simulation models at a fraction of the computational cost. Furthermore, Markov models lend themselves to application of optimization techniques if appropriate objectives can be defined.

The computer system model developed in Section 3. 2. 3 is a Markovian queueing model. Specification of system parameters and a scheduling policy for this model results in a value for the throughput of the system. Additionally, we may define as our objective the maximization of system throughput and use this model in the formulation of the optimization problem as a Markovian Decision Process [22, 23]. Part of this research effort is devoted to improving the techniques available for handling this type of

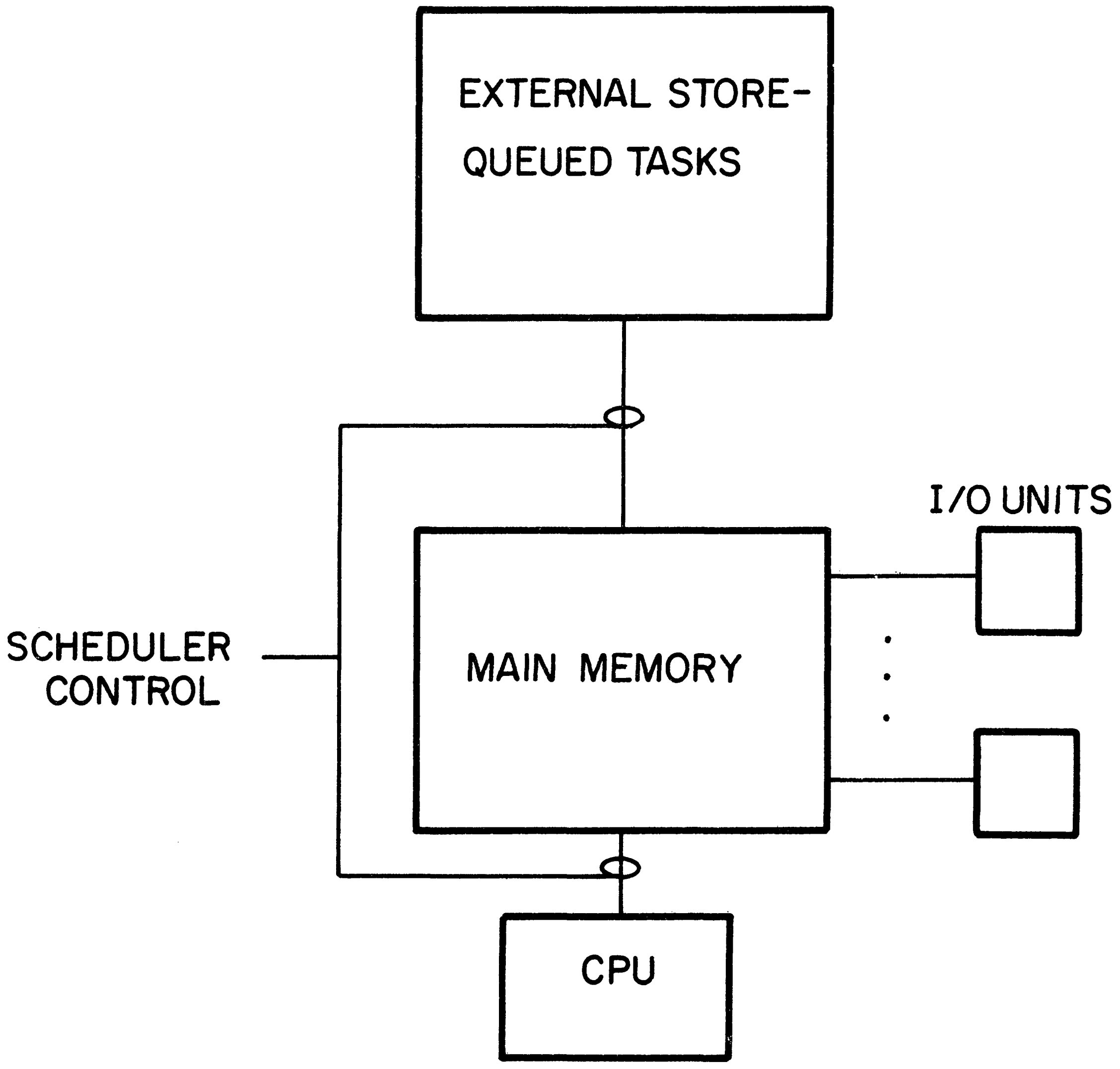


Figure 3-1. System Configuration.

optimization problem.

3.2.3 Scheduling Model

In this section a model of a multiprogrammed computer system is developed. This model will be used to determine system throughput as a function of system parameters and the scheduling policy used. The model will be used to find scheduling policies which maximize the system throughput.

Consider the multiprogrammed computer system shown in Figure 3-1. A large external store is used to hold tasks which are queued awaiting execution: the scheduler controls the movement of these tasks to main memory where they may execute. The tasks multiprogrammed in main memory are served by a CPU and a multiplicity of input/output devices, and the scheduler determines which task has priority at the CPU. Thus the scheduler takes action at two places in the system, and it should be clear that these two types of decisions are not independent.

We assume M distinct kinds of tasks, each of which makes use of a distinct reentrant procedure. Therefore there are M memory queues in the system: one to hold each type of task. An arriving task is placed in the appropriate queue. The probability that an arriving task is of type i is a_i and is independent of previous arrivals or of the number of other tasks of type i in the system. Of course,

$$\sum_{i=1}^{M} a_i = 1$$

since an arriving task must enter some one of the M queues.

There are M distinct task types in the system and the model must distinguish among them. Thus at each point in the system where queueing can occur, there must be M such queues. The following description is for the generic queues and servers which make up the model while the reader should keep in mind that each queue has multiplicity M.

A task arriving at the system queues up outside main memory. The scheduler decides, constrained by availability of main memory, when to load this task into main memory. Once loaded, the task queues at the central processing unit. Again, the scheduler determines task priority at the CPU. A task uses the CPU until it requests an input/output operation. The task then relinquishes the CPU and carries out the I/O operation. When this operation is complete the task either terminates or goes back to the CPU for more execution and the cycle continues. Thus the scheduler makes interrelated decisions at two points in the system: at the main memory and at the CPU.

A diagram of the model is shown in Figure 3-2. Tasks flow through the system as indicated by the lines. Tasks within squares are being served while those in circles are enqueued. Constrained by available memory the scheduler decides what type of task to load. When the CPU becomes available the scheduler determines which of the tasks enqueued there gets use of the CPU. This is the basic model to which optimization techniques will be applied. Future plans also call for the addition of paging to this basic model.

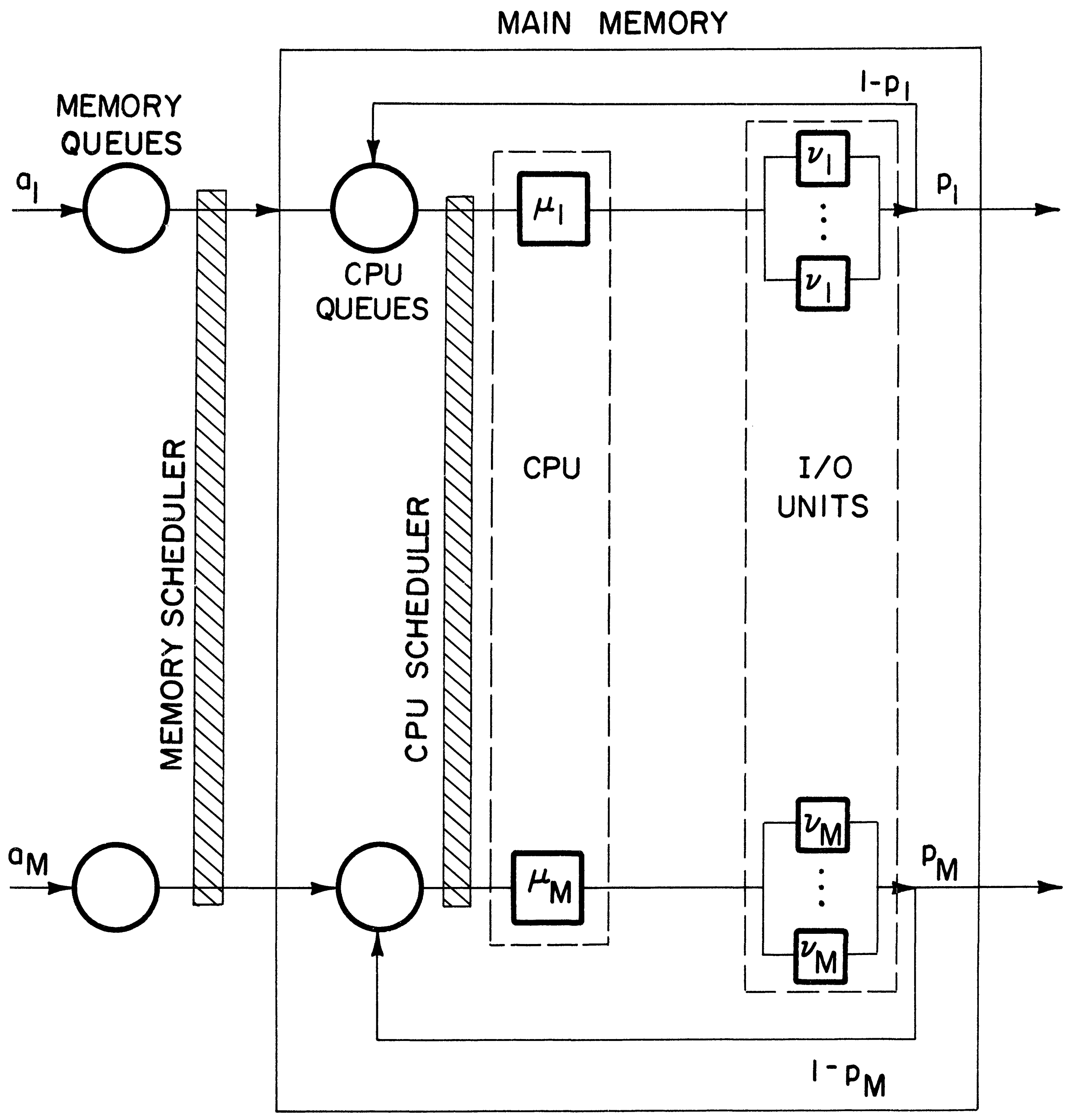


Figure 3-2. Scheduling Model.

References for Section 3

1. R. J. Pankhurst, "Program Overlay Techniques," Comm. ACM, 11, February 1968, pp. 119-125.

2. C. V. Ramamoorthy, "The Analytic Design of a Dynamic Look-Ahead and Program Segmenting System for Multiprogrammed Computers," Proc. ACM 21st National Conference, August 1966, pp. 199-239.

3. Informatics Incorporated, "Program Paging and Operating and Operating Algorithms," Technical Documentary Report, TRGB-793-1, September 1968.

4. E. L. Lawler, K. N. Levitt, and J. Turner, "Module Clustering to Minimize Delay in Digital Networks," IEEE Trans. on Computers, vol. C-18, No. 1, January 1969, pp. 47-57.

5. B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," The Bell Systems Technical Journal, Vol. 49, No. 2, February 1970, pp. 291-307.

6. B. Arden, B. Galler, T. O'Brien and F. Westervelt, "Program and Addressing Structure in a Time-Sharing Environment," Journal of the ACM, Vol. 13, No. 1, January 1966, pp. 1-16.

7. J. B. Dennis, "Segmentation and the Design of Multiprogrammed Computer Systems," Journal of the ACM, Vol. 12, No. 4, October 1965, pp. 589-602.

8. B. Arden, "Time-Sharing Systems: A Review," Proc. of the IEEE International Convention, Vol. 15, Part 10, March 1967, pp. 23-35.

9. D. R. Cox and W. L. Smith, Queues, Wiley, New York, 1961.

10. P. M. Morse, Queues, Inventories, and Maintenance, Wiley, New York, 1958.

11. T. L. Saaty, Elements of Queueing Theory, McGraw-Hill, New York, 1961.

12. T. B. Pinkerton, "Program Behavior and Control in Virtual Storage Computer Systems," Technical Report 4, CONCOMP, The University of Michigan, April 1968.

13. A. L. Scherr, "An Analysis of Time-Shared Computer Systems," MAC-TR-18, Cambridge, Massachusetts Institute of Technology, Project MAC, 1965.

14. L. Kleinrock, Communication Nets: Stochastic Message Flow and Delay, McGraw-Hill, New York, 1964.

15. L. Kleinrock, "Time-Shared Systems: A Theoretical Treatment," Journal of the ACM, Vol. 14, No. 2, April 1967.

16. L. Kleinrock, "Certain Analytic Results for Time-Shared Processors," Proc. 1968 IFIPS Conference, August 1968, pp. D119-D125.

17. V. L. Wallace and R. S. Rosenberg, "RQA-1, The Recursive Queue Analyzer," Report 07742-1-T, Systems Engineering Laboratory, The University of Michigan, February 1966.

18. D. W. Fife, "The Optimal Control of Queues, with Applications to Computer Systems," Technical Report No. 170, Cooley Electronics Laboratory, The University of Michigan, October 1965.

19. D. W. Fife, "An Optimization Model for Time-Sharing," Proc. 1966 AFIPS SJCC, Vol. 28, pp. 97-104.

20. J. L. Smith, "Markov Decisions on a Partitioned State Space, and the Control of Multiprogramming," Report 07742-4-T, Systems Engineering Laboratory, The University of Michigan, April 1967.

21. J. L. Smith, "Multiprogramming under a Page on Demand Strategy," Communications of the ACM, Vol. 10, No. 10, October 1967, pp. 636-646.

22. R. Bellman, "A Markovian Decision Process," Journal of Mathematics and Mechanics, Vol. 6, No. 5, 1957, pp. 679-684.

23. R. A. Howard, Dynamic Programming and Markov Processes, Massachusetts Institute of Technology Press, Cambridge, 1960.

4. CENTRAL PROCESSOR OPTIMIZATION

This section focuses on research into optimization of two facets of digital computer central processors. The general study areas are

1) Choice of data paths in the CPU;

2) Choice of micro-instructions in a micro-programmed CPU.

In this work we wish to make these choices in a way which minimizes a suitable cost-performance ratio.

4.1 Data Path Optimization

The problem under consideration here is that of formalizing the design of a digital computer and developing algorithms and procedures which can be applied in optimizing the design of the central processor. Specifically, it is the problem of creating the optimum data path for a central processing unit when the system architecture is given. The term data path is used here to refer to a set of hardware logic units such as registers, adders, and counters and the interconnections between them for data transfers. System architecture means a description of the computing system as it appears to the programmer and is composed of definitions of such items as data and instruction word formats, addressing and indexing structure, and the operation of each instruction. Optimization requires maximizing the ratio of performance, as measured by a weighted average instruction execution time, to the cost of the data path.

4.1.1 Example

The concepts involved here can be made clear through a simple example. Assume the system description of a central processing unit includes the definition of two registers, R1 and R2, which are accessible to the programmer and an instruction which swaps the contents of R1 and R2. Many different arrangements of hardware logic units can be used to implement this instruction and which will result in different hardware costs for the computer and different execution times for the instructions.

Figure 4-1 shows a data path having one additional register, T1, which is not accessible to the programmer and an adder. The instruction can be executed on this data path in three steps:

1) R1 —> ADDER —> T1

2) R2 —> ADDER —> R1

3) T1 —> ADDER —> R2

Figure 4-2 shows a data path which has an additional shifting unit. This allows the instruction to be executed in just two steps as follows:

1) R1 —> ADDER —> T1

2) T1 —> SHIFTER —> R2 and R2 —> ADDER —> R1

Finally, in Figure 4-3 a new path has been introduced connecting R2 to the shift unit. This allows the instruction to be performed in just one step:

1) R1 —> ADDER —> R2 and R2 —> SHIFTER —> R1

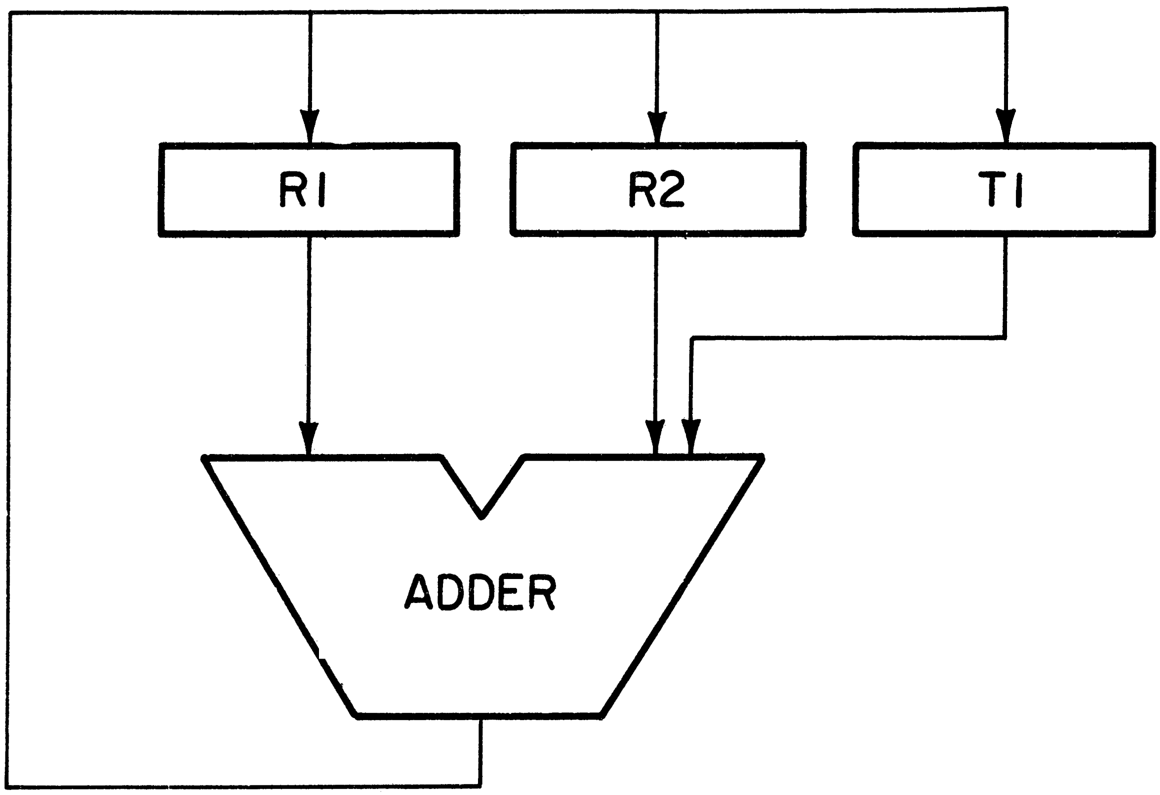


Figure 4-1.

CPU Data Path I

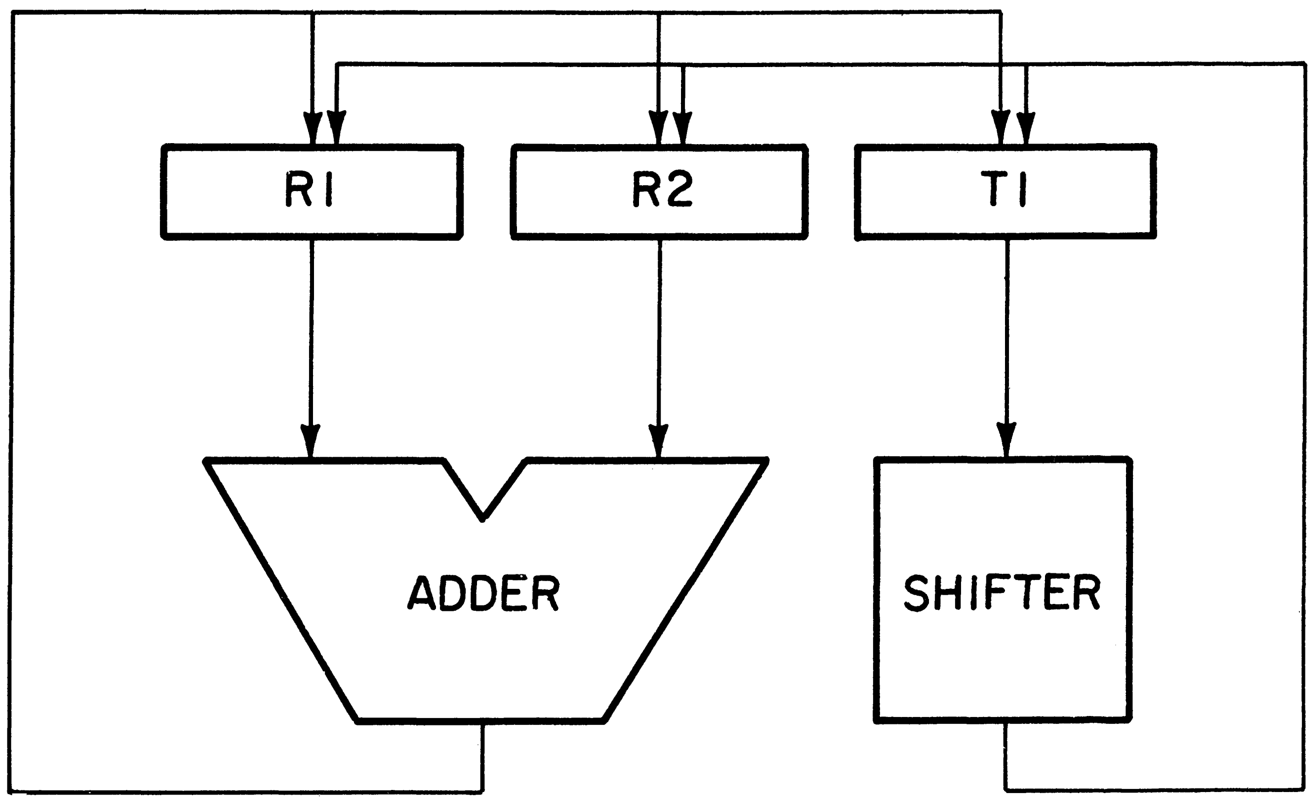


Figure 4-2.

CPU Data Path II

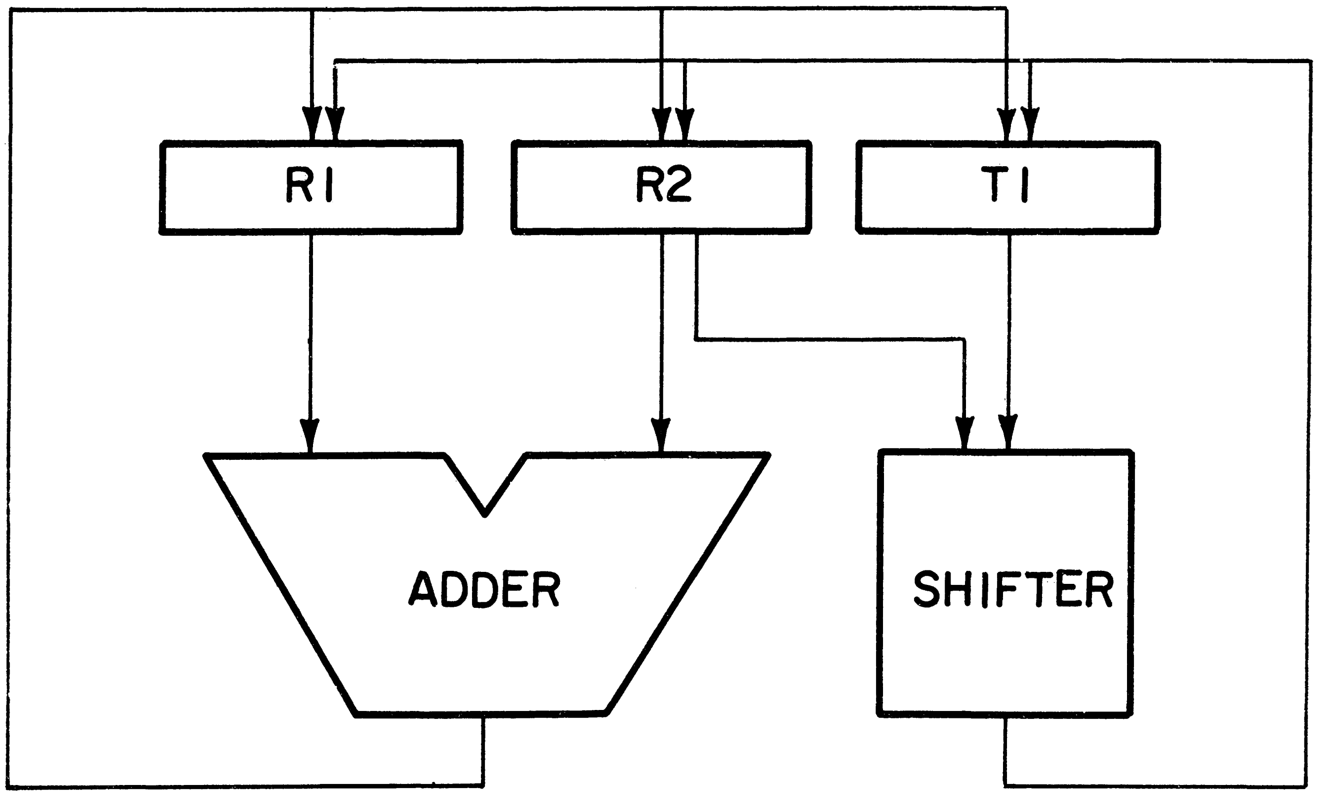


Figure 4-3.

CPU Data Path III

Clearly, each of these data paths has a different cost and allows different execution times for the instruction. The problem of determining which of these is optimum requires calculating the cost of each of the data paths and then comparing their performance-cost ratio.

Even for a computer with a very small instruction set this problem quickly becomes unmanageable. The number of different data paths which could be used becomes large and cost evaluation requires a detailed design study of each one. As a result, the normal solution of this problem involves a considerable amount of intuitive judgment on the part of the designer since it is just not possible to carefully consider all of the possibilities. The objectives of the research on this problem are to gain an understanding of the relationships between the definition of the instruction and the design of the optimum data path to implement it, and to provide algorithms which will allow the computer designer to systematically explore the set of possible data paths to find the optimum one.

4.1.2 Model for the Study

The overall structure of the model being used in this study is shown in Figure 4-4. The architecture description is taken as input to the data path design process. A catalogue of hardware unit designs and a library of algorithms are used to produce a particular data path and a set of flow charts indicating the implementation of each instruction on the data path.

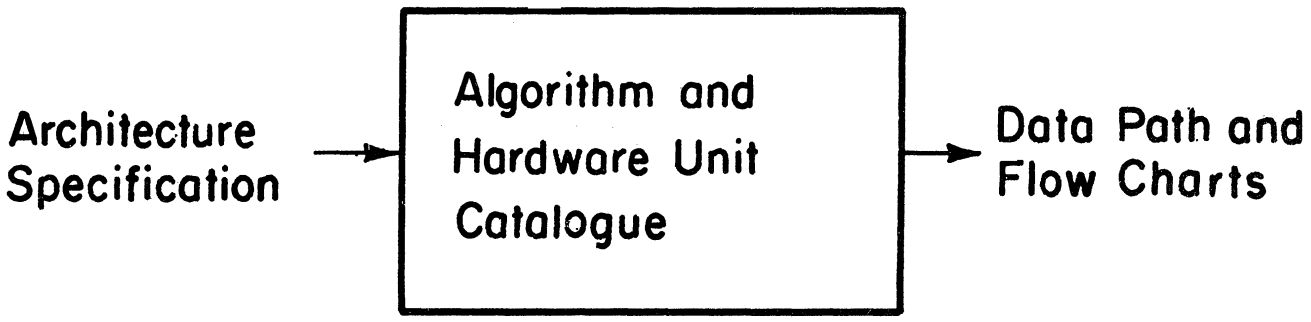

Architecture Specification
Algorithm and Hardware Unit Catalogue
Data Path and Flow Charts

Figure 4-4.

Block Diagram of Model

4.1.2.1 Architecture Specification

The architecture is specified in three parts:

1) Facilities
2) Instructions
3) Performance requirements.

The definition of the facilities is composed of a list of the programmable registers, data and instruction word formats, and interfaces of the processor to external units.

The instruction set is described in a language which allows each instruction to be defined completely and unambiguously but does not specify a particular algorithm. This is accomplished by using a rich set of operators in the language and by providing a means to distinguish those steps in an instruction which must be done sequentially from those that can be more freely reordered.

The computing power is measured by the mix method of performance evaluation, that is, by computing a weighted average of instruction execution times over the entire instruction set. The data which is specified in the performance requirement section of the architecture definition is the weighting factor or percent usage figure for each instruction. Also a maximum allowable execution time is specified for each instruction as well as maximum and minimum values for the weighted average execution time.

4.1.2.2 Hardware Unit and Algorithm Catalogues

The hardware unit catalogue is a library of designs for the hardware units that can be used in a data path—registers of various types, adders, counters, etc. Each entry gives a list of the transformations that unit performs and functions for calculating the delay and cost of the unit from the manner it is used in the data path (the width in bits of the unit and the loading on the unit ports).

The algorithm catalogue is a library of algorithms for translating the set of operator symbols used in the instruction definition language into the set of transformations that can be performed by hardware units. An operator is translated by many different algorithms allowing a wide selection of data path hardware units to be considered in implementing an instruction.

4.1.2.3 Data Path and Flow Charts

The results of the design and optimizing process are presented by specifying a data path and a flow chart of each instruction.

The description of the data path is given as:

1) A list of the hardware units selected from the hardware unit catalogue.

2) A list of the connections between the hardware units.

The flow charts or sequencing charts for the instruction set are cycle by cycle descriptions of the operation of the data path for each instruction. The information specified for each cycle is:

1) Which connecting links between units are open and which are closed.

2) Which transformation each hardware unit is performing (most units can perform more than one function).

4.1.2.4 Criteria for Optimization

The desired solution to the problem, that is, the optimum data path and flow charts, is one which satisfies the following conditions:

1) Maximize $\left[\frac{\text{weighted average instructional execution time}}{\text{total cost of hardware units}}\right]$

2) Weighted average instruction execution time is between the minimum and maximum values given in the architecture specification.

3) Each instruction execution time is less than the maximum given in the architecture specification.

4.1.3 Status of Research

At present, a mathematical model has been completed which allows a precise statement of this problem. Effort will now be directed to developing methods of selecting hardware units and algorithms for a design and to searching systematically for an optimization.

4.2 Micro-program Control

This study centers on micro-program control of computer systems. Micro-program control of central processing units has increased in the last few years. Currently, there are micro-programmed channel controllers, display devices, and other pheripheral devices. Therefore, both efficient design and efficient use of micro-programmed control units is very important in a large computer system.

4.2.1 What is Micro-programming?

The invention of micro-programming is generally credited to M. V. Wilkes [1, 2] who in the early 1950's suggested micro-programming as a more orderly approach to the design of computer control circuitry. He was particularly concerned with the signals needed to control the flow of information among the registers and transformation units that comprise the central processing unit. Wilkes felt that the execution of a machine instruction* by the central processing unit (CPU) could be partitioned into a series of register to register transfers in the CPU. Some of the transfers of data in the CPU would, of course, be made to pass through a transformation unit while going from the source register to the destination register. A typical transformation unit might be capable of shifting, adding, logical disjunction, etc.

A representation of Wilkes' scheme is shown in Figure 4-5 with the nonvolatile "control memory" divided into a part A and a part B. Each column of part A represents a micro-operation. The excitation of a micro-operation causes a predetermined action by one or more elements of the CPU. Some of the actions that a micro-operation might cause are: data

----

*Machine instruction: the binary code for an operation that is performed by the central processing unit. The code is normally held in the main memory until fetched into the CPU for instruction decoding and execution.

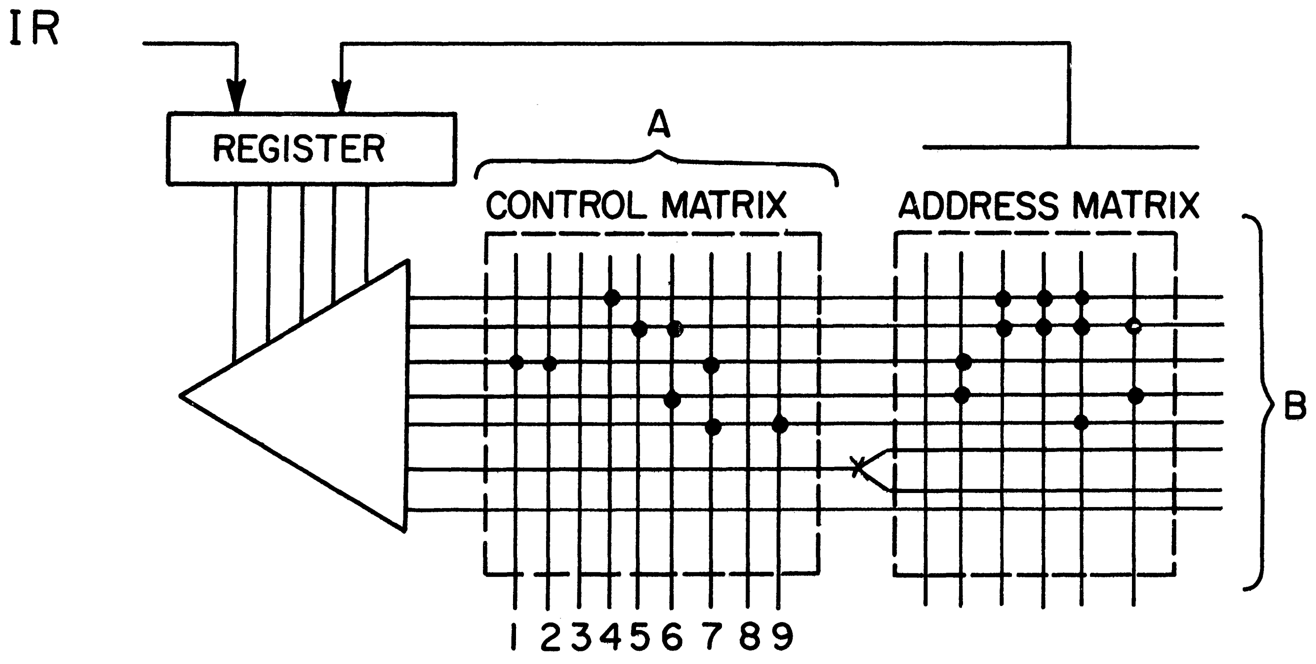


Figure 4-5. Wilkes Microprogram Control

gated out of a register, data gated into a register, data gated through a transformation unit. Each row in the control memory represents a micro-instruction. Part A of the micro-instruction controls gating in the central processing unit (CPU), and part B of the micro-instruction contains the address of the next micro- instruction to be executed. Thus the micro-program controls the gating within the CPU and hence, controls the transfer of data within the CPU.

The control of a central processing unit, according to Wilkes, is implemented by the execution of a series of micro-instructions. The excitation of the micro-operations specified by each micro-instruction causes register to register transfers to occur. The series of register to register transfers specified by the micro-instructions implement the desired machine instruction.

There are two different approaches to micro-program control. In the first, horizontal micro-programming, each micro-operation usually controls one gate (either input gate or output gate) per flip flop for each flip flop in a register. In some cases the micro-operation may control the flow of information into or out of a portion of a register. In the second, vertical micro-programming, each micro-operation controls all the gating necessary to implement a register to register transfer.

Horizontal micro-programming employs the micro-operations of the micro-instruction word in selecting the data paths within the CPU. Each micro-operation controls gates (either input gates or output gates) within the CPU, thereby causing data to flow into or out of registers and transformation units. A register to register transfer is executed by the proper selection of micro-operations, each execution at the same time. It is the freedom of selecting the data paths, and therefore the register to register transfers, which gives the horizontal micro-programming control method great flexibility.

The flexibility offered by the use of horizontal micro-programming has its price. The length of micro-instruction words for a horizontal micro-program controlled machine are longer than those for a vertical micro-program controlled machine. Also, the programming of a horizontal micro-program machine is tedious. It is much easier to specify a register to register transfer than to specify all the micro-operations required to implement the register to register transfer.

Vertical micro-programming employs the individual micro-operations of the micro-instruction word to specify register to register transfers. The register to register transfer must be executable in one micro-instruction cycle time. For most CPU's the number of different register to register transfers that may be implemented is very large. Normally, a small subset of all possible transfers is chosen as the set of micro-operations. Since a restricted set of transfers is normally available in a vertically micro-programmed machine, it is not as flexible as a horizontally micro-programmed machine. However, the micro-instruction word length for vertical micro-programming is normally smaller than the micro-instruction word length for horizontal micro-programming. In addition, the vertical micro-program machine is easier to micro-program than a horizontal micro-program machine. Vertical micro-programming is very similar to machine language or assembly language programming of a conventional computer.

4.2.2 Advantages and Disadvantages of Micro-programmed Computer Systems

The advantages and disadvantages of using micro-program control will be listed and discussed in this section. The number of advantages presented in the discussion far outnumber the disadvantages which are presented. This was not done with any intentional bias, but is simply the situation as it appears to us. Some of the advantages listed for micro-program control are based on equipment and methods not yet fully developed. However, in all such cases current research supports the claims made. The disadvantages of micro-program control will be presented first.

One disadvantage resulting from the replacement of logic control by micro-program control is the decrease in execution speed. This occurs because the time spent accessing the control memory is usually greater than the time required for the logic control network to act. The effective micro-instruction access time can be shortened by using a micro-instruction look ahead facility. However, the look ahead facility will increase the cost of the micro-program control unit.

The use of a changeable control memory implies some potential problems. First, the control memory can be changed accidentally. Any accidental change in the control memory is equivalent to a component failure in a hardwired control unit; therefore, some method must be used to check the validity of the control memory. This can be done with control memory against a copy held in the main memory. The problem is not insurmountable, but does require consideration.

Secondly, the micro-programs contained in the control memory should not be changed haphazardly [2, 3]. It would be hoped that any changes in the micro-programs would still allow older software programs to be run under the new micro-programs. The integrity and continuity of the micro-programs must be maintained with greater care than even present day executive systems.

Micro-program computers were first proposed [1, 2] with the expectation that such computers would be easier to design and hence less expensive. Currently, the design and development of micro-program computers can be justified for many other reasons. The advantages of micro-programmable computers can be seen in areas of diagnostics emulation, system compatibility, and special purpose applications.

The use of micro-programming does allow a more systematic approach to the design of computer systems. The basic design method involves choosing a set of micro-operations which are to be implemented by the control memory. The desired machine instruction is then formed from sequences of micro-operations. This method has a simplicity and ease of understanding which is not available in the timing diagrams, ring counters, and logic diagrams of hardwired control circuitry. Because micro-programs reside in a control memory, they are changed more easily than hardwired control circuitry. Therefore, correction of mistakes in machine instruction implementation or addition of new machine instructions may be accomplished more easily and at a later date in the design schedule.

In addition, the interaction of the machine designers and the system programmers has a greater possibility of occurring. The reason for the increased interaction is the necessity of programmers and logic designers interacting an intermediate stage in the design of a computer. Micro-programs can be written by system programmers and need not be written by logic designers. However, the micro-programmers must have a detailed understanding of the micro-operations if they are to write efficient micro-programs. Also, if the logic designers are to select the most useful set of micro-operations, they must have knowledge of the machine instructions that the system software designers require. Thus it can be seen that micro-programming techniques will indeed require more interaction between the logic designer and the system programmer.

The use of micro-programming for economic reasons and system compatibility was fully exploited by IBM in their system 360 series of computers. The IBM [4] system 360 series of computers spans a broad range of speed and price. However, the system design philosophy requires that a program written for one machine shall run on any other 360 series machine unless basic I/O timing considerations are a factor. However, this re quires that the least expensive machine have as large an instruction set as the most expensive computer in the series.

One very nice attribute of a micro-programmed machine is the low cost required to add more machine instructions to an existing set. In a micro-programmed machine there is a fixed cost associated with the circuitry to execute and cycle the micro-operations stored in read only memory. However, once the fixed cost is absorbed the addition of more machine instruction involves only the cost of the memory as long as the address bounds of the micro-program controller are not exceeded. Thus the cost of the control for a micro-programmed computer with 100 machine instructions is not that much more expensive than one with ten machine instructions. This is not the case with hardwired machines where the cost of control circuitry is almost a linear function of the number of machine instructions. Thus the lower models of the IBM series 360 are micro-programmed mainly to achieve a large instruction set at reasonable cost. Micro-programming can indeed provide cost advantages and system compatibility.

One of the most fruitful applications of micro-programming techniques is in the area of emulation [5, 6]. Emulation is the execution of a program written for one machine on another machine with no change in the original program.

Emulation was used extensively in the IBM system 360 [6] series computers. The emulation of the IBM 1401 and 7090 series computers, which were being replaced, allowed computing centers and computer users to switch computers and yet retain much of their investment in software for the older machine. Even if it is anticipated that software will be written on the new computer to replace the old software, emulation can be very valuable. Emulation, and hence utilization of the older software, allows gradual replacement of the existing software.

The gradual conversion of existing software to the new computer has two advantages. First, a gradual replacement will not require large amounts of overtime and the added costs that are associated with any rush job. Secondly, it is generally true that there are a limited number of people intimately familiar with the existing programs. Therefore, if a rush conversion is done those most familiar with the programs may not be able to work on all the programs that need converting. The expanded time scale will allow those most capable and most familiar with the programs to actually implement the conversion. It should be pointed out that the reason the software need be rewritten at all is to take advantage of the speed of the new machine. It is very seldom that a host machine can perform a task written in the emulated language as swiftly as the task can be performed in the host's machine language. If the older programs run infrequently enough so that the cost of rewriting the programs exceeds the savings in execution costs, the older programs should not be rewritten, but run under emulation.

In some cases to speed up emulation some minimal amounts of hardware have been added. However, the addition of hardware was for reasons of speed and not completeness. The emulation could still have been performed without hardware, but at a much lower speed.

It is conceivable that emulation [5, 7] may be used in the future so that a new machine may replace several older machines with no intention of rewriting existing programs in the host's machine language, if, in fact, the host machine even has its own machine language. Since the cost of computing power is decreasing and the cost of software preparation increasing, more effort will be made to save existing hardware. It can be envisioned that a person purchasing a new computer will buy the computer on the basis of cost speed characteristics in the language it emulates with little consideration given to any inherent machine language the computer calls its own.

There are also two areas of micro-programming use that appear to have great future potential. The first is machine diagnostics utilizing microprogramming. Currently, most diagnostic programs reside in the main memory. Therefore, if diagnostic tests are to be performed a large part of the computer must be functioning correctly. The minimal amount of correctly working computer hardware would consist of the memory required to hold the diagnostic programs, the channels connecting the CPU to the memory, CPU hardware to sequence the diagnostic program, and CPU hardware to implement the diagnostic tests. If the diagnostic routines are located in the control memory, a smaller percentage of the computer needs to be working correctly to perform the diagnostic tests. The basic philosophy of micro-program diagnostic routines is that they can execute micro-order tests and hence perform diagnostic tests to a finer degree than can a core resident diagnostic program. Also, if the machine is designed with micro-program diagnostics in mind, it is expected that even greater improvements will result.

The second area of micro-programming that holds great promise is the area of special purpose languages. Work done at Northwestern University [3, 8] has shown that it is possible to improve speed by a factor of ten for nonarithmetic operations run on the same machine but under specially oriented languages. Thus, if many languages are provided, it is expected that computing power could be increased. It is expected that to utilize those languages, dynamically alterable control memories will be utilized. This should present no problem in the future, however, since several manufacturers are either currently offering read/write control memories or plan to offer such memories in the future.

4.2.3 Problem Statement

The objective of the study currently being undertaken is to understand the relationships between control memory size, central processor organization, micro-instruction organization, and the execution speed of higher level instructions. One very useful result of the study is the method used to model micro-program control. The model is useful for describing both a micro-control machine and the central processing unit organization. The model is described in Section 4.2.4.

In this study certain simplifying assumptions will be made. The most fundamental of these concerns the central processing unit organization. Many different types of CPU organization will be studied. But for each optimization problem the CPU architecture will remain fixed. The term CPU architecture encompasses the registers, transformation or functional units, and the data paths. We distinguish between data paths and controlled data paths in the following fashion. A data path is a connection capable of being controlled which allows data to flow between the registers and transformation units of the CPU. The most common example of a data path is the register to register data path which allows the contents of one register to be transferred into another register. There is a choice of how the data paths are controlled. In the register-to-register data path all the bits of a register could be transferred or each subgrouping of the register bits may be transferred by a separate control. Once the choice is made, the data paths are controlled. We will always assume the registers, transformation units and data paths are fixed when attempting any optimization. In addition, much of the time we will assume that the data paths are controlled and will not change.

One of the questions that will be studied is that of choosing a branching scheme for the micro-program sequencing unit. The greater the number of unconstrained branches that are allowed, the greater the length of the micro-instruction word. However, greater freedom in branching will often result in shorter programs. Therefore, it may be possible by lengthening the micro-instruction word to allow greater freedom in the method of addressing the next micro-instruction word and shorten the length of the micro-programs. Shortening the length of the micro-programs could result in an overall decrease in the size of the control memory and also decrease the execution time.

Another possibility is to allow several different addresses but constrain the choices of addresses and thereby shorten the micro-instruction word length. One example of this method is Wilkes' suggestion to allow one bit of an address to be controlled by a test flip flop. This allows two address branching using only one address word. Of course, the two addresses are different in only one bit of the address.

How micro-program branching effects the number of micro-instructions required to implement higher level instructions is the relevant question.

4.2.4 A Mathematical Model of Micro-program Control

The problems suggested for study in this report (Section 4.2.3) all involve optimization in one form or another. Before any optimization can be attempted, an accurate description of the micro-program control system to be optimized must be available. With this end in sight, a mathematical model of micro-program control systems will be developed. The model will allow both a standardized and an accurate description of a micro-programmed computer to be realized. The mathematical model will provide the framework needed to allow the optimization to be attempted. In addition, the model will provide two other useful attributes. First, the model is applicable to some nonmicro-control computer systems. Secondly, it is believed that the model aids in the understanding of the operation of central processing units.

The model is subdivided into three parts. The three sub-divisions of the model have been given the names control function, branching function, and testing function.

The control function is concerned with the modification and movement of data within the central processing unit. The control section of each micro-instruction will determine the change in the state of the register of the CPU resulting from execution of that micro-instruction. The control function controls the gating within the CPU, and therefore controls the flow of data from register to register.

The testing function is used to describe tests specified by the micro-instructions. The results of the test determine the sequencing of the micro-program. The tests are specifically limited to prewired tests which may be performed on the states of the register of the CPU or upon status information provided by either the transformation units or input/output devices.

The branching function describes the range of addresses of the next micro-instruction. The manner in which the sequencing of the micro-program is accomplished is determined by a combination of the testing function and the branching function. The branching segment of each micro-instruction contains the possible alternate addresses of the next micro-instruction. The results of the test specified by the testing segment of each micro-instruction determines which address is chosen.

Why should the model of micro-program control be divided into the control, branching and testing functions? First, from a hardware viewpoint the separation is quite natural. The control function is determined by the registers, transformation units, and data paths of the CPU, and is dependent upon the CPU architecture. The branching function is determined by the type hardware sequencing unit used to control the micro-programs and is dependent upon the micro-control sequence unit design. The testing function is determined by both the CPU architecture and the micro-control sequence unit.

Secondly, the problems that are to be studied in this report are well matched to the proposed model of micro-program control. For example, one problem to be studied is that of selecting the best branching function to combine with a preselected control function, testing function, and the desired instruction set. Another problem is to determine the best testing function to combine with a given branching function. A whole class of problems similar to those given above are to be studied, each having the characteristic that either branching, testing or control functions need to be optimized.

References for Section 4

1. M. V. Wilkes, "Microprogramming," Proc. 1958 AFIPS EJCC Vol. 12, pp. 18-19.

2. M. V. Wilkes, "The Growth of Interest in Micro-programming: A Literature Survey," Computing Surveys, Vol. 1, No. 3, September 1969, pp. 139-145.

3. R. W. Cook and M. J. Flynn, "System Design of a Dynamic Microprocessor," Department of Electrical Engineering, Northwestern University, Evanston, Illinois, 1969.

4. S. G. Tucker, "Micro-program Control for System/360," IBM Systems Journal, 6(1967), p. 222.

5. R. F. Rosin, "Contemporary Concepts of Microprogramming and Emulation," Computing Surveys, Vol. 1, No. 4, December 1969, pp. 197-212.

6. S. G. Tucker, "Emulation of Large Systems," Comm. of the ACM, Vol. 8, No . 12, December 1965, pp. 753-761.

7. L. L. Rakoize, "The Computer-Within-a-Computer: A Fourth Generation Concept," Computer Group News, March 1969, pp. 14-20.

8. A. B. Tucker, and M. J. Flynn, "A Dynamic Micro-programmed Processor and its Programming Implications," Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, Illinois, 1969.

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* University of Michigan (Systems Engineering Laboratory) Ann Arbor, Michigan 48104	2a. REPORT SECURITY CLASSIFICATION Unclassified
	2b. GROUP

3. REPORT TITLE
A Study of Information Flow in Multiple-Computer and Multiple-Console Data Processing Systems

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
First Annual Report (April 1969 to March 1970)

5. AUTHOR(S) *(First name, middle initial, last name)*

Keki B. Irani	David L. Hinshaw	Ashby M. Woolf
John W. Boyse	George A. McClain	
Don M. Coleman	Kevin M. Whitney	

6. REPORT DATE May 1970	7a. TOTAL NO. OF PAGES 81	7b. NO. OF REFS 45
8a. CONTRACT OR GRANT NO. F30602=69-C-0214	9a. ORIGINATOR'S REPORT NUMBER(S) Annual Report No. 1	
b. PROJECT NO. 5581		
c. Task 558102	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)* RADC-TR-70-87	
d.		

10. DISTRIBUTION STATEMENT
This document is subject to special export controls and each transmittal to foreign governments or foreign nationals may be made only with prior approval of RADC (EMBIH), GAFB, NY 13440.

11. SUPPLEMENTARY NOTES RADC Project Engineer Rocco F. Iuórno/EMBIH 315-330-7010	12. SPONSORING MILITARY ACTIVITY Rome Air Development Center (EMBIH) Griffiss Air Force Base, NY 13440

13. ABSTRACT

This report documents the achievements from April 1969 to March 1970 of continuing research into the development and application of mathematical techniques for the analysis and optimization of multiple-computer, multiple-user systems.

Section 2 of the report details the progress made in the design of computer storage systems, data base structures and communication networks. Section 3 reports on the research into the application of optimization theory to problems of program organization and program scheduling in multiprogrammed computer systems. Finally, Section 4 reports on efforts to apply mathematical techniques to the analysis and optimization of the hardware configuration of the central processor.

DD FORM 1 NOV 65 1473

UNCLASSIFIED
Security Classification

UNCLASSIFIED
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Queuing Models Multiprogramming Multiplexing Paging Algorithms Microprogramming Data Base Structures File Systems						

UNCLASSIFIED
Security Classification