TX/AUSTIN/CS/TR 48

TR 48
May 1975

COMPUTER SCIENCES
TECHNICAL LITERATURE
CENTER

FILE ASSIGNMENT IN MEMORY HIERARCHIES

by

Derrell Van Foster

May 1975 TR 48

This report constituted the author's Ph.D. dissertation in Computer Sciences at The University of Texas at Austin and was supported in part by National Science Foundation Grant #GJ-1084.

DEPARTMENT OF COMPUTER SCIENCES
THE UNIVERSITY OF TEXAS AT AUSTIN

TABLE OF CONTENTS

CHAPTER PAGE

I INTRODUCTION 1

II GENERAL DEFINITIONS AND BACKGROUND 4

2.1 Hierarchy Topology and Management 5

2.1.1 Management of Executable Memories 6

2.1.1.1 Centralized Execution 7

2.1.1.2 Distributed Execution 9

2.1.1.3 Supporting Memory 10

2.1.2 Management of Non-executable Memories 12

2.1.2.1 Single Device Management 12

2.1.2.2 Multiple Device Management 13

2.2 Models of Memory Hierarchies 15

2.2.1 Models without Internal Queuing 16

2.2.1.1 Stressing Executable Memories 16

2.2.1.2 Stressing Non-executable Memories 22

2.2.2 Models with Internal Queuing 30

III FILE ASSIGNMENT IN MEMORY HIERARCHIES 31

3.1 Introduction 31

3.1.1 Approach 34

3.1.2 Design of System Models 35

3.2 Definitions 38

3.2.1 System Throughput 38

3.2.2 Activity Profile 39

CHAPTER PAGE

			Page
	3.2.3	Degree of Multiprogramming	44
	3.2.4	System Model -- Hardware Characteristics	45
3.3		Analysis Technique	48
	3.3.1	Assumptions	49
		3.3.1.1 Activity Profile	49
		3.3.1.2 System Model	49
	3.3.2	Hybrid Model	51
		3.3.2.1 Simulation Model	57
		3.3.2.2 Analytical Model	58
		3.3.2.3 Iterative Procedure in Detail	60
	3.3.3	An Example	65
3.4		Static File Assignment	74
	3.4.1	Simulation Model Verification	75
		3.4.1.1 One Executable Memory	75
		3.4.1.2 Two Executable Memories	84
	3.4.2	Strategies	93
	3.4.3	Experiments	95
	3.4.4	Results	96

IV THE UT2D PERIPHERAL PROCESSOR LIBRARY -- A CASE STUDY 102

			Page
4.1		Introduction	102
4.2		The Model	105
4.3		Model Parameterization	108
	4.3.1	Activity Profile	109

CHAPTER PAGE

4.3.2 Degree of Multiprogramming 112

4.3.3 System Model -- Hardware Characteristics 113

4.4 Model Validation 115

4.5 Model Results 116

4.5.1 Throughput and Assigned Memory 117

4.5.2 Program Assignment 120

V CONCLUSIONS 123

5.1 Summary of Results 123

5.2 Extensions 125

BIBLIOGRAPHY 127

LIST OF FIGURES

FIGURE		PAGE
2-1	Mattson's Model	17
2-2	Ramamoorthy and Chandy's Model	25
2-3	Arora and Gallo's Model	28
3-1	Customer Satisfaction Function	36
3-2	Activity Profile	40
3-3	System Model	52
3-4	Simulation Model	54
3-5	Analytical Model	54
3-6	Initial File Assignment to IO1 of Example	67
3-7a	Terminating File Assignment to IO1 of Example	70
3-7b	Terminating File Assignment to IO2 of Example	71
3-7c	Terminating File Assignment to IO3 of Example	72
3-8	Improvement in System Throughput of Example	73
3-9	Simulation Model Verification Results	77
3-10a	Worst Case Verification Results	79
3-10b	Worst Case Service Time Distributions	80
3-11a	Best Case Verification Results	81
3-11b	Best Case Service Time Distributions	82
3-12	Verification Results using Multiple Channels	83
3-13	Verification Results using a Single Channel	35
3-14	Verification Results for Memory Queuing	86
3-15	Simulation Model Verification Results using PS CPU Discipline	89

FIGURE		PAGE
3-16	Worst Case Verification Results using PS CPU Discipline	90
3-17	Best Case Verification Results using PS CPU Discipline	91
3-18	Throughput Convergence of Different Loading Strategies for the Load Bound Case	99
4-1	PPU System Model	106
4-2	PPU Simulation Model	106
4-3	Activity Profile of PPU Transient Programs	110
4-4	Results of PPU Transient Program Assignment	118
4-5a	PPU Transient Program Assignment to ECS	121
4-5b	PPU Transient Program Assignment to Disk	122

CHAPTER I

INTRODUCTION

It appears that during every stage of development of computer systems that the demand for computer memory has increased. Two major reasons causing this increase are 1) larger memories have made it possible to attempt larger processing tasks (often these new tasks expose other tasks which require even larger memories), and 2) the development of multiprogramming and time-sharing computer systems. Professor C. V. Ramamoorthy has prophesied [R1] that in the foreseeable future, computer systems will always be able to utilize all of their memories, even a trillion bits. In other words, memory access time and capacity will be a major bottleneck of computer system applications. Consequently, efficient utilization of memory is essential to satisfy the demands for memory.

A memory hierarchy results from physical and economic considerations which make it impossible to provide unlimited storage in single memory. Several memory levels with different access times, capacities, and costs are necessary. Such a hierarchy consists of executable levels such as core memory and non-executable levels (IO devices) such as drums, disks, and tapes. It is a well-known fact that computer system performance is critically governed by the choice of a cost-effective memory hierarchy.

The goal of memory hierarchy management is to use strategies (algorithms which assign information (files) to the level which is warranted by several factors included in their activity profile (defined later). System designers have been trying to exploit the

heterogeneous nature of program and data files so that files which are the least frequently accessed (have a low activity profile) are loaded in slower memories and files that are the most frequently accessed are loaded in fast memories. Management of executable levels has long been considered and many of the associated problems are well understood. However, the parallel management of all IO devices when viewing the computer system as a whole has been performed on a more or less heuristic basis. Strategies for the parallel management of non-executable memories is the subject of the dissertation. By adaptively applying these strategies to dynamically changing workloads, the achievement of dynamic flow of files (described by Opler [O1]) in the memory hierarchy can be realized.

It has long been recognized that the assignment of program and data files to different memory levels is a very critical factor in computer system performance. For example, a critical factor in determining the response time for time-sharing users is the swapping memory to which the files are assigned. This factor is accentuated even more by the shift toward the automated management of the hierarchy. In some cases all responsibilities for the management of a hierarchy have been assumed by the operating system and/or system hardware (making the hierarchy invisible to the user.) Therefore, a controlled evaluation of file assignment yielding strategies employing the major rate determining factors for hierarchy management is justified.

Work on optimization of file assignment to levels of a memory hierarchy show that queuing delays associated with IO devices

is a very important factor affecting system throughput. The dominant effort of this research is the construction, parameterization, and verification of a hybrid model (using both simulation and analytical solution techniques) which includes queuing delays. Note that the term 'hybrid model' refers to the solution technique itself. Simulation is used to provide a realistic model; an analytical model is used to guide the simulation so that evaluation of non-optimal file assignment can be avoided. Experiments utilizing the hybrid model as a tool to isolate the cause-effect relationship of important domain variables on system throughput are performed. Memory management strategies are then proposed from the insight gained from these experiments.

The approach used here differs from previous approaches in at least three ways: 1) the entire system is included in attempt to optimize total system performance, 2) queuing delays are accounted for at all servers in the model, and 3) non-executable memories are managed on a logical information block (file) basis rather than on a physical information block (page) basis. All significant holding times at all servers throughout the entire system is included in this approach. However, many important simplifications (such as first-come-first-serve queuing disciplines) are used.

Chapter II contains the general definitions and background (including citations to the literature) necessary for optimal file assignment generation techniques discussed in Chapter III. Chapter IV applies these techniques to the files of the UT2D operating system.

CHAPTER II

GENERAL DEFINITIONS AND BACKGROUND

The term 'memory hierarchy' is a convenient way of referring collectively to the various levels used in computer systems. Each successive storage level contains a memory with characteristics which follow these two fundamental principles: 1) the larger the memory, the slower the access time (i.e., the average time needed to locate a storage unit) and 2) the slower the access time, the lower the cost per storage unit. The memory hierarchy itself is necessitated because of physical and economic considerations, that is, for a given performance measure, a computer system which satisfies this measure for the least cost is desired. System designers must balance the cost savings accruing from a memory hierarchy against the performance degradation caused by the hierarchy.

In this chapter, the two background topics discussed are 1) hierarchy topology and management, and 2) models of memory hierarchies. The main purpose of detailing this background of memory hierarchies is to provide a perspective as to where the memory management strategies presented in the next chapter belong with respect to the problem domain.

2.1 Hierarchy Topology and Management

This section is divided into the topics of managing executable and non-executable storage levels of a memory hierarchy. In theory, a distinction between executable memory and non-executable memory is its access time, an executable memory having a fast access time (commensurate with the CPU cycle time) and a non-executable memory having a slower access time. In other words, the faster memories are executable. In practice, the distinction is whether or not the CPU instruction fetch circuitry is connected to the memory drives. The management of executable memories has been the primary concern of the literature [A2, A4, B1, B3, B4, B5, C4, D2, D3, D4, D5, F1, F2, L1, L2, R3, T2, W1, W2, V2] leaving the (practical) management of non-executable memories largely ignored when this set of memories (IO devices) is incorporated into an integrated computer system. However, as is noted later, the management of individual IO devices under non-varying workloads has been fairly well explored [D1, F3, S1, T1].

2.1.1 Management of Executable Memories

The management of executable memories is discussed using the following two approaches: centralized versus distributed execution. The distinguishing feature is that centralized management allows execution from the first level only, whereas distributed management allows execution from either of the first two levels. There exist many papers in the literature concerning centralized execution [B1, B3, B4, B5, D2, D3, D4, F2, L2, R3, T2, W1, W2] but not as many concerning distributed execution [A2, A4, C4, D4, D5, F1, L1, V2]. Both approaches employ a memory hierarchy normally consisting of no more than three levels, of which the first two are electronically accessed. The third level, a supporting memory, may be electronically or mechanically accessed (e.g., large capacity storage (LCS), extended core storage (ECS), drum).

The common goal of management strategies of executable memories is to keep the most frequently accessed storage block in the fastest memory level. Centralized execution achieves this goal by using management algorithms which produce no access queuing due to either overflow removal (with an instruction stack) or mandatory updating of all copies of a modified storage block(with a cache). A totally general solution to the selective transfer problem associated with distributed execution is currently unknown. Consequently, optimal storage block management can only be approximated (with the core/LCS combination). The third level, a supporting memory, is managed by algorithms which may require access queuing (especially if mechanically

accessed memories are used).

The common memory management strategies (for centralized, distributed, and supporting) are now summarized. Centralized execution is discussed from the viewpoint of the first memory level. Distributed execution is discussed from the viewpoint of the core/LCS combination. Strategies for the management of supporting memory are only highlighted.

2.1.1.1 Centralized Execution

A few modern processors employ an instruction stack which provides storage for the most recently referenced instructions [T2]. There are two major advantages of this strategy. First, the instruction fetch is much faster in many cases especially since it allows processing to proceed at the speed of the faster stack for loops contained within the stack. Second, fewer storage conflict conditions are possible since fewer actual storage references are made. Its major disadvantage is its small capacity. However, even if a program's working set cannot be contained in the stack, performance is not appreciably degraded due to the small disparity in access times between the stack and core memory.

Another current memory management strategy employs the use of a high speed buffer memory, called a cache, between the CPU and core memory to partially compensate for the disparity of their access times [L2, W1]. The cache is managed so as to hold the contents of those portions of core memory that are currently being accessed. Most CPU memory fetches can then be handled by referring to the cache, so that

most of the time the CPU has a short access time. The net effect of the cache is to reduce the number of required core memory fetches, or to make core memory 'look faster' as seen by the CPU, in that the expected access time to core memory is smaller. The result is the principal advantage of this strategy: a program's working set accumulates in the cache.

Closer inspection of cache operation illuminates a major disadvantage. Two CPU cycles are required to fetch data residing in the cache. The first cycle is used to examine registers and validity bits to determine if the data is within the cache. If so, the second cycle is used to retrieve it. If the data is not in the cache, additional cycles are required while the page is loaded from core memory. Rollout of the replaced data from the cache to core memory is never necessary since store operations always cause core memory to be updated in most cache designs. Hence, a major disadvantage is whenever a page in the cache is modified, its copy in core memory must also be modified (at core access time) causing an enforced wait upon the CPU. However, the mandatory updating produces no access queuing for core memory. It should be noted that if a particular working set generates much storing, the cache offers little advantage.

Another problem of using the cache occurs in a multiprogramming environment. If the degree of multiprogramming is one and if the application program is one in which the accessing of memory is not random but follows localized patterns, then the use of a

cache can normally be justified. If the degree of multiprogramming is large, then determining the accessing patterns to memory is more complicated (even if all jobs are the same application program) and use of a cache is more difficult to justify. One factor governing the competition for the cache is the CPU scheduling algorithm. The cache would be defeated if the time quantum for a job expires just when that job has retrieved its working set into the cache from core memory. The fact that typical jobs when run separately do not produce random memory accessing does not imply that typical jobs when run concurrently do not produce (near) random memory accessing.

Finally, like the instruction stack, the cache reduces the effects of channel interference upon core memory. However, if the entire computer system is IO bound, the cache again offers little advantage.

Management strategies of a single core memory have long been a topic that has been widely explored in the literature [B1, B3, B4, B5, D2, D3, D4, R3, W2]. One of the best known of these is the work of Belady which evaluates the performance of paged memories in terms of page fault rate as a function of page size and core memory allotment. Denning has also contributed the highly regarded concept of working set to page replacement.

2.1.1.2 Distributed Execution

In the distributed execution approach, the CPU may access

and execute information stored either in core memory or LCS. The goal of the management strategy is to keep the most frequently accessed information in core. Consequently, the selective transfer problem immediately arises; when should information be moved into LCS? Some of the conclusions incorporated into management strategies are tradeoffs based on the cost of not moving the information and accessing at a slower rate, and the cost of moving the information to access it at a fast rate [C4, D4, D5, F1]. The strategies are based on having a value for the frequency of access to the desired information. This value is generally not known and can only be approximated by measurements taken either statically [C4] or dynamically [F1]. Because of the disparity of access times between the CPU and LCS is even larger than between the CPU and core memory, the transfer of information between these two levels (which is performed by the CPU) is very time consuming. Without the use of additional hardware to make the core/LCS combination a self-contained memory unit, it appears that distributed execution is not a likely path for future architecture.

2.1.1.3 Supporting Memory

The supporting memory level of executable memory is normally comprised of either a bulk core memory such as ECS or LCS, or a fast IO device such as a paging drum. This level is primarily used as a swapping medium in a multiprogramming/interactive environment. Both of the bulk core memories are at least three orders of magnitude

faster than the fastest IO device. ECS has an advantage over LCS in that its transfer time, a function of the number of words moved, is non-linear (e.g., it takes the same time to transfer ten words as one word, but it takes longer to transfer 500 words than one word). Consequently, further improvements in response time of bulk core memory probably must come from the use of 'ECS-like' block transfer memories. Current management strategies of bulk core memories involves managing them as a fast IO device of zero latency time and of very low probability of delay due to access queuing [B1, F1, F2, L1, V2]. Management strategies of fast IO devices are the same as those of single device management strategies of non-executable memories and are discussed later under this topic.

2.1.2 Management of Non-executable Memories

Some strategies for managing single non-executable memories have long been practiced, in particular, preplanned data layout on mechanically accessed IO devices so as to reduce latency time. However, concepts underlying queuing algorithms [D1, F3, T1] and device folding [S1] have only recently been studied. The management of multiple IO devices when viewed as a system has been largely ignored. New hardware such as device-to-device channels has further added to this management problem.

2.1.2.1 Single Device Management

The goal of preplanned data layout on mechanically accessed IO devices is to keep the information that is most likely to be referenced next almost under the read heads by the time it is referenced. (This strategy was used on the IBM 650 drum memory). Of course, preplanned data layout is a static technique and depends upon correctly collecting/predicting external reference characteristics. In a multiprogramming environment, this strategy is most amenable if a single IO device can be assigned to an individual application program.

Several queuing algorithms currently exist which optimize the response time of the IO device which they schedule. These algorithms include shortest-seek-time-first, SCAN, and minimal-total-processing-time policies [D1, F3, T1]. For the most part, these algorithms are implementations of results from non-executable single

server models discussed in the next section. A tradeoff must be made between the overhead required to execute these algorithms and the expected improvement in response time. In some computer systems, it does not pay to schedule individual IO devices either because the overhead of the algorithms is too large or because the mean queue length of the device is small.

Folding of a rotating IO device is an allocation and accessing management strategy which sacrifices device capacity to gain a reduction in access time [S1]. An IO device is folded by keeping several copies of an information block equally spaced over the device's surface so that when an information block is accessed, it is serviced by retrieving the block closest to the read heads. Experience with this strategy is somewhat limited especially concerning the overhead of maintaining multiple copies of an information block on a potentially slow IO device.

2.1.2.2 Multiple Device Management

Heretofore, multiple device management of non-executable memories has been largely ignored. An exception to this is that the same information block may be allocated across a device set in order to reduce access time (e.g., a file is allocated across a successive disk sequence of a disk set (on a one track/disk basis, if possible) to attempt a reduction in seek time when accessing the file). The success of this strategy is highly correlated with external referencing characteristics.

The strategies presented in the next chapter are those for managing non-executable memories. They differ from previous strategies in at least three ways. First, in attempts to optimize total system performance, say system throughput, they include the entire system. In other words, they do not view the non-executable memories as stand-alone memories. Second, they account for queuing or waiting time delays not only for the individual devices, but also for the CPU, for the channels, and for executable memory. Buzen [B7] has shown that whenever a choice of IO devices can be made, it should be made so that a greater proportion of accesses flows to the fastest device (so that it is always busy) for optimal system performance; the faster the device implies the greater its utilization. Strategies for proportioning these accesses for optimizing system throughput are given. Third, the non-executable memories are managed on a file or segment (logical information block) basis, rather than on a page (physical information block) basis. This is a more realistic situation for many operating systems.

New hardware concerned with non-executable memories greatly advance the management of these memories on a dynamic basis. Specialized hardware is already available for the management of particular IO devices (e.g., CDC's Direct Data Path allows information to flow into ECS from a lower storage level without flowing through core memory). These advances make it easier for the more frequently loaded files to percolate to faster non-executable memories. However, models incorporating these advances are currently embryonic.

2.2 Models of Memory Hierarchies

This section classifies the models of memory hierarchies into two categories: models which assume no internal queuing, and models which permit internal queuing. Internal queuing means that queues are allowed to form before the servers inside the model. However, for open single server models which permit the formation of a queue, the queue itself is considered to be outside the model and is treated as a holding area for arriving requests. When no internal queuing is assumed, the literature contains many reports concerning the models of memory hierarchies [A1, A3, A4, C6, C5, D1, F3, M1, R2, T1, T3, Y1]. But when internal queuing is permitted, relatively few reports are found [G1, S2]. Such models, be they exact or approximate, are currently an area of active research.

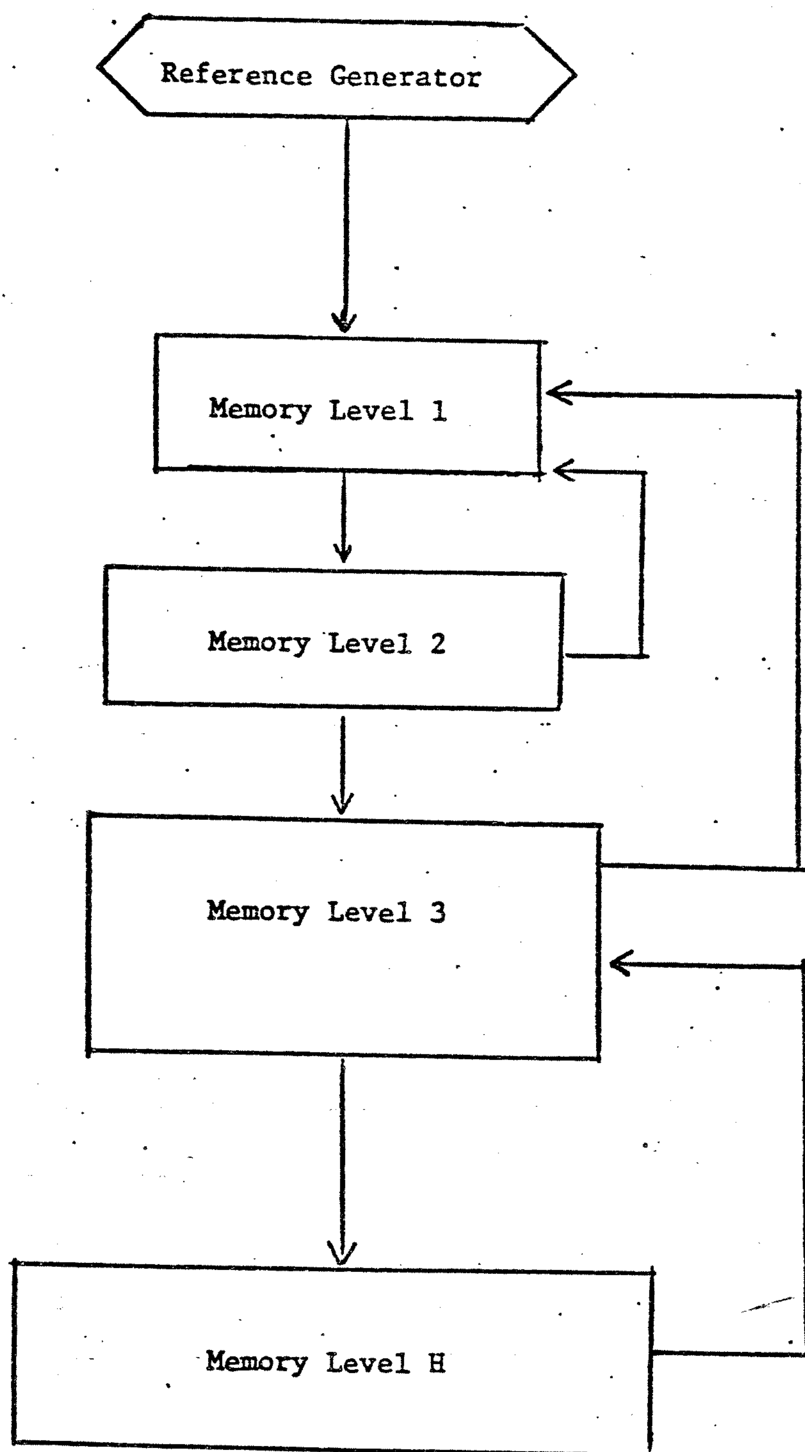
2.2.1 Models Without Internal Queuing

Models which assume no internal queuing are discussed using the following two approaches: those stressing the modeling of executable memories [A3, C5, M1, T3], and those stressing the modeling of non-executable memories, the IO devices [A1, A4, C6, D1, F3, R2, T1, Y1]. Special notice should be taken of the models of Ramamoorthy and Chandy [R2], and Arora and Gallo [A4] because the analysis technique described in the next chapter employs an extension of their models.

2.2.1.1 Stressing Executable Memories

The model used by Mattson, et. al., [M1] yields an efficient procedure that determines the frequency of access to each level in a memory hierarchy as a function F of the capacity of each level. It assumes that a page address trace has already been collected and that the replacement algorithm is a stack algorithm. The effects on F of other variables such as page size and number of levels in the hierarchy can be determined during the same pass of the address trace. Consequently, this is an analysis/evaluation model of a single address trace.

The model is given in Figure 2-1. The address trace has been replaced by a page reference generator. Note that this abstraction does not differentiate between CPU and channel memory references. Since centralized execution is assumed, the model



Mattson's Model

Figure 2-1

operates as follows: At a given time, the reference generator presents a request for a page. Under demand paging, if this page is located in the first level, the reference proceeds and no page movement occurs. Otherwise the referenced page is moved into the first level from a lower level. The retrieval time depends on the level number (distance) of the nearest level containing the referenced page. If the first level is already full, a stack replacement algorithm (normally least recently used) selects some other page for deletion. Downward percolation from the first level is never necessary since whenever a page is modified in the first level, its copies in all lower levels (except the last) must also be modified. However, percolation between the next-to-last level and the last level must be performed.

The function H(C), called the success function or the hit ratio function, is the probability of finding a referenced page already loaded in the first level memory of capacity C of the memory hierarchy. Briefly, it can be evaluated in one pass of the address trace as follows:

a. maintaining a stack of the referenced pages ordered from most frequently used to least frequently used
b. measuring the distance from the top of the stack to the referenced page (each new reference coming from the address trace)
c. generating H from the frequency of these distances (values of H for all capacities are generated

simultaneously since capacities are fixed distances from the top of the stack).

It is important to understand the derivation of this function since it is an integral part of other works [C5, T3].

Only two minor faults can be found with the assumptions of this model. First, it assumes that no memory interference exists between the references of the CPU and the channels. Furthermore, it assumes that channel references are directed to the first memory level. This is not the case in the real system behind this model (i.e., the IBM 360/85 cache/core/drum). Second, it assumes that no temporary storage is required for staging a page from the last level to the next-to-last level of the memory hierarchy. In the real system, this is again not the case.

Three points must be stated concerning the potential misuse of this model. First, the address trace which is used to generate the success function H(C) is assumed to characterize the workload. However, common address traces are collected in a uniprogramming environment and represent a particular workload only. The success function may be unpredictable if the address trace is collected in a multiprogramming environment with a large degree of multiprogramming and inappropriate management policies. (The designer must be aware of how closely the composite success function of many jobs approximates the success function of a single job.) This important assumption is often ignored. Second, this model may lull the designer into managing the entire memory hierarchy on a page basis. This is impractical for a large capacity hierarchy, one reason being

the enormous page location tables which must be maintained. Third, this model assumes no internal queuing (e.g., no (or constant) interference at the drum level). This is immaterial for generating the success function (since it is not a function of access time) but may lead the designer into over-specifying the capacity of the next-to-last level in order to prevent queuing.

The major contribution of this work aside from producing an efficient procedure that evaluates the frequency of access to each level in a memory hierarchy is that it develops a class of replacement algorithms (stack algorithms) such that the probability of a page fault is a monotonic decreasing function of memory capacity. This is not always true, contrary to intuition. For example, FIFO replacement is known to have an increasing section in its fault probability curve as memory capacity is increased for particular address traces [B5]. The importance of this work is also emphasized by noting that it is the father of many design models [T3, G1, C5].

Chow [C5] generalizes the success function H in order to obtain analytical solutions for optimal design problem of memory hierarchies. In the solution process, the following three assumptions are made: 1) each memory level is characterized by its access time, capacity, and cost per unit storage, 2) the memory management strategy is completely characterized by the success function H, and 3) the fault rate function F (=1 - H) and the memory cost function, both being nonlinear, are representable by power functions. A closed form solution for the minimum average access time per memory reference is obtained as a function of system capacity and cost. The solution technique uses the theory of

geometric programming to solve this nonlinear programming problem. Using this solution, answers to questions such as 'What is the optimum capacity of each storage level?' and 'What is the optimum number of levels in the hierarchy?' are obtained.

This set of solutions on the optimization of memory hierarchies is the most mathematically rigorous to date, but is most applicable in a uniprogramming environment. For this solution set to be completely valid in a multiprogramming environment, three implicit assumptions about the nature of a computing system must be made. First, all programs, including the operating system, must have the same success function characteristics. While this assumption aids in mathematical tractability, it disallows a dynamically changing workload. How close a static (but general) success function approximates a dynamic success function is a subject for future research. Second, the degree of multiprogramming must not be too large and inappropriate management policies must not be used to avoid making H unpredictable (which defeats the memory hierarchy management strategies). And third, since a constant access time is assumed for the last level, queuing at the last level is disallowed. For counterexample, in an actual computing system a given program is allowed to execute until it references information in the last level. While the referenced information is being transferred into the next-to-last level, the CPU is switched to another program which may immediately do the same thing. The mean waiting time could be used in the calculation of the constant access time except that it depends on the capacity of

the next-to-last level, a variable which is being solved for. So this solution set may cause the designer to over-specify the capacity of the next-to-last level.

2.2.1.2 Stressing Non-executable Memories

The second approach of models which assume no internal queuing is that of stressing the modeling of non-executable memories, the IO devices [A1, A4, C6, D1, F3, R2, T1, Y1]. This approach can be further classified into single server models [A1, C6, D1, F3, T1, Y1] and multiple server models [A4, R2]. Management algorithms for ordering a queue of references according to some discipline (such as shortest-seek-time-first) are proposed from single server modeling results. By management algorithms is meant the scheduling of waiting memory references according to the current state and the characteristics of the IO device. These models do not consider the nature of the computing system that generates the references (i.e., they are stand-alone models), nor do they include intrafile organization. The multiple server models extend results by proposing management strategies for more than one device under the assumption of no queuing delays. They also consider on a very general basis the processes that generate the references to the hierarchy.

Primarily, single server models concerned with scheduling moveable head disks are contained in the reports by Denning [D1] and Teorey and Pinkerton [T1]. Their models are used to evaluate

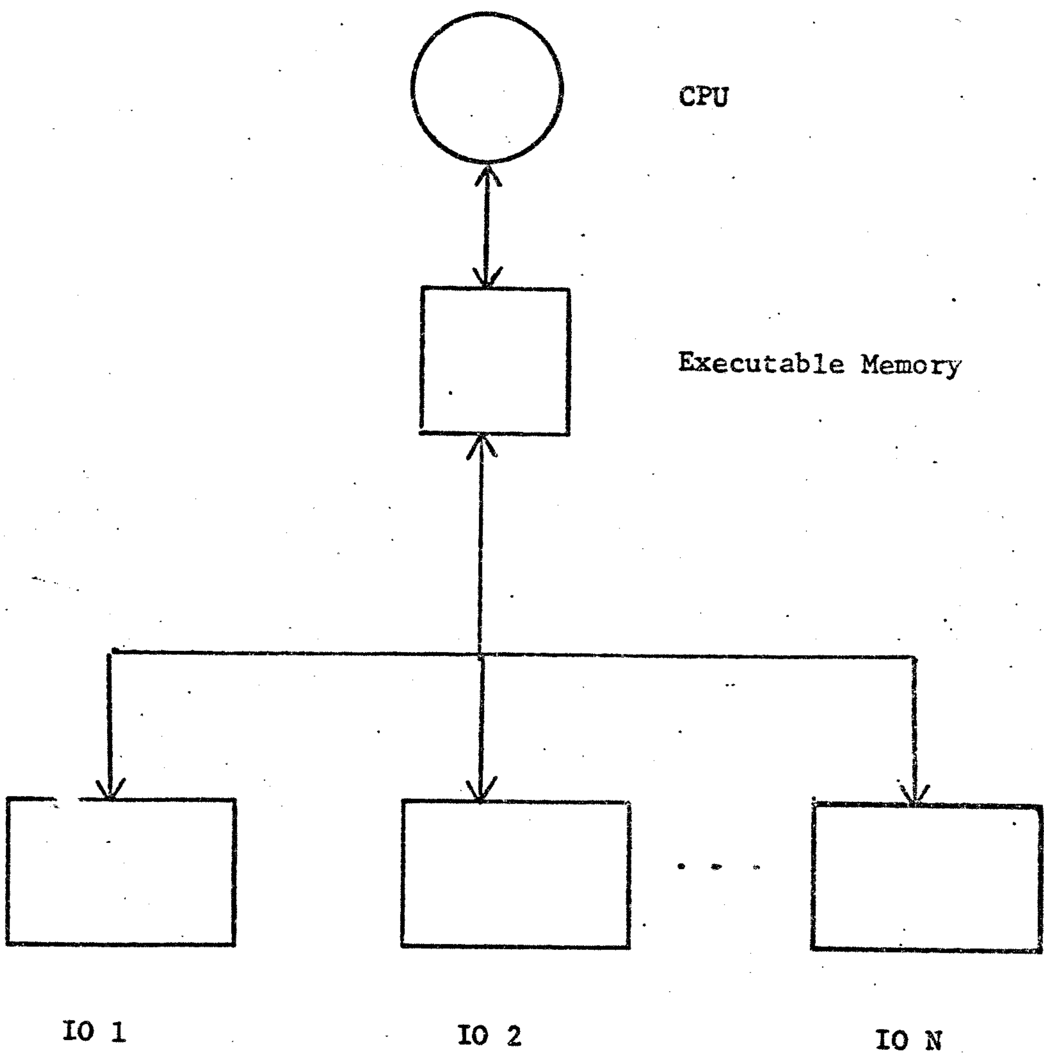
many queuing disciplines over the first-come-first-serve (FCFS) discipline (which induces a random seek pattern since it does not take advantage of positional relationships among the queued references). The shortest-seek-time-first (SSTF) discipline is rejected because it may allow discrimination against those references whose requested information requires a long seek time (i.e., the variance of the mean seek time is large). With the SCAN access method [D1], the disk arm sweeps back and forth across the cylinders changing direction only at the inner and outer cylinders to avoid the discrimination of SSTF (i.e., to reduce the variance but at the expense of the mean). Teorey and Pinkerton note that under heavy loading conditions (i.e., the disk is a system bottleneck), the difference in performance between SSTF and SCAN is nearly indistinguishable. Of course many other queuing disciplines are possible, but the above three are probably the most popular and most widely discussed in the literature.

A single server model concerned with drum analysis is in an early report by Coffman [C6]. He provides an exact analysis of an approximate drum model under the assumption of Poisson arrivals. Results for particular disciplines are derived which represent bounds on the performance of actual drum systems. A most recent model concerned with the scheduling of both drum and disk is the minimum-total-processing-time (MTPT) discipline reported by Fuller [F3]. The MTPT discipline orders the set of pending IO

requests (upon each new request arrival) such that the order minimizes the total processing time for the set. However Fuller notes that, even for the efficient implementation of the MTPT algorithm, the difference in drum performance between shortest-latency-time-first and MTPT is nearly indistinguishable under heavy loading conditions.

The multiple server models are used in the optimization of computer systems with more than one IO device [A4, R2]. The object is to minimize cost per access to the memory hierarchy [R2] or to maximize throughput of the system [A4]. However, their solution techniques impose assumptions such as no internal queuing which results in their proposed management strategies not being widely adopted. An extension of their models which includes internal queuing is given in the next chapter.

Ramamoorthy and Chandy [R2] describe a technique for cost-performance evaluation of a memory hierarchy in terms of a total memory cost per memory access, the selection criterion being that of the lowest ratio. Their model, consisting of several IO devices (non-executable memories) all directly connected to an implicit executable memory and that to a CPU, is given in Figure 2-2. Note that memory-to-memory channels do not exist. The model operates as follows: the CPU retrieves a unit of information from a given memory level, transforms it, and then stores it back into the same level. In the solution process, the following two major assumptions are made: 1) each memory level is characterized by its access time,



Ramamoorthy and Chandy's Model

Figure 2-2

capacity, and cost per unit storage, and 2) memory requirements and frequencies of usage for programs and their associated data are known a priori (like the success function H for a particular workload). An algorithm for the minimum average access time per memory reference is obtained satisfying system capacity and cost constraints. (Note that the cost of associated hardware such as channels and controllers is not included in the model.) Then integer programming techniques are used to solve for optimum capacity of each memory when that memory is constructed of individual modules.

For these solutions to be valid in a multiprogramming environment, the following additional assumptions must be made. First, the activity profile must characterize the activity of all appropriate programs and data, including the operating system. Also, the activity profile is assumed to be constant (static) in time (i.e., it is assumed to be a steady state representation of activity). This disallows a dynamically changing workload. Second, the probability of accessing any block must not depend on past accessing history. Finally, competition for resources (which induces queuing) for the programs must be ignored.

The major contribution of this work is not so much that it is the first to produce quantitative solutions to this problem (by applying operations research techniques) as it is the first to quantitatively formulate the problem. In other words, it is the first to define the optimization problem, the objective function,

the variables, and the constraints. Note the similarity between this work and the later work of Chow [C5] especially since both assume a static workload in a uniprogramming environment.

Arora and Gallo [A4] extend the use of the activity profile to aid in maximizing the throughput of the entire computer system. A typical system configuration is given in Figure 2-3. Since optimal system throughput greatly depends on where program and data files are loaded and processed, a linear programming (LP) problem is formulated whose objective is to minimize the sum of the following times for all files:

a. instruction execution times in the executable memories,

b. access and transfer times from the non-executable memories.

Note that the above sum does not include queuing time for system resources.

The variables of the LP problem are as follows:

1. hardware parameters:

   b_m : transfer time from the mth executable memory,

   a_n : access time for the nth non-executable memory,

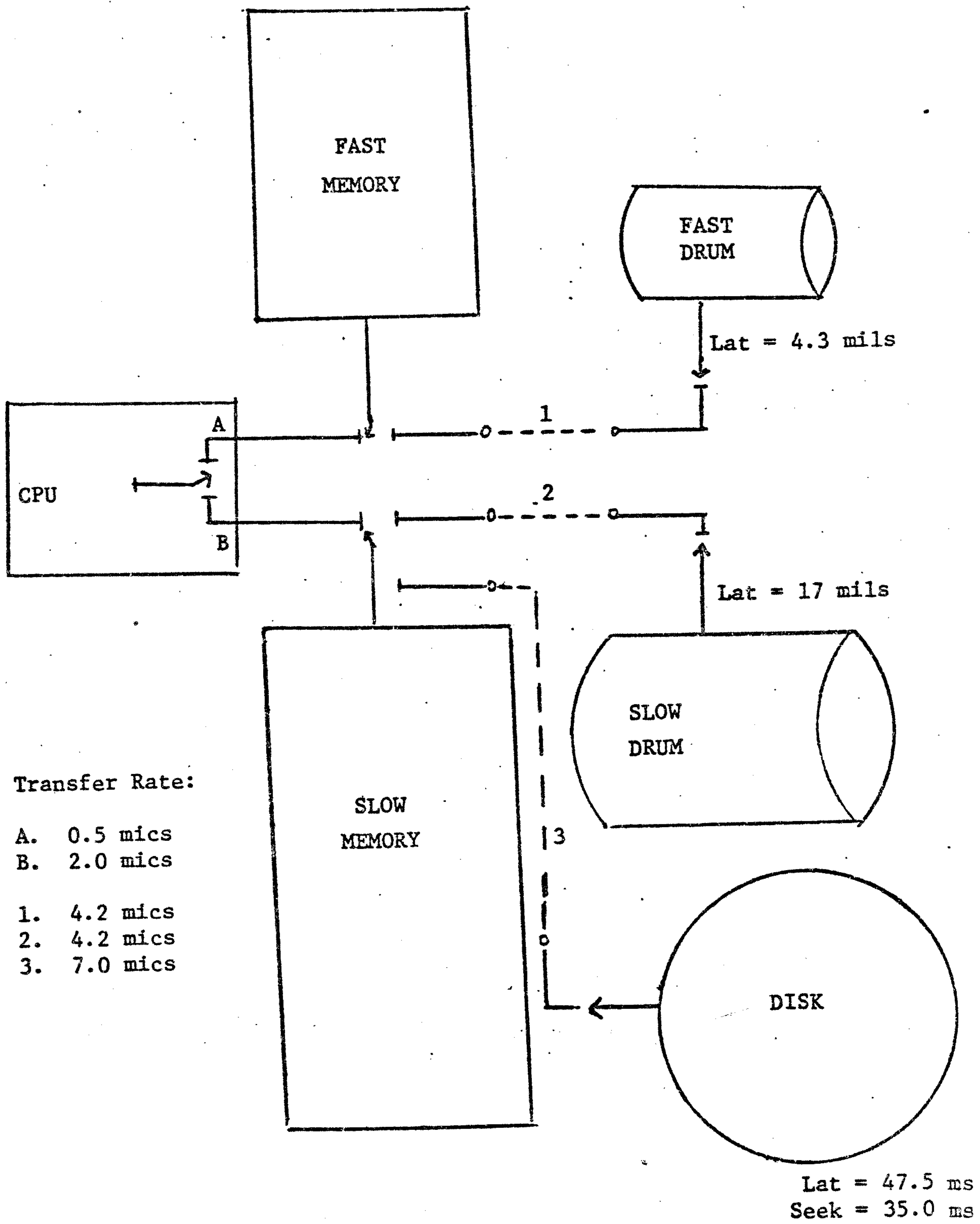
   $c_{n,m}$: transfer time from the nth non-executable to the mth executable memory.

2. job characteristics (given in the activity profile):

   v : file size

   r : average record size that must be loaded

   i : average number of executed instructions



Arora and Gallo's Model

Figure 2-3

f : frequency of file execution

f' : frequency of file loading.

Note that f' is a function of many variables, including where a file is loaded which is being solved for. By defining a loading factor per file as a time (execution time + access time + transfer time) per unit size ratio, a simple loading strategy is shown to be optimal and operates as follows: the file with the greatest loading factor is loaded into the fastest memory, the next file with the greatest factor into the same memory, and so on until the capacity of that memory is exceeded, then the process is repeated on the next fastest memory.

This solution is valid in a uniprogramming environment where the loading frequencies (f') are known. Assuming prior knowledge of the loading frequencies, this strategy loads the most frequently loaded files (everything else being equal) in the fastest memory. So if only two files are contained in the activity profile and the capacity of the fastest memory is large enough, both files are loaded there, even if a second memory is available and is only slightly slower than the first. In a multiprogramming environment, this would normally produce sub-optimal system throughput. The model of the next chapter rectifies this problem.

2.2.2 Models With Internal Queuing

The literature contains relatively few reports of models which permit internal queuing [G1, S2]. The most notable example of such a model is that of Shedler [S2]. The system modeled is a CPU accessing and processing files stored in a two-level memory hierarchy, a drum containing a file directory and a disk containing the files themselves. A Markovian model is constructed and operates as follows. After jobs are served by the CPU according to a geometric distribution of service times, they flow to the drum for a geometrically distributed number of references in the file directory, and then flow to the disk to be served by an arbitrary distribution. The power of this technique is illustrated by the relatively simple derivation of utilizations, mean queue lengths, and mean waiting times. It should be noted that a job has only one path to follow after it determines where the requested file is located (from the directory on the drum). Thus optimal device usage is not a problem.

A model permitting internal queuing is given in the next chapter. The model also includes multiple non-executable memories (e.g., parallel disks in Shedler's model). The key to the analysis technique is optimally proportioning file accesses among the IO devices. When this is done, optimal system throughput is obtained.

CHAPTER III

FILE ASSIGNMENT IN MEMORY HIERARCHIES

3.1 Introduction

Fundamental to the generation of file assignment strategies in memory hierarchies is a model which maps domain variables, notably the job characteristics, the degree of multiprogramming (potential job interference due to queuing delays), and the system model, into values of system performance metrics, say a value of system throughput. If the jobs themselves are requests for the loading (transferring a record of a file from secondary memory into executable memory before processing, if not already there) and processing (executing program files or accessing intra-record information of data files) of files then the job characteristics are commonly given in an activity profile, one entry per file. Each entry contains characteristics of a given file (e.g., the frequency with which a file is requested by different jobs. See section 3.2.2). The object of the model, generating optimal system throughput, is accomplished by deciding 1) where individual files are to be loaded and 2) where they are to be processed in the memory hierarchy. This decision process is referred to as file assignment. Note that the optimization of file assignment is the fundamental problem of analysis/evaluation of memory hierarchies. The analysis technique described later is an approximate solution to this problem.

The model which permits internal queuing is then used as a primitive to gain insight into the load factors (of the files in the

activity profile. See section 3.2.2), the degree of multiprogramming, and the hardware characteristics (of the system model) effecting system throughput and corresponding file assignment. Memory management strategies (practical assignment algorithms), a function of load factors, degree of multiprogramming, and hardware characteristics, are given for obtaining file assignment under using a static activity profile. (This implies that the time period for measuring the requests for all files is sufficiently long so that they may comprise a single file request distribution.) These are called static memory management strategies.

Multiple file request distributions can be considered by using a dynamic activity profile (i.e., a profile in which the frequency of requesting a file is not known a priori and varies with different time periods). A dynamic activity profile may be viewed as a sequence of static activity profiles, each being valid for one time period. Consequently, a strategy is suggested for managing dynamically the non-executable levels of the memory hierarchy. It uses the same analysis technique as that used for a static activity profile, but it is triggered dynamically by monitoring file accesses. This strategy attempts to adapt the computer system to the workload and not the workload to the computer system.

It is an important simplification to separate the management of executable memories from the management of non-executable memories. This allows more frequently processed files freedom to percolate to faster executable memories without regard to their loading require-

ments, and the more frequently loaded files freedom to percolate to faster non-executable memories without regard to their processing requirements. (It is assumed that appropriate data transfer paths exist among the memory levels.)

It has been demonstrated that executable memories and specialized IO devices can be managed on a page basis [C5, G1, M1, T3]. However, management of the entire non-executable memories on a page basis appears doubtful primarily because of enormous page location tables that must be maintained for large capacity hierarchies. Consequently analysis techniques assuming that the entire hierarchy is managed on a page basis seems implausible at the present state of the art. Note that the strategy referenced above manages non-executable memories on a file or segment (logical information block) basis, rather than on a page (physical information block) basis.

3.1.1 Approach

The management strategies are generated in the following steps (corresponding to sections of this chapter):

A. Analysis Technique

A model which includes queuing delays is constructed which maps three throughput variables, activity profile, degree of multiprogramming, and the system hardware/software characteristics into system throughput and corresponding file assignment.

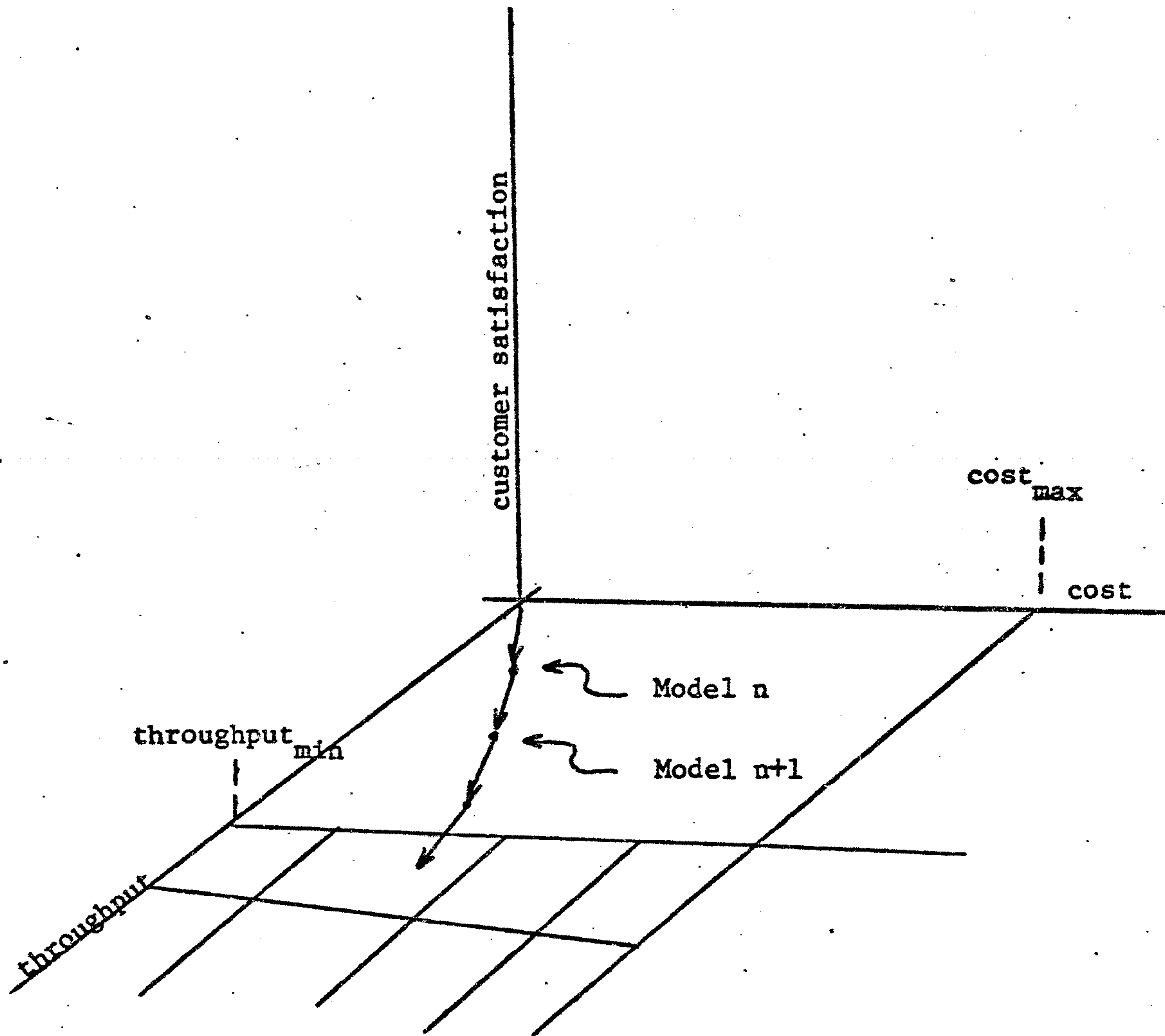
B. Static File Assignment Strategies

1. The cause/effect relationship of hitherto unstudied throughput domain variables on file assignment is isolated via the model constructed in A.
2. Memory management strategies are given for static file assignment from the insight gained in B.1.

3.1.2 Design of System Models

Before proceeding to the definition section, a brief look at computer system design is given. The reader should observe two points: 1) the model of A is a primitive of the hierarchy design process, and 2) the primitive does not spend effort to minimize system model cost; it only evaluates the system model for performance metrics.

The fundamental problem of computer system design is mapping design inputs (such as job characteristics, individual hardware/software component costs and timings, maximum cost, minimum throughput) into a proposed system while at the same time ensuring customer satisfaction, a function of several variables including various performance metrics (e.g., throughput, response time, etc.) and total cost of the proposed system. Let the domain variables be throughput and cost and let their corresponding input constraints be $throughput_{min}$ and $cost_{max}$. The customer is satisfied when any point in the shaded area of Figure 3-1 is spanned by a proposed system. Since there are normally many systems that ensure satisfaction, the designer can choose one according to other strategies (such as maximizing the cost of a system for the given $throughput_{min}$ constraint at the expense of the customer). Here, the strategy of minimizing the cost of the system for the given $throughput_{min}$ constraint is assumed not only to make the system the most competitive possible for the design inputs, but also to eliminate the need of minimizing system cost at primitive levels of the design process. For example, the problem of memory sizing does not arise [as in R2]



Customer Satisfaction Function

Figure 3-1

if the design process steps from one intermediate minimal cost system model to the next minimal cost system model which increases throughput in its attempt to generate a proposed system.

For the purpose of solving memory hierarchy design problems such as those described above, the analysis primitive is used to calculate performance. The design procedure is composed of a set of nested loops, one for each design parameter. The innermost loop uses the parameter settings and the primitive to compute a cost-performance product. Normally, the hierarchy yielding the lower product (subject to customer cost-performance constraints) is selected. Intelligent search strategies are employed to reduce search time. Note however that the analysis primitive is the subject of this chapter.

3.2 Definitions

3.2.1 System Throughput

Throughput is the rate at which requests are serviced per unit time and is measured at various points within the system model. System throughput is throughput which is measured only at points corresponding to job arrival/departure. Throughout this chapter, the throughput optimization problem is approached from the viewpoint of optimizing the throughput of the entire system (measured at the CPU), and not of individual subsystems. (Often when individual subsystems are optimized and then interfaced together, the entire system performs in a sub-optimal manner. For example, a sophisticated scheduling algorithm is used to minimize waiting time of requests at an individual IO device. But when the device is included within the system, the mean queue length is one. Hence, the throughput of the CPU is sub-optimized due to the overhead induced by executing the algorithm.)

The system throughput function, being in non-closed form, is defined as a function of the following three domain variables: the activity profile of the files, the degree of multiprogramming, and the system model. All three variables must be given since the analysis process does not specify the workload nor design the system model; it merely analyses it for system throughput and file assignment.

3.2.2 Activity Profile

The first domain variable is the activity profile, a representation of the job characteristics abstraction. In this case the following five parameters are included in the profile:

a. (Serial) Reusability of a File (i.e., single copy sharable)

   Its values are:

   P - Program File, Not Reusable

   D - Data File, Not Reusable

   PR - Program File, Reusable

   DR - Data File, Reusable

   Note that reusable files which are loaded are not reloaded for additional usage and that program files which are loaded but otherwise unused are not swapped-out on replacement.

b. Request Frequency of the File

c. Instructions Executed/Request
   (for CPU service time)

d. Words Loaded (record size)/Request
   (for loading service time)

e. Volume of the File

The last four parameters are used by Arora and Gallo [A4] and represent easily measured statistics of a file in a transaction oriented system. Refer to Figure 3-2 for an example of an activity profile.

j	Reusability	Frequency	Instructions	Record Size	Volume
1	PR	3.1	6000	2112	2112
2	PR	1.2	3000	6000	6000
3	PR	.26	3000	3456	3456
4	PR	.26	3100	1920	1920
5	PR	.76	2900	254	254
6	PR	.32	2900	3200	3200
7	PR	.32	2400	2752	2752
8	PR	.14	800	254	254
9	PR	.176	1800	1088	1088
10	PR	.14	1800	1792	1792
11	PR	.15	1000	832	832
12	PR	.15	600	640	640
13	PR	.15	1000	384	384
14	P	.296	10	64	21504
15	D	.228	1700	1700	167040
16	D	.268	10	150	24000
17	D	.820	1	2	264000
18	D	1.420	304	304	516800
19	D	.492	3	6	90024
20	D	.176	1	2	330
21	D	.492	10	2000	113154
22	DR	.001	20	462	462
23	D	.296	27	27	216000
24	D	.101	30	35	1033352
25	D	1.130	40	46	5630400
26	D	.843	47	47	135360
27	D	.180	30	33	68360
28	D	.001	20	28	26032
29	D	.912	30	33	600000
30	D	.090	1	1	1320
31	D	.560	50	202	126000
32	D	.140	4	2	2112
33	D	.068	1	1	4400
34	D	.068	7	7	93060
35	D	.0001	50	245	21102
36	D	.001	100	209	30030
37	D	.750	7	7	54012
38	D	.0001	100	1000	60060
39	D	.001	50	66	20064
40	D	.001	3	3	240042
41	D	5	75	1056	3168
42	P	.001	1000	500	482608

Activity Profile

Figure 3-2

Notice that the activity profile includes a parameter representing a priori knowledge of request frequency of a given file. This knowledge of inter-file (job) behavior can be obtained by a variety of different techniques. Probably the easiest of these is to assume certain file behavior dictated by experience with a particular environment. A more sophisticated technique of obtaining an activity profile is monitoring the files themselves. The needed parameter can be measured especially easily with a software monitor since it represents knowledge only at a very gross level of detail.

The analysis problem of improving system throughput is approached by using two different types of activity profiles: first, that of using a static activity profile, and second that of using a dynamic activity profile. A static activity profile is a constant; it is independent of time throughout the analysis process. In other words, a static activity profile remains insensitive to transient fluctuations of file requests because steady state request frequencies of files are known. Since it is time independent, values of system throughput and corresponding file assignment are static and need be calculated only once. Therefore, a static activity profile implies static memory management strategies.

A dynamic activity profile is one whose frequency of file request parameters are time dependent and allowed to vary during the analysis process. This time dependency stems from the facts

that 1) the files may experience high usage at some periods of time and low usage at other periods, and 2) more importantly, the assumption of a priori knowledge must be relaxed since system designers may have imprecise knowledge as to the steady state request frequencies of files for a single period of time. Consequently, the files themselves may percolate in the memory hierarchy, at one period being loaded in a memory of fast access time and at another period of relatively slow access time. The problem is not doing the physical file movement, but knowing when to do it. If one percolates too frequently, one induces sub-optimal performance due to the overhead cost of excessive file movement; if one does not percolate frequently enough, one induces a sub-optimal performance due to use of incorrect file assignment. Therefore, a dynamic activity profile implies dynamic memory management strategies.

Later experiments are made to gain insight into load factors effecting memory management. These load factors are defined here for conciseness. They are simply parameter permutations of the activity profile of Figure 3-2 defined as follows:

a. All Files are Reusable
   (1) Case RL, all files are load bound
   (2) Case RE, all files are balanced
   (3) Case RP, all files are processing (CPU) bound
b. All Files are Non-resuable
   (1) Case NL, all files are load bound
   (2) Case NE, all files are balanced
   (3) Case NP, all files are processing bound

The load bound cases (RL and NL), the balanced cases (RE and NE), and the processing bound cases (RP and NP) are obtained from the activity profile of Figure 3-2 by setting the reusability parameter such that all files are either reusable (R) or not (N=nill), and by dividing the given words loaded/request parameter by 1, 5, and 10 for the respective cases.

3.2.3 Degree of Multiprogramming

The second domain variable of the system throughput function is the degree of multiprogramming. It is the variable which specifies the amount of potential queuing interference (job competition) for servicing requests or for allocating resources.

A common definition of degree of multiprogramming is the number of jobs circulating in the model which have already queued for and been allocated executable memory. However, here for simplicity, the degree of multiprogramming is the number of jobs circulating in the model regardless of their location. For example, jobs waiting in the memory queue are counted in the degree of multiprogramming.

3.2.4 System Model -- Hardware Characteristics

The third domain variable is the system model, a representation of the hardware/software characteristics. In this case it is composed of the following parameters:

a. Hardware

   (1) Numer of CPUs and Associated Mean Execution Time/Instruction

   (2) Number of Executable Memories and Associated

      (a) Capacity

      (b) Transfer Time/Word

   (3) Number of IO Devices and Associated

      (a) Capacity

      (b) Mean Latency Time

      (c) Transfer Time/Word

   (4) Number of Channels

   (5) Interconnection Topology

b. Software

   (1) Queuing Discipline Algorithms

   (2) Resource Management Algorithms

The hardware characteristics that are used in the later experiments are now given (the number and capacity of executable memories are defined with the experiments themselves). Primarily, they are the same as those used by Arora and Gallo [A4] and represent the hardware of a transaction oriented system. They are defined as follows (times in microseconds):

A. Single CPU with Mean Execution Time/Instruction of 10.0

B. Two Executable Memories

1. Fast Memory

a. Capacity of 32K Words

b. Transfer Time/Word of 0.5

2. Slower Memory

a. Capacity of 32K Words

b. Transfer Time/Word of 2.0

C. Three IO Devices

1. Fast Drum

a. Capacity of 1.2M Words

b. Mean Latency Time of 4,300

c. Transfer Time/Word of 4.2

2. Slower Drum

a. Capacity of 1.2M Words

b. Mean Latency Time of 17,000

c. Transfer Time/Word of 4.2

3. Disk

a. Capacity of 3M Words

b. Mean Latency Time of 47,500

c. Tranfer Time/Word of 7.0

D. Two Pooled Channels

E. Interconnection Topology of Figure 3-3.

The software algorithms (e.g., FCFS queuing disciplines, least frequently used replacement algorithm) are chosen for convenience of analysis since extensive work has been done on the effect of these algorithms on system performance.

A variable number, greater than or equal to one, of CPUs, executable memories, and non-executable memories may exist in the system model, each number being independent of the remaining numbers. Each CPU may potentially be connected to each executable memory. The problems of memory interference are not dealt with directly. Each executable memory may potentially be connected to each non-executable memory (IO device) via assigned channel(s). A channel is connected to only one device; however, a channel may be connected to more than one executable memory. Note that device-to-device channels are not allowed.

3.3 Analysis Technique

This section deals with the following three topics: 1) assumptions imposed on the system throughput domain variables, 2) construction of a model which maps an activity profile, a degree of multiprogramming, and a system model into a value for system throughput and corresponding file assignment, and 3) an example illustrating the analysis technique.

3.3.1 Assumptions

3.3.1.1 Activity Profile

The conditions imposed on the activity profile are minimal. First, the activity profile is static in time for a given arrival period. In other words, the frequency of requesting a file is known a priori and does not vary. Second, there must exist an initial feasible file assignment, one in which all the files of the activity profile can be loaded (assigned) on the IO devices of the system model. Also the IO service request of a file (given by the words loaded/request parameter) must be characterizable by its mean service time; likewise for the CPU service request.

3.3.1.2 System Model

The conditions imposed on the hardware of the system model concern the interconnections and the transfer times associated with the various memories and devices, as well as memory interference generated by multiple CPUs and/or channels. The interconnections must pose a realistic model. For instance, if an IO device is included within the model, then an interconnection between some executable memory and that device is assumed. Also the system model assumes that the transfer time of the channels is governed by the device transfer time and not the executable memory transfer time. This is a reasonable assumption since the transfer time of the fastest device is larger than the transfer (acceptance) time of the slowest memory. If this is not true, a design error exists since

it would be more practical to transfer executable instructions from the device rather than from memory. It is assumed that the executable memories can be characterized by their capacity and transfer time/word and that the IO devices can be characterized by their capacity, mean latency time (seek time + rotation time), and transfer time/word. Finally, it is assumed that memory interference does not exist (or is small enough to be negligible). This means that bandwidths of memory busses may sometimes be overspecified.

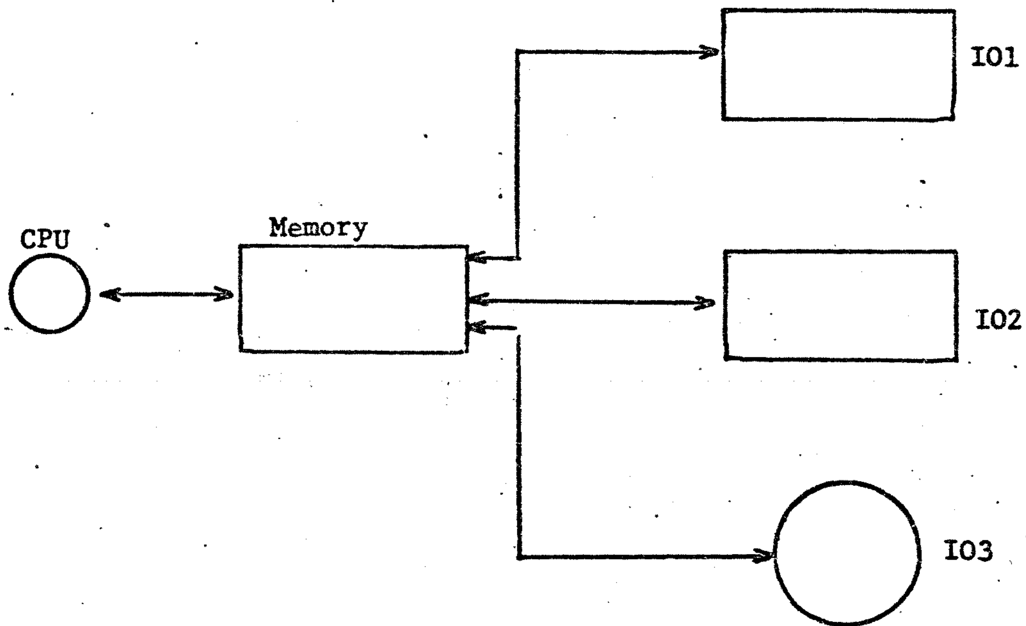
The conditions assumed by the software of the system model are as follows: a dispatching discipline of first-come-first-serve for all queues with no preemption of servers, and a least frequently used replacement algorithm for executable memory management. These assumptions are made to allow analysis of the analytical model and ease of verification of the simulation model.

3.3.2 Hybrid Model

A conventional approach for improving system throughput is to construct a simulation model corresponding to the system model, to postulate an assignment of the files, and then to gather the throughput metric by executing the model. This approach has the obvious flaw that the postulation may use the system model sub-optimally (due to device queuing), especially for reusable files since an IO device may be overused at the expense of the others. An arbitrary postulation of file assignment is made to keep this feasible by avoiding a simulation model evaluation for each of the file assignment combinations. Clearly, this ad hoc approach must be improved.

A reference system used to illustrate the new approach taken in this dissertation for optimizing system throughput is given in Figure 3-3. It consists of one CPU, one level of directly executable memory, and three secondary memories, say a fast drum, a fast disk, and a slow disk. Of course a channel interconnection exists between each of the secondary memories and the executable memory. There are a total of two channels servicing these intermemory. There are a total of two channels servicing these interconnections. The primary purpose of using this example is to help the reader keep his perspective in the following discussion.

The construction of a model which maps the previously defined domain variables into system throughput and corresponding file assignment utilizes two internal models -- first, a detailed



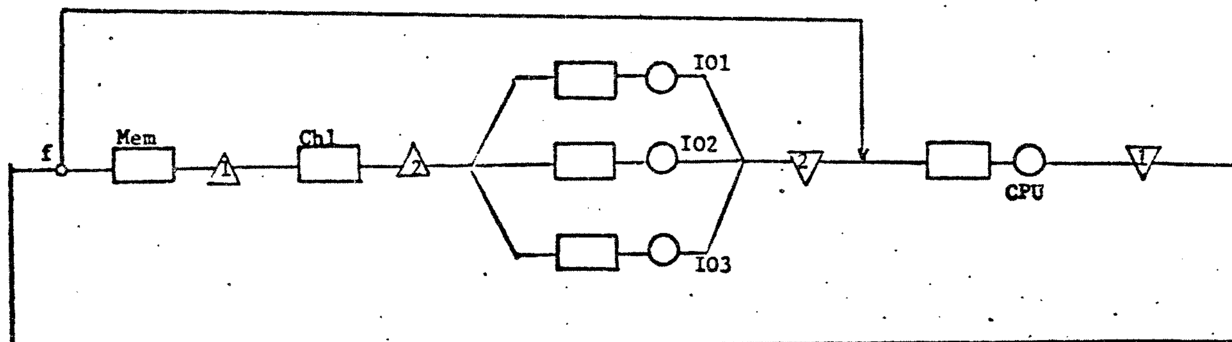
System Model

Figure 3-3

level simulation model and second, a gross level analytical queuing model. These two models are referred to collectively as a hybrid queuing model. The use of a two component model is suggested by the necessity of carrying out the interaction shown in Figure 1 and described succeedingly.

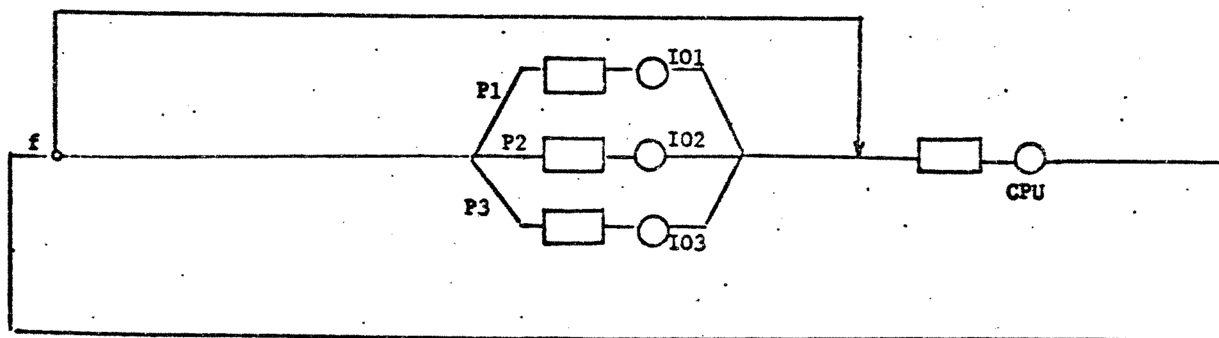
The gross level model shown in Figure 3-5 reflects the effect of file assignment through the service times of the IO devices and the branching probabilities to each device. Thus for the three IO devices included in this figure, there are only six dependent parameters regardless of the number of files to be assigned. For a given set of device service rates and for a given value of f, the probability that a file is already loaded in executable memory when it is requested, there exists a set of branching probabilities, P_1, P_2,..., P_i,..., P_m, which maximizes system throughput. Thus the optimality of a file assignment can be estimated by how accurately the model yields the values of P_i. A knowledge of the characteristics of rotating devices reveals that mean service rates are effected by file assignment somewhat less than are the branching probabilities.

Since the identity of the files is not maintained in the gross level model, a model with a greater level of detail (i.e., one which fully resolves the identity of files) is necessitated to generate file assignments. This is accomplished using the simulation model shown in Figure 3-4.



Simulation Model

Figure 3-4



Analytical Model

Figure 3-5

The exploration of the parameter space of possible file assignments is a very formidable task if carried out without guidance (as in conventional approaches). The solution of the gross level model suggests the following iterative procedure:

(i) Select some initial file assignment.

(ii) Evaluate the detailed model to obtain the service rates for the IO devices and f (and the performance metric, system throughput).

(iii) Determine from the evaluation of the gross level model the values of P_i which yield optimal throughput using the service rates and f determined from the solution of the detailed model.

(iv) Select a new file assignment whose accumulative frequency of file request on an assigned device is as close as possible to the optimal branching probabilities determined in step (iii).

(v) Evaluate the detailed model for new values of device service rates and f. (Note that new file assignments may return different service rates due to differing record sizes and positioning time.)

(vi) Iterate on steps (iii) - (v) until no changes in file assignment occur or the P_i are stable within a predetermined tolerance.

Note that the optimization is actually carried out via evaluation of the gross level model.

The points remaining to be discussed include the algorithm used for determining the optimal branching probabilities and the selection of new file assignments. The determination of the initial file assignment (loading strategy) is discussed later.

The determination of the optimal values of the branching probabilities in the gross level model is computed by an exhaustive grid search. The throughput of the model is evaluated for the values of P_i at intervals of 0.5 between 0 and 1. During the course of the development of this work, Hogarth and Chandy [H1] established the convexity of system throughput with respect to the set of P_i. This result would allow a more efficient determination of the optimal set of P_i. However, the grid search method was retained because it was already embedded in the computer program and because the determination of P_i consumed only about 20% of the processing time of model evaluation.

Each new file assignment for matching the optimal P_i is generated by the following procedure:

(i) For all files, compute the ratio of frequency request to volume.

(ii) Select an unassigned file with the largest ratio for assignment to the fastest device as long as the device capacity is not exceeded.

(iii) If step (ii) does not assign all files, force

assignment of the remaining files to the fastest device as long as the device capacity is not exceeded.

The reader should note that 1) the above procedure terminates since an initial assignment guarantees sufficient capacity, and 2) the solution of step (ii) is a very economical approximation to the costly solution generated by the corresponding branch and bound integer program. It has been shown [K2] that for a large number of files, this approximation is almost exact.

A discussion of the major components of the hybrid model (the simulation model, the analytical queuing model, and the iterative procedure) follows in more detail. The reader may wish to continually refer to the above discussion in order to maintain his perspective.

3.3.2.1 Simulation Model

The simulation queuing model corresponding to the above system model is given in Figure 3-4. It is interpreted in the following way. First, a request for a given file reaches branch point f. At branch point f a decision is made as to whether the file is loaded in the executable memory. If it is, then the request is scheduled for processing on the CPU. If it is not, the requested file must be loaded. So branch point f is answering the question "is the request file loaded?" Assume it is not. The request must queue for executable memory and then must queue for a channel. After obtaining these two resources, it must queue for the secondary memory on

which the file is loaded. Finally the request is serviced transferring the file into executable memory. Upon the completion of the loading process the channel is released and the file is scheduled for processing by the CPU. After completing CPU service the server is released and the file request recirculates to the beginning of the simulation model becoming a new request. Note that if f accumulates the number of successes of finding of requested file in executable memory, then in the limit f represents the success function of Mattson [M1].

3.3.2.2 Analytical Model

The model presented here takes advantage of the fact that optimal system throughput can be obtained from an analytical model of known service time distributions, known branching probabilities, and a known degree of multiprogramming. An example of an analytical queuing model corresponding to the system model of Figure 3-3 is given in Figure 3-5. It is interpreted in the following way. A request for a file reaches point f. At point f a branch is probabilistically made corresponding to whether or not the file is loaded in executable memory. If the bypass (vertical) path is taken, then it is assumed to be loaded and the request is scheduled for processing on the CPU. If the other (horizontal) path is taken, then it is assumed that the requested file must be loaded. Hence, it is scheduled for loading on probabilistically selected IO device (secondary memory). Upon completion of the loading process, the file is then scheduled for processing by the CPU.

After completing CPU service the file request recirculates to the beginning of the analytical model becoming a new request.

Notice that in this analytical model memory and channels are not explicitly represented. Consequently, the model makes assumptions that sufficient memory and channels are always available so that a request never has to wait for these resources. Notice also that the analytical model determines the probabilities, P1, P2, and P3, such that accessing the IO devices with these probabilities will produce optimal system throughput. All else has been specified such as the 'already loaded' probability f, the service time means, and the degree of multiprogramming.

An important assumption not previously mentioned is that file requests are independent and identically distributed. While a systematic treatment for serially dependent file request sequences is possible using simulation models, it certainly requires more sophisticated analytical models of job/file behavior. The present state of the art of such models is still embryonic and is an area of current research.

Now to the problem of mapping the activity profile, the degree of multiprogramming, and the system model into optimal system throughput. If this is done by simulation alone, a point by point evaluation method of each of the combination of file assignments must be used. If there are 42 files and 3

secondary memories, there exist $\binom{42}{3}$ combinations. Clearly, a point by point simulation evaluation for large combinations becomes infeasible simply because of time required for the evaluations. Consequently, the solution of this combinational optimization problem by simulation alone is infeasible.

At the present state of the art of queuing theory, it is intractible for the analysis of an analytical model for a large number of files since each state must 'remember' whether or not a particular file is loaded. Therefore, the procedure for the generation of optimal system throughput involves analyzing a hybrid model. A hybrid model is one which employs the advantages of both the simulation model and the analytical model. The simulation model maintains information on each file while the analytical model partitions the possible file assignments.

3.3.2.3 Interative Procedure in Detail

The iterative procedure for the analysis of the hybrid model is as follows:

a. Initially load the IO devices according to some loading strategy, also compute the mean service time.
b. Analyze the simulation model to obtain: f, CPU throughput, P1, P2, P3, and mean service times of the CPU and IO devices.
c. Analyze the analytical model to obtain: P1*, P2*, and P3* using f and the mean service times from