TASK RESPONSE TIME AND MODULE ASSIGNMENT FOR REAL-TIME DISTRIBUTED PROCESSING SYSTEMS

Kin Kwong Leung

December 1985
CSD-850041

UNIVERSITY OF CALIFORNIA

Los Angeles

Task Response Time and Module Assignment for Real-Time Distributed Processing Systems

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Computer Science

by

Kin Kwong Leung

1985

The dissertation of Kin Kwong Leung is approved.

[illegible]
James MacQueen

[illegible]
Harold Borko

[illegible]
D. Stott Parker

[illegible]
Milos D. Ercegovac

[illegible]
Wesley W. Chu, Committee Chair

University of California, Los Angeles

1985

To

Miranda (Mei-Ling)

and

our Moms and Dads

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF NOTATION

Roman Capitals

A - Module assignment matrix with elements A_{ij}

D_{net} - Average interconnection network delay

$D_{net}(i,j)$ - Average interconnection network delay incurred by messages from module i to module j

$D(A)$ - Vector of thread response time discrepancies for module assignment A with elements $d_i(A)$

F - The objective function which represents the task response time model

G - Task control-flow graph

$LWP(A)$ - The Processor with the longest average waiting time for module assignment A

R_w - Thread waiting time ratio

S - The set of all modules of the application task

S_i - The set of modules of thread i

S_L - The set of modules residing on LWP

S_R - The set of candidate modules for further replication on SWP

$SWP(A)$ - The Processor with the shortest average waiting time for module assignment A

T_c - Matrix of average IMC times with elements $\overline{t}_c(i,j)$

T_{obj} - Objective criterion for replicated module assignment problem

T_{pn} - Penalty delay function for replicated module assignment problem

T_{ptp} - Mean task response time

$T(i)$ - Average response time for module i

$U_i^*(s)$ - Laplace Transform of execution time of i^{th} module bulk invocation

$W(R)$ - Vector of required mean module waiting times with elements $\overline{w}_i(R)$

X - Vector of module execution times with elements $\overline{x}(i)$

Small Roman

a - Scaling constant for the penalty delay of the objective function in replicated module assignment problem

c_i^2 - Squared coefficient of variation of execution time for module i

$d_i(A)$ - Response time discrepancy for thread i for module assignment A

p - Enabling probability of a consecutive module pair

p_{ij} - Probability that module i enables module j upon completion of execution

r - Mean execution time ratio for a consecutive module pair

$t(i)$ - response time (a random variable) for module i

$\overline{t_i}(A)$ - Average response time for thread i for module assignment A

$\overline{t_i}(R)$ - Average response time requirement for thread i

$\overline{t_c}(i,j)$ - Average IMC time from module i to j

$\overline{u_i^n}$ - n^{th} moment of execution time (service time) for i^{th} module bulk invocation

$\overline{w}$ - Average module bulk waiting time

$\overline{w_i}(R)$ - Required mean module waiting time for thread i

$\overline{x}(i)$ - Average execution time for module i

Greek

α_i - Initial Multiplicity of Module i

λ - Task invocation rate

λ_i - Invocation rate for the i^{th} module bulk or module i

σ^2_{net} - Variance of interconnection network delay

σ^2_w - Variance of module bulk waiting time

$\sigma^2_c(i,j)$ - Variance of IMC time from module i to j

$\sigma^2(c)$ - matrix of variances of IMC times with elements $\sigma^2_c(i,j)$

$\sigma^2_t(i)$ - Variance of response time for module i

$\sigma^2_x(i)$ - Variance of execution time for module i

$\sigma^2(x)$ - Vector of variances of module execution times with elements $\sigma^2_x(i)$

ρ - total processor utilization (loading)

ρ_i - Processor utilization due to module i

ρ_M - Average processor utilization due to a module of the task

ACKNOWLEDGEMENTS

I would like to express my most sincere thanks to my committee chairman, Professor Wesley W. Chu, for his guidance, friendship, encouragement, and insightful comments during the course of this research. Thanks are also due to Lance M. Lan formerly of UCLA for sharing with me his DPAD Simulation results, James Huang and Dennis Townsend of Titan Systems, Inc. for their technical information on the Sentry System. I also thank my committee members: Professors Milos Ercegovac, D. Stott Parker, Harold Borko and James MacQueen for their encouragement and serving in the committee. I also want to thank Jung Min An, Joseph Bannister and Joseph Hellerstein for their stimulating discussions during this research, as well as Laurel Hendrix for her secretarial and administrative assistance. I am grateful to members of Chinese Bible Church at West Los Angeles for their concern and spiritual support.

Additionally, I would like to thank my parents, Kwok and Shun-Yeung, and other family members for their love and emotional support. Finally, I owe a great debt of thanks to Miranda (Mei-Ling) whose love, patience, sensitivity, and devotion kept my spirit and motivation from waning and, having completed my Ph.D., she made it all worthwhile.

This work was supported in part by U.S. Army Contracts No. DASG60-79-C-0087 and subsequent renewals of the same contract, and Univ. of California MICRO Contract No. 4-482516-19900.

VITA

January 6, 1958	Born, Hong Kong
1976-1980	B.S. in Electronics, The Chinese University of Hong Kong, Hong Kong
1979	Project Assistant, Royal Observatory of Hong Kong
1980-1982	M.S. in Computer Science, University of California, Los Angeles
1981-1983	Teaching Assistant/Associate, Computer Science Dept., University of California, Los Angeles
1983-1985	Postgraduate Research Engineer, Computer Science Dept., University of California, Los Angeles

PUBLICATIONS

"Task Response Time Model & Its Applications for Real-Time Distributed Processing Systems," coauthored with W.W. Chu, *Proceedings of 5th Real-System Symposium*, Texas, Dec.4-6, 1984, pp.225-236.

"Reservation Channel Access Protocol for High Speed Local Networks with Star Configurations," coauthored with W.W. Chu and W. Haller, *IEEE Trans. on Computer*, Vol.C-32, No.8, Aug. 1983, pp.763-766.

"A Contention Based Channel Reservation Protocol for High Speed Local Networks," coauthored with W.W. Chu and W. Haller, *Proceedings of 7th Conf. on Local Computer Networks*, Minneapolis, MN, Oct. 1982, pp.110-117.

"On the Effects of Non-Uniform Propagation Delays on the Performance of Multi-Access Protocols," Working Paper No.82008, Advanced Teleprocessing Systems Group, Computer Science Dept., UCLA, Spring 1982.

"A New Look at Partial Fraction Expansion from a High-Level Language Viewpoint," coauthored with C.F. Chen, *Computer & Maths. with Appls.* (Great Britain), Vol.7, No.5, 1981, pp.361-367.

ABSTRACT OF THE DISSERTATION

Task Response Time and Module Assignment

for Real-Time Distributed Processing Systems

by

Kin Kwong Leung

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1985

Professor Wesley W. Chu, Chair

Task response time is an important system performance measure for real-time systems. An analytic model is introduced to estimate task response time for loosely coupled distributed processing systems with real-time applications. The model considers such factors as assignment of modules to computers, module precedence relationships, interprocessor communications, interconnection network delay and module scheduling policy. Simulation experiments are used to validate the model assumptions and to show the accuracy of the model.

The analytic model is first employed to investigate the effects of module precedence relationships on response times. Our study reveals that the distributions of module execution times and the mean execution time ratio for a pair of consecutive modules are major factors for the effects.

The task response time model is then used to study module assignment for distributed systems. Based on the model, a new local search algorithm for module assignment is developed. Firstly, each module is assumed to be allocated to a single computer. Task response time is the optimality criterion, and the analytic model becomes the objective function. Search strategies are established to search for better module assignments. Further, the algorithm is extended to handle module replications; that is, modules may be replicated on several computers. The design objective for replicated module assignment is to minimize task response time with the thread response time constraints. A new objective criterion which is the sum of task response time and possible penalty delay to account for the violations of thread response time constraints is introduced. With this objective function, not only module copies are optimally assigned to computers, the proper module multiplicities are also iteratively determined by the algorithm so as to achieve the objective.

The algorithm is validated by applying to two distinct distributed systems for space defense applications. One system does not require module replications while the other does. The sub-optimal module assignments generated by the algorithm provide excellent response time performance on both systems since the analytic model has considered all major factors that affect task response time. Therefore, the algorithm can serve as a valuable tool for distributed systems design.

CHAPTER 1

INTRODUCTION

1.1 REAL-TIME DISTRIBUTED PROCESSING SYSTEMS

In real-time systems, a task is a sequence of system functions which must finish within a specified time period if the systems are to perform properly (e.g., process control and space defense applications). To meet the real-time requirements, it is desirable to share the processing workload among several computers (processors [1]). This technique is referred to as *distributed processing*. In addition to execution speedup, distributed processing can also provide cost-effective system designs, incremental system growth, better system reliability, and grace performance degradation in case of failures, etc.

The spectrum of distributed processing systems ranges from multiprocessor systems in which processors share common memory to sets of processors which are geographically dispersed and communicate via message exchanges. Herein, we consider real-time distributed processing system (RTDPS) (e.g., the Distributed Processing Architecture Design (DPAD) System [GREE80]) which is commonly known as *loosely coupled distributed processing system*. This system is solely dedicated for a

[1] "Computers" and "processors" are used interchangeably in this dissertation.

given real-time application task. The RTDPS consists of a set of processors connected together by an interconnection network which may be a fully connected network, a store-and-forward network or a local area network (e.g., a multi-access communication channel). Communications among processors are provided in the form of message exchanges.

Each processor has its own memory and self-autonomy thus all processors are equally important and no master-slave relationship among processors exists. A *distributed operating system* (DOS) is residing on each computer. Besides performing the functions such as memory allocation, scheduling and file management, the DOS also participates in the concurrency control, interfaces with communication subsystems, and exchanges information with other computers to guard against overloading and system failures. A typical execution cycle on each computer runs in the following sequence: processing messages from remote computers, dispatching a job for execution, sending output messages to remote computers, and other miscellaneous system management work. For real-time applications, the DOS usually does not provide interrupt facility as: (1) to eliminate interrupt overheads; (2) to avoid complicated data inconsistency problems which may occur during interruptions; and (3) to keep the variance of response time small, otherwise may be largely increased due to program interruptions.

In a RTDPS, the application task is partitioned into a set of *software modules* (or simply, *modules).* The task is repeatedly invoked to meet the processing requirements (e.g., processing radar return signals). *Task response time* or *port-to-port (PTP) time* is defined as the time from the task is invoked to the completion of the task execution. In some cases, the response time for the executions of a sequence of modules, which is referred to as a *thread*, is of interest. The *thread response time* (or *PTP Time of the thread*[1]) is the time from the request for the first module to the completion of the last module execution in the thread.

During the task execution, modules need to communicate with other modules via message exchanges. These messages are called *intermodule communication* (IMC). For a specific application task, the volume of IMC messages among modules is determined by the process of *task partitioning*. Therefore, it is desirable to partition the task to minimize the IMC among modules. In distributed systems, IMC is usually facilitated by sharing common data files and/or direct message exchange. The overhead for message exchanges on a local computer is usually small and can be assumed to be negligible in most systems. However, if IMC messages are sent to a remote computer via the interconnection network, the messages become *interprocessor communication* (IPC). Clearly, the IPC load in a distributed system depend on the IMC and the assignment of modules to computers. These IPC messages requires such extra processing as communication protocol and management of the distributed data files. IPC presents extra processing load on both the transmitting and the receiving computers. Therefore, IPC has significant impact on the system performance and response time. The importance of these impacts have been recognized by many

[1]Unless specified for a thread, PTP time is referred to the response time of the entire application task.

researchers [JENN77, CHU78, MA82, CHU84].

If data are shared among modules residing on different computers, to provide fast local accessing and to enhance file availability, some of the shared data files are *replicated* on several computers. However, maintaining the data consistency of the replicated copies requires the use of a concurrency control mechanism. A number of consistency control techniques such locking, timestamp and exclusive-writer protocol [BERN81,CHU85a] have been proposed. Besides the extra processing overhead due to the concurrency control protocol, file replications also increase the total IPC on the system because more than one file copy need to be updated. In the contrary, if a file is not replicated on a local computer, accesses to the file during module executions become read/write requests on remote computers where the file is located. These remote file accesses as a form of IPC of course incur the interconnection network delay and wait for response on the remote computers, thus degrading the task response time.

Therefore, planning a distributed system is complicated by many such complex and interdependent design issues as task partitioning, module and file assignment, data base management algorithm, etc. Presently, there is no systematic methodology for designing distributed systems. Existing system designs use ad hoc methods which result in a trial-and-error approach. Since response time is an important performance measure in real-time systems, we conduct this research to develop an analytic model for estimating response time for distributed systems. The model can then be used as a unified approach for studying various distributed design issues and exploring the tradeoffs among different choices.

1.2 SOME RELATED DESIGN ISSUES OF RTDPS

The merits of distributed processing: response time, throughput, system availability and incremental system growth, have made this system architecture appealing to many applications. However, the expected system performance and merits may not be accomplished unless the distributed system is properly designed. Here we discuss some important design issues of RTDPS which are related to the main theme of this research. The design issues discussed in the following include: (1) task partitioning, (2) module assignment (or task allocation [1]), (3) module replication and (4) file allocation.

1.2.1 Task Partitioning

The main purpose of distributed processing is to fully utilize the available computing resources by distributing the processing workload on several computers. Thus, the first step to RTDPS design is to partition the application task into a set of smaller and well-defined subtasks which are implemented in programming languages. These software modules are ready for allocation to computers. In general, task partitioning is strongly dependent upon the nature and inherited parallelism of the application task under consideration. Although few results on this issue were published in the literature, system designers usually find the following guidelines useful for task parti-

[1] In this dissertation, "task allocation" is to assign modules to computers. Thus we refer this process as to module assignment. Nevertheless, "task allocation" is a popular term.

tioning.

(a) It is desirable to exploit the inherited parallelism of the application task. That is to partition the task such that synchronizations among modules are eliminated as much as possible. Therefore, modules can be executed concurrently on several computers.

(b) Modules' execution times should not differ so much from each other because similar module sizes often facilitate even load balancing among computers. In addition, too large a module execution time may delay the response to the requests of some urgent module executions if module execution is non-interruptable. Too small a module usually wastes the computing capacity as much operating system overhead is incurred in scheduling when comparing with the amount of useful computation performed by the module.

(c) The data input and output characteristics of each module are determined by task partitioning. It is advantageous to partition the task such that amount of IMC's among modules is small. As a result, the total IPC on the system can be kept small and most of the computing power is used for actual application processing.

Clearly these design factors are interrelated with each other. For example, in order to reduce IMC, in one extreme, one might keep the entire application task unpartitioned as no IPC will be generated on the computers. However, it totally neglects the possibility of concurrent module processing. In some situations, a large module should not be further partitioned due to the nature of its processing requirements and/or avoidance of generating a large volume of IMC. Although we know little about the general approach to efficient task partitioning, a good task partitioning ought to be a proper balance and tradeoff among these factors.

In this context, we shall assume that the task of the RTDPS under consideration is well partitioned into modules and the precedence relationships among them can be represented by a task control-flow graph.

1.2.2 Module Assignment

The assignment of modules to processors affect the response time, throughput and system reliability. Several approaches to module assignment in distributed processing systems have been proposed [STON77, RAO79, PRIC79, MA82, CHOU82, EFE82, SHEN85, CHU85b, etc.]. These techniques include graph-theoretic, mathematical programming, and heuristic approaches. Instead of doing a survey on various module assignment strategies, here we discuss: (1) the key factors considered in these methods, and (2) their shortcomings when applying to the RTDPS. These drawbacks have motivated us to investigate some better approach to module assignment for RTDPS.

The key parameters considered in these approaches mainly are (a) module execution times (costs) and (b) interprocessor communication times (costs). Their basic notion in performing module assignment is to balance the computational load, including (a) and (b), among the processors such that either the total system time (cost) is minimized or computer loads are well balanced. Besides balancing the processing load, logical and precedence relationships among modules, queueing effect and interconnection network delay also have significant influence on the performance of module assignments. Especially for RTDPS where the response time is an important performance measure, these effects should be considered in module assignment.

[STON77] and [RAO79] use graph-theoretic algorithms which are tractable only for systems with two computers. Algorithms proposed in [MA82], [EFE82], [CHOU82] and [SHEN85] balance the workload on computers but neglect the impacts of module precedence relationships. Further, the approaches proposed in the literature usually assume that the application task is invoked only once. As a result, the queueing effect is ignored which affects the response time. This assumption used in the algorithms do not represent the actual operating environment in a RTDPS. Therefore, their "optimal" module assignment does not necessarily provide good response time. Moreover, the interconnection network delay is disregarded in the module assignment methods, but the network delay may have tremendous effect on system response time, especially if the network bandwidth in the RTDPS is not sufficiently large. In one extreme, modules may prefer to be allocated to a single processor instead of several processors to avoid the prohibitively high network delay. In summary, all these aforementioned factors substantiate that mere balancing the workload among processors is inadequate in module assignment to achieve a low system response time.

1.2.3 Module Replication

More importantly, all module assignment algorithms proposed so far do not consider module replications. This is, modules may be allocated to several processors instead of a single processor for executions. Sometimes, certain modules need to be replicated in RTDPS due to the considerations of response time requirements, system reliability, and/or some other constraints. In general, module replications may improve the system performance under one or more of the following circumstances:

(a) To meet strict response time specifications

In general, some modules of a given real-time application task are more urgent (or crucial) than others. And, these modules require rapid response from the processors. For example, the process of discrimination of offensive missiles is far more critical in terms of response time than search operations for new incoming objects in a space defense application. Therefore, it is desirable to satisfy these urgent modules with a much stringent response time requirement. Apparently, one way to meet the requirements is merely to assign a high execution priority to these modules thus receiving quicker response at the expense of other modules. However, if the requests for the urgent module executions come in a bursty manner, this approach may violate their real-time constraints. On the other hand, these modules may be replicated on several processors in addition to being assigned with a high priority. Thus, bursty execution requests can be routed to and executed on several processors to share the processing load for achieving the desired response times. Hence, besides load balancing, module replication is also a good alternative to meet strict response time specifications.

(b) To enhance system reliability

Module replication is similar to file replication in RTDPS in terms of enhancing system reliability. By replicating the important modules on several processors, requests for these modules can be processed on more than a single site. Therefore, module replications improve the system availability for the essential module executions in the presence of system failures.

(c) To achieve load balancing

The loading on each computer depends on the execution times and invocation rates for the residing modules. Clearly, the module execution times are predetermined by the task partitioning process in the early stage of system design. Of course, these execution times cannot be changed at the phase of module assignment. However, if modules are replicated, requests for these modules can be routed to the several computers for executions. Thus replicating modules in effect reduces the invocation rate for each replicated module copy on a computer. By adjusting the multiplicity of a replicated module, one can reduce the invocation rates for the replicated modules on each computer to a desirable level. As a result, the processing load due to the replicated modules can be evenly balanced on several computers. More importantly, module replications therefore do provide a new degree of flexibility for efficient system design in the hope that the computing resources are better utilized and the response time requirements can be satisfied.

The current module assignment algorithms neglect: repeated task invocations, IPC, module precedence relationships, queueing effects, interconnection network delays, and module replications. The new approach should remedy these drawbacks.

1.2.4 File Allocation

The other related design consideration for RTDPS is to replicate and distribute data files so that they can be efficiently accessed and generate less amount of IPC. This is commonly referred to as *file allocation* (or *file assignment*).

The optimal file allocation problem for multiprocessor systems with multiple files was firstly studied by Chu [CHU69]. Chu considered storage and transmission costs, file lengths, and read and update rates. The optimal allocation yields minimum overall operating costs. Subsequently, a lot of research effort [CASE72, MORG77, CHEN80, COFF81, RAMA83, etc.] has been dedicated to the problem. Most of them partitioned the multiple file allocation problem to multiple single file allocation subproblems. Minimizing the communication and storage costs is commonly used as an optimality criterion. The allocation models are usually formulated in the form of a 0-1 integer programming problem. In particular, both [CASE72] and [COFF81] attempted to determine the optimal number of copies of a data file. [MORG77] considered the placement of files and programs which use these files. [RAMA83] showed the isomorphism between the simple file allocation problem (without response time consideration) and the single commodity warehouse location problem in operations research, therefore some results by operations researchers can be adopted for solving the file allocation problems.

In RTDPS design, module and file assignment problems are coupled together: (1) since file accesses are generated by module executions, file access rates on each computer are dependent upon how modules are assigned to computers; and (2) module waiting times and processing loading on computers also depend on the volume of IPC's generated due to file accesses. Morgan and Levin [MORG77] considered the assignment of program and data files to computers in which file accesses are generated from program executions. However, the followings are their drawbacks when applying to RTDPS environments:

(a) The model only considers the communication and storage costs for file accesses and programs, but neglects IPC, processor loadings and queueing times.

(b) "Programs" in [MORG77] are equivalent to modules in our RTDPS. However, [MORG77] does not take the logical and precedence relationships among modules into consideration.

(c) Communication and storage costs are assumed to be known in the allocation model. Since response time is an important performance measure in RTDPS, it is desirable to approximate the response times by proper selections of communication and processing costs. However, these costs are usually difficult to estimate in a loaded environment like RTDPS.

Of course, it will be advantageous to perform module and file assignments at the same time. As a result, the interactions among modules and data files can be considered and the task response time for the RTDPS can then be truly minimized. However, simultaneous assigning modules and files to computers in order to minimize the task response time is very complicated because actually four design sub-issues are involved in the problem: (1) to determine the optimal number of copies for each module; (2) to allocate the module copies to computers; (3) to determine the optimal number of copies of each data file; and (4) to allocate file copies to computers. Even a simple module assignment problem in a multiprocessor system has been proved to be a NP-complete problem [GARE77] for which a polynomial time bound algorithm is very unlikely to exist. Thus all four sub-issues together further increase the problem complexity and make it more difficult to formulate. As a first step to this complex problem, we concentrate ourselves on module assignment aspect. In this research, we treat the file allocation aspect by assuming each data file which is shared by several modules is stored (or replicated) on each computer which needs access (read and/or write) to the file.

1.3 CONTRIBUTIONS OF THIS RESEARCH

The major contributions of this research lie in three areas: (1) a new analytic model to estimate task response time for RTDPS is introduced; (2) the model is employed to study the impacts of module precedence relationships on response times; and (3) based on this model, a new module assignment algorithm for the distributed systems is developed.

Response time is an important design criterion for real-time systems. Therefore, it is desirable to design a RTDPS such that its response times are minimized. In general, either analytical or simulation techniques can be used to study response times for RTDPS. However, simulation methods usually tend to be more expensive and time-consuming. These shortcomings have led us to pursuing analytical approaches. To overcome the inadequacies of current analytical approaches, a new analytic model is introduced to estimate the task response time for the distributed systems. The model considers such factors as module precedence relationships, IPC, interconnection network delay, module scheduling policy, and assignment of modules and files to computers. Simulation experiments are used to validate the model assumptions and show the accuracy of the model.

The analytic model can be used in various design study for distributed systems. Firstly, the model is employed to study the importance and the effects of module precedence relationships on response times. Secondly, since the analytic model accurately estimates task response times for various module assignments, the model can be used to perform module assignment. Based on the model, a new local search algorithm for module assignment in RTDPS is developed. In this algorithm, the task response time becomes the objective criterion which is optimized over the solution space -- module assignments, and the model is used as the objective function for the problem. Search strategies are established in the algorithm to look for better module assignments. We first consider the cases where each module is allocated to a single computer. Then, we extend the algorithm to handle cases where each module may be replicated on several computers. For the distributed systems where module replications are required, not only the module copies are optimally allocated to computers, but the appropriate module multiplicities are also iteratively determined by the algorithm.

The module assignment algorithm is applied to two distinct distributed systems. One of them requires module replications while the other does not. The solutions from the algorithm provide excellent response time performance because the task response time model has considered all major factors that affect the task response time in the distributed systems. Therefore, the module assignment algorithm can serve as a valuable tool for distributed systems design.

CHAPTER 2

TASK RESPONSE TIME MODEL FOR RTDPS

2.1 INTRODUCTION

With the advent of low-cost VLSI and communication technologies, distributed processing has become an economically and technologically attractive computer architecture. The distributed system considered here consists of multiple computers, each with its own memory and peripherals, connected by an interconnection network. This type of systems is commonly referred to as *loosely coupled distributed processing systems*.

Recall that an application task in a RTDPS is often partitioned into several sub-tasks (i.e., software modules) which are assigned to a set of computers for processing. An example of a task consisting of fifteen modules assigned to a system with three computers is shown in Figure 2.1. The logical structure and precedence relationships among the software modules may be represented by a task control-flow graph. The task is repeatedly invoked to meet the processing requirements (e.g., processing return signals from a radar). After a module completes its execution, it sends messages to enable (invoke) its succeeding module(s) as indicated in the task

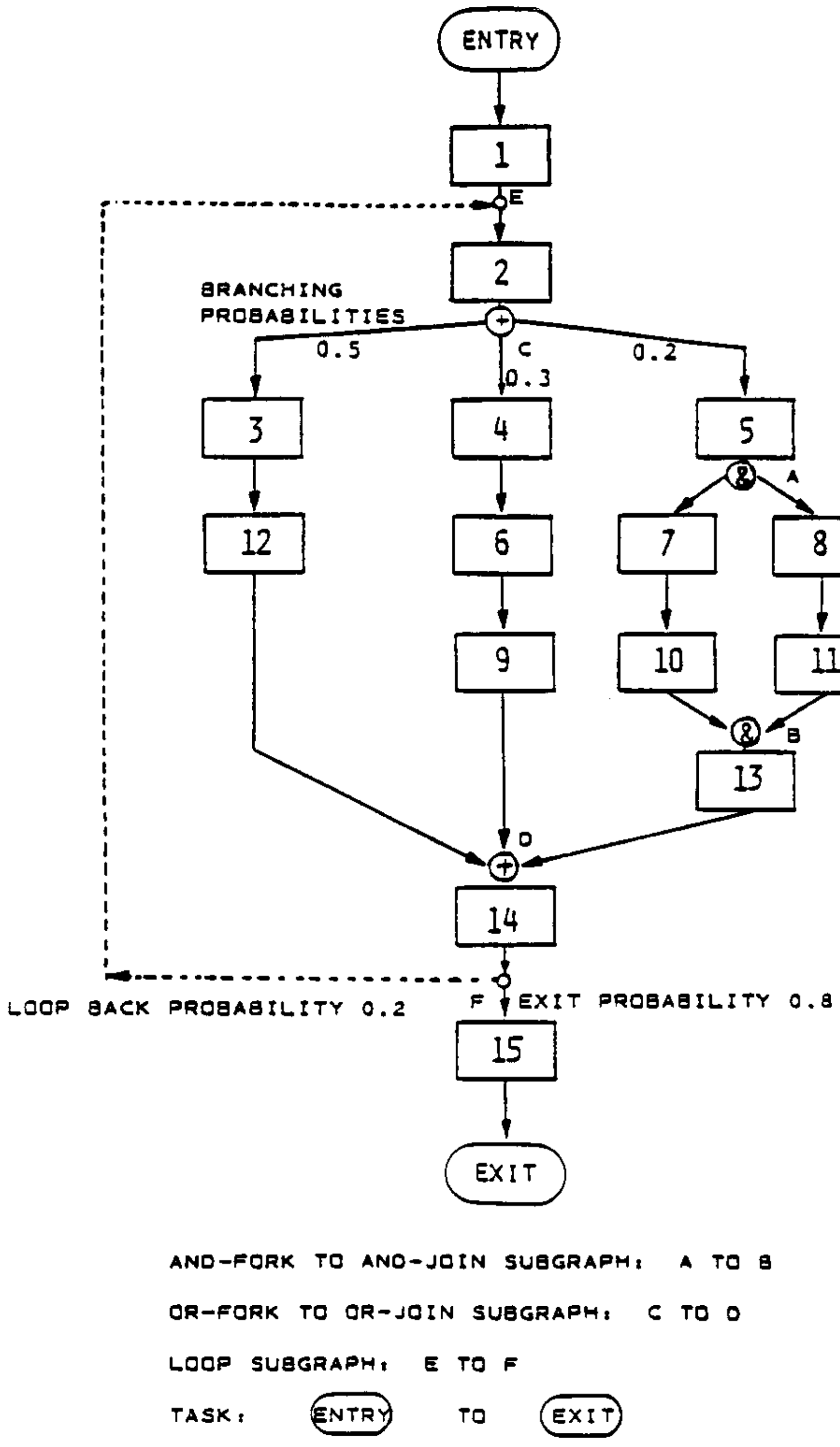


Figure 2.1a A Sample Task Control-Flow Graph

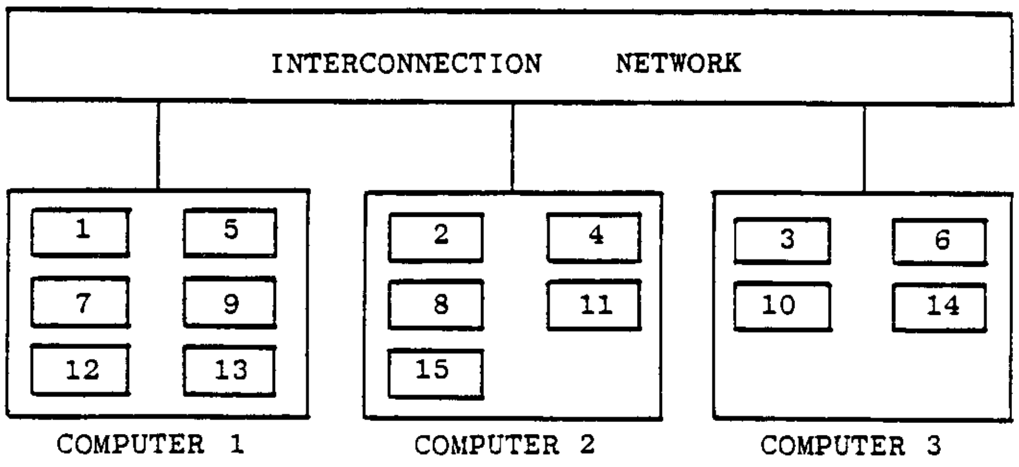


Figure 2.1b Assignment of Modules to Computers
(A Loosely Coupled Distributed System)

control-flow graph. In addition, when a module finishes its execution, it may also send messages to update the shared data files on other computers. Such message exchanges among modules are referred to as intermodule communication (IMC) [CHU84]. The overhead for communications among modules that reside on the same computer is usually small and can be assumed to be negligible. If messages are sent between modules that reside on different computers, the messages are called interprocessor communication (IPC). IPC requires such extra processing as communication protocol and management of the distributed data files, and incurs interconnection network delay. Therefore IPC has significant impact on the system performance and response time.

If data are shared among modules residing on different computers, some of the shared data files are replicated on several computers. However, maintaining the data consistency of the replicated copies requires the use of a concurrency control mechanism (e.g., locking, timestamp, exclusive-writer protocol). Therefore, planning a distributed system is complicated by many such complex and interdependent design issues as module and file assignment [CHU80], module scheduling policy, database management algorithm, etc. Presently, there is no systematic methodology for designing distributed systems. Existing system designs use ad hoc methods which result in a trial-and-error approach. Further, since RTDPS often are required to perform time critical functions, response time is an important performance measure. Simulation techniques are used to estimate the response time, but such approaches are time-consuming and expensive. This motivates us to develop an analytic model for estimating the response time for these systems. The model can be used as a unified approach for studying various RTDPS design issues and exploring the tradeoffs among different design choices.

We shall first present our task response time model based on module response times and the weighted task control-flow graph. Next, we present a set of simulation experiments to validate the assumptions used in the model for various types of logical structures and precedence relationships among modules. Finally, we discuss the use of the model to study the interrelationships among task response time, module assignment, precedence relationships, scheduling policy for module executions, and database management algorithms.

2.2 A TASK RESPONSE TIME MODEL

Queueing networks [BASK75, HEID82, LAZO84] are commonly used to model distributed processing systems. In such models, computers are represented as servers, modules as customers, and task invocations correspond to external arrivals. Customers are routed for service in accordance with the task control-flow graph and the module assignment. In distributed systems, a module may enable more than one modules. This is referred to as a FORK in the graph. Alternatively, a module may have several immediate predecessor modules which must complete their executions before the succeeding module can be executed. This is referred to as a JOIN. When a control-flow graph consists of FORKs and JOINs, the routing scheme in the queueing network model is inadequate to represent the logical relationships among modules.

Thus the system cannot be represented by a tractable queueing network model. Therefore, we introduce a new model to estimate the task response time.

Task response time, or *port-to-port* (PTP) time, is the time from the request of a task invocation to the completion of its execution. Since a task may be repeatedly invoked and the modules are enabled according to the sequence as indicated in the control-flow graph, task response time consists of module waiting times, module execution times and precedence waiting times. *Module waiting time* is the time from a module invocation arrival until it starts its execution on a computer. This waiting time is the time spent waiting for module executions and input IPC processings. *Module execution time* is the sum of a module's execution time and its output IPC time. Let the sum of a module's waiting time and execution time be denoted as *module response time.* The *precedence waiting time* is the intermodule synchronization delay due to the precedence relationships among modules. Our task response time model consists of two sub-models: *module response time model* and *weighted control-flow graph model.* The first sub-model computes the module response times, while the latter considers the precedence waiting times.

2.2.1 Module Response Time Model

For a given module assignment, each computer will execute a fixed set of modules. The response time of a module is the time from its invocation to the completion of its execution. Thus module response time includes waiting (queueing) time and module execution time. If a module needs to send messages to other computers, the output IPC time is included as a part of the module execution time. Further, these IPC's are transmitted over the interconnection network, and eventually arrive at their destinations. These input IPC's on the destination computers can be viewed as a special module which also contends for processing. Based on the module assignment and IMC's among modules, IPC processing times can be obtained. Let the module execution times be characterized by probability distribution functions (PDF's). Then each computer can be modeled as a queueing system with several modules (customers of different types) with specified service distributions. Based on the logical structures among modules and task invocation rate, the invocation rate of each module on the computer can be determined. In queueing terminology, module invocations are customer arrivals. If several modules on the same computer are invoked simultaneously, this results in a bulk module invocation.

In our model, we assume that (1) the module invocation arrival (single or bulk) processes are independent of each other, and (2) module invocation interarrival times are Poisson distributed. To illustrate the concept, let us determine the modules' response times on a computer that uses *first-come-first-serve* (FCFS) scheduling policy[1] for module executions.

Consider a computer that has m distinct module invocations (single or bulk invocations), and the arrival rate for the i^{th} module invocation be λ_i and the Laplace Transform (L.T.) of the service requirement be $U_i^*(s)$ for $i=1,2,...m$. One of these m

[1] The model can be applied to other module scheduling policies with the use of appropriate queueing delay equations.

module invocations (say the c^{th}) represents the input IPC on the computer. Thus λ_c and $U_c^*(s)$ are the arrival rate and L.T. of processing time for the input IPC. For the i^{th} bulk invocation that invokes a set S_i of distinct modules (referred to as *module bulk),* the corresponding service requirement is $U_i^*(s) = \prod_{j \in S_i} X_j^*(s)$, where $X_j^*(s)$ is the L.T. of the service time of module j.

Based on the assumptions 1 and 2, this queueing system is an extension of the regular FCFS M/G/1 queue with total arrival rate $\lambda = \sum_{i=1}^{m} \lambda_i$, and the L.T. of service time for each invocation arrival is $U^*(s) = \sum_{i=1}^{m} \frac{\lambda_i}{\lambda} U_i^*(s)$. For the M/G/1 queue, the first two moments of the module bulk waiting time from the bulk invocation arrival until its first module starts to execute are

$$\bar{w} = \frac{\sum_{i=1}^{m} \lambda_i \overline{u_i^2}}{2(1-\rho)} \tag{2.1}$$

and $$\overline{w^2} = 2(\bar{w})^2 + \frac{\sum_{i=1}^{m} \lambda_i \overline{u_i^3}}{3(1-\rho)} \tag{2.2}$$

where:

$\overline{u_i^n} = n^{th}$ moment of service time for i^{th} module invocation,

ρ = server utilization $= \sum_{i=1}^{m} \lambda_i \overline{u_i^1}$,

$\bar{w}$ = average module bulk waiting time.

From Eqs.(2.1) and (2.2), we obtained the variance of module bulk waiting time as

$$\sigma_w^2 = \overline{w^2} - (\overline{w})^2 = 2(\overline{w})^2 + \frac{\sum_{i=1}^{m} \lambda_i \overline{u_i^3}}{3(1-\rho)} - \left\{ \frac{\sum_{i=1}^{m} \lambda_i \overline{u_i^2}}{2(1-\rho)} \right\}^2 \qquad (2.3)$$

In a bulk invocation, a set of modules are invoked at the same time. Based on the resource requirements, the operating system schedules the execution sequence for these modules. Let the sequence be $j_1, j_2, \ldots j_{k-1}, j_k, j_{k+1}, \ldots$. The response time (a random variable) for module j_k is

$$t(j_k) = w + \sum_{i=1}^{k} x(j_i) \qquad (2.4)$$

where:

w = module bulk waiting time,

$x(j_i)$ = execution time for module j_i.

The average response time $T(j_k)$ for module j_k can be obtained by taking the expected values of Eq.(2.4). We have

$$T(j_k) = \overline{w} + \sum_{i=1}^{k} \overline{x}(j_i) \qquad (2.5)$$

Since w, $x(j_i)$ and $x(j_k)$ are independent random variables, the variance $\sigma_t^2(j_k)$ of the response time for module j_k is the sum of variances of each component in Eq.(2.4). Hence

$$\sigma_t^2(j_k) = \sigma_w^2 + \sum_{i=1}^{k} \sigma_x^2(j_i) \qquad (2.6)$$

where $\sigma_x^2(j_k)$ is the variance of execution time for module j_k and σ_w^2 is given in Eq.(2.3). For the case of a single module invocation, there will be only a single module in the execution sequence.

2.2.2 Weighted Control-Flow Graph Model

To take into consideration the precedence waiting times due to the intermodule relationships as indicated in the task control-flow graph, we map the mean and variance of the module response times (computed by the module response time model) onto the control-flow graph as arc weights (Figure 2.2). The response time for module i is assigned as the weights for all arcs emerging from module i in the control-flow graph. After the execution of module i, if it enables module j which is residing on a different computer, the module enablement message is transmitted via the interconnection network. Since the network delay is independent of module response times, the mean and variance of network delay [1] can be added to the weight of the arc from module i to j. Then the task response time can be estimated from this weighted control-flow graph model.

According to the logical structures and precedence relationships among software modules, there are four common types of control-flow subgraphs: sequential thread, And-Fork to And-Join, Or-Fork to Or-Join, and loop (Figures 2.3 to 2.6). A task control-flow graph may contain a combination of these basic logical relationships

[1] Network delays among any pair of computers may be different depending upon the characteristics of the interconnection network.

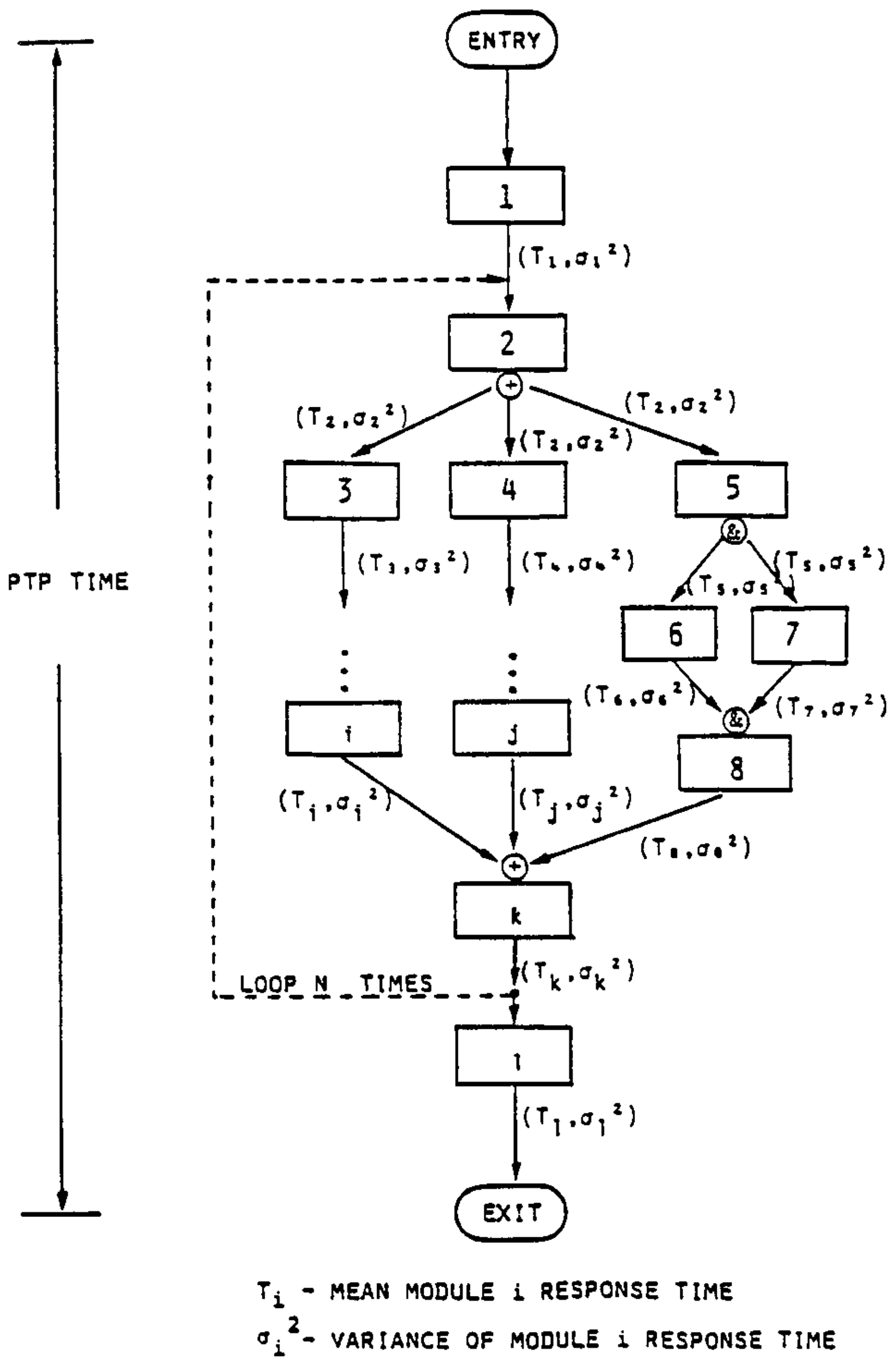


Figure 2.2 Weighted Control-Flow Graph for Response Time Estimations

among modules. Each of these graphs can be reduced to a single node graph. Such successive graph reductions yield the estimation of the task response time.

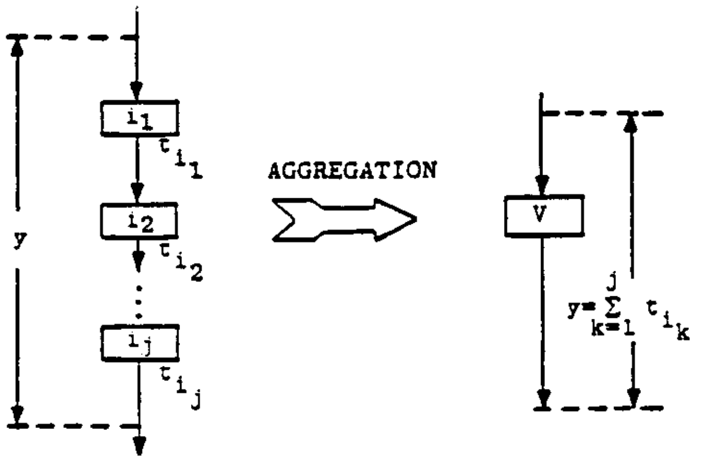
2.2.2.1 Sequential Thread Subgraph

Sequential thread subgraph (Figure 2.3) is a sequence of modules connected in series in which each module (except the last) has a single successor. Modules execute in the sequence indicated by the thread. Assuming that module response times represented by arc weights are random variables, then the total response time of the thread is the sum of all arc weights of each module.

2.2.2.2 And-Fork to And-Join Subgraph

This subgraph begins from a module which simultaneously enables several succeeding modules *(an and-fork)* and ends at a module which is enabled only when all of its preceding modules have completed their executions *(an and-join)* as shown in Figure 2.4. This subgraph may correspond to the case in which the modules assigned to different computers require concurrent processing. Since sequential threads can be reduced to a single node as mentioned above, the and-fork to and-join subgraph can be aggregated into several nodes V_i with response time y_i for $i=1,2,\ldots n$ (Figure 2.4). Because of the and-join function, the response time of the subgraph is the maximum of y_i's.

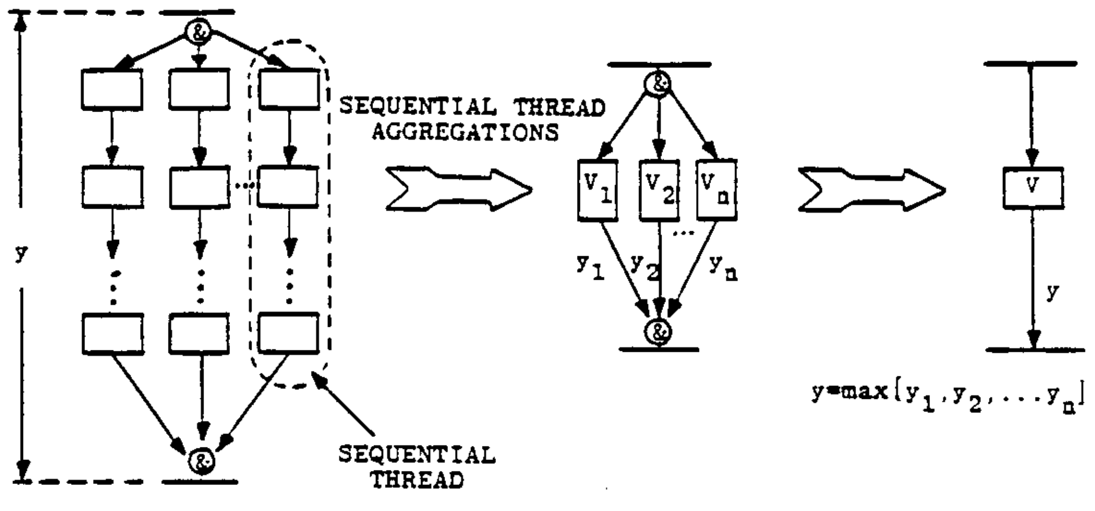
Computing the response time for this subgraph requires the knowledge of the PDF's for y_i's, which is rather complicated. In this study, we shall emphasize mainly



t_{i_j} : RESPONSE TIME OF MODULE i_j (RANDOM VARIABLE)

y : RESPONSE TIME FOR THE SEQUENTIAL THREAD (RANDOM VARIABLE)

Figure 2.3 Sequential Thread



&
y
SEQUENTIAL THREAD AGGREGATIONS
SEQUENTIAL THREAD
V_1
V_2
V_n
y_1
y_2
y_n
V
y
$y=\max\{y_1, y_2, \ldots y_n\}$

Figure 2.4 And-Fork to And-Join Subgraph

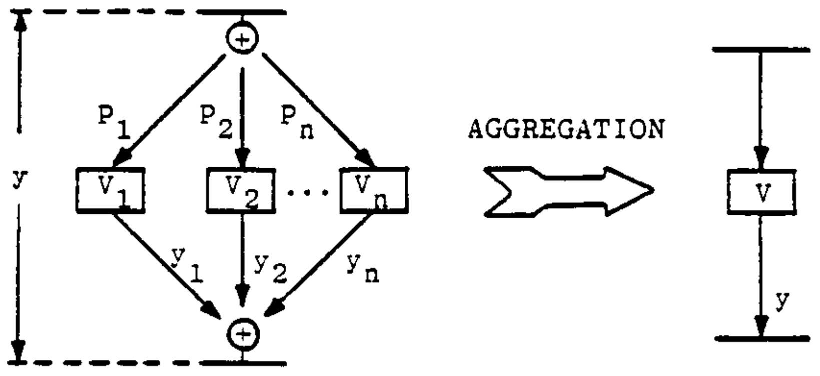
the *average* task response time, which usually can be determined by the first two moments of module response times. Therefore, these moments are derived from the module response time model. According to the coefficients of variation of y_i's, they can be approximated by either Erlangian or hyper-exponential distribution functions [SAUE81]. Assuming that y_i's are independent, the joint PDF for y_i's can be computed. Thus the mean and variance of the response time for the subgraph can be obtained (See Appendix A).

2.2.2.3 Or-Fork to Or-Join Subgraph

This type of the subgraph consists of an or-fork and an or-join as depicted in Figure 2.5. At the or-fork, the module enables one of its succeeding modules. This type of subgraph facilitates the system to process one out of several threads based on certain selection criteria. The branching probability to execute each thread can be measured or estimated from the IMC data. The response time for the subgraph is the sum of all these threads' response times weighted by their branching probabilities.

2.2.2.4 Loop Subgraph

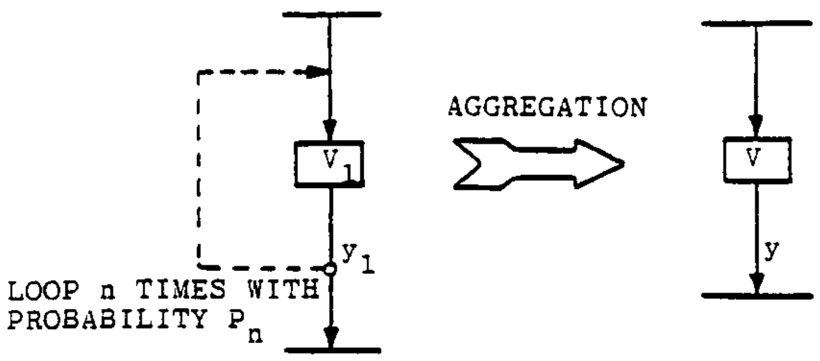
Loops are often contained in a task control-flow graph for repeatedly processing a set of modules for a task invocation. A loop may contain any of the aforementioned subgraphs. After aggregating these subgraphs, a loop may be represented by a single cyclic node graph as shown in Figure 2.6. The arc weight is the response time of executing a single loop. The response time of the loop subgraph can be computed from the average number of times that the loop is executed multiplied by the time re-



P_i : BRANCHING PROBABILITY (TO ENABLE MODULE V_i)

$$\sum_{i=1}^{n} P_i = 1$$

Figure 2.5 Or-Fork to Or-Join Subgraph



y_1 : RESPONSE TIME FOR A SINGLE LOOP

y: RESPONSE TIME FOR THE LOOP SUBGRAPH

Figure 2.6 Loop Subgraph

quired to execute a single loop.

2.2.3 Module Response Times With Dependent Module Invocations

In Section 2.2.1, module invocations are assumed to be independent and their interarrival times are Poisson distributed (assumptions 1 and 2). Thus, the logical dependency and the precedence relationships among modules are neglected when computing the module response times. The independence assumption is based on the following observations. Each computer is allocated with several modules which are enabled by modules residing on other computers. Since the operation of each computer is independent of each other, the module invocation arrival processes at each computer are random and thus can be approximated by independent Poisson processes. However, if a module is invoked by another module residing on the same computer (e.g., assigning a sequential thread to the computer), then the module invocations are dependent and non-Poisson arrivals. The error introduced in computing the mean module response times in such cases may be unacceptable. Therefore we introduce the following generalized model to compute the mean module response times for dependent module invocations.

2.2.3.1 Partitioning the Control-Flow Subgraphs

Based on a module assignment, we partition the control-flow graph into a set of subgraphs such that the modules of each subgraph are allocated to the same computer. Each control-flow subgraph on a computer is invoked by other computers via the interconnection network. Examples of such subgraphs are shown in Figure 2.7.

Due to the relationships among modules as indicated in the subgraphs, the invocations of these modules are dependent upon each other. In addition, the dependency among the modules at the *forks* and *joins* increases the computation complexity for module response times. For tractability while considering the precedence relationships among modules, we further partition the subgraphs into several smaller ones at the forks or joins. As a result, the partitioned subgraphs become sequential threads (Figure 2.8). Figure 2.8a is a special case where two sequential threads are invoked simultaneously via bulk module invocations as they succeed an and-fork in the original control-flow subgraph. Further, if a sequential thread has an or-fork (Figure 2.8f) and the control branches to a module residing on another computer, then the execution terminates at the or-fork.

2.2.3.2 Mean Module Response Times for Partitioned Subgraphs

Since computing the mean module response time is simpler than computing its variance, we are able to relax assumptions 1 and 2. Let us refer to the first module of each sequential thread in a subgraph as the ***entry module***, and other modules as ***non-entry modules***. We assume: (1a) the invocations for the entry module(s) of each subgraph are independent of each other, (2a) the interarrival times of these invocations are exponentially distributed (i.e., Poisson arrival processes). In this case, only the invocations for the entry modules are independent and Poisson arrivals, and the invocations for non-entry modules may be dependent and non-Poisson arrivals. Thus the

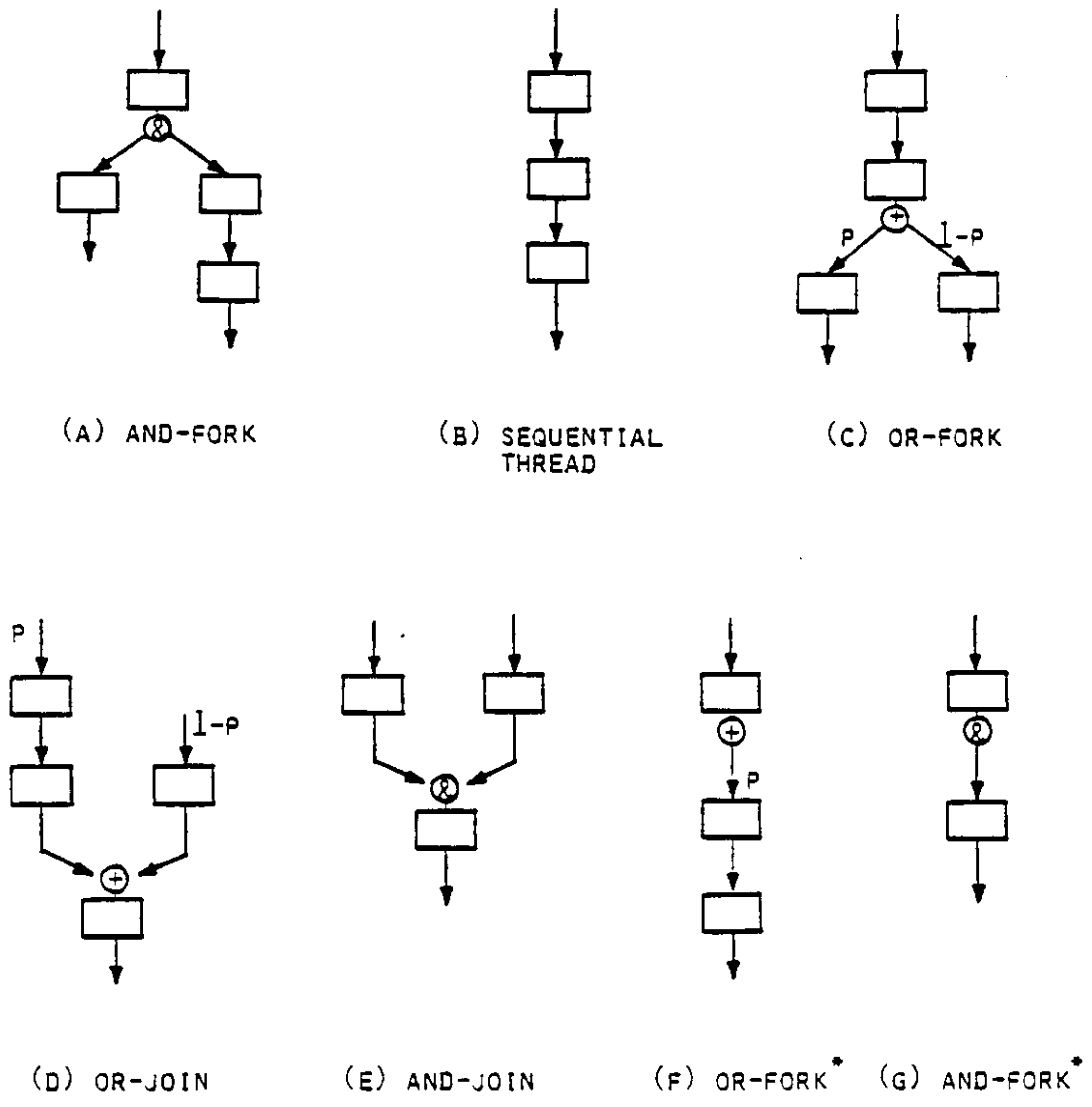


Figure 2.7 Examples of Control-Flow Subgraphs that Allocated on a Computer

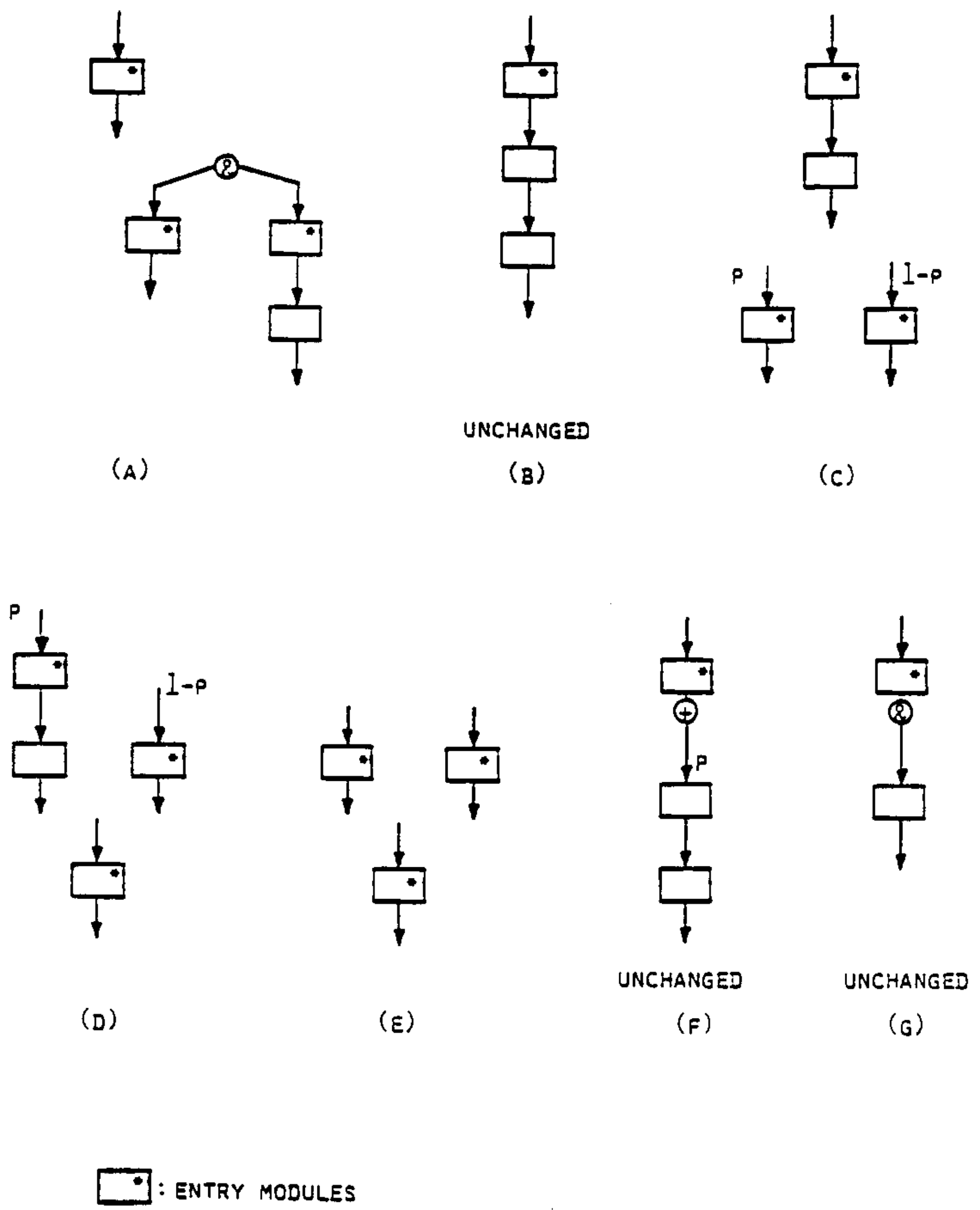


Figure 2.8 Partitioned Control-Flow Subgraphs of Figure 2.7

mean module response times [1] computed under these relaxed assumptions include such module precedence relationships as sequential threads, bulk module invocations at and-forks, and branching at or-forks.

Let us consider the response times for entry modules. Due to Poisson arrivals, the average waiting time for a given entry module is the processing time required to execute all the module invocations existing (waiting or being executed) on the computer upon the arrival of the entry module invocation. When several entry modules are invoked simultaneously, these modules are executed in a predefined sequence. Except the first module in the sequence, the mean module waiting time for a given entry module is the sum of the module bulk waiting time and the execution times of those modules processed prior to the module (Same as Eq.(2.4)).

Let us now consider the waiting times for non-entry modules. After an entry module finishes its execution, it enables its succeeding module as indicated in the subgraph. Since the invocation arrivals for the non-entry modules no longer form a Poisson arrival process, we need to keep track of the 'history' of the module executions since the arrival of that entry module invocation. During the waiting time of the entry module, new module invocations may arrive from other computers, and some of modules waiting in front of the entry module may invoke their succeeding modules. These module executions will become the waiting time for the non-entry module, which can be divided into three components, and computed as shown in the Appendix

[1] For mathematical tractability, the variances of module response times are computed under the independent Poisson assumptions.

B. The module response times can be obtained by summing the respective waiting and execution times.

Our study reveals that for most subgraphs, the module response times based on independent and Poisson module invocation assumptions are very close to those of the dependent module invocations. The dependent module invocation approach provides more accurate module response times only when the modules assigned on a computer form a long sequential thread. This reveals that assumptions 1 and 2 are reasonable, and provide good approximations for most cases.

2.3 MODEL VALIDATION

To validate the proposed task response time model, simulation experiments were performed via two simulation packages: a queueing network based simulation package PAWS [BERR82], and a simulator of the Distributed Processing Architecture Design (DPAD) System [GREE80] for real-time space defense applications. In the PAWS simulation, computers are modeled as servers, and module invocations are represented as customers which request services from the servers. The service times correspond to the module execution times. After receiving service, a customer is transferred to another server queue according to the task control-flow graph and the module assignment. A customer goes through the interconnection network if it is transferred from one server to another. The network is represented by a server which always delays each customer according to the network delay distribution function before passing the customer to its destination server. As a result, the module invocations are dependent upon each other, and their arrivals are non-Poisson distributed. Further, the queueing discipline on a computer is also used for the corresponding server queue. For an AND-FORK operation, the module invocation is split into several modules and routed to their appropriate servers. For an AND-JOIN operation, the module following the join waits until all precedent modules complete their executions. The precedence and logical relationships among modules are preserved in the simulation. Therefore, PAWS provides a flexibility for testing different types of task control-flow graphs. However, it uses idealized external inputs (e.g., Poisson task invocation arrivals) and does not include the detailed operating system overhead.

We have performed the simulation to obtain the mean PTP times for selected types of task control-flow graphs. To reach the steady state of the queueing systems, the task is invoked ten thousand times for each simulation run. In order to aviod the instability of computers due to overloading in the simulations, the maximum task invocation rate is thus chosen that utilization of each computer is less than 80%. Further, each simulation experiment is repeated five times with different initial random numbers to reduce the statistical fluctuation. Here let us consider the sample task control-flow graph in Figure 2.1a with its parameters given in Table 2.1. It consists of sequential threads, an And-Fork to And-Join, an Or-Fork to Or-Join, and a loop. These modules are assigned to three identical computers for processing, and the system has a constant network delay of 0.2 second for message exchanges among computers. Figures 2.9 to 2.12 present the mean response times for these subgraphs and the whole task for the module assignments (Table 2.2). We aggregate the response

MODULES	MEAN EXECUTION TIME (in sec)	DISTRIBUTION
1, 2, 3, 4, 5	1	EXPONENTIAL
6, 7, 8, 9, 10	2	EXPONENTIAL
11,12,13,14,15	3	EXPONENTIAL

Table 2.1 Module Execution Times for the Sample Control-Flow (Figure 2.1a)

COMPUTERS / LOADING / ASSIGNMENTS	CPU 1		CPU 2		CPU 3	
	MODULES ASSIGNED	PROCESSING LOAD (SEC) PER TASK INVOCATION	MODULES ASSIGNED	PROCESSING LOAD (SEC) PER TASK INVOCATION	MODULES ASSIGNED	PROCESSING LOAD (SEC) PER TASK INVOCATION
A	1, 5, 7, 9, 12, 13	5.125	2, 4, 8, 11, 15	5.875	3, 6, 10, 14	5.625
B	1, 3, 5, 7, 13	3.125	2, 6, 8, 9, 11, 12, 15	8.875	4, 10, 14	4.625
C	1, 5, 6, 7, 12, 13	5.125	2, 3, 8, 9, 11, 15	6.875	4, 10, 14	4.625
D	3, 4, 11, 14	5.5	1, 5, 6, 8, 15	5.5	2, 7, 9, 10, 12, 13	5.626
E	9, 13, 14	5.25	1, 2, 10, 15	5.75	3, 4, 5, 6 7, 8, 11, 12	5.626

Table 2.2 Module Assignments & Computer Processing Load for the Sample Control-Flow Graph

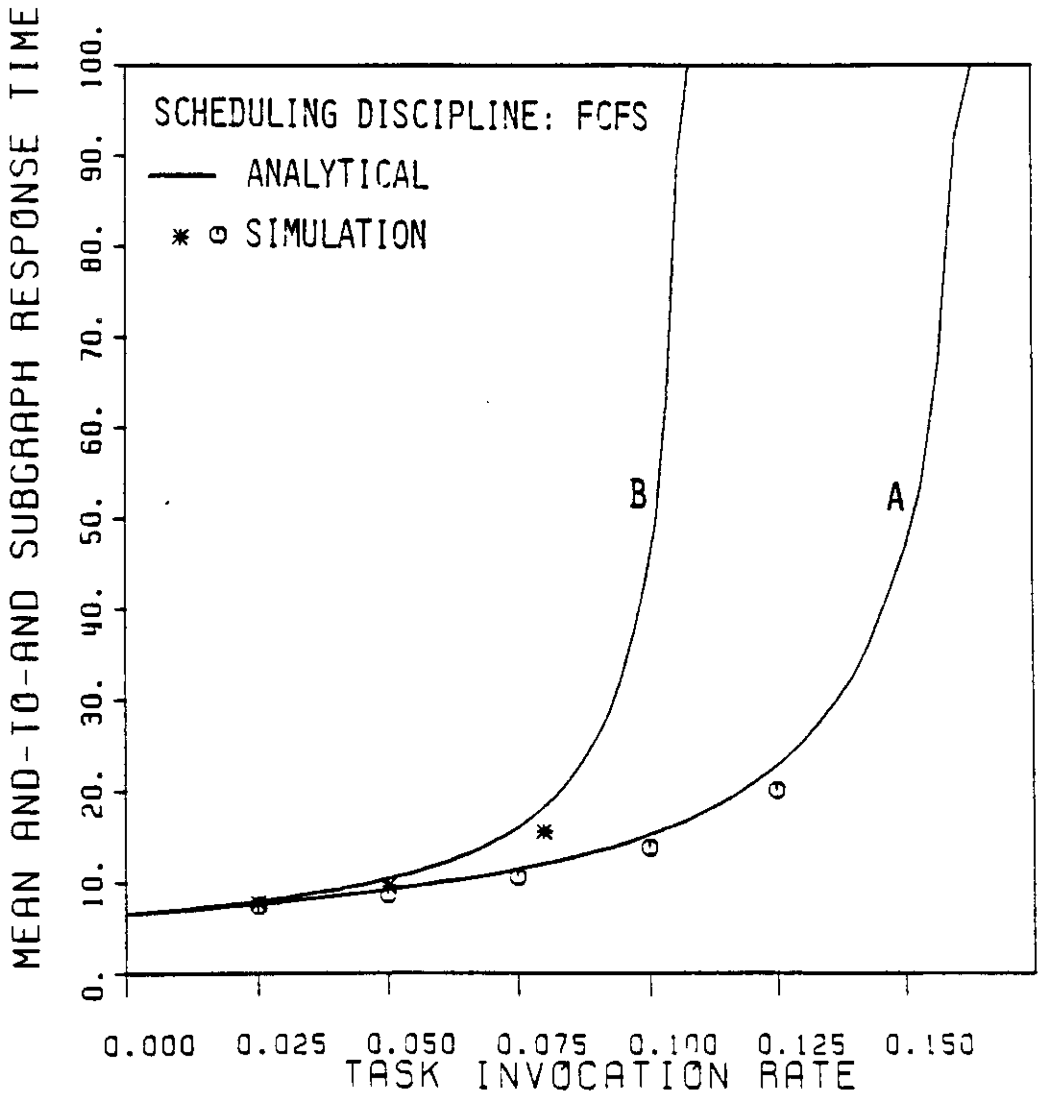


Figure 2.9 Mean Response Time for the And-Fork to And-Join Subgraph of the Sample Control-Flow Graph (Figure 2.1a)

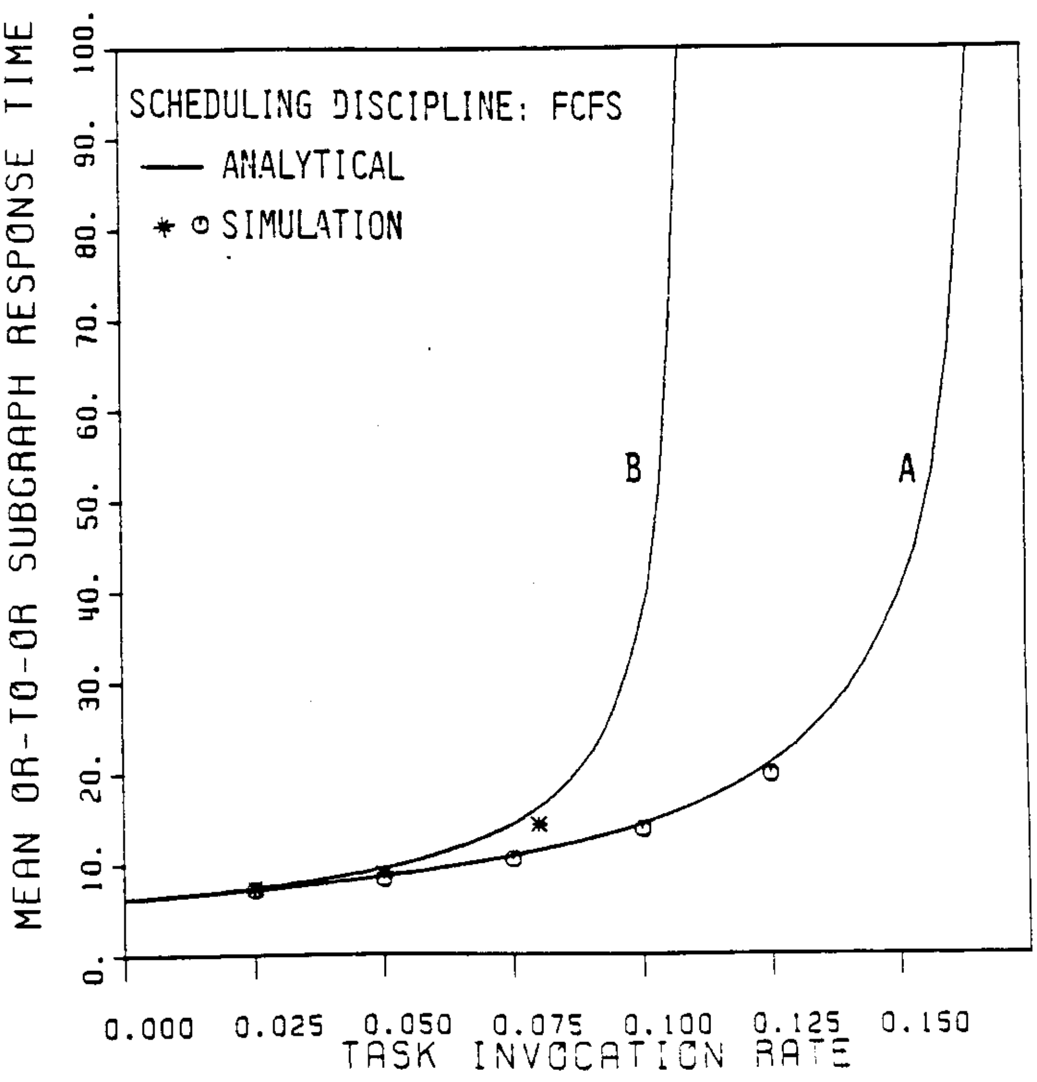


Figure 2.10 Mean Response Time for the Or-Fork to Or-Join Subgraph of the Sample Control-Flow Graph

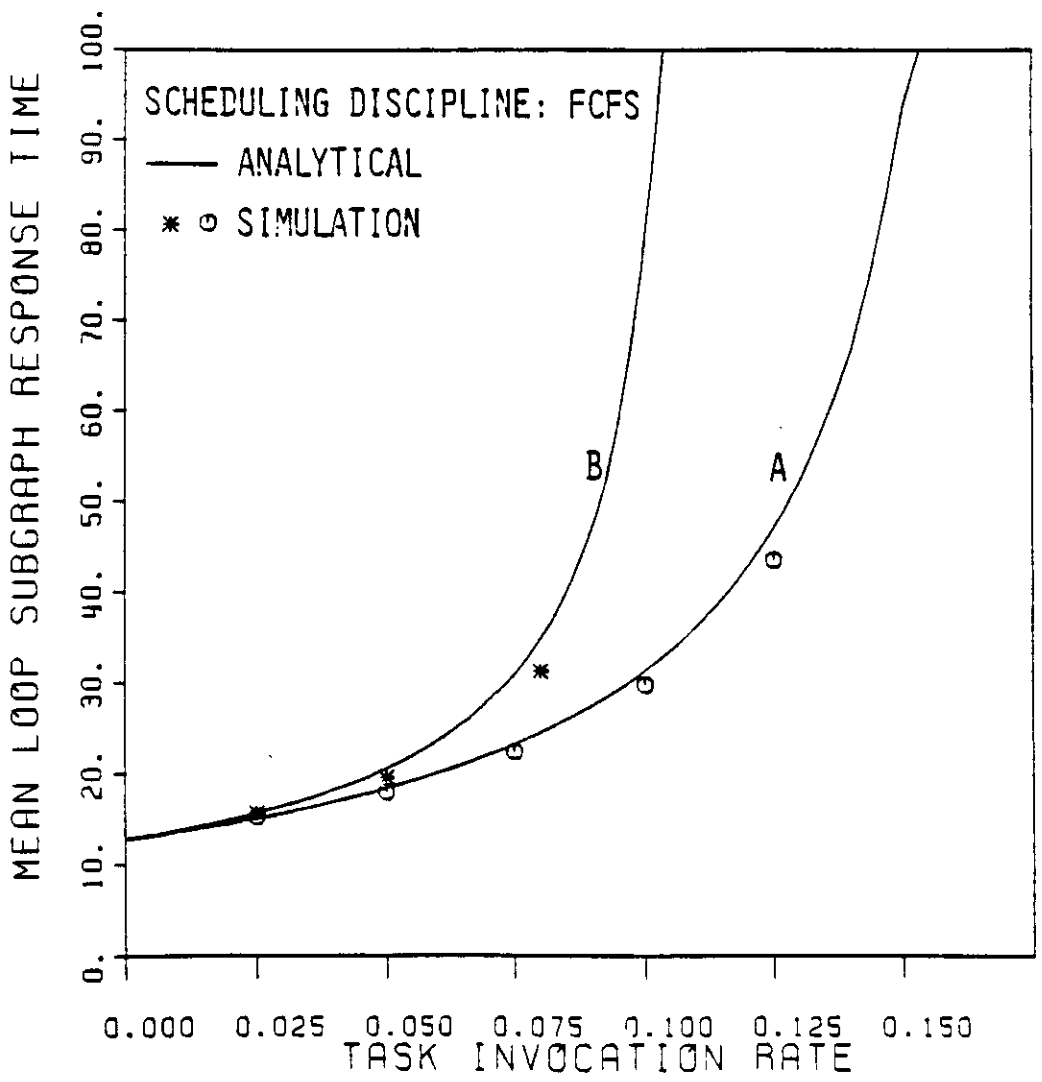


Figure 2.11 Mean Response Time for the Loop Subgraph of the Sample Control-Flow Graph

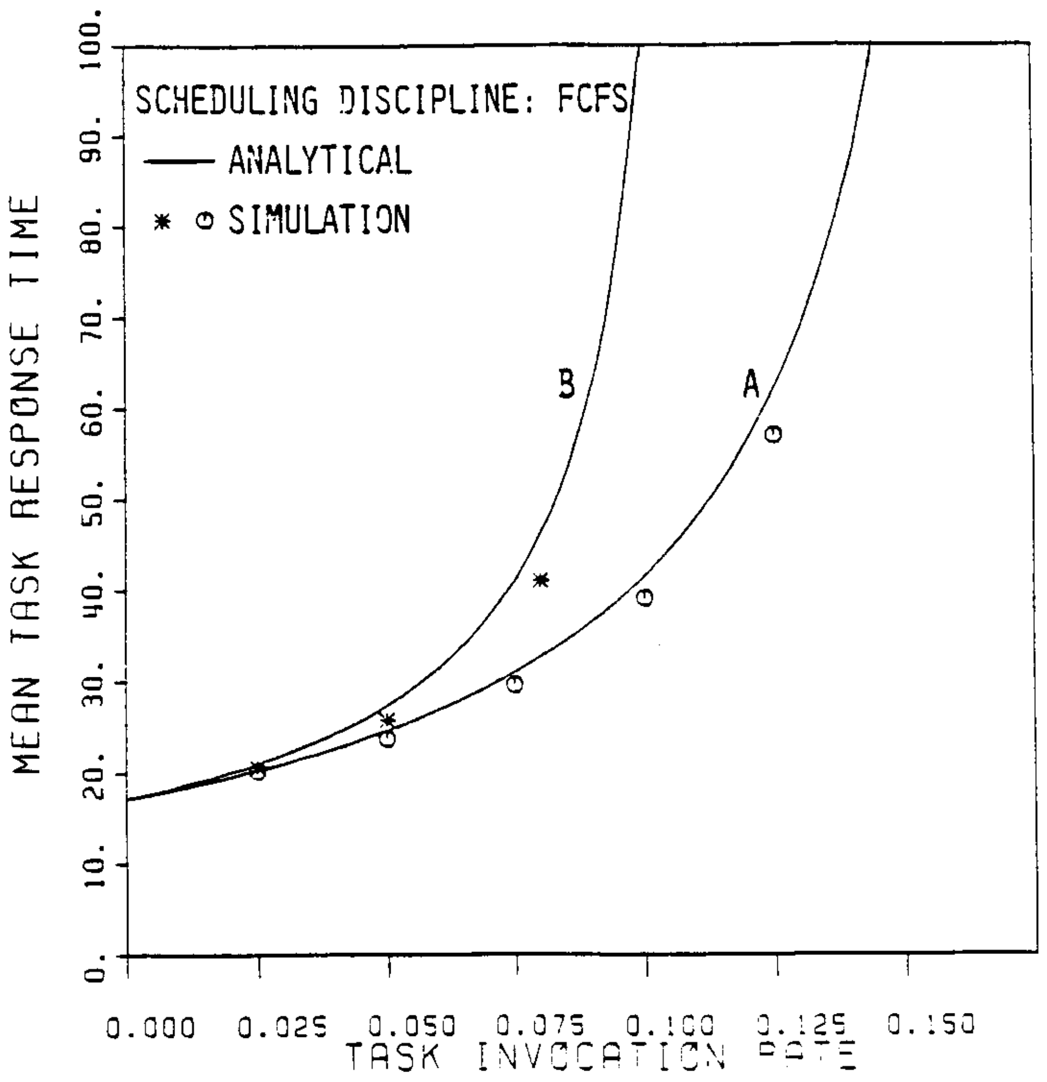


Figure 2.12 Mean Task Response Time for the Sample Control-Flow Graph

times of sequential threads, the and-fork to and-join, the or-fork to or-join, and the loop, and finally obtain the PTP time for the entire graph. The comparisons of analytical PTP time predictions to simulation measurements for all module assignments in Table 2.2 are shown in Table 2.3. These simulation experiments reveal that the analytic estimations show a relative error of less than 10% even when the heaviest loaded computer has a processor utilization up to 70%. Besides this control-flow graph, we have also studied the performance of the analytical model for various types of control-flow structures. The fact that mean response times from the analytical model compare closely with that of simulations reveals that the assumptions used in the analytical model (independent and Poisson module invocation arrivals) are good approximations for response time estimations.

The PAWS simulation is very time-consuming. Depending on task invocation rates and control-flow graphs, each simulation point requires five to eight hours of VAX-11/780 processing time. While for the analytical model, the response time computation for a given module assignment under various loading environments requires less that one minute of CPU time. This represents a reduction of three orders of magnitude in computation time!

We now describe the model validation via the DPAD simulator[1]. The DPAD system is a RTDPS which processes radar return signals for space defense applications. The DPAD simulator provides detailed operating system operations for module

[1]The DPAD simulator was originally developed at TRW and subsequently enhanced at UCLA to include facilities for measuring IMC data, module execution time and invocation statistics.

Assignments	Task Invocation Rates (Per Sec)	Maximum Processor Utilization	Simulated Task Response Times (sec)	Analytical Predictions (sec)	Relative Error
A	0.05	0.294	23.67	24.64	+4.01%
A	0.1	0.588	39.09	41.59	+6.40%
B	0.05	0.444	25.75	27.33	+6.14%
B	0.08	0.710	40.97	46.10	+12.52%
C	0.05	0.344	23.96	25.01	+4.38%
C	0.1	0.688	41.48	45.95	+10.78%
D	0.05	0.281	24.10	24.54	+1.83%
D	0.1	0.562	39.47	40.85	+3.50%
E	0.05	0.288	23.43	24.18	+3.20%
E	0.1	0.575	38.23	40.38	+5.62%

Table 2.3 Comparisons of Simulation Results with Analytical Predictions

scheduling and IPC message exchanges among computers. Further, non-Poisson task invocation arrivals are used. Its task control-flow graph is shown in Figure 2.13. The module assignment and module priorities are shown in Table 2.4. The processing thread for precision track function is indicated by shaded modules in Figure 2.13. For input data to the analytical model, we collected the IMC data, module execution times (Table 2.5) and invocation rates in every 100-msec time interval from the DPAD simulator. Since the DPAD System uses a head-of-line (HOL) priority module scheduling policy rather than FCFS, queueing formulas were derived to compute the module response times for this scheduling discipline (See Appendix C). The PTP time was generated for each of these time intervals. To obtain the 90% confidence intervals for the task response time, the simulation was repeated five times. From Figure 2.14, we note that the PTP time predictions are close to the simulation measurements. This indicates that the model also provides a good response time estimation for non-Poisson task invocation arrivals with priority module scheduling policy and IPC overhead.

2.4 MODEL APPLICATIONS

The proposed model can be used to study the effect on response time of such design issues as module assignment and precedence relationships, module scheduling disciplines and database management algorithms. With the response time as a performance measure, the model can be used to study the tradeoffs among various design choices and to provide us insight into planning and evaluating distributed systems.

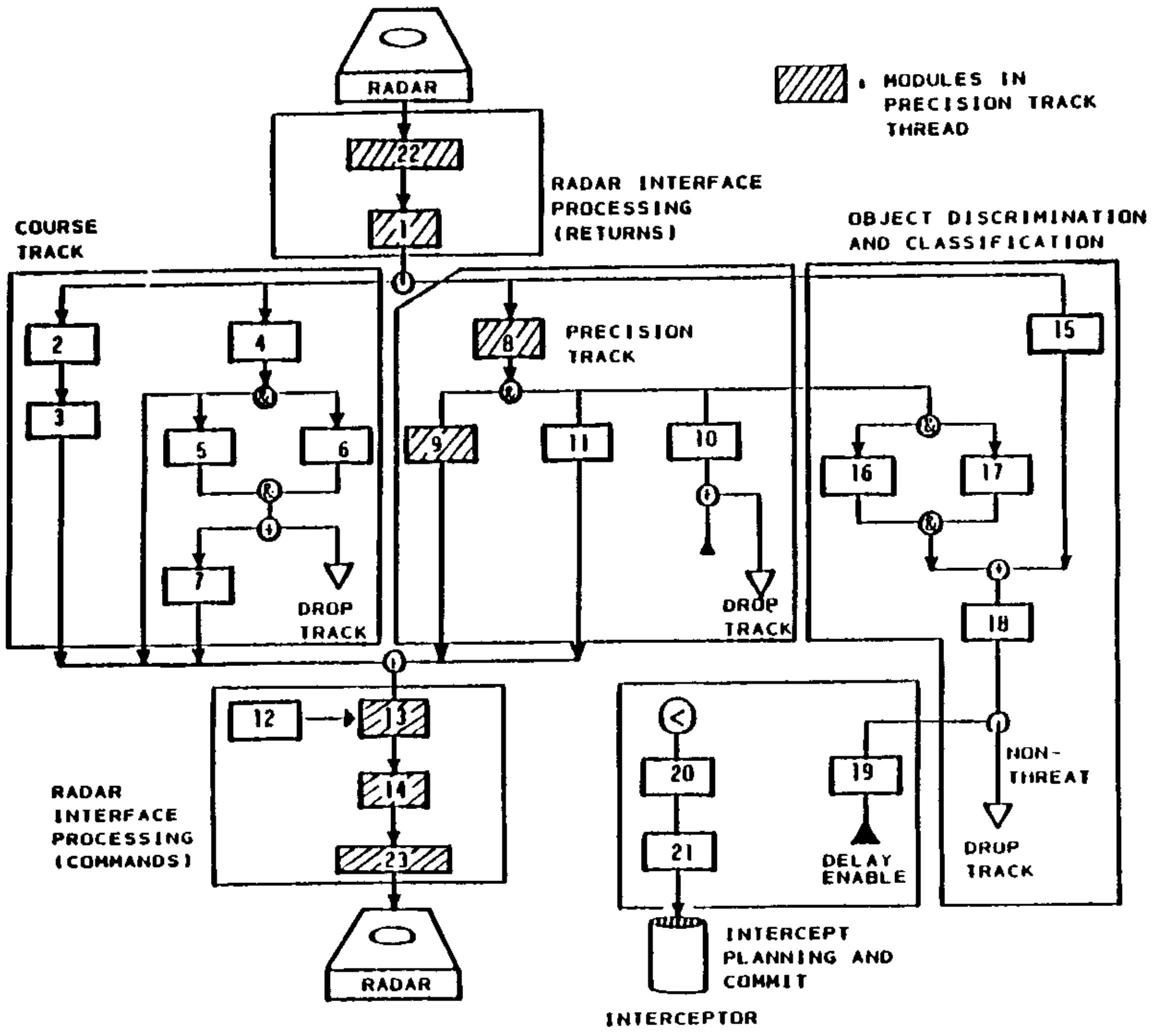

RADAR
22
1
RADAR INTERFACE PROCESSING (RETURNS)
▨ • MODULES IN PRECISION TRACK THREAD
COURSE TRACK
2
3
4
5
6
7
DROP TRACK
PRECISION TRACK
8
9
11
10
DROP TRACK
OBJECT DISCRIMINATION AND CLASSIFICATION
15
16
17
18
NON-THREAT
DROP TRACK
12
13
14
23
RADAR INTERFACE PROCESSING (COMMANDS)
RADAR
20
21
19
DELAY ENABLE
INTERCEPT PLANNING AND COMMIT
INTERCEPTOR

Figure 2.13 The Task Control-Flow Graph for the DPAD System

COMPUTERS	MODULE ASSIGNMENT
CPU 1	$M_1(1)$, $M_2(1)$, $M_4(1)$, $M_6(1)$, $M_8(1)$, $M_{10}(1)$, $M_{16}(1)$, $M_{22}(4)$
CPU 2	$M_3(1)$, $M_5(1)$, $M_9(1)$, $M_{17}(1)$, $M_{18}(1)$, $M_{19}(5)$, $M_{20}(6)$, $M_{21}(6)$
CPU 3	$M_7(1)$, $M_{11}(1)$, $M_{12}(1)$, $M_{13}(3)$, $M_{14}(2)$, $M_{15}(1)$, $M_{23}(4)$

$M_x(i)$: Module x with priority i. $M_x(i)$ has higher priority than $M_y(j)$ if $i > j$.

Table 2.4 A Module Assignment for the DPAD System

MODULES	MEAN EXECUTION TIME (ms)	SQUARED COEF. OF VARIATION
1	0.157043	0.000163
2	0.313522	0.023414
3	0.397477	0.000066
4	0.422061	0.885611
5	0.379197	0.000042
6	0.321836	0.252115
7	0.325322	0.178972
8	1.128163	0.007556
9	0.659989	0.000000
10	0.535785	0.000236
11	0.000000	0.000000
12	0.000000	0.000000
13	0.334381	0.002888
14	0.131086	0.003647
15	0.000000	0.000000
16	0.717703	0.000090
17	1.017310	0.000000
18	0.656880	0.002384
19	3.339637	0.000003
20	6.695341	0.000012
21	0.730000	0.000000
22	0.080373	0.017631
23	0.162269	0.005265

Table 2.5 Module Execution Times (Including Output IPC) for the DPAD System Averaged over 35 100-msec Time Intervals

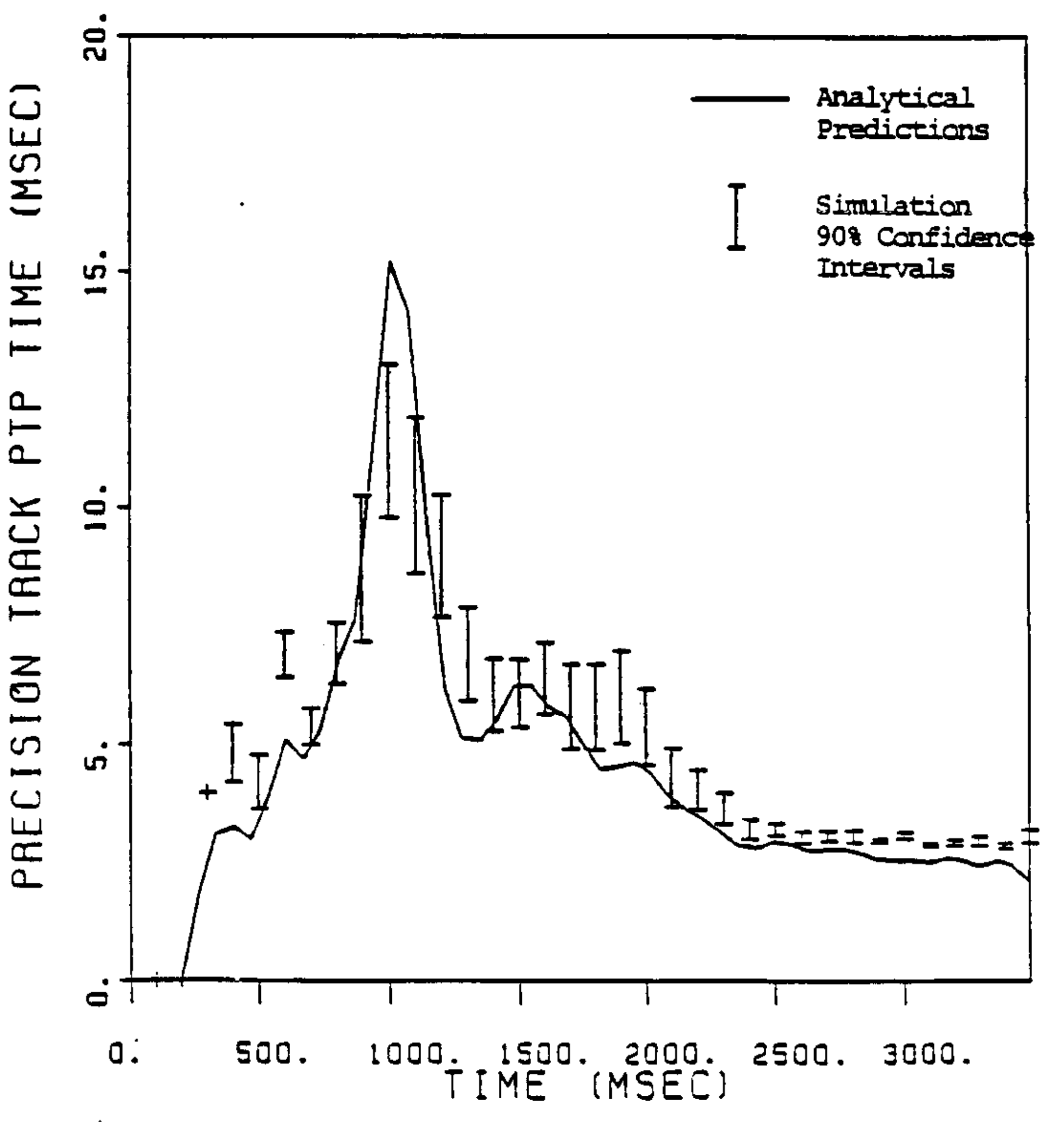


Figure 2.14 Comparing Analytical Predictions with the DPAD Simulation Results

The assignment of modules to computers is an important problem in distributed processing system design. Module assignment affects the response time, throughput, and system reliability. The factors that affect the module assignments are: (a) computer processing capacities and their utilization factors, (b) IMC among modules, and (c) logical and precedence relationships among modules. Several approaches to the assignment problem in distributed systems have been proposed [STON77, RAO79, MA82, CHOU82, SHEN85]. However, each of these approaches has its shortcomings such as neglecting queueing effect and precedence relationships. Therefore, the 'optimal' module assignments generated by them do not provide low response times on the actual systems. Our proposed model takes both computer load and precedence relationships into consideration. For a given module assignment, and module scheduling policy, the task response time can be estimated from the proposed model. In the following chapters, we shall use the proposed model to (1) investigate the performance impacts in terms of response time due to module precedence relationships; (2) develop a new module assignment algorithm for RTDPS. This study provides insight into the interrelationship among precedence relationships, module assignment and task response time.

2.5 SUMMARY

A new task response time model is presented for estimating the PTP time for distributed processing systems. The model maps the module response times into the task control-flow graph as arc weights and estimates the PTP time from the weighted task control-flow graph model. Since this approach considers the queueing effects, the interconnection network delays, and the logical relationships among modules. the model provides accurate PTP time prediction. Simulation experiments reveal that the proposed model provides fairly accurate PTP time. The model can be used to study module assignment problem and the effect of precedence relationships among modules on the PTP time. In addition, it can be used to study other design issues such as module scheduling policy, database management algorithm, etc. Thus this model serves as a valuable tool for the systematic planning and designing of distributed processing systems.

CHAPTER 3

PRECEDENCE RELATIONSHIP EFFECTS ON RESPONSE TIMES

3.1 INTRODUCTION

Precedence relationships (PR) among modules of an application task are inter-module synchronization requirements which require each module not to start its execution until all its preceding modules have finished their executions. Some common types of PR among modules such as sequential thread, and-fork to and-join, and or-fork to or-join have already been discussed in the previous chapter. Here, we shall investigate the impacts of module PR on response time. The work that illustrates these impacts for distributed systems was first documented in [LAN85]. By using simulation experiments, [LAN85] demonstrates how PR affects response times. In this chapter, we mainly use analytical techniques to portray the interrelationships between PR and response times under different loading environments. And, we shall attempt to interpret the reasoning behind these relations. Finally, heuristic rules for module assignment are derived to account for the PR effects. A simulation experiment will be presented as an example to show the importance of PR effects and the usefulness of these rules in module assignment.

3.2 PR EFFECTS FOR A CONSECUTIVE MODULE PAIR

There exists various types of module PR such as sequential thread, fork and join. For the reason of mathematical tractability, here we shall concentrate on sequential threads which may possibly contain an or-fork as shown in Figure 2.8(F). To isolate the effect due to PR, we assume there is no IMC among modules and no IPC overhead in the distributed system. Therefore, processing load on computers is solely due to module executions. Given a module assignment, modules are enabled on the processors according to the sequence as depicted in the task control-flow graph. Each processor has a distributed operating system to schedule modules for executions. The modules' response times on a processor will depend on the module scheduling policy. Since first-come first-serve (FCFS) policy is the most commonly used scheduling discipline on computer systems, we shall focus on the PR effect under this policy.

In order to illustrate the PR effect, it is desirable to compare the response times for two identical systems except one system possesses PR among modules while the other does not. To start with a simple example, let us consider two processors in Figure 3.1. In this example, processor #1 is allocated with a sequential thread of two modules, while processor #2 is assigned with another two independent modules. Let M_i denote module i. Suppose M_1 and M_3, and M_2 and M_4 have identical execution time distributions and invocation rates respectively. On processor #1, M_1 and M_2 have PR as, with probability p, M_2 is enabled and placed at the end of the processor's job queue (according to FCFS discipline) immediately after a M_1's

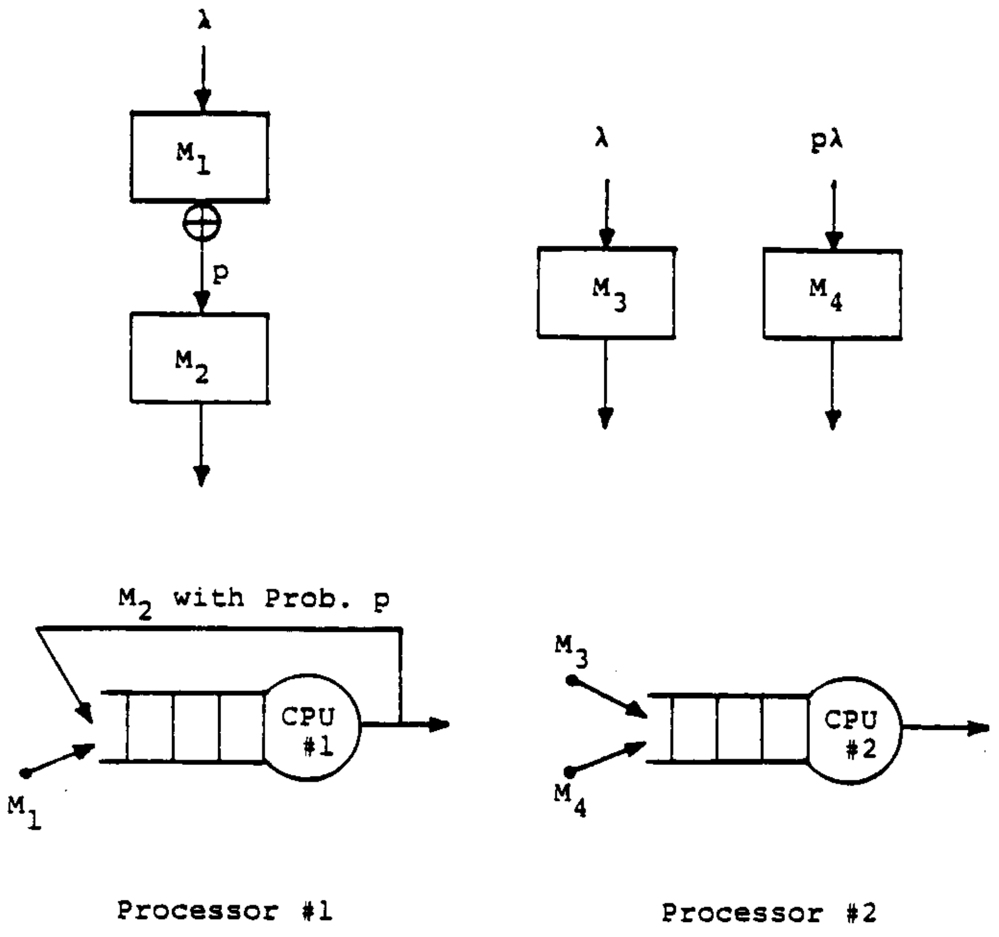


Figure 3.1 Two Processors: To Contrast PR Effect

execution is completed. For processor #2, M_3 and M_4 are independently invoked by different preceding modules or sources, thus they do not possess PR. Note that both processors #1 and #2 have identical processing load as well as module execution time distributions, any discrepancy in modules' response times on these two processors is solely due to the PR effect. The factors that cause the PR effect include: module execution time distributions, and ratios of average execution times for consecutive module pairs (i.e., M_1 and M_2 on processor #1). These factors will be elaborated later in this chapter.

Now, let us compute the modules' response times on both processors. According to Section 2.2.3, M_1, M_3 and M_4 are entry modules and M_2 is a non-entry module for this example. Invocations for these entry modules are assumed to be independent and their interarrival times have Poisson distributions. Let the invocation rates for M_i be λ_i. Thus we have $\lambda_1 = \lambda_3 = \lambda$ and $\lambda_2 = \lambda_4 = p\,\lambda$. Suppose the mean and second moment of execution time for M_i are $\bar{x}_i$ and $\overline{x_i^2}$ respectively. Then the processor utilization due to M_i is $\rho_i = \lambda_i\,\bar{x}_i$. In addition, let the coefficient of variation of execution time for M_i be c_i.

Clearly, processor #2 becomes a M/G/1 queueing system with two types of 'customers' M_3 and M_4. Let us use W_i to denote the average waiting (queueing) time for M_i. By M/G/1 queue results, the mean module waiting times are given by

$$W_3 = W_4 = \frac{W_{o2}}{1 - \rho_3 - \rho_4} \tag{3.1}$$

where:

$W_{o2} =$ mean residual module execution time on processor #2

$$= \frac{1}{2} (\lambda_3 \overline{x_3^2} + \lambda_4 \overline{x_4^2})$$

$$= \frac{1}{2} [\lambda (1 + c_3{}^2) \bar{x}_3{}^2 + p \lambda (1 + c_4{}^2) \bar{x}_4{}^2]$$

By the method in Appendix B, the average waiting times W_1 and W_2 on processor #1 can be obtained as follows. The mean waiting time for M_1 under FCFS scheduling policy is the average time to complete the current module execution plus all module invocations waiting in the job queue when the invocation for M_1 arrives. Thus, we have

$$W_1 = W_{o1} + \bar{n}_1 \bar{x}_1 + \bar{n}_2 \bar{x}_2 \tag{3.2}$$

where:

$W_{o1} =$ mean residual module execution time on processor #1

$$= \frac{1}{2} (\lambda_1 \overline{x_1^2} + \lambda_2 \overline{x_2^2})$$

$$= \frac{1}{2} [\lambda (1 + c_1^2) \bar{x}_1{}^2 + p \lambda (1 + c_2^2) \bar{x}_2{}^2]$$

$\bar{n}_i =$ average number of invocations for M_i waiting in the job queue.

To find the waiting time for M_2, we need to keep track of the queueing behavior since the arrival of the invocation for M_1. Let us consider a particular tagged invocation for M_1. After the completion of this tagged M_1's execution, its succeeding tagged invocation for M_2 is placed at the end of the job queue. The waiting time for this tagged M_2 invocation consists of three components. The first component is due to the new invocations for M_1 that arrive during the waiting plus execution time of the tagged M_1 invocation. The second component is due to the executions of all M_2 invocations which are possibly enabled by M_1 invocations waiting in the job queue when the tagged M_1 invocation arrives at the system. The last component is due to the execution of a M_2 invocation possibly enabled if the module in execution upon the arrival of the tagged M_1 invocation is M_1. Thus, by adding these components together, we have

$$W_2 = (W_1 + \bar{x}_1) \lambda \bar{x}_1 + p \ \bar{n}_1 \bar{x}_2 + p \ \rho_1 \bar{x}_2 \tag{3.3}$$

Apply Little's result [LITT61] (i.e., $\bar{n}_i = \lambda_i W_i$) and put $\rho_i = \lambda_i \bar{x}_i$ in Eq.(3.2) and (3.3). Then, these equations become

$$W_1 = W_{o1} + \rho_1 W_1 + \rho_2 W_2 \tag{3.4}$$

and

$$W_2 = (W_1 + \bar{x}_1) \rho_1 + \rho_2 W_1 + p \ \rho_1 \bar{x}_2 \tag{3.5}$$

From Eq.(3.4) and (3.5), W_1 and W_2 can be solved as

$$W_1 = \frac{W_o + \rho_1 \rho_2 (\bar{x}_1 + p \ \bar{x}_2)}{1 - \rho_1 - \rho_2 (\rho_1 + \rho_2)} \tag{3.6}$$

and

$$W_2 = \frac{W_o + \rho_1 \rho_2 (\bar{x}_1 + p\,\bar{x}_2)}{1 - \rho_1 - \rho_2 (\rho_1 + \rho_2)} (\rho_1 + \rho_2) + \rho_1 (\bar{x}_1 + p\,\bar{x}_2) \tag{3.7}$$

where $W_o = W_{o1} = W_{o2}$ as the corresponding modules on both processors have identical execution time distributions.

Recall that processors #1 and #2 have equal processing load except processor #1 has PR effect but not processor #2. To illustrate the PR effect, the thread response time for M_1 and M_2, which is $W_1 + \bar{x}_1 + W_2 + \bar{x}_2$, is compared with the sum of response times for M_3 and M_4, which is $W_3 + \bar{x}_3 + W_4 + \bar{x}_4$. Since $\bar{x}_1 = \bar{x}_3$, $\bar{x}_2 = \bar{x}_4$ and these mean execution times are constants, we define the *thread waiting time ratio* $R_w = \frac{W_1 + W_2}{W_3 + W_4}$. If $R_w \neq 1$, then the modules' response times on processor #1 are shorter or longer than those on processor #2 solely due to the PR between the consecutive modules M_1 and M_2. Since $W_3 = W_4$, the thread waiting time ratio can be rewritten as

$$R_w = \frac{1 + W_2 / W_1}{2\,W_3 / W_1} \tag{3.8}$$

In the following, we shall separately consider these two factors of R_w: W_2 / W_1 and W_3 / W_1 to examine the conditions under which PR effects provide better response times (i.e., $R_w < 1$).

3.2.1 The Waiting Time Ratio: W2/W1

The ratio W_2 / W_1 can be obtained from Eq.(3.6) and (3.7). We put $W_o = \frac{1}{2} [\, \lambda(1 + c_1^2) \bar{x}_1^2 + p\lambda(1 + c_2^2) \bar{x}_2^2 \,]$ and after simplification, the ratio can be expressed as

$$\frac{W_2}{W_1} = (\rho_1 + \rho 2) + \frac{2(1 + \rho_2/\rho_1)[\, 1 - \rho_1 - \rho_2 (\rho_1 + \rho_2)\,]}{(1 + c_1^2) + \rho_2/\rho_1(1 + c_2^2)\bar{x}_2/\bar{x}_1 + 2\rho_2 (1 + \rho_2/\rho_1)} \tag{3.9}$$

Let us define processor utilization for processor #1 or #2 as $\rho = \rho_1 + \rho_2 = \rho_3 + \rho_4$, and the *mean module execution time ratio* $r = \bar{x}_2 / \bar{x}_1 = \bar{x}_4 / \bar{x}_3$. Then, it yields $\rho_2 / \rho_1 = \rho_4 / \rho_3 = pr$, $\rho_1 = \rho_3 = \frac{\rho}{1 + pr}$ and $\rho_2 = \rho_4 = \frac{pr\rho}{1 + pr}$ where p is the enabling probability from M_1 to M_2. To further simplify Eq.(3.9) with these relations, we get

$$\frac{W_2}{W_1} = \rho + \frac{2\,[\, 1 + pr - \rho - pr\rho^2 \,]}{(\, 1 + c_1^2\,) + (\, 1 + c_2^2\,)\, pr^2 + 2pr\rho} \tag{3.10}$$

As expressed in the above equation, W_1 / W_2 is a function of p, r, ρ and c_i's. Therefore, we shall discuss this ratio under three selected types of distributions of module execution times below:

Case I: Deterministic Module Execution Times

As modules have fixed execution times, we put $c_1 = c_2 = 0$ into Eq.(3.10). Then, we have

$$\frac{W_2}{W_1} = \rho + \frac{2(1 + pr - \rho - pr\rho^2)}{1 + pr^2 + 2pr\rho} \tag{3.11}$$

Given the values for p (enabling probability) and ρ (processor loading), it is desirable to find r (execution time ratio for the consecutive module pair) such that $W_2 / W_1 < 1$. After some algebraic manipulation, the range of r such that $W_2 / W_1 < 1$ for all $\rho \varepsilon [0,1]$ is that

$$r < 1 + \sqrt{1 + \frac{1}{p}} \tag{3.12}$$

In case M_1 always enables M_2 (a regular sequential thread), $p = 1$. Hence, if $r > 1 + \sqrt{2}$, then $W_2 / W_1 < 1$.

Case II: Exponential Module Execution Times

If modules' execution times have exponential distributions, $c_1 = c_2 = 1$. Substitute these values in Eq.(3.10). We find

$$\frac{W_2}{W_1} = \rho + \frac{1 + pr - \rho - pr\rho^2}{1 + pr^2 + pr\rho} \tag{3.13}$$

From this equation, it can be shown that $W_2 / W_1 < 1$ if $r \geq 1$ for all p, $\rho \varepsilon [0,1]$. This means that if $W_2/W_1 < 1$ is independent of the enabling probability p and processor load ρ but only depends on the execution time ratio r for exponential module execution times.

Case III: Hyper-Exponential Module Execution Times

Since hyper-exponential distributions have higher coefficients of variation than those of exponential distributions, we select $c_1^2 = c_2^2 = 2$ as an illustrative example. From Eq.(3.10), we obtain

$$\frac{W_2}{W_1} = \rho + \frac{2\,(\,1 + pr - \rho - pr\rho^2\,)}{3\,(\,1 + pr^2\,) + 2pr\rho} \tag{3.14}$$

With the squared coefficients of variation of module execution times equal to 2, it can be shown from Eq.(3.14) that W_2 / W_1 is always less than 1 for all p, $\rho \in [0,1]$ and real positive r. In other words, the PR effect always yields less waiting time for M_2 comparing with M_1 if modules have highly variating execution times.

3.2.2 The Waiting Time Ratio: W3/W1

From Eq.(3.1) and (3.6), the waiting time ratio W_3 / W_1 is given by

$$\frac{W_3}{W_1} = \frac{[\lambda(1 + c_3^2)\bar{x}_3^2 + p\lambda(1 + c_4^2)\bar{x}_4^2\,][1 - \rho_1 - \rho_2(\rho_1 + \rho_2)]}{(1 - \rho_3 - \rho_4)[\lambda(1 + c_1^2)\bar{x}_1^2 + p\lambda(1 + c_2^2)\bar{x}_2^2 + 2\rho_1\rho_2(\bar{x}_1 + p\bar{x}_2)]} \tag{3.15}$$

After simplifying, Eq.(3.15) can be rewritten as

$$\frac{W_3}{W_1} = \frac{[\,(\,1 + c_3^2\,) + pr^2\,(\,1 + c_4^2\,)\,]\,[\,1 + pr(\,1 + \rho\,)\,]}{(\,1 + pr\,)\,[\,(\,1 + c_1^2\,) + pr^2\,(\,1 + c_2^2\,) + 2pr\rho\,]} \tag{3.16}$$

Once again, we also compute the ratio W_3 / W_1 for three cases of module execution time distributions as discussed above.

Case I: Deterministic Module Execution Times

Substitute $c_i = 0$ for $i = 1$ to 4 in Eq.(3.16). The ratio W_3/W_1 is given by

$$\frac{W_3}{W_1} = \frac{(1+pr^2)[1+pr(1+\rho)]}{(1+pr)[1+pr^2+2pr\rho]} \tag{3.17}$$

Although Eq.(3.17) takes a different form from Eq.(3.11), it can be readily shown from the above equation that $W_3 / W_1 > 1$ if $r > 1 + \sqrt{1 + \frac{1}{p}}$ for all $\rho \varepsilon [0,1]$. Coincidentally, this condition is exactly identical to that for $W_2 / W_1 < 1$ if module execution times are deterministic. Similar coincidence also occurs in the following two cases.

Case II: Exponential Module Execution Times

We substitute $c_i = 1$ for $i = 1$ to 4 into Eq.(3.16) and obtain

$$\frac{W_3}{W_1} = \frac{(1+pr^2)[1+pr(1+\rho)]}{(1+pr)[1+pr^2+pr\rho]} \tag{3.18}$$

With exponential distributed execution times, we can show from Eq.(3.18) that $W_3 / W_1 > 1$ if $r > 1$ for all $p, \rho \varepsilon [0,1]$.

Case III: Hyper-Exponential Module Execution Times

Substituting $c_i^2 = 2$ for $i = 1$ to 4 into Eq.(3.16), it yields

$$\frac{W_3}{W_1} = \frac{(3+3pr^2)[1+pr(1+\rho)]}{(1+pr)[3+3pr^2+2pr\rho]} \tag{3.19}$$

With $c_i^2 = 2$, the module execution times have higher variations than exponential distributions. Once again, it can shown that $W_3 / W_1 > 1$ for all positive-valued r, and p and $\rho \in [0,1]$.

3.2.3 Cutting Points for the PR Effect

The three cases discussed above represent a spectrum of distributions for module execution times. The thread waiting time ratio R_w for these cases can be computed by substituting the values of W_2 / W_1 and W_3 / W_1 into to Eq.(3.8). Clearly, if $W_2/W_1 < 1$ and $W_3/W_1 > 1$, $R_w = \dfrac{1 + W_1/W_2}{2W_3/W_1} < 1$. Therefore,

(a) for deterministic module execution times, if $r > 1 + \sqrt{1 + \frac{1}{p}}$,

(b) for exponential module execution times, if $r > 1$,

(c) for hyper-exponential module execution times with squared coefficients of variations greater than 2.0, for all $r > 0$

we have $R_w < 1$ under any processor loading $\rho \in [0,1]$. From the ranges of r derived above, it can be concluded that whether the PR effect yields shorter response times (i.e., if $R_w < 1$) only depends upon the ratio of mean execution times for the consecutive module pair r, and the enabling probability p.

These relationships are depicted in Figures 3.2 to 3.4. In these figures, processor load is bounded by $\rho = \rho_1 + \rho_2 \leq 1$ by which the feasible loading environment is

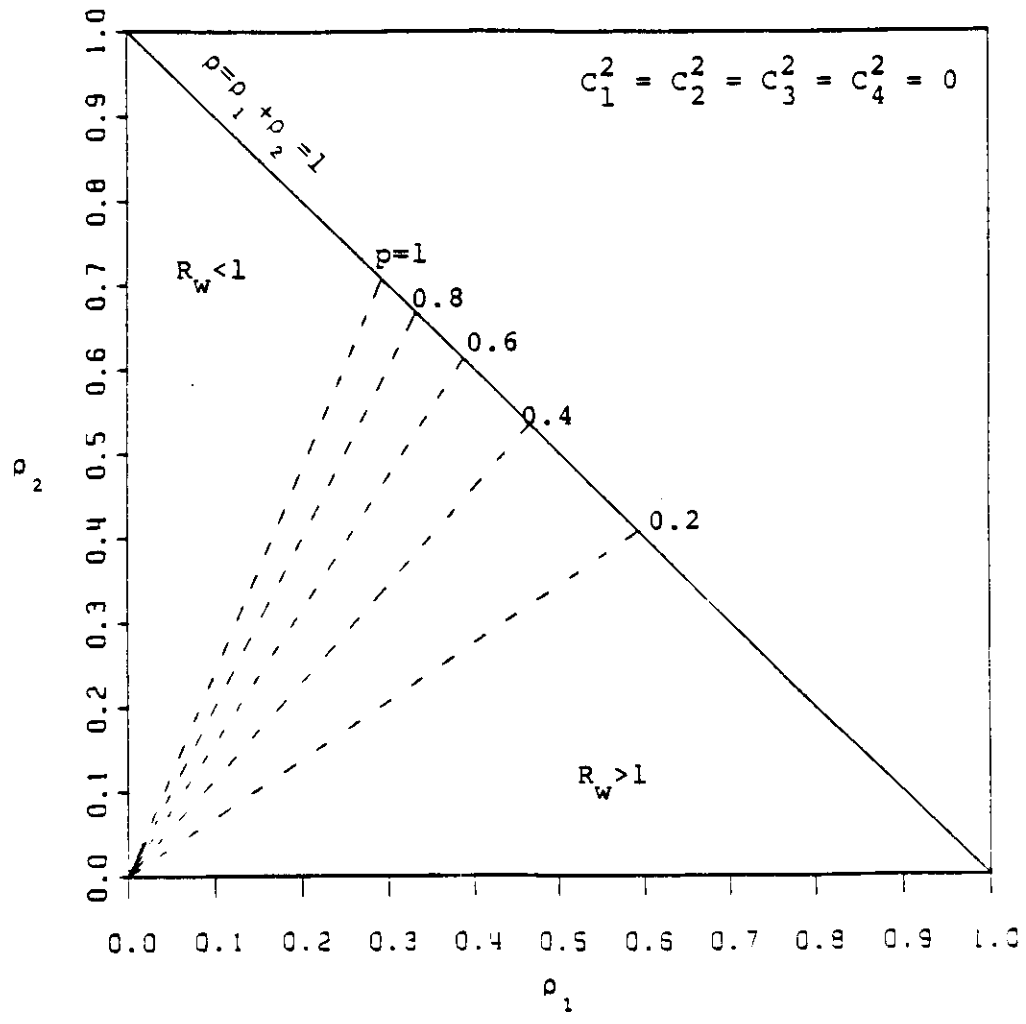


Figure 3.2 Conditions for the PR Effect for Deterministic Module Execution Times

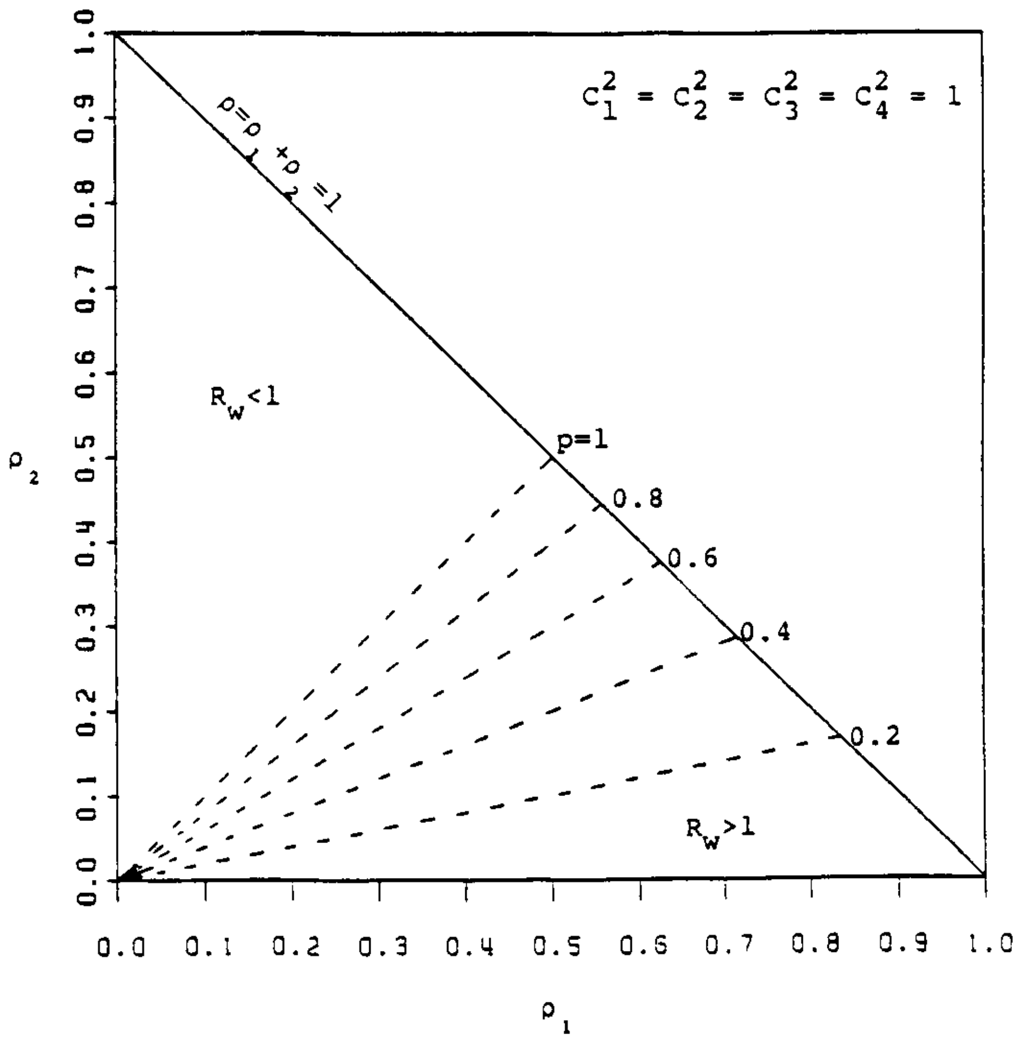


Figure 3.3 Conditions for the PR Effect for Exponential Module Execution Times

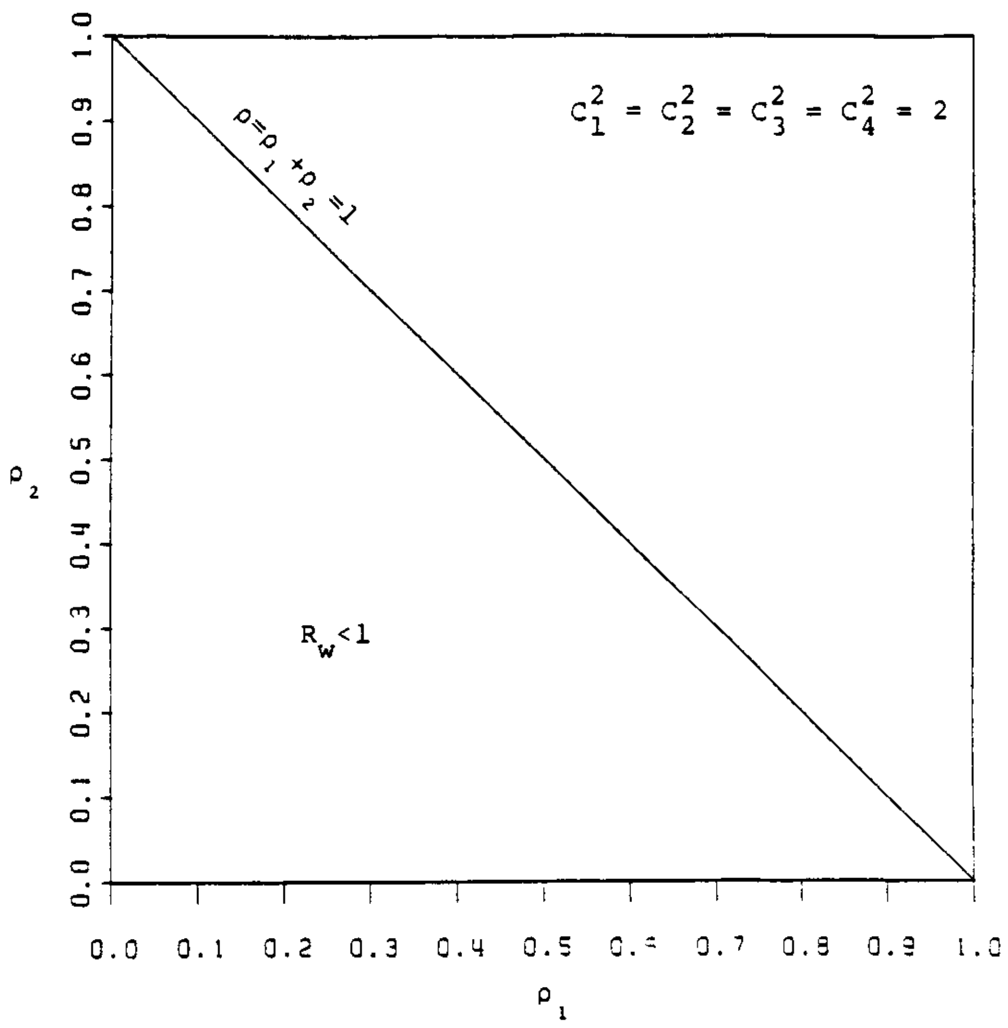


Figure 3.4 Conditions for the PR Effect for Hyper-Exponential Module Execution Times with Squared Coefficients of Variations Equal to 2

the triangular area at the lower left part of the quadrant. The dotted lines for various enabling probability p divide this triangular area into two regions: the lower region with $R_w > 1$ while the upper one with $R_w < 1$. This means that if ρ_2 / ρ_1 falls into the upper region for a specific p value, the thread response time can be improved by the PR effect; otherwise, the PR effect will prolong the thread response time. Further, although whether $R_w < 1$ is only determined by r and p for the given module execution time distributions, the value of R_w does depend on r and p as well as the processor utilization ρ. To consider $p = 1$ (M_1 and M_2 become a regular sequential thread), the thread waiting time ratio versus computer utilization for some typical execution time distributions and ρ_2 / ρ_1 values are plotted in Figures 3.5 to 3.7. These figures also show the values of ρ_2/ρ_1 (i.e., $\bar{x}_2/\bar{x}_1$ for $p = 1$) such that $R_w < 1$ for the selected types of module execution time distributions. Note that, due to the PR effect, the response times have more variation with computer utilization if module execution times have less variation.

3.3 INTUITIVE REASONING FOR THE PR EFFECT

Now we understand the major factors that cause the PR effect including: the characteristics of module execution time distributions and mean execution time ratio for the consecutive module pair.

If the module execution time distributions have large variations, PR effect yields better response times than those without PR effect. As shown in Figure 3.1, this is mainly due to M_2 invocations will never encounter any residual module execution,

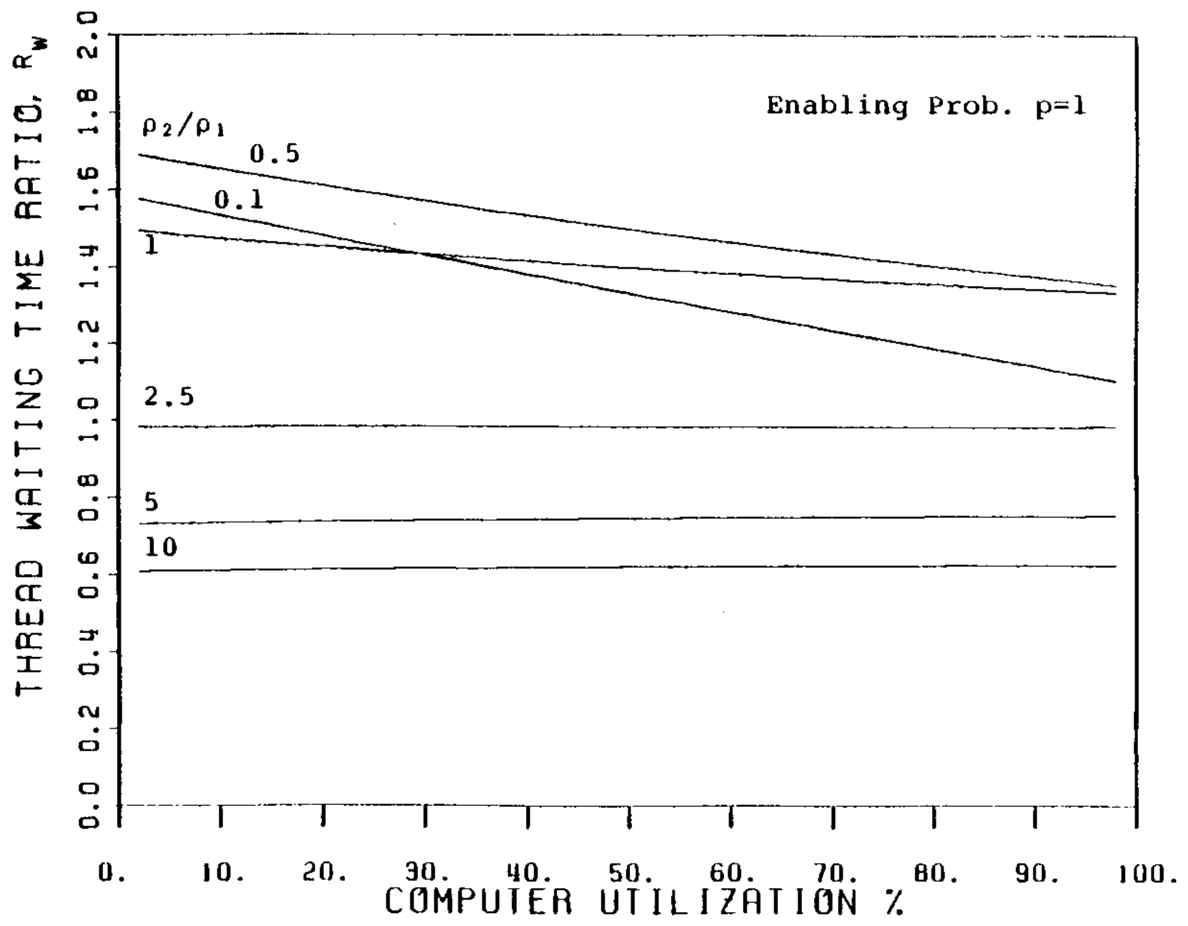


Figure 3.5 Thread Waiting Time Ratio as a Function of Computer Utilization and ρ_2/ρ_1 for Deterministic Execution Times

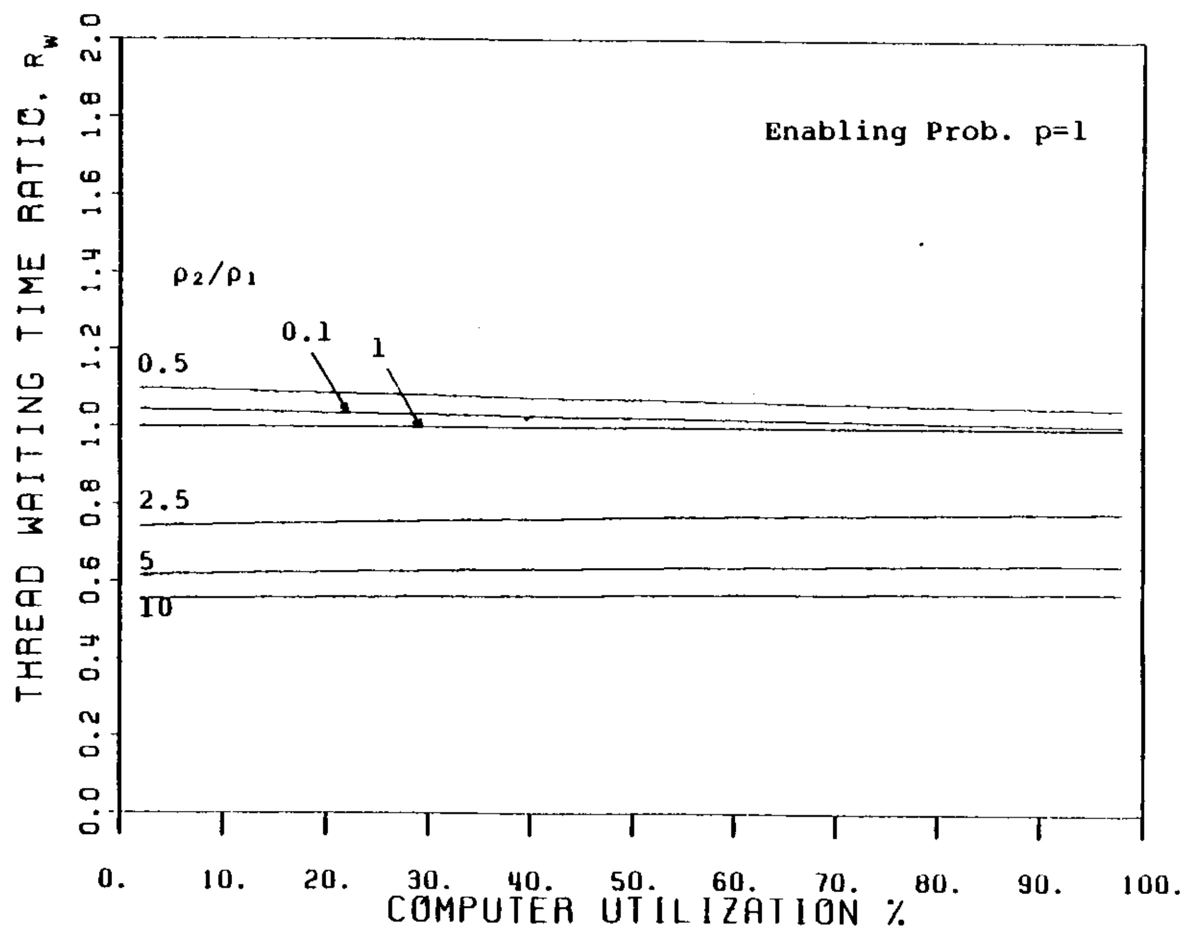


Figure 3.6 Thread Waiting Time Ratio as a Function of Computer Utilization and ρ_2/ρ_1 for Exponential Execution Times

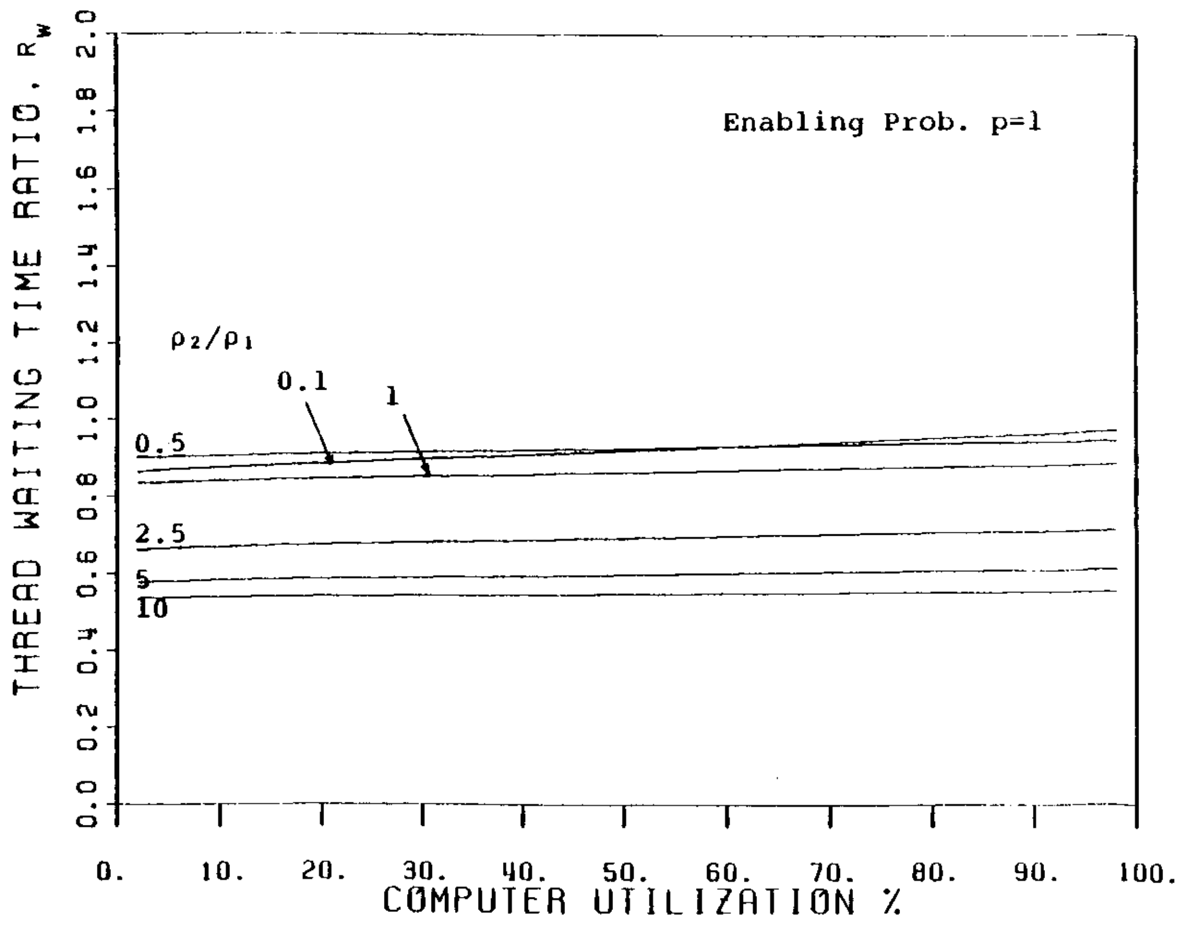


Figure 3.7 Thread Waiting Time Ratio as a Function of Computer Utilization and ρ_2/ρ_1 for Hyper-Exponential Execution Times with Squared Coefficients of Variation Equal to 2

which is possibly large for highly variating distribution functions, on processor #1 as M_2 is only enabled immediately after a M_1 invocation finishes its execution. However, the waiting times for M_3 and M_4 invocations on processor #2 always include the waiting for the current module in execution to complete.

Secondly, the mean module execution time ratio, r, plays an important role in the PR effect especially for modules which have more deterministic execution times. In general, if r is larger than a certain threshold, which depends on the modules' execution time distributions, PR effect will reduce response times. The reason for this phenomenon is similar to the explanation for the superior performance in terms of average customer waiting time of the shortest-job-first (SJF) scheduling policy which minimizes the waiting time by serving the shortest job first. Consider the systems in Figure 3.1. If r is small, then M_1 requires a longer execution time than M_2. Since M_1 invocation arrivals are random, new M_1 invocation arrivals will most likely find a M_1 invocation in execution. Including these two and other M_1 invocations waiting in the job queue, the processor needs to process these long jobs (M_1 invocations) before executing their succeeding M_2 invocations which require shorter execution times. Therefore, the queueing behavior is opposite to that of SJF scheduling policy. Conversely, if r is large, the execution time for M_1 is shorter than M_2. According to the PR requirement, M_1 invocations are executed prior to their succeeding M_2 invocations. Therefore, due to the similar reasoning for SJF scheduling discipline, the thread response time for M_1 and M_2 is improved because of the PR effect. In particular, when the processor utilization is low, the queueing behavior closely resembles to that of SJF policy. Thus the PR effect is more profound under lightly loaded environment as depicted in Figure 3.5.

3.3 PR EFFECT FOR SEVERAL CONSECUTIVE MODULES

Although we can extend the previous results to a sequential thread that consists of more than two consecutive modules, the number of variables involved in the thread waiting time ratio increases as the number of modules in the thread increases. Therefore, it is difficult to examine the PR effect and explore its tradeoffs with many variables. However, the characteristics of PR effect on response times for consecutive module pairs provides us understanding of the PR effect for a long sequential thread:

(1) If modules' execution times are highly variant, PR of a long sequential thread that is executed on a processor always yields shorter response times because invocations for succeeding modules in the thread need not incur the waiting for a large residual module execution time.

(2) If the ratio of mean execution times for two consecutive modules in the thread is larger than a certain threshold, the mean execution times for a module is always larger than that of its preceding module. According to the module PR, preceding modules with shorter execution times are processed before the succeeding modules which have longer execution times. Under these situations, the queueing behavior is similar to that of shortest-job-first scheduling. Therefore, PR among modules on a processor provides shorter response times.

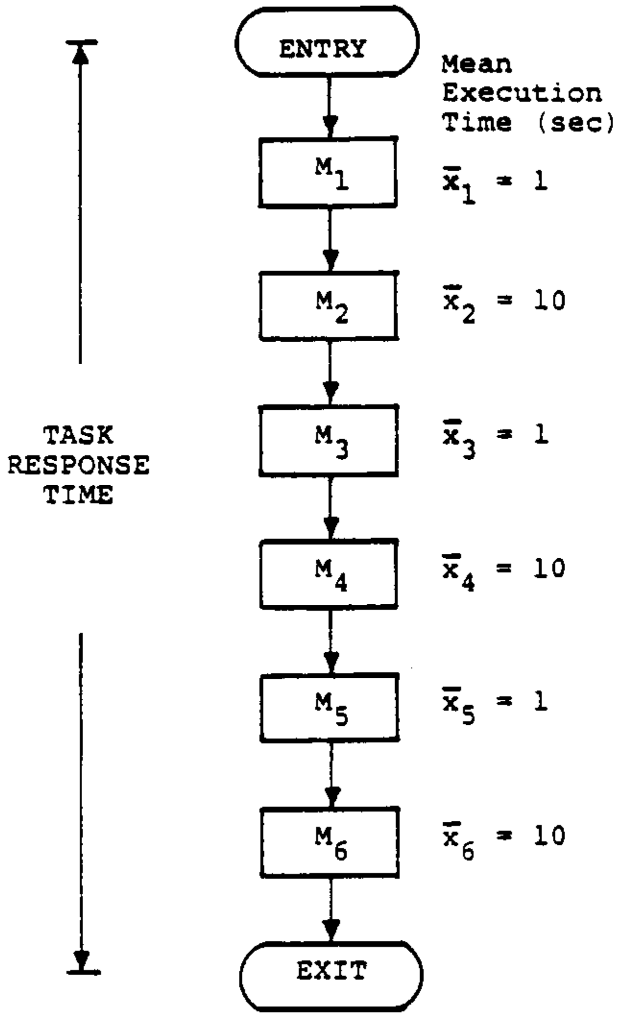
3.4 HEURISTIC RULES TO CONSIDER PR IN MODULE ASSIGNMENT

Based on the understanding on how PR among modules affect response times, we shall discuss some heuristic rules for module assignment. We need to emphasize that these rules are only based upon the considerations of PR effects. These module assignment rules include:

Rule #1: If module execution times have large variances, consecutive modules should be co-located on a same computer.

Rule #2: For the given module execution time distributions, if the mean execution time ratio of two consecutive modules fall in the range specified in Section 3.2.3 such that $R_w < 1$, the consecutive modules should be allocated to a same computer. Otherwise, if $R_w > 1$, these two modules should be separated on two distinct processors to avoid the unfavorable PR effect.

To demonstrate the effect of PR and to validate these rules, let us consider a task which consists of six modules and are assigned to three computers (Figure 3.8). All modules have deterministic execution times, and form a long sequential thread. By applying these rules to this task, we select three assignments: A, B and C as shown in Figure 3.8. Assignment A is obtained based on Rule #2 to yield favorable PR effect. In this assignment, the ratio r on each CPU is 10 which is far greater than the required threshold $1 + \sqrt{2}$ for deterministic execution times to produce $R_w < 1$. Conversely, assignment B is so selected to demonstrate the unfavorable PR effect as r is 0.1 on CPU2 and CPU3 in this assignment. Assignment C is another load balanced



Assign-ments	CPU#1	CPU#2	CPU#3
A	M_1, M_2	M_3, M_4	M_5, M_6
B	M_1, M_6	M_2, M_3	M_4, M_5
C	M_1, M_4	M_2, M_5	M_3, M_6

Figure 3.8 Task Control-Flow Graph & Module Assignments for Illustrating PR Effects

assignment but with no PR effect on processors as each module is enabled by its preceding module from a remote computer. The task response times for these three assignments are measured from PAWS simulation [BERR82], and portrayed in Figure 3.9. Due to PR effects, assignment A yields the best response time performance. Assignment B gives the worst task response time. And, assignment C provides moderate task response time.

3.5 SUMMARY

In this chapter, our study reveals that PR does influence module response times and thus affects the overall task response time. It is difficult to understand the exact effects for various types of PR among modules. However, by considering two consecutive modules, we are able to illustrate the PR effects. Heuristic rules are developed for considering PR effects in module assignment. Since other factors such as load balancing and IPC also affect the performance of module assignments, a new algorithm for module assignment which takes all these factors into considerations will be introduced in the next chapter.

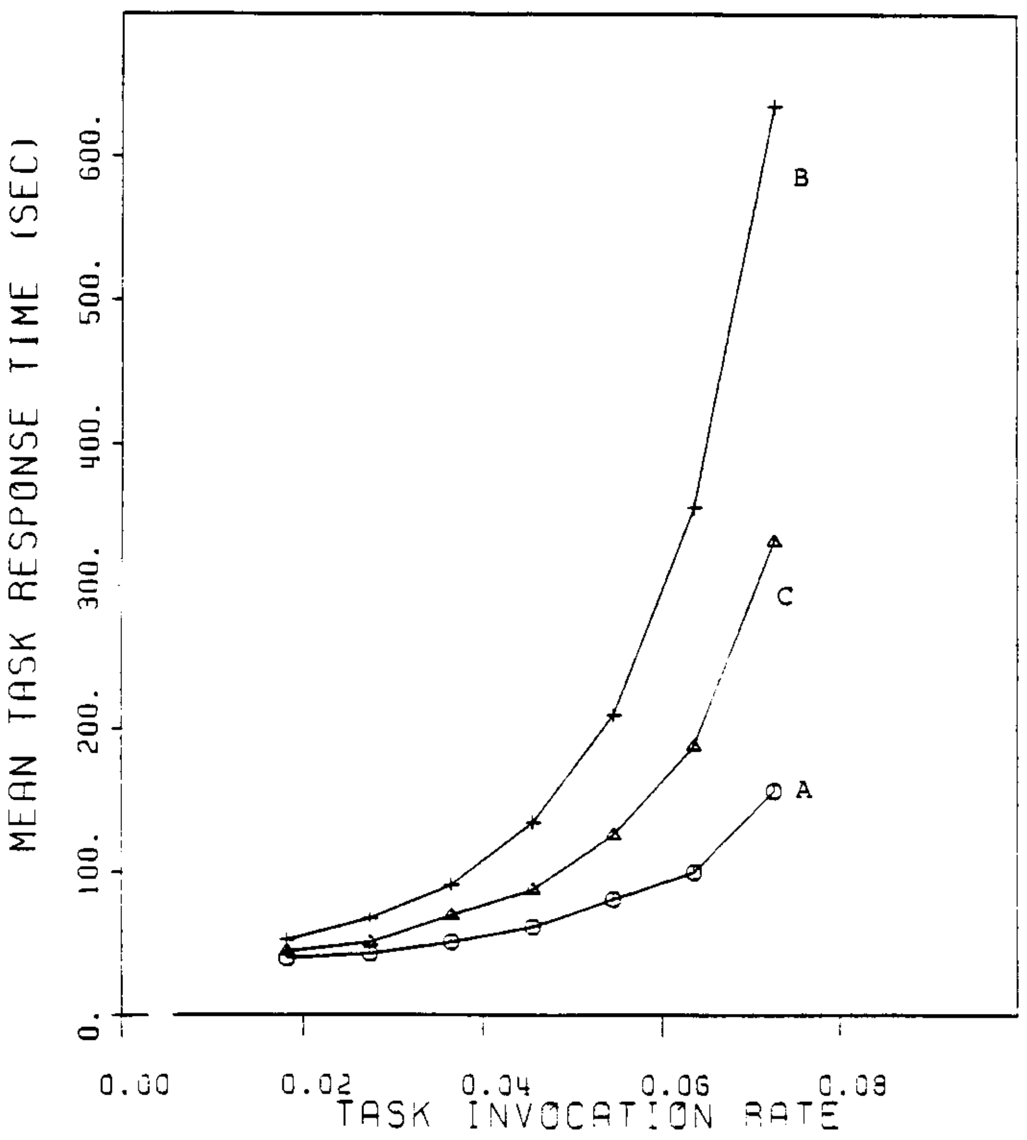


Figure 3.9 Task Response Times for the Assignments

CHAPTER 4

MODULE ASSIGNMENT WITHOUT MODULE REPLICATIONS

4.1 INTRODUCTION

Although distributed processing may provide such advantages over uniprocessor systems as response time improvement and grace degradation in case of failure, there are additional design problems which require careful considerations. One of these design problems is the *module assignment problem* (MAP). The basic objective of module assignment in a RTDPS is to allocate the set of modules of the application task to computers such that the prescribed performance requirements such as response times and system throughput can be satisfied.

A number of module assignment algorithms [STON77, RAO79, PRIC79, MA82, CHOU82, EFE82, SHEN85, CHU85b] have been proposed in the past decade. These methods can be classified into three categories: graph-theoretic, integer programming and heuristic approaches. The processing load on a distributed system is due to both module executions (i.e., for application task) and IPC's for file accesses and control functions. Without careful considerations, a module assignment can cause computer saturation [CHU80] due to excessive IPC. Therefore, a good module assignment should balance the processing workload among processors and generate minimum amount of IPC in the system.

Since task response time[1] is an important performance measure for RTDPS, minimizing the response time becomes the major goal of module assignment. The key factors (parameters) that affect the task response time include IPC, processor loading, module precedence relationships and interconnection network delay. However, current module assignment techniques usually neglect one or more of these factors. These have motivated us to investigate the module assignment with these key parameters and we shall use the task response time model for estimating task response times for various module assignments.

In this chapter, we shall study the MAP without module replication; this is, each module is allocated to a single processor. We shall first present our formulation of the MAP. Next, we discuss the complexity and possible approaches for performing module assignment. Then a new module assignment algorithm will be introduced and finally the application of the algorithm to DPAD System is presented.

4.2 ASSUMPTIONS AND PROBLEM FORMULATION

Let us make the following assumptions for the RTDPS:

(1) There is no memory space constraint on each processor in the system. This assumption becomes acceptable due to the advents of semiconductor technology.

[1] In this Chapter, we use *task response time* to refer to its mean response time unless otherwise stated.

(2) A data file is stored (or replicated) on a processor where a resident module reads and/or updates the file.

(3) The scheduling discipline (e.g., first-come first-serve, head-of-line priorities) for module executions on each processor is given.

(4) All processors in the system are identical. Thus the execution time for each module is the same on all processors.

(5) The network delay is independent of module assignment. Under this assumption, although different module assignments may generate different volume of IPC traffic in the network, we assume the network has sufficient bandwidth and the delay remains unchanged.

Assumptions (4) and (5) can be relaxed by re-computing modules' execution times and the interconnection network delay for each module assignment.

Let us define the following system parameters:

n = total number of processors in the system,

m = total number of modules in the application task,

G = the control-flow graph of the application task;

$\bar{x}(i)$ = average execution time for module i;

$\sigma_x^2(i)$ = variance of execution time for module i;

$X = [\bar{x}(i)]$ = vector of average module execution times where $i \in [1, m]$;

$\sigma^2(x) = [\sigma_x^2(i)]$ = vector of variances of module execution times where $i \in [1, m]$;

D_{net} = average interconnection network delay;

σ^2_{net} = variance of interconnection network delay;

λ = task invocation rate.

Upon the completion of execution, a module may require to communicate with other modules. The processing time for sending message from a module i to another module j is referred to as *IMC time* for the module pair. For some systems, IMC time for a module pair which are co-located on a same processor is assumed to be negligible. However, if the communicating modules are allocated on different processors, IMC becomes IPC which requires processing on both the transmitting and receiving computers. Then the processing time for the IPC on both processors is equal to the IMC time plus the protocol processing overhead. Thus we define

$\bar{t}_c(i,j)$ = average IMC time from module i to j after a module i's execution;

$\sigma_c^2(i,j)$ = variance of IMC time from module i to j after a module i's execution;

$T_c = [\,\bar{t}_c(i,j)] =$ average IMC time matrix where $i,j \in [1, m]$;

$\sigma^2(c) = [\sigma_c^2(i,j)] =$ variance of IMC time matrix where $i,j \in [1, m]$;

Let the module assignment matrix $A = [A_{ij}]$ such that

$$A_{ij} = \begin{cases} 1 & \text{if } \textit{module } j \textit{ resides on processor } i \\ 0 & \textit{otherwise} \end{cases}$$

where $i \in [1, n]$ and $j \in [1, m]$.

Given the module assignment matrix A and all invariant system parameters [1] as defined above, the task response time for the RTDPS can be computed by the task response time model (See Chapter 2). The task response time T_{ptp} can be expressed as a function F of these system parameters. Therefore, the MAP for the RTDPS can be formulated as a zero-one integer optimization problem which is to determine A in order:

To minimize $\quad T_{ptp} = F(G, A, X, \sigma^2(x), T_c, \sigma^2(c), D_{net}, \sigma^2_{net}, \lambda, m, n)$

with constraints $\quad \sum_{i=1}^{n} A_{ij} = 1 \quad$ for *all* $j \in [1,m]$.

Note that the constraints represent each module residing on a single processor (i.e., no module replication). Clearly, this optimization problem involves a non-linear objective function which does not possess an empirical form and can only be computed numerically.

4.3 PROBLEM COMPLEXITY OF MAP

Many sequencing and scheduling problems for multiprocessor systems (See Appendix in [GARA79]) have been proved to be NP-complete. It is believed that these NP-complete problems are intractable in the sense that no polynomial time algorithms can possibly solve them. In fact, the RTDPS under investigation involves repeated task invocation and the task consists of various logical and precedence rela-

[1] Based on the means and variances, the distribution functions for these parameters are approximated by Erlangian or hyper-exponential distributions. Higher order moments of these parameters are computed from the distributions.

tionships among modules. These factors increase the complexity of the MAP as compared with the multiprocessor scheduling problems [GARA79]. Let us now prove that the MAP is indeed a NP-complete problem.

Theorem : The MAP in the RTDPS is NP-complete.

Proof : To prove the MAP is NP-complete, we shall first convert the MAP optimization problem into the corresponding decision problem and then prove the decision problem to be NP-complete. The MAP decision problem is described as follows:

Instance: Given the system parameters of a RTDPS: G, X, $\sigma^2(x)$, T_c, $\sigma^2(c)$, D_{net}, σ^2_{net}, λ, m and n and a real positive value R. Each module is allocated to a single processor. The task response time model is used to compute the task response time.

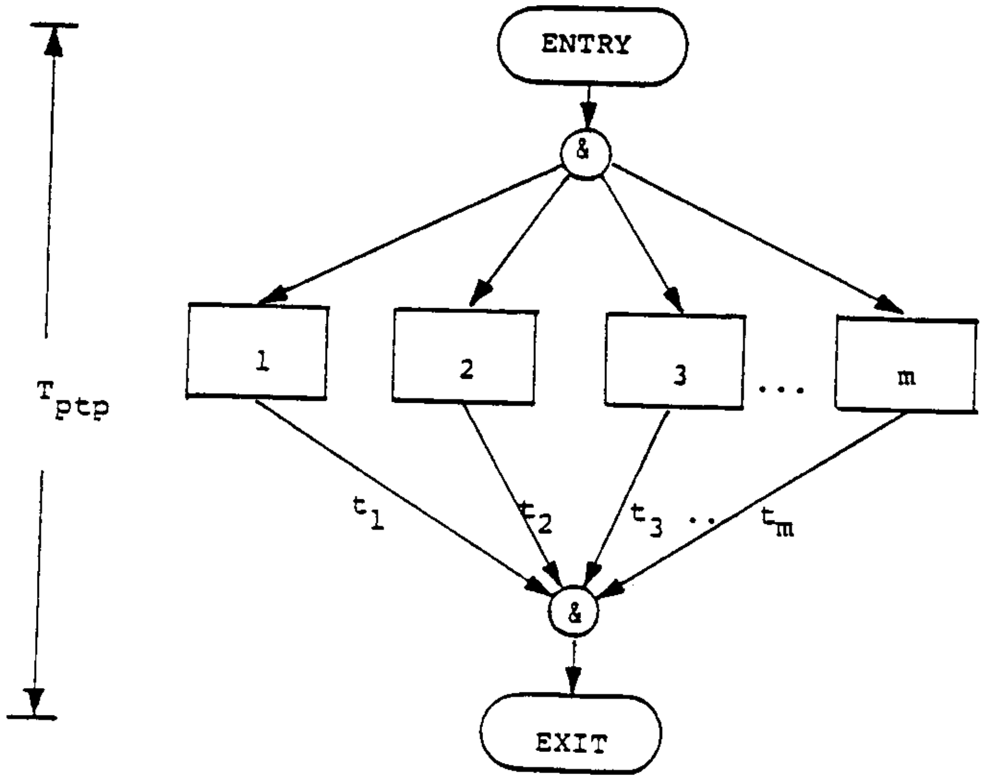
Question: Is there existing a module assignment (represented by matrix A) such that the task response time

$$T_{ptp} = F(G, A, X, \sigma^2(x), T_c, \sigma^2(c), D_{net}, \sigma^2_{net}, \lambda, m, n) \le R \ ?$$

It is apparent that this decision problem is NP since a nondeterministic algorithm need only guess an appropriate module assignment and check in polynomial time if $T_{ptp} \le R$.

Next, we shall transform a well-known NP-complete problem -- multiprocessor scheduling problem (MSP) (pp.65 in [GARA79]) to the MAP of a RTDPS. Consider an instance of MSP: a finite set S of modules, a module execution time ("length") $l(a) \varepsilon Z^+$ for each $a \varepsilon S$, a number $n \varepsilon Z^+$ of processors, and a "deadline" $D \varepsilon Z^+$. Let the RTDPS have n processors and the same module set S. We set T_c, $\sigma^2(c)$, $\sigma^2(x)$, D_{net} and σ^2_{net} to be zero matrices/values in the MAP because the MSP does not consider these factors. Suppose $|S| = m$. We refer to the modules in S by indices 1,2,...m. Thus the corresponding average module execution time vector X can be constructed from $l(a)$ for each $a \varepsilon S$. In addition, let the task invocation rate in the RTDPS $\lambda \rightarrow 0^+$ as the modules are invoked once in the MSP. Since the modules in the MSP do not possess precedence relationship and are independent of each other, we construct a task control-flow graph G (an and-fork to and-join graph) for the MAP as shown in Figure 4.1.

Due to the and-fork function, all modules will be invoked simultaneously (i.e., there will be bulk module invocations) on the processors of the RTDPS. As modules are independent of each other, the modules residing on a same processor can be executed in any order. Further the task response time is only determined by the module assignment regardless of the execution sequence. Therefore processors can execute the bulk module invocations in an arbitrary sequence for the RTDPS. By the definition of and-fork to and-join control-flow graph, the task response time $T_{ptp} = \max \{ t_1, t_2, ..., t_m \}$ where t_i is the response time for module i. Hence T_{ptp} of this RTDPS is equal to the completion time of all module executions in the MSP. Finally, we set $R = D$. It is obvious that the MSP has been reduced to the MAP of a RTDPS and this transformation only requires a polynomial time algorithm. □



t_i : RESPONSE TIME FOR MODULE i

Figure 4.1 The Corresponding Task Control-Flow Graph for the RTDPS

The common methods to tackle these hard combinatorial optimization problems include approximation algorithms, probabilistic algorithms, branch-and-bound and local search techniques [PAPA83]. Due to the complexity and characteristics of the MAP, it is very difficult to pursue in using approximation and probabilistic approaches. Further, since the objective criterion in the MAP is task response time, it is also inefficient to use branch-and-bound approach for the solution. It is because a tight lower bound is difficult to obtain as the task response time can be computed only after all modules have been allocated to computers. Therefore we use local search technique for the solution for the MAP.

4.4 A NEW APPROACH TO MODULE ASSIGNMENT PROBLEM

The major factors that affect the performance of a module assignment in a RTDPS are: (a) processor loading, (b) IPC, (c) logical and precedence relationships among modules, and (d) interconnection network delay. The importance of each of these factors on the task response time depends upon the specific distributed system under consideration. Since our task response time model (Chapter 2) considers all these factors and can be used to compute task response time for a given module assignment, the model becomes the optimization function.

Let us now introduce our new module assignment algorithm which consists of three components:

(a) to generate feasible initial module assignments,

(b) to define the "search neighbor region" of each assignment, and

(c) to search for the best assignment in its neighbor region.

The algorithm repeats searching until it reaches a local optimum. The final suboptimal module assingment is the one that gives the shortest task response time (computed by the task response time model) from all the feasible initial solutions.

4.4.1 Algorithm for MAP

MODULE ASSIGNMENT ALGORITHM (WITHOUT MODULE REPLICATION)

1. Generate a random module assignment A as an initial assignment.

2. Based on the invariant parameters: G, A, X, $\sigma^2(x)$, T_c, $\sigma^2(c)$, D_{net}, $\sigma^2{}_{net}$, λ, m and n, compute the assignment dependent parameters for assignment A (including module execution times, IPC arrival rate & processing time for each processor).

3. Compute the processor utilization on each computer for assignment A. If any computer(s) is saturated (i.e., its utilization $\geq$ 100%), stop; Otherwise, continue.

4. Invoke the task response time model:

   4.1 Compute the task response time $T_{ptp}(A)$ for assignment A, and

   4.2 Identify the computers with the longest and shortest average module waiting times (Denote them as $LWP(A)$ and $SWP(A)$ respectively).

```
5. Let $S_L$ be the set of modules residing on $LWP(A)$. For each module $i \in S_L$, perform
    5.1 Temporarily reallocate module $i$ from $LWP(A)$ to $SWP(A)$ thus becomes assignment $A_i$;
    5.2 Compute the assignment dependent parameters and processor utilization factors for assignment $A_i$ (* As Steps 2 and 3 do *);
    5.3 If computer(s) is saturated, set the task response time $T_{ptp}(A_i) = \infty$; otherwise, invoke task response time model to compute and record $T_{ptp}(A_i)$, $LWP(A_i)$ and $SWP(A_i)$ (* As Step 4 does *).
6. If there exists $T_{ptp}(A_i) \leq T_{ptp}(A)$ for any $i \in S_L$ tested in Step 5, then perform:
    6.1 Set $A = A_i$, $T_{ptp}(A) = T_{ptp}(A_i)$, $LWP(A) = LWP(A_i)$ and $SWP(A) = SWP(A_i)$ where $T_{ptp}(A_i) = \min_{j \in S_L} \{ T_{ptp}(A_j) \}$. (* Finalize the single module reallocation from LWP to SWP -- A greedy step! *)
    6.2 Go to Step 5.
7. Otherwise, stop. (* Reach a local optimal assignment A *)
```

For a given module assignment, the algorithm will identify the processors with the longest and shortest average module waiting (queueing) times [1]. The basic operation of this algorithm (Step 5) is to reallocate some module(s) from the longest average waiting time processor (LWP) to the shortest average waiting time processor (SWP) and to hope for better load balancing among computers thus to yield a shorter task response time. However, such module reallocations may not necessarily provide a shorter response time because they may increase IPC on the system. Therefore, the task response time is recomputed for each module reassignment (Step 5.3). If the response time is improved after the reallocations of modules, the algorithm finalizes the one that yields the shortest task response time -- a greedy step (Step 6). The algorithm continues to reassign module(s) from LWP to SWP until it cannot further reduce the response time for the distributed system. In order to generate a better final suboptimal module assignment, the algorithm is repeated with a number of randomly selected initial assignments. The final assignment is the one that achieves the shortest task response time among all the local optimal assignments.

By reducing the response time based on module reassignments, the algorithm implicitly attempts to equally balance the computational load and decrease overall IPC in the system. Other factors such as maximizing the parallelism among module executions and the effects due to precedence relationships and interconnection network delay have also been considered in the process of searching for a shorter response time assignment.

[1] In case waiting times for distinct modules are different for the given scheduling policy on a processor, the average module waiting time is taken to be the sum of modules' individual waiting times weighted by their invocation rates normalized to the total module invocation rate on the processor.

4.4.2 The Required Number of Random Initial Assignments

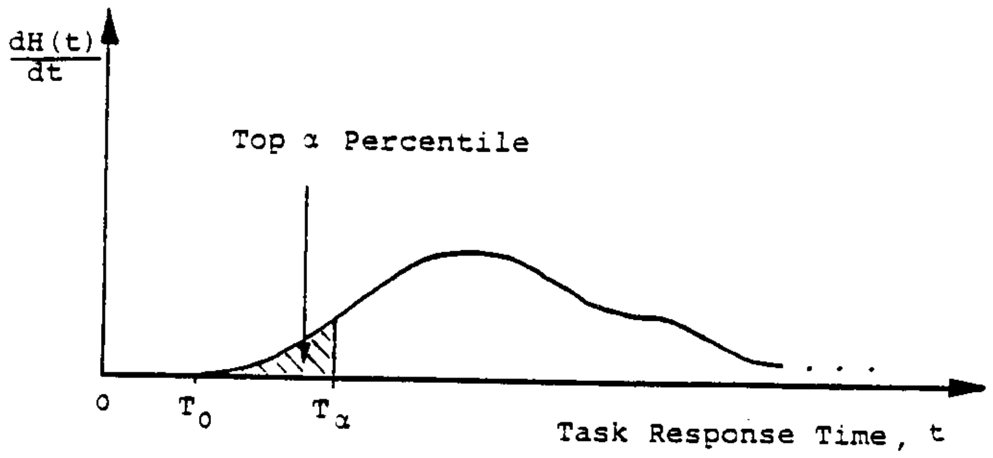
In order for the local search to generate a good module assignment, we need to repeat the algorithm with randomly selected initial assignments. Such technique was also used by [LIN65] for traveling salesman problem. The required number of initial assignments depends on such system characteristics as loading, module execution times and IMC of the RTDPS, and the expected performance of the final solution. In the following, we discuss the desired performance of "good" module assignments. Next, we illustrate how to estimate the required number of initial assignments for generating a suboptimal module assignment with the specified performance.

To find the best (global optimal) assignment that yields the minimum task response time for a RTDPS is very time-consuming even if it is not impossible. Let the probability distribution function of task response times T_{ptp}'s for all feasible module assignments (i.e., assignments with finite task response times or do not cause processor saturation) be $H(t)$. That is,

$$H(t) = Pr[\, T_{ptp} \leq t \,] \tag{4.1}$$

It is desirable to generate a module assignment which is ranked as top α percentile (e.g., 0.1%) of the distribution $H(t)$.

Let the task response time of a module assignment of the top α percentile be less that T_{α} (See Figure 4.2). This means



T_0: Optimal Task Response Time

T_α: Maximum Task Response Time for a Top α Percentile Assignment

Figure 4.2 A Sample Distribution of Task Response Times for Module Assignments

$$H(T_\alpha) = Pr\ [\ T_{ptp} \le T_\alpha\ ] = \alpha \tag{4.2}$$

To start with a feasible initial assignment, the algorithm searches for better assignments. During the search process, the algorithm will test out a large number of module assignments with different task response times. Suppose the algorithm records the task response time of a module assignment which is shorter than the best obtained so far during the algorithm run. Let the sequence of these response times recorded (a descending sequence) consist of l elements which are designated as $R_1, R_2, \ldots, R_l$. Clearly, R_1 and R_l are the task response times for the initial assignment and the local optimum respectively. Note that the length of this sequence depends on the characteristics of the algorithm and the RTDPS. Let P_α be the probability that the task response time of the local optimal assignment is less than T_α (i.e., within top α percentile of the distribution). Thus we have

$$\begin{aligned} P_\alpha &= Pr[\ \min_{1 \le i \le l} \{R_i\} \le T_\alpha\ ] \\ &= 1 - Pr[\ R_1 > T_\alpha, R_2 > T_\alpha, \ldots, R_l > T_\alpha\ ] \end{aligned} \tag{4.3}$$

Since $R_i > R_j$ if $j > i$, the R_i's are dependent variables. However, given the events that $(R_1 > T_\alpha)$, $(R_2 > T_\alpha)$, ..., and $(R_i > T_\alpha)$, they do not have a significant influence on the occurrence of event $(R_{i+1} > T_\alpha)$. And, the probability $Pr[\ R_{i+1} > T_\alpha\ ]$ is close to unity when α is small (i.e., T_α is close to T_o). Therefore, we can approximate the probability in Eq.(4.3) as

$$Pr[\,R_1 > T_\alpha\,, R_2 > T_\alpha\,, ..., R_l > T_\alpha\,] = \prod_{i=1}^{l} Pr[\,R_i > T_\alpha\,] \tag{4.4}$$

Further, due to the search effort of the algorithm, R_i's form a descending sequence (i.e., $R_1 \geq R_i$), we have

$$Pr[\,R_i > T_\alpha\,] \leq Pr[\,R_1 > T_\alpha\,] \tag{4.5}$$

for all $i = 2$ to l. We substitute these inequalities into Eq.(4.4). It yields

$$Pr[\,R_1 > T_\alpha, R_2 > T_\alpha\,, ..., R_l > T_\alpha\,] \;\leq\; \{\,Pr[\,R_1 > T_\alpha\,]\,\}^l \tag{4.6}$$

Since the initial assignment is randomly selected, by definition we put $Pr[\,R_1 > T_\alpha\,] = 1 - H(T_\alpha) = 1 - \alpha$ into Eq.(4.6) and then substitute (4.6) in (4.3). We obtain

$$P_\alpha \geq 1 - (\,1 - \alpha\,)^l \tag{4.7}$$

Now, let us estimate the required number N_f of feasible initial assignments to assure that the suboptimal solution is within top α percentile with certain confidence P_α. For the j^{th} feasible initial assignment, suppose the sequence $\{\,R_i\,\}$ of the "improving" task response times recorded by the algorithm consists of l_j elements. By the same arguments for the derivation of Eq.(4.7), the probability $P_\alpha{}'$ that the suboptimal solution from these N_f feasible initial assignments falls within top α percentile of $H(t)$ is

$$P_\alpha{}' \geq 1 - (\,1 - \alpha\,)^{l_1 + l_2 + \cdots + l_{N_f}} \tag{4.8}$$

To find the number N_I of required initial assignments (some of which may be infeasible), we conduct a pilot run of the algorithm with a small number (e.g., 20) of random initial assignments. From this pilot run, the proportion r of initial assignments that are feasible can be determined. In addition, the average sequence length $\bar{l}$ of the "improving" task response times can be estimated from l_j's. Then, we have $N_f = r\, N_I$ and $l_1 + l_2 + \cdots + l_{N_f} = N_f\, \bar{l}$. Substituting these two relations in Eq.(4.8), it becomes

$$P_\alpha' \geq 1 - (1 - \alpha)^{r\, N_I\, \bar{l}} \tag{4.9}$$

Once the desired performance (in terms of α percentile) of the suboptimal solution and the confidence probability P_α' to obtain such an assignment are given, the required number of initial assignments input to the algorithm, N_I, can be solved from Eq.(4.9).

4.4.3 Weighted Task Response Time

In Chapter 2, task response time is defined as the time from a task invocation until the completion of the task execution. Very often, the application task consists of several processing threads. A task invocation may only branch to invoke a particular thread. This branching to process a certain thread is represented by an or-fork in the task control-flow graph and the task response time is the sum of thread response times (PTP times of the threads) weighted by the threads' invocation rates. However, the importance of various threads in terms of responsiveness requirements may not be necessarily proportional to their invocation rates. For example, processing a thread for missile discrimination is far more important thus requiring rapid response than a thread of detecting new incoming object in a space defense application, although the latter thread is more frequently invoked than the former one. Therefore, the task response time can be redefined as the sum of threads' response times weighted by some constants properly chosen to reflect the importances of these threads for the particular application under consideration. With this new definition of task response time, the algorithm will search for module assignments to meet the expected response times for various threads. In other words, system designers may have more flexibility in defining the task response time so as to search for good module assignments for their specific applications.

4.4.4 Application of the Algorithm to DPAD System

We apply the module assignment algorithm to the DPAD System [GREE80, LAN85] which was used for model validation in Chapter 2. The task control-flow graph of this system is shown in Figure 2.13. Since the system input load (module invocation rates and execution times) for the DPAD System is a time-variating function, we collect the system parameters during the peak-load period (from 1.0 to 2.0 seconds) as input to the algorithm. These parameters include module invocation rates, module execution times [1], and IMC's [1] as shown in Tables 4.1 to 4.3. Due to the complexity of the application task in DPAD System, its task response time is redefined as

[1] Only the mean and second moments of these parameters are measured from the DPAD simulator. Based on these measurements, the distribution functions for these parameters are approximated by Erlangian distributions. Other moments of higher orders are computed from these functions.

$$T_{ptp} = \sum_{i=1}^{4} \frac{\lambda_i}{\lambda_{tot}} T_i + \frac{\lambda_5}{\lambda_{tot}} T_{\text{and}}(5,6) + \sum_{i=7}^{15} \frac{\lambda_i}{\lambda_{tot}} T_i + \frac{\lambda_{16}}{\lambda_{tot}} T_{\text{and}}(16,17)$$

$$+ \sum_{i=18}^{23} \frac{\lambda_i}{\lambda_{tot}} T_i + \sum_{i=1}^{23} \frac{\lambda_i}{\lambda_{tot}} \sum_{j=1}^{23} p_{ij}\, \delta_{ij}\, D_{net}$$

where:

$\lambda_i =$ the invocation rate for module i,

$\lambda_{tot} =$ sum of λ_i's for all $i=1$ to 23,

$T_i =$ response time for module i,

$T_{\text{and}}(i,j) =$ response time for the and-fork to and-join subgraph formed by modules i and j,

$p_{ij} =$ probability that module i enables module j upon the completion of execution,

$$\delta_{ij} = \begin{cases} 0 & \text{if } \textit{modules } i \text{ and } j \textit{ resides on a same processor} \\ 1 & \textit{otherwise} \end{cases}$$

$D_{net} =$ average interconnection network delay.

In this definition, task response time is the sum of modules' response times weighted by their normalized invocation rates. To consider two and-fork to and-join subgraphs (formed by modules 5 and 6, and 16 and 17 respectively) in the task control-flow graph, the response times for the subgraphs, instead of individual module response times, are used in the task response time definition. The last summation term in the definition represents the interconnection network delays incur by the module enablement messages if the preceding and succeeding modules are residing on different computers.

Modules	Invocation Rates	Modules	Invocation Rates
1	0.432	13	0.509
2	0.063	14	1.0
3	0.032	15	0.0
4	0.169	16	0.042
5	0.035	17	0.042
6	0.035	18	0.042
7	0.036	19	0.018
8	0.202	20	0.002
9	0.203	21	0.002
10	0.042	22	1.399
11	0.0	23	1.0
12	0.0		

Table 4.1 Module Invocation Rates (no. of enablements/msec) for the DPAD System

Modules	Execution Times (ms)	Squared Coefficients of Execution Time
1	0.133	0.0
2	0.273	0.03586
3	0.335	0.00021
4	0.380	0.15240
5	0.333	0.00002
6	0.327	0.02875
7	0.312	0.05535
8	1.033	0.0
9	0.600	0.0
11	0.0	0.0
12	0.0	0.0
13	0.327	0.00071
14	0.112	0.00040
15	0.0	0.0
16	0.667	0.0
17	1.0	0.0
18	0.628	0.00038
19	3.333	0.0
20	6.672	0.00001
21	0.6	0.0
22	0.078	0.00034
23	0.113	0.00041

Table 4.2 Module Execution Times During Peak-Load Period for DPAD System

Sending Modules	Receiving Modules	Avg. IMC* Times(ms)	Variances of IMC Times	File No.**
2	3	0.01300	0.00017	114
3	13	0.03000	0.00000	115
4	5	0.01239	0.00054	116
4	6	0.01239	0.00054	117
4	7	0.01239	0.00054	117
4	13	0.00443	0.00007	117
5	7	0.01200	0.00000	119
6	7	0.01200	0.00000	121
7	13	0.01257	0.00017	122
8	6	0.25766	0.00023	124
8	9	0.05166	0.00023	123
8	10	0.25766	0.00023	124
8	13	0.00204	0.00004	120
8	16	0.25766	0.00023	124
8	17	0.05166	0.00023	123
8	19	0.05166	0.00023	123
8	20	0.05166	0.00023	123
9	13	0.02600	0.00000	125
13	14	0.39791	0.22615	131
14	13	0.01200	0.00000	147
14	23	0.03346	0.00009	132
16	18	0.01600	0.00000	135
17	18	0.01600	0.00000	136
18	18	0.00569	0.00004	138
18	19	0.03619	0.00072	137
18	20	0.03856	0.00646	139
19	20	0.20600	0.00000	139
20	21	0.20600	0.00000	140
21	Radar	0.11200	0.00000	141
22	1	0.02188	0.00105	113
22	2	0.02188	0.00105	113
22	4	0.02188	0.00105	113
22	8	0.02188	0.00105	113
22	23	0.00115	0.00006	142
23	Radar	0.03400	0.00000	112
Radar	22	0.07020	0.00000	111

*IMC's between all other module pairs not listed here are zero.
** If a sending module communicates with several receiving modules via a same file and those receiving modules are co-locating on a remote processor, the update message (IPC) is sent to the processor once.

Table 4.3 IMC's for the DPAD System

The algorithm is repeated for two cases: with 100 and 500 random initial assignments. Their respective algorithm times are about 20 and 93 minutes on a VAX 11/780 machine. Our study shows that about 60% of these initial assignments are feasible solutions (i.e., they do not cause computer saturation thus do not stop at Step 3). The algorithm produces two suboptimal module assignments, A and B, for these two cases as shown in Table 4.4. To evaluate their response time performance, we use the DPAD simulator to simulate these two module assignments. Figures 4.3 and 4.4 depict the comparison of response times for two important processing threads for the assignments. We note that the assignment B performs better than assignment A during the peak-load period. It is because assignment B is the suboptimal solution generated from 500 initial assignments, while assignment A is based on 100 initial assignments. However, due to the fluctuation of input loading, the performance of assignment B may not always be superior than that of A outside the peak-load period. This can be observed from 0 to 1.0 second period for the Search/Verify thread in Figure 4.3. Therefore, system designers may have the flexibility to select system parameters of a proper time period to input to the algorithm according to the requirements of the system application. For the run with 500 initial assignments, the algorithm tested out 12,285 feasible assignments and 195 infeasible assignments in total. The difference between these two figures reveal that the algorithm is very efficient in searching for better assignments (most likely, feasible assignments) as module(s) are reallocated from LWP to SWP. During the search for assignment B, our algorithm records that $N_I = 500$, $r = 0.6$ and $\bar{I} = 10$ for Eq.(4.9). From this equation, these figures infer that,

Processors / Modules / Suboptimal Assignments	CPU 1	CPU 2	CPU 3
A (FROM 100 INITIAL ASSIGNMENTS)	M3, M4, M6, M7, M8, M10, M16, M17	M13, M14, M21, M22, M23	M1, M2, M5, M9, M18, M19, M20
B (FROM 500 INITIAL ASSIGNMENTS)	M6, M8, M10, M15, M19, M20	M9, M13, M14, M21, M23	M1,M2,M3,M4,M5, M7, M17, M18, M22

Table 4.4 Suboptimal Module Assignments for Cases with 100 and 500 Random Initial Assignments

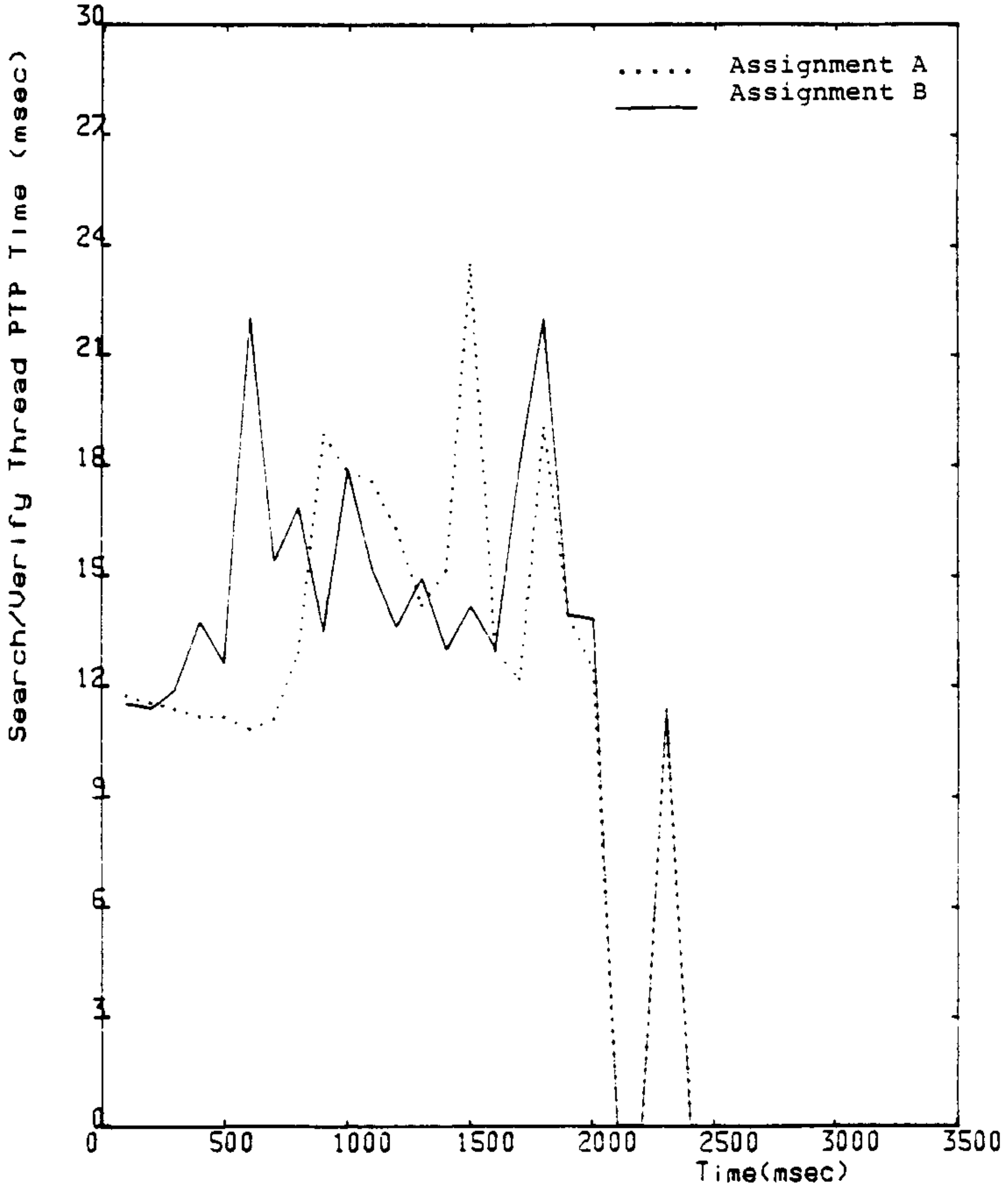


Figure 4.3 Comparison of Search/Verify Thread PTP Time for Assignments A and B

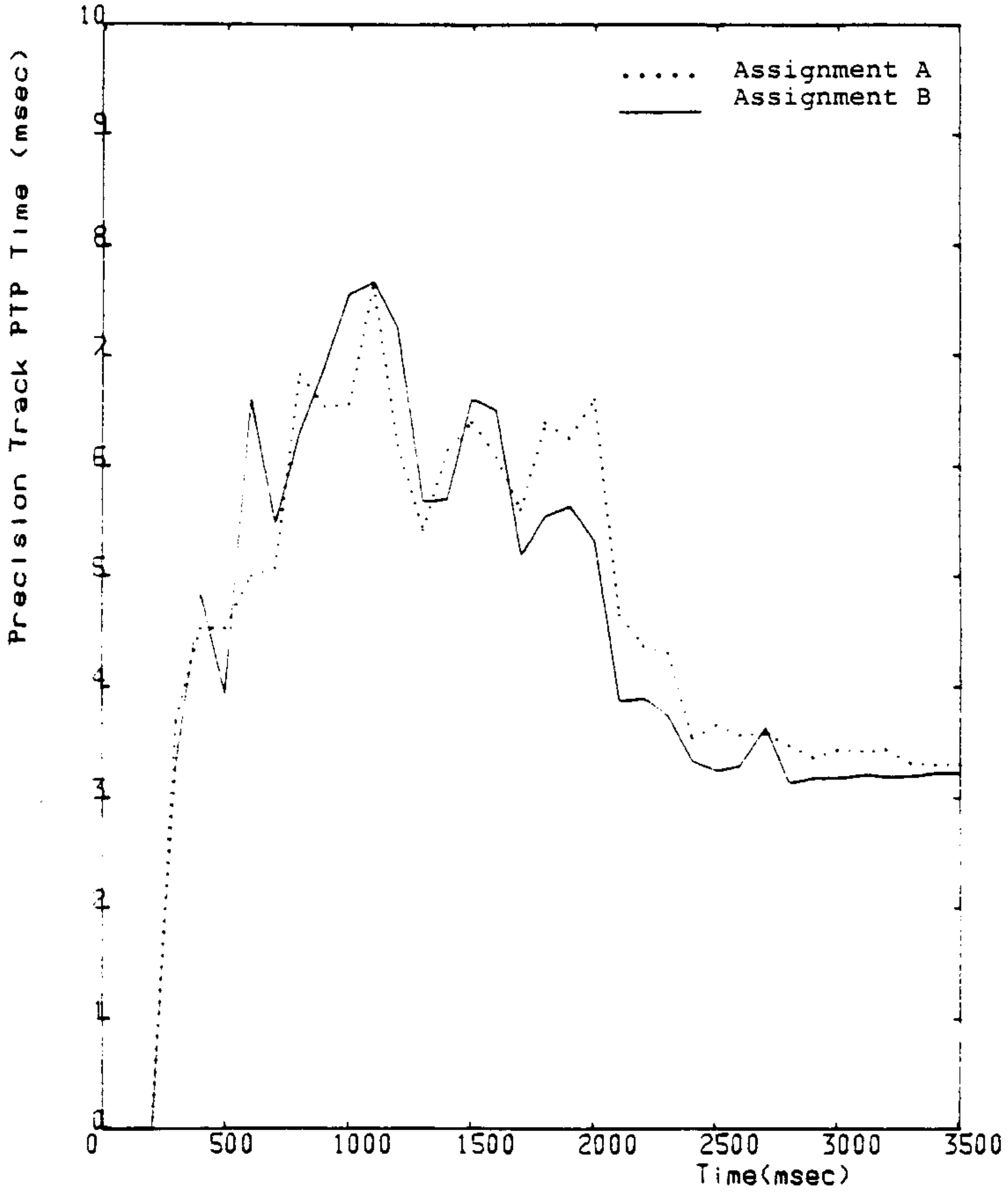


Figure 4.4 Comparison of Precision Thread PTP Time for Assignments A and B

with probability more that 99%, the suboptimal assignment B is one of top 0.15% assignments.

Recently, another module assignment algorithm was proposed by [LAN85] which is based on co-locating heavily communicating modules on a same processor and minimizing the utilization of most heavily loaded processor (referred to as *bottleneck).* To find the minimum bottleneck assignment, it requires an exhaustive search from all possible solutions which takes a prohibitively large amount of CPU time. To reduce the CPU time, a heuristic search algorithm was also developed in [LAN85]. Figures 4.5 and 4.6 show the response time performance for assignment B from our algorithm, the minimum bottleneck assignment based on exhaustive search and that based on heuristic search reported in [LAN85]. We note the results are comparable which also serves as another validation of the analytic model as well as the algorithm. The major advantages of our new algorithm over that proposed by [LAN85] are:

(1) Less computation time because a significant amount of CPU time is required to repeat the algorithm in [LAN85] for selecting the threshold values: α, β and γ to yield good solutions.

(2) The new algorithm can be extended to handle module assignment with module replications which have not been considered in the existing module assignment algorithms. We shall discuss this extension in the following chapter.

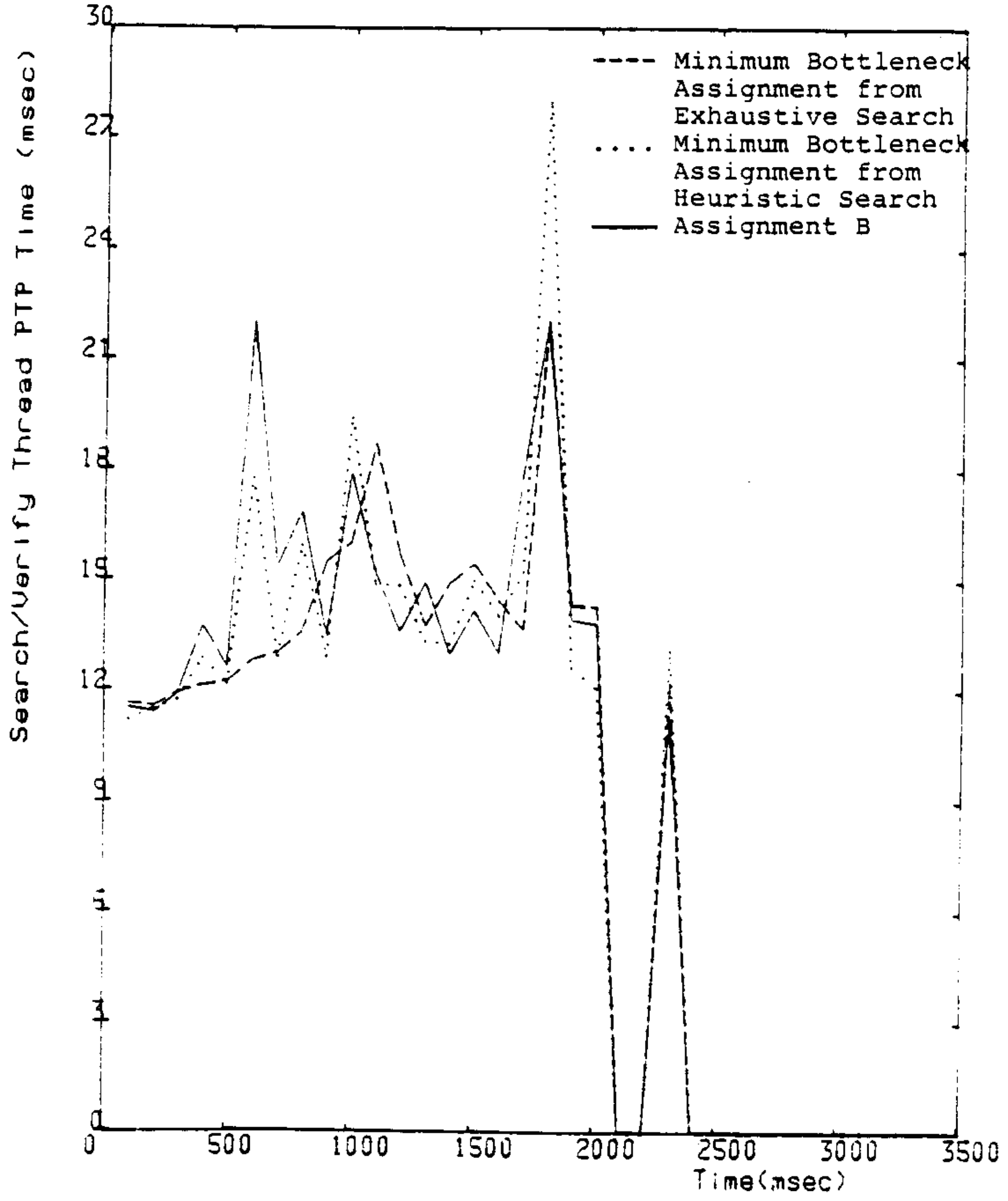


Figure 4.5 Comparison of Search/Verify Thread PTP Time for the Minimum Bottleneck Assignments and Assignment B

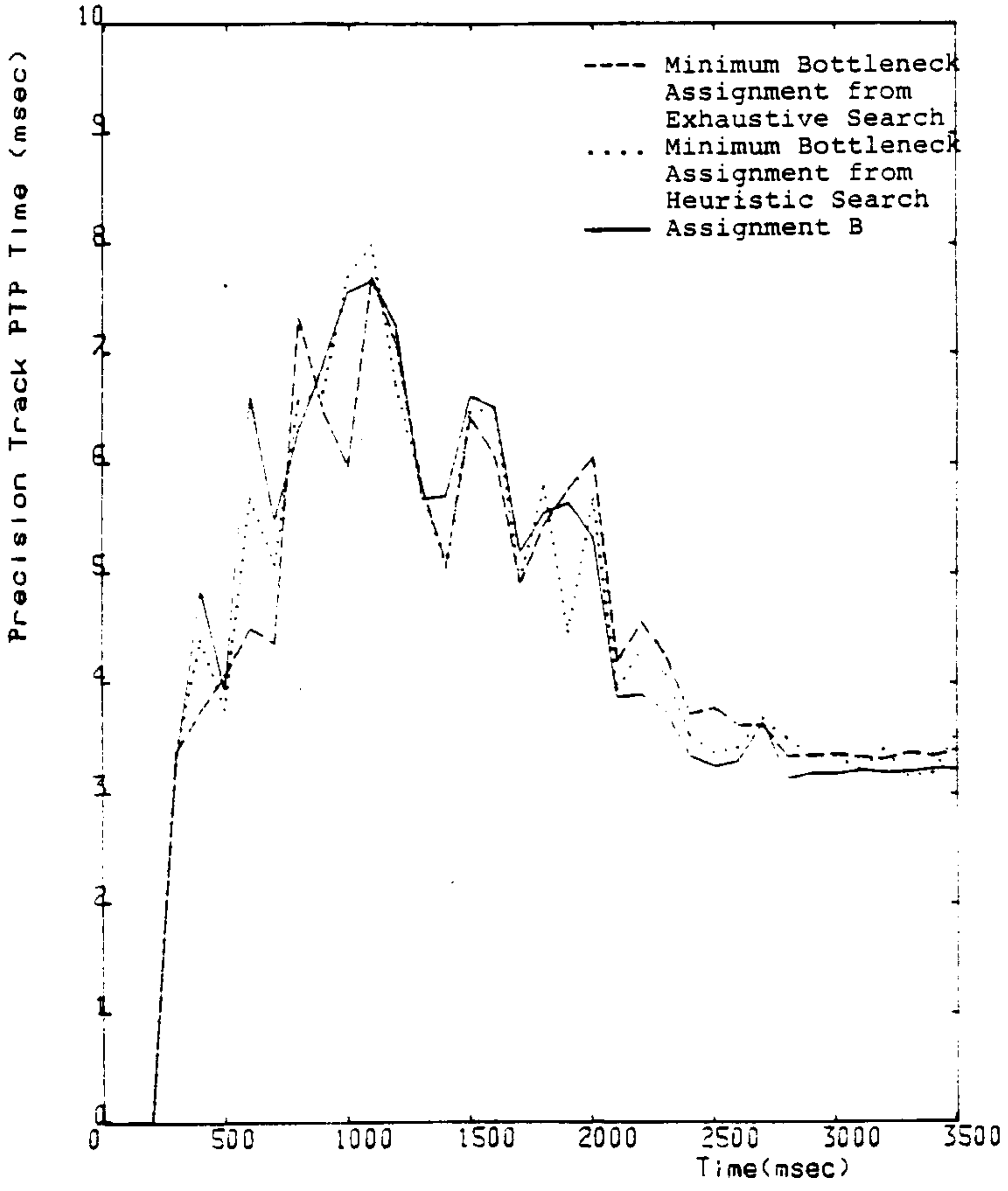


Figure 4.6 Comparison of Precision Thread PTP Time for the Minimum Bottleneck Assignments and Assignment B

4.5 SUMMARY

In this chapter, the problem of module assignment without module replication for RTDPS has been formulated as a zero-one integer optimization problem. The objective function of this problem does not posses an empirical form and can only be computed numerically by the task response time model. A new algorithm for the MAP which is based on the response time model has been introduced. Application of this algorithm to the DPAD System has demonstrated the efficiency of the algorithm, and further validated the analytic model. Since module replications may be required for some real-time applications, we shall discuss the extension of this MAP algorithm for handling module replications in the following chapter.

CHAPTER 5

MODULE ASSIGNMENT WITH MODULE REPLICATIONS

5.1 INTRODUCTION

Another alternative to share the processing workload among processors in a distributed system is to *replicate* some modules on several processors instead of allocating each module to a single processor. As a design alternative for distributed systems, this technique is referred to as *module replication*. Invocations for these replicated modules are routed to and executed on their resident processors according to the loading condition of the processors. Module replications may improve the system performance in the following aspects. Firstly, the processing load on a computer depends on the invocation rates and execution times of its residing modules. As invocations for a replicated module are executed on a number of computers, processor load due to executions of replicated modules may be evenly balanced among computers by proper module replication and allocation. Secondly, with better balanced load on processors, system response times can be improved to meet their strict requirements. Thirdly, if computer malfunction occurs, invocations for the replicated modules still can be executed on other available computers. Thus, module replication improve the system reliability and availability.

To support module replications, a distributed system should have a special algorithm to route the invocations for a replicated module to its resident computers. When a replicated module is invoked, if a copy of the module is located on a local computer, then the module invocation will be executed locally. Otherwise, the enablements for the module have to be routed to some remote computers for execution. The routing algorithm ought to be properly designed so that it will not become the bottleneck for the system. A simple strategy is to route the invocations to their residing computers in a round-robin manner.

The replication and assignment of modules to computers in a distributed system is referred to as the *replicated module assignment problem* (RMAP). Actually, a RMAP consists of two sub-problems:

(1) To determine the optimal module multiplicities (i.e., number of copies for each module), and

(2) To allocate these module copies to computers

such that the specified system performance requirements can be satisfied. Since both module multiplicities and the assignment of module copies to computers will affect the resultant system performance, these two issues cannot be considered separately. For simplicity, we shall still refer the replication and assignment of modules to computers in the replicated processing environments as *module assignment* (or *module allocation)* in the following discussion.

In this chapter, we shall first discuss the objectives for the RMAP and a new objective function is constructed for efficient search of good module allocation. Next, we extend the module assignment (without module replications) algorithm introduced in Chapter 4 to handle module replications. Application of this algorithm for RMAP to the Sentry System will be presented and finally the performance of the algorithm will be discussed.

5.2 THE OBJECTIVE FUNCTION FOR RMAP

The primary goal of module assignment in distributed systems is to satisfy the prescribed response time requirements with the considerations of IPC, module precedence relationships, load balancing and interconnection network delay. One way to achieve this design goal is to minimize the mean task response time. In the previous chapter, we have demonstrated the effectiveness of this objective in searching for good assignments without module replication. Module replications provide us with more flexibility in system design. The key question is to determine which modules require replications, and how many copies of them are needed.

An application task is often decomposed into several threads of modules. To respond to each external stimulus (task invocation), often only a single thread (or a portion) of the task is invoked. Besides the response time requirement for the complete task, users always specify the response time constraints for various threads of the task. Therefore, for the case of module replications, the following are the objectives for RMAP:

(1) the average response time of the application task is minimized, and

(2) response time constraints for various threads are satisfied.

Depending on the definition of task response time for a RTDPS, minimizing task response time does not assure that the threads' response time constraints are satisfied. For real-time applications, to satisfy the response time constraints is usually more important than to merely minimize their task response times. Thus, thread response time specifications should be satisfied before minimizing the task response time. In other words, the goal of RMAP for a RTDPS is to minimize the task response time with the conditions that thread response time specifications are met.

5.2.1 A New Objective Function

An algorithm for an optimization problem can perform more efficiently in search of solutions if there exists an objective criterion (function) for the problem. Now, the RMAP for RTDPS has two objectives. Therefore, it is desirable to combine them into a single objective function.

One approach to combining these two objectives is to define a new objective function for the RMAP as

$$T_{obj} = \begin{cases} T_{ptp} & \text{if } \textit{all thread response time constraints are met} \\ T_{ptp} + aT_{pd} & \textit{otherwise} \end{cases} \tag{5.1}$$

where:

T_{ptp} = mean task response time (as defined in Section 4.2),

T_{pd} = A positive-valued penalty delay function (to be defined later),

a = a positive scaling constant to account for the impacts of violating thread response time constraints.

This new objective criterion is the sum of mean task response time T_{ptp} and the possible *penalty delay* aT_{pd}. Both T_{ptp} and T_{pd} depend upon the module assignment in consideration. For a given module assignment, the penalty delay may be added to the objective criterion to "penalize" the violations of thread response time requirements for the assignment. Clearly, if the penalty delay aT_{pd} is chosen to be sufficiently large when compared with T_{ptp}, the objective criterion T_{obj} will be largely increased when some threads violate their response time specifications. Since any algorithm for the RMAP will definitely search for a module assignment with the minimum value of T_{obj}, the algorithm will implicitly avoid those solutions which yield unsatisfactory thread response times.

5.2.2 Properties of the Penalty Delay Function

Before we define the penalty delay function, let us define some variables and discuss the properties of the function. Suppose the system performance specifications consist of response time constraints for k distinct threads. Let each thread be designated by indices from 1 to k. We define

$t_i(A)$ = the average response time for thread i for module assignment A,

$t_i(R)$ = the average response time requirement for thread i,

$$d_i(A) = \begin{cases} t_i(A) - t_i(R) & \textit{thread } i \textit{ violates response time requirement} \\ 0 & \textit{otherwise} \end{cases}$$

= response time discrepancy for thread i for module assignment A,

$D(A) = [d_i(A)] =$ thread response time discrepancies vector for module assignment A.

For a given module assignment A, if the response time constraint for thread i is violated; that is, $t_i(A) > t_i(R)$, then $d_i(A)$ represents the discrepancy between the response time and its requirement. In addition, let thread i consist of a set of modules S_i. For each $i \in [1,k]$, we define the ***required mean module waiting time*** for thread i as

$$\bar{w}_i(R) = \frac{t_i(R) - \sum_{j \in S_i} \bar{x}_j}{|S_i|} \tag{5.2}$$

where $\bar{x}_j =$ mean execution time for module j. Let

$W(R) = [\bar{w}_i(R)] =$ required mean module waiting times vector.

Given the response time requirement for thread i, the numerator in Eq.(5.2) is the maximum allowable sum of waiting times for all modules of thread i. Thus, $\bar{w}_i(R)$ represents the average module waiting time for each module in the thread. Obviously, the smaller value of $\bar{w}_i(R)$ means that the faster response does thread i require from the processor.

To provide an efficient search for good module assignments, we define the penalty delay function, T_{pd}, as a function of $D(A)$ and $W(R)$. Let us discuss the properties of $T_{pd}(D(A), W(R))$ in the following.

Property 1:

$T_{pd}(D(A), W(R))$ is an increasing function of $d_i(A)$ for all $i \in [1,k]$.

The penalty delay increases as the amount of a thread response time exceeding its requirement increases.

Property 2:

For all $i,j \in [1,k]$ and $i \neq j$, if $d_i(A) = d_j(A) > 0$ and $\bar{w}_i(R) > \bar{w}_j(R)$, we have

$$\left. \frac{\partial T_{pd}(D(A), W(R))}{\partial d_i(A)} \right|_{D(A)} < \left. \frac{\partial T_{pd}(D(A), W(R))}{\partial d_j(A)} \right|_{D(A)}$$

Provided that the threads' response time discrepancies are the same, the thread with the stricter response time requirement (i.e., with a smaller $\bar{w}_i(R)$) yields a higher penalty delay.

Based on these properties of the penalty delay function, the objective function, T_{obj}, in Eq.(5.1) can be used to efficiently search for better module assignments which reduce the thread response time discrepancies and attempt to firstly satisfy the threads which have stricter response time constraints. Therefore, we define the penalty delay function according to these properties as follows:

$$T_{pd}(D(A),W(R)) = \sum_{i=1}^{k} l_i \; d_i(A) \tag{5.3}$$

where $l_i = \dfrac{\max\{\, w_1(R),\, w_2(R), \ldots,\, w_k(R)\,\}}{w_i(R)}$ for all $i \; \varepsilon \; [1,k]$.

Apparently, this function satisfies Property 1. If $\overline{w}_i(R) > \overline{w}_j(R)$, then $l_i < l_j$. Thus, the function also satisfies Property 2. Further, by the definition of $d_i(A)$'s, when all thread response time constraints are satisfied for the assignment A, $T_{pd}(D(A),W(R))$ in Eq.(5.3) equals zero. Substituting Eq.(5.3) into (5.1), we obtain the objective function for the RMAP as

$$T_{obj}(A) = T_{ptp}(A) + a \sum_{i=1}^{k} l_i \; d_i(A) \tag{5.4}$$

5.3 A NEW SEARCH ALGORITHM FOR RMAP

Given a module replication and assignment for a distributed system, each computer will execute a fixed set of modules. Based on the task invocation rate and the routing algorithm for replicated module invocations, module invocation rates on each computer can be obtained. Therefore, with other system parameters (as defined in Section 4.2), the response times for the complete task and various threads can be computed by the task response time model. From the thread response times and their requirements, the penalty delay is computed thus the value of the objective function for the module assignment.

5.3.1 Problem Formulation

Using the same assumptions and definitions as in Section 4.2, the RMAP for the distributed system is to determine the module assignment: *To minimize*

$$T_{obj}(A) = T_{ptp}(A) + a\ T_d(D(A), W(R))$$

$$= F(G, A, X, \sigma^2(x), T_c, \sigma^2(c), D_{net}, \sigma^2_{net}, \lambda, m, n) + a \sum_{i=1}^{k} l_i\ d_i(A) \qquad (5.5)$$

with constraints $\quad 1 \le \sum_{i=1}^{n} A_{ij} \le n \quad$ for *all* $j \in [1, m]$.

The constraint inequalities indicate that each module may be allocated to at least one processor or replicated on as many as all processors in the system. Module assignment with replications is much more complicated than that without module replication as module multiplicities are also key parameters that need to be determined. In the following, we shall extend the MAP Algorithm to handle module replications.

5.3.2 Algorithm for RMAP

The RMAP algorithm consists of three major components:

(1) Module Reallocations from LWP to SWP

For a given module assignment, modules may be reallocated from the LWP to the SWP without changing module multiplicities until a local optimal assignment is reached. This portion is identical to the MAP Algorithm (Chapter 4) except the objective function T_{ptp} is replaced by T_{obj}. Actually, the MAP Algorithm becomes a subroutine in this RMAP Algorithm.

(2) Further Module Replications on SWP

After module reallocations from the LWP to the SWP has reached a local optimum, the algorithm attempts to balance the processing workload by further replicating certain modules on the SWP. In case some thread response time constraints are violated, the modules for further replications on the SWP are the modules of those threads which violate their response time requirements. If all thread response time constraints are satisfied, those modules currently residing on the LWP becomes the candidates for further replications on the SWP. If T_{obj} is improved by these replications, the single module replication on the SWP that yields the minimum T_{obj} is finalized. However, T_{obj} may not always be improved with these replications of modules on the SWP because of the possible increase of IPC and/or the violations of thread response time requirements.

(3) Module Deletions from LWP

If further module replications on the SWP does not improve T_{obj}, then the algorithm tries to delete some modules from the LWP in the hope that (a) IPC in the system is reduced, and/or (b) T_{obj} can be improved by deleting modules of some threads with less stringent response time requirements. The algorithm also takes a greedy step as finalizing the single module deletion from the LWP that yields the lowest T_{obj}.

The RMAP algorithm is listed in the following:

REPLICATED MODULE ASSIGNMENT ALGORITHM

1. Determine the initial module multiplicities; Or, use the module multiplicities of the previous local optimal assignment.

2. Generate a random module assignment A based on these multiplicities.

3. Invoke the task response time model to compute $T_{obj}(A)$ for assignment A, and reallocate module copies from $LWP(A)$ to $SWP(A)$ (without changing module multiplicities) until reaching a local optimal assignment A_o. (* As Steps 2 to 7 in the MAP Algorithm do except T_{ptp} is replaced by T_{obj} *)

4. Compute thread response time discrepancies $d_i(A_o)$ for all threads i where $i \varepsilon [1,k]$ and identify $LWP(A_o)$ and $SWP(A_o)$ for assignment A_o.

5. If there exists $d_i(A_o) > 0$ for any $i \varepsilon [1,k]$, then let S_R be the set of modules of all threads i where $d_i(A_o) > 0$ for all $i \varepsilon [1,k]$; (* some thread response time constraints violated *)

   Otherwise, let S_R be the set of modules residing on $LWP(A_o)$. (* all thread response time constraints satisfied *)

6. For each module $j \varepsilon S_R$ not residing on $SWP(A_o)$, perform

```
    6.1 Temporarily replicate $j$ on $SWP(A_o)$ thus becomes assignment $A_j$;
    6.2 Compute $T_{obj}(A_j)$ and reallocate module copies from $LWP(A_j)$ to
        $SWP(A_j)$ until reaching a local optimal assignment $A_{jo}$. (* As Step 3
        does *)
7. If there exists $T_{obj}(A_{jo}) < T_{obj}(A_o)$ for any $j \varepsilon S_R$ from Step 6, then
    7.1 Set $A_o = A_{jo}$, $T_{obj}(A_o) = T_{obj}(A_{jo})$, $LWP(A_o) = LWP(A_{jo})$, and
        $SWP(A_o) = SWP(A_{jo})$ where $T_{obj}(A_{jo}) = \min_{i \varepsilon S_R}\{ T_{obj}(A_{io}) \}$; (* To
        finalize a single module replication on SWP *)
    7.2 Go to Step 4.
8. Otherwise, let $S_L$ be the set of modules residing on $LWP(A_o)$. For each module
   $j \varepsilon S_L$ where j has more than one copy, perform
    8.1 Temporarily delete j from $LWP(A_o)$ thus becomes assignment $A_j$;
    8.2 Repeat Step 6.2 to obtain the local optimal assignment $A_{jo}$.
9. If there exists $T_{obj}(A_{jo}) < T_{obj}(A_o)$ for any $j \varepsilon S_L$ from Step 8, then
    9.1 Set $A_o = A_{jo}$, $T_{obj}(A_o) = T_{obj}(A_{jo})$, $LWP(A_o) = LWP(A_{jo})$, and
        $SWP(A_o) = SWP(A_{jo})$ where $T_{obj}(A_{jo}) = \min_{i \varepsilon S_L}\{ T_{obj}(A_{io}) \}$; (* To
        finalize a single module deletion from LWP *)
```

9.2 Go to Step 4.

10. Otherwise, Stop. (* Reach a local optimum *)

5.3.3 Initial Module Multiplicities

During the execution of the algorithm, module multiplicities are changed due to the module replications and deletions in searching for better assignments. To provide a good sub-optimal module assignment as a final solution for a given distributed system, the algorithm is re-iterated with a number of randomly selected assignments. In additional, to explore different assignments with the same module multiplicities, the module multiplicities of a local optimal solution are used to generate the next random module assignment (Step 1 in the RMAP Algorithm). However, the algorithm should start with a set of feasible initial module multiplicities. Therefore, the selection of initial module multiplicities is important and will be discussed in the following.

There are many ways to select the initial module multiplicities. The basic requirement is to select these initial multiplicities such that the processing requirement for each module copy does not saturate a processor. That is, the processor utilization due to each module copy, which is the product of the invocation rate and the mean execution time of the module copy, on each processor is less than unity. In addition, it is desirable to select the initial module multiplicities so that the processing workload can be easily balanced among the processors in the system. Based on these considerations of processor utilization, the following procedure is used to determine the initial module multiplicities for the RMAP Algorithm:

1. Assume the system has m distinct modules. Let ρ_i be the processor utilization of module i where $i \in [1,m]$. Based on the invocation rate and the mean execution time for module i, compute ρ_i for all $i \in [1,m]$.

2. Compute the mean processor utilization due to a module, $\rho_M = \frac{\sum_{i=1}^{m} \rho_i}{m}$.

3. Let α_i be the initial multiplicity for module i. For each $i \in [1,m]$, perform

   3.1 Set $\alpha_i = \left\lceil \frac{\rho_i}{\rho_M} \right\rceil$;

   3.2 If $\frac{\rho_i}{\alpha_i} > 1$ then reset α_i as the smallest integer such that $\frac{\rho_i}{\alpha_i} < 1$.

Note that if there exists any α_i such that $\alpha_i > n$ (the total number of processors in the system), then the system is bound to be saturated and no module assignment can provide satisfactory response times. Although the initial module multiplicities determined by the above procedure may not provide satisfactory thread response times at the beginning, these multiplicities are subsequently modified in searching for the module assignment with the minimum value of T_{obj}.

5.4 APPLICATION OF RMAP ALGORITHM TO THE SENTRY SYSTEM

The Sentry System [TITA85] is another real-time distributed system that processes radar signals for space defense applications. In the following, we shall first describe the system characteristics of the Sentry System. Then, we discuss the performance of the RMAP algorithm when applied to the Sentry System for the selected sets of thread response time requirements.

5.4.1 The Characteristics of the Sentry System

The Sentry System is a loosely coupled distributed system which consists of six processors interconnected by a high-speed bus. The application task comprises of 12 modules and its control-flow graph is given in Figure 5.1. The task response time for the system is defined as the weighted sum of the average module response times; that is,

$$T_{ptp} = \sum_{i=1}^{12} \frac{\lambda_i}{\lambda_{tot}} T_i + \sum_{i=1}^{12} \frac{\lambda_i}{\lambda_{tot}} \sum_{j=1}^{12} p_{ij}\, \delta_{ij}\, D_{net}(i,j)$$

where:

$\lambda_i =$ the invocation rate for M_i,

$\lambda_{tot} =$ the total invocation rate for all M_i for $i = 1$ to 12.

$T_i =$ the mean response time for M_i (averaged over the response times for all copies if M_i has several copies).

$p_{ij} =$ probability that M_i enables M_j upon the completion of execution,

$$\delta_{ij} = \begin{cases} 1 & \textit{if } M_i \textit{ enables } M_j \textit{ on a remote processor} \\ 0 & \textit{otherwise} \end{cases}$$

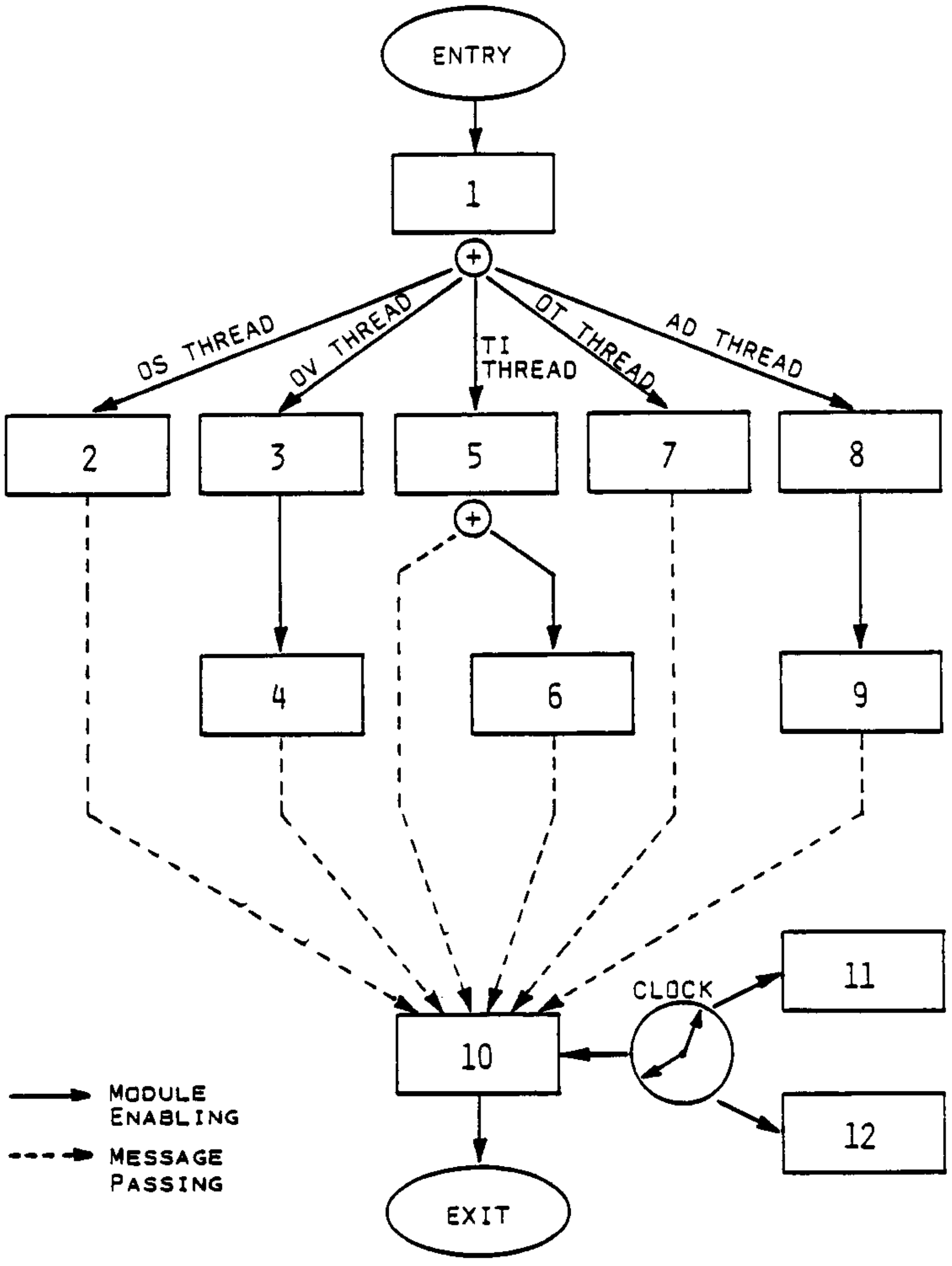


FIGURE 5.1 TASK CONTROL-FLOW FOR THE SENTRY SYSTEM

$D_{net}(i,j) =$ average interconnection network delay incurred by messages from M_i to M_j.

Among these modules, three modules (M_{10}, M_{11} and M_{12}) are periodically enabled by the system, while the rest of them are invoked according to the arrivals of radar return signals. When a return signal arrives, M_1 is invoked. After M_1's execution, control is branched to process a particular thread dependent upon the type of the return signal. The Sentry System gives names for various threads as shown in Figure 5.1. The response time for a thread is defined as the time from the arrival of a return signal at the system (i.e., M_1 is invoked) until the message sent by the last module of the thread to M_{10} is processed by the resident processor of M_{10}. Due to the thread response time and the loading requirements of the system, modules require to be replicated on several processors. In addition, since M_{11} performs some special functions, it is not allocated to any of the six processors in the system.

Data flow due to shared file access is presented in Figure 5.2. Each ellipse represents a data file. An arc pointing from a module to a file indicates a file-update, while pointing from a file to a module designates a file-read. An arc with double arrows means that the module will perform read and update to the file during the module execution.

The Sentry System has a detailed operating system for module scheduling and IPC processing. Module executions incur scheduling overheads. All module execution times (including their scheduling overheads and file access times) are deterministic

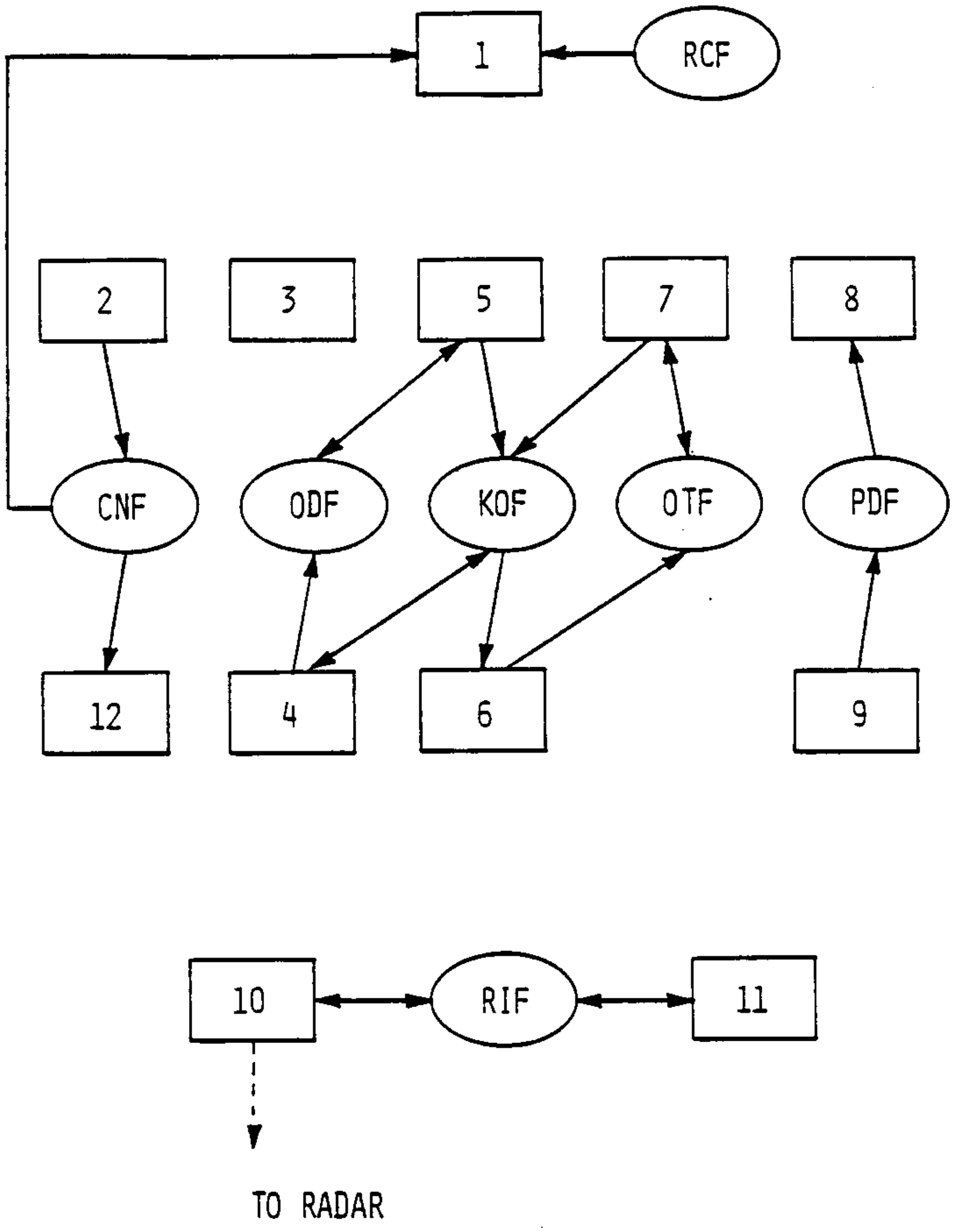


FIGURE 5.2 SHARED DATA FILES IN THE SENTRY SYSTEM

and shown in Table 5.1. Modules can communicate with other modules through sharing common data files and/or direct message exchange. The processing time for the IMC from M_i to another M_j is referred to as IMC time (IMC_{ij}) for the module pair. The IMC times for various module pairs are tabulated in Table 5.2. If the communicating modules (i and j) are located on distinct computers, the IMC becomes IPC which requires processing on both the transmitting and the receiving processors. The processing time for the IPC is called *IPC time* (IPC_{ij}). In the Sentry System, the IPC times on transmitting and receiving processors are different as shown below:

$$IPC_{ij} = \begin{cases} 80\ \mu sec & \textit{on the transmitting processor} \\ IMC_{ij} & \textit{on the receiving processor} \end{cases}$$

For the applications of the Sentry System, the invocation rates for the modules are given in Table 5.3. The branching probabilities of the control-flow graph can be determined by the invocation rates of those modules involved at the or-forks. The interconnection network delay is the bus delay in the Sentry System. This delay depends on the length of the message being transferred via the bus, and ranges from 0.165 to 0.2 msec.

5.4.2 Performance of the RMAP Algorithm

To study the performance of the Algorithm, we repeat the Algorithm with the selected sets of thread response time requirements, initial module multiplicities and the penalty delay scaling constant. Four selected sets of thread response time requirements, which are designated by R_A, R_B, R_C and R_D respectively, are shown in Table

Modules	Exec. + Scheduling Time	Read File/Time	Write File/Time	Total Exec. Time
1	138	RCF/5 CNF/63	-	206
2	199	-	CNF/98	297
3	1144	-	-	1144
4	286	KOF/66	ODF/149 KOF/138	639
5	1049	ODF/64	KOF/138 ODF/149	1400
6	355	KOF/66	OTF/149	570
7	1406	OTF/64	OTF/149 KOF/133	1752
8	1286	PDF/97	-	1383
9	981	-	PDF/215	1196
10	660	RIF/16	RIF/94	770
11	1137	RIF/26	RIF/84	1247
12	269	CNF/102	-	371

Note: All times are in micro-second.

Table 5.1 Average Module Execution and File Access Times for the Sentry System

Sending Modules	Receiving Modules	Fixed IMC Times (μs)
1	2	61
1	3	61
1	5	61
1	7	61
1	8	61
2	10	54
3	4	77
4	10	54
5	6	77
5	10	54
6	10	54
7	10	54
8	9	54
9	10	54
10	RADAR	127

Note: All other module pairs not listed here have zero IMC time.

Table 5.2 IMC Times for Various Module Pairs

Modules	Invocation Rates
1	1.58
2	0.57
3	0.1695
4	0.1695
5	0.6795
6	0.0075
7	0.1015
8	0.0595
9	0.0595
10	0.2
11	0.01
12	0.2

Table 5.3 Module Invocations Rates (No. of Enablements/msec)

for the Sentry System

5.4. Among these requirements, R_A represents a set of less stringent response time constraints, R_D is the most stricter one, and R_B and R_C are moderate. Besides the thread response time requirements, the Algorithm also requires two other sets of initial parameters: initial module multiplicities, α_i's, and the penalty delay scaling constant, a. Two different sets, α_A and α_B, of initial module multiplicities are tested, which are displayed in Table 5.5. Set α_A is obtained by the procedure discussed in Section 5.3.3. For set α_B, only M_5 is selected to have two copies as the processor utilization for M_5 is 95% which may easily saturate a processor, while other modules have a single copy. The penalty delay scaling constant is chosen to be 1, 10 or 1000. The Algorithm is repeated with 11 different combinations of these requirements, initial module multiplicities and scaling constant. The requirements and parameters for these 11 experiments are presented in Table 5.6. Experiments #1 through #9 use set α_A as initial module multiplicities, while Experiments #10 and #11 use set α_B. The scaling constant for Experiment #1 is 1. Experiments #2 to #5 and #10 use 10, while the rest use 1000 as the scaling constant. In each experiment, the RMAP Algorithm is re-iterated with 500 or 1000 randomly selected initial module assignments. For each initial assignment, the Algorithm generates a local optimal assignment. Dependent on the thread response time requirements, only certain proportion of the local optimal assignments can satisfy the constraints. The final sub-optimal solution is the local optimum that yields the minimum T_{obj}. The performances of the final sub-optimal module assignment generated for these these experiments, as well as some statistics of the Algorithm runs are portrayed in Table 5.7. The sub-optimal module assign-

Thread \ Thread Response Time Requirements (msec)	Sets of Requirements			
	R_A	R_B	R_C	R_D
OS	2.5	1.8	1.75	1.7
OV	7.0	6.6	6.55	6.5
TI	4.0	3.2	3.15	3.1
OT	4.5	4.0	3.95	3.9
OD	5.5	5.0	4.95	4.9

Table 5.4 Selected Sets of Thread Response Time Requirements

Modules	Initial Multiplicities	
	α_A	α_B
1	2	1
2	1	1
3	1	1
4	1	1
5	5	2
6	1	1
7	1	1
8	1	1
9	1	1
10	1	1
11	1	1
12	1	1

Table 5.5 Two Sets of Initial Module Multiplicities

Experiment No	1	2	3	4	5	6	7	8	9	10	11
Thread Response Time Requirements	R_B	R_A	R_B	R_C	R_D	R_A	R_B	R_C	R_D	R_B	R_B
Initial Module Multiplicities	α_A	α_A	α_A	α_A	α_A	α_A	α_A	α_A	α_A	α_B	α_B
Penalty Scal- ing Constants	1	10	10	10	10	1000	1000	1000	1000	10	1000

Table 5.6 11 Experiments with Selected Sets of Requirements, Module Multiplicities and Scaling Constants

Experiment No.		1	2	3	4 *	5 *	6	7	8 *	9 *	10	11
Thread Response Times (msec)	OS	1.741	1.394	1.699	1.653	1.678	1.441	1.586	1.616	1.678	1.642	1.767
	OV	6.051	5.894	6.052	6.136	6.153	5.657	6.236	6.309	6.131	6.044	6.261
	TI	3.204	3.186	3.173	3.101	3.169	3.247	3.171	3.128	3.160	3.167	3.189
	OT	3.710	3.639	3.838	3.667	3.743	3.657	3.578	3.767	3.738	3.687	3.776
	OD	4.977	5.057	4.954	4.797	4.848	4.556	4.772	4.798	4.844	4.990	4.995
T_{obj}		1.138	1.058	1.148	1.119	1.837	1.062	1.106	1.151	61.59	1.131	1.151
% of Local Optimum Meeting Response Time Requirements		15.8%	99.8%	15.9%	0.44%	0%	99.8%	14.5%	0.33%	0%	13.2%	14.3%
No. of Module Assignments Tested		21241	23868	19621	43755	43764	22960	19155	43455	45045	19572	19648
CPU Time (Hour)		1.62	1.52	1.48	3.48	3.42	1.55	1.50	3.35	3.37	1.48	1.53

Note: Experiments with * indicate that the Algorithm was re-iterated with 1000 random initial module assignments, while others with 500 random assignments.

Table 5.7 The Performance of the Sub-Optimal Module Assignment and the RMAP Algorithm After Iterating with 500/1000 Random Initial Assignments

ments for all these 11 experiments are shown in Table 5.8. We notice that since requirement set R_D is so stringent that no module assignment which meets the thread response time requirements can be generated for Experiments #5 and #9. During each of these experiments, 19,500 to 45,000 module assignments with various module multiplicities are tested. The CPU time for the algorithm runs ranges from 1.48 to 3.48 hours on a VAX 11/780 machine. Further, from these experiments, we recognize the following characteristics of the Algorithm:

(1) Higher T_{ptp} for the more stringent requirements

For Experiments #2 to #4 and #6 to #8, T_{ptp} ($= T_{obj}$) is higher for the more stricter thread response time requirements. The reason is because the modules of a thread with a strict response time constraint need to be allocated to some lightly loaded processors, while other modules are residing on the relatively highly-loaded processors. As a result, T_{ptp} cannot be further reduced during the search of better module assignments by either module reallocations, replications, or deletions; otherwise, the stringent thread response time constraints are violated. Conversely, if the threads have less stringent response time requirements, the Algorithm has more flexibility in searching for better assignments with lower T_{ptp}.

(2) Selections of the penalty delay scaling constant

Clearly, the selection of the scaling constant does not affect the performance of the Algorithm if the threads have less strict response time requirements. But, the

Experiment No.	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6
1	5 8 9 10	1 2 5 8 9	1 2 5 12	1 2 3 8 9	2 4 5 6 8 9	5 7
2	1 2 3 12	3 5 7	5 8 9	1 2 4 6 12	3 5 7	5 10
3	1 2 5 8 9	5 7 8 9	4 5 6 8 9	1 2 8 9 10	1 5	1 2 3 8 9 12
4	1 2 10	5 8 9	1 2 4 8 9 12	3 4 6 7	5 8 9	5 8 9
5	4 5 8 9	1 2 8 9 10	1 2 5 8 9	5 7	1 2 5	1 2 3 6 8 9 12
6	5 7	1 2 3 4 6 12	3 8 9 10	5 7	3 5	1 2 3 4 6 12

Table 5.8 The Sub-Optimal Module Assignments for the 11 Experiments

Experiment No.	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6
7	2 5	1 2 8 9 12	5 6 8 9	1 2 10	4 5	3 7 8 9
8	1 2 5 8 9	1 2 5 8 9	1 2 10 12	4 5 6 7	1 2 5 8 9	3 4 5
9	1 2 5	1 2 8 9 10	1 2 3 4 6 8 9 12	5 7	1 2 5 8 9	3 5 8 9
10	5 7 8 9	4 5 6 8 9 10	1 2 5 8 9 12	3 4 5	1 2 5 8 9 12	1 2 5 8 9
11	1 2 4 5 6	5 8 9 10	1 2 5 8 9 12	1 2 5 8 9	2 3 5	4 5 7

Table 5.8 (cont.) The Sub-Optimal Module Assignments for the 11 Experiments

scaling constant should be properly chosen for stringent sets of constraints. By comparing the response time performance of the sub-optimal solution for Experiment #1 to its requirements (Table 5.4), we notice that TI thread violates its response time specification. In fact, many local optimal assignments which satisfy the requirements, R_B, have been generated in Experiment #1. However, the scaling constant of 1 is too small for the Sentry System. Thus, although the TI thread slightly violates its response time constraint, the "sub-optimal" assignment yields the minimum T_{obj} and is considered to be the "best" assignment. Therefore, the penalty scaling constant should be chosen sufficiently large in order to correctly generate the sub-optimal solutions.

The scaling constant should be so selected that T_{obj} for an assignment which meets the response time constraints should be less than that of another assignment which violates some of the response time requirements. For the Sentry System, T_{ptp} is about unity and thread response times are a few times of T_{ptp}. These experiments indicate that setting the scaling constant equal to or greater than 10 is sufficient. Except Experiment #1, all other experiments use 10 or 1000 as the scaling constant. Our results show that the Algorithm is able to generate sub-optimal solutions with very close response time performance. In addition, we also repeated the experiments with $a = 50$, the Algorithm also generated sub-optimal assignments with similar response time performance. Therefore, the experimental results show that the performance of the Algorithm is insensitive to the selections of the penalty delay scaling constant within this wide range from 10 to 1000.

(3) Insensitivity of the initial module multiplicities

Experiments #10 and #11 use the set α_B as initial multiplicities, and the threads have the moderately stringent set of response time requirements, R_B. The results for Experiments #10 and #11 are closely compared to those of #3 and #7. It reveals that the performance of the Algorithm is also insensitive to the initial module multiplicities, provided that no single module copy for the initial multiplicities can saturate a processor.

5.5 SUMMARY

The design objective for replicated module assignment in RTDPS is to minimize the task response time with the constraints that thread response time requirements are satisfied. A new objective function which is sum of the task response time and the possible penalty delay to account for the violations of thread response time requirements has been introduced. Based on this objective function, the RMAP Algorithm has been developed by extending the MAP Algorithm for handling module replications. The RMAP Algorithm consists of three components to: (1) reallocate module from LWP to SWP, (2) further replicate modules on SWP, and (3) delete modules from LWP. This new algorithm was applied to the Sentry System which is a loosedly coupled distributed system with six processors interconnected by a bus. The algorithm optimally allocates module copies to computers and the module multiplicities are iteratively determined so as to achieve the design objective. The algorithm has been validated and proved to be efficient for generating sub-optimal module assignments which meet the response time specifications while the overall task response time is minimized. Further, the Algorithm is insensitive to the selections of initial module multiplicities and the penalty delay scaling constant if they are properly chosen. Therefore, the RMAP Algorithm is a robust tool for performing module assignment in replicated processing environments.

CHAPTER 6

CONCLUSIONS & FUTURE RESEARCH WORK

6.1 CONCLUSIONS

Response time is an important design criterion for real-time systems. Therefore system designers have to design RTDPS such that the prescribed response time specifications can be satisfied. In general, either analytical or simulation techniques can be employed to study response times for RTDPS. However, simulation methods usually tend to be expensive and time-consuming. These shortcomings have led us to pursuing analytical approaches for the problem. To overcome the inadequacies of current analytical approaches, a new analytic model has been introduced for estimating task response time for loosely coupled distributed processing systems. Our model considers such factors as IPC, module precedence relationships, module scheduling, interconnection network delay, and assignment of modules and files to computers. Simulation experiments have been used to validate the model assumptions and reveal that our model can provide good task response time estimations.

Our model is applicable in various design and performance studies of distributed systems. First of all, we have used the analytic model to study the effects of PR on module response times, and the importance of PR effects in module assignment for distributed systems. Our study reveals that the mean execution time ratio of a pair of consecutive modules plays an important role in the PR effects. Besides PR effects, IPC, load balancing and interconnection network delay also affect the task response time for a distributed system. Since the analytic model considers all these factors, it can accurately estimate task response times for various module assignments.

Based on the model, we have developed a new local search algorithm for module assignment in RTDPS. In this algorithm, the task response time model becomes the objective function of the module assignment problem. And, the task response time is optimized over all possible module assignments. Search strategies are established in the algorithm to look for module assignments which provide shorter task response times. We first considered the cases where each module is allocated to a single computer. Then, we have extended the algorithm to handle module replications; that is, modules may be replicated and processed on several computers. For the distributed systems where module replications are required, not only the module copies are optimally allocated to computers, but the appropriate module multiplicities are also iteratively determined by the algorithm.

The module assignment algorithm has been successfully applied to two distributed systems for space defense applications: the DPAD and Sentry Systems. System specifications do not require module replications for the DPAD System, while it does require for the Sentry System. The applications of the algorithm to these systems have demonstrated the efficiency and versatility of the algorithm, and also served as another validation of the analytic model. The module assignments generated by the algorithm provide excellent response time performance because the algorithm is based on the task response time model which has considered all the major factors that affect task response time in the distributed systems. Therefore, the module assignment algorithm should serve as a valuable tool for designing distributed systems.

6.2 FUTURE RESEARCH WORK

A number of research areas which need further investigations. We shall describe these research problems in the following:

6.2.1 Database Management Strategies

Distributed systems require protocols to ensure internal and mutual data consistency for simultaneous access of replicated data files. These protocols require extra IPC, processing, and increase module response delays. Common consistency controls include locking, timestamp, and exclusive-writer protocol [BERN81, CHU85a]. In the task response time model, the processing of IPC on a processor is treated as a special module execution. If the module execution has to be delayed for handling the data consistency problem, the module execution time is correspondingly prolonged. It is desirable to incorporate these effects of concurrency controls into the response time model. Hence, the model can be used to study the overhead in terms of task response time of several commonly used database concurrency control algorithms such as locking, timestamping, and the exclusive-writer protocol. The results of these investigations should provide insight into the performance as well as overhead of concurrency control algorithms for distributed systems at various operating environments.

6.2.2 Module and File Allocations

In this research, we assume files are stored (or replicated) on a computer where its residing modules may access to the files. However, for some applications, this file allocation strategy may generate a large volume of IPC in the system because a lot of files are replicated on a large number of computers and all these file copies require to be updated. To overcome this potential processor saturation, it is desirable to replicate and allocate files such that both IPC and response times are minimized. Since file access are generated from the module executions, module assignment and file allocation need to be considered simultaneously. Therefore, we need to determine both the module and file multiplicities, and allocate these module and file copies to computers in order to minimize the task response time and to satisfy the system response time specifications.

6.2.3 Tightly Coupled Distributed Systems

Since the loosely and tightly coupled systems share a lot of common characteristics such as data consistency requirements, module scheduling, precedence relationships and IPC, the task response time model can be generalized to be applicable to tightly coupled distributed systems. In tightly coupled systems, processors communicate via sharing common memory, which may cause memory access conflicts. The delay due to these memory conflicts can be treated by prolonging the module execution time in the task response time model. In addition, our response time model needs to be extended to model the generalized class of data concurrency and synchronization mechanisms such as ADA's rendezvous operations which are commonly used in tightly coupled systems.

REFERENCES

BASK75 F. Basket, K.M. Candy, R. Muntz and F.G. Palacios, "Open, Closed, and Mixed Networks of Queues with Different Classes of Customers," *Journal of the ACM*, vol. 22, No. 2, pp. 248-260, April 1975.

BERN81 P.A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, vol. 13, no. 2, pp. 185-221, June 1981.

BERR82 R. Berry, K.M. Chandy, J. Misra, and D. Neuse, "PAWS 2.0 — Performance Analyst's Workbench System: User's Manual," Information Research Associates, Austin, Texas, December 1982.

CASE72 R.G. Casey, "Allocation of Copies of a File in an Information Network," *AFIPS, SJCC*, 1972, pp. 617-625.

CHAU83 M.L. Chaudhry and J.G.C. Templeton, A First Course in Bulk Queues, John Wiley & Sons, Inc., New York, 1983.

CHEN80 P.P.S. Chen and J. Akoka, "Optimal Design of Distributed Information System," *IEEE Trans. on Computer*, vol. C-29, pp. 1068-1080, Dec. 1980.

CHOU82 T.C.K. Chou and J.A. Abraham, "Load Balancing in Distributed Systems," *IEEE Trans. on Software Eng.*, vol. SE-8, no. 4, pp. 401-412, July 1982.

CHU69 W.W. Chu, "Optimal File Allocation in a Multiple Computer System," *IEEE Trans. on Computers*, vol. C-18, no. 10, pp. 885-889, Oct. 1969.

CHU78 W.W. Chu, D. Lee and B. Iffla, "A distributed processing system for naval data communication networks," in *Proc. AFIPS National Computer Conf.*, vol. 47, pp. 783-793, 1978.

CHU80 W.W. Chu, L.J. Holloway, M.T. Lan and K. Efe, "Task Allocation in Distributed Data Processing," *Computer*, vol. 13, no. 11, pp. 57-69, Nov. 1980.

CHU84 W.W. Chu, M.T. Lan and J. Hellerstein, "Estimation of Intermodule Communication (IMC) and Its Applications in Distributed Processing Systems," *IEEE Trans. on Computers*, vol. C-33, no. 8, pp. 691-699, Aug. 1984.

CHU85a W.W. Chu and J. Hellerstein, "The Exclusive-Writer Approach to Updating Replicated Files in Distributed Processing Systems," *IEEE Trans. on Computer*, vol. C-34, no. 6, pp.489-500, June 1985.

CHU85b W.W. Chu and L.M.T. Lan, "Task Allocation and Precedence Relations for Distributed Real-Time Systems," Submitted to *IEEE Trans. on Computers* for publication.

COFF81 E.G. Coffman, Jr. et al., "Optimization of the Number of Copies in a Distributed Data Base," *IEEE Trans. on Software Eng.*, vol. SE-7, no. 1, pp. 78-84, Jan. 1981.

CONW67 R.W. Conway, W.L. Maxwell and L.W. Miller, Theory of Scheduling, Addison-Wesley (Reading, Mass.), 1967.

EFE82 Kemal Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *Computer*, vol. 15, no. 6, pp. 50-56, June 1982.

GARE79 M.R. Garey and D.S. Johnson, Computers and Intractability, W.H. Freeman and Company, San Francisco, 1979.

GREE80 M. L. Green, E. Y. S. Lee, S. Majumdar, and D. C. Shannon, "Phase III of Distributed Processing Architecture Design (DPAD) Program -- the DDP Underlay Simulation Experiment: Tactical Applications and d-RTOS Models," TRW Defense and Space Systems Group, Special Report 35010-79-A005, Redondo Beach, Calif., May 15, 1980.

HEID82 P. Heidelberger and K.S. Trivedi, "Queueing Network Models for Parallel Processing with Asynchronous Tasks," *IEEE Trans. on Computers*, vol. C-31, no. 11, pp. 1099-1109, Nov. 1982.

JENN77 C.J. Jenny, "Process Partitioning in Distributed Systems," *Proc. NTC 1977*, pp. 31:1-1 -- 31:1-10.

KLEI76 L. Kleinrock, Queueing System Volume II: Computer Application, Wiley, New York(1976).

LAN85 L.M.T. Lan, "Characterization of Intermodule Communications and Heuristic Task Allocation for Distributed Real-Time Systems," Ph.D. dissertation, Computer Science Dept., Univ. of California, Los Angeles, 1985.

LAZO84 E.D. Lazowska, J. Zahorjan, G.S. Graham and K.C. Sevcik, Quantitative System Performance: Computer System Analysis Using Queueing Network Models, Prentice-Hall, New Jersey (1984).

LIN65 S. Lin, "Computer Solutions of the Traveling Salesman Problem," *Bell Syst. Tech. Journal*, December 1965.

LITT61 J.D.C. Little, "A Proof of the Queueing Formula $L = \lambda W$," *Operations Research*, 9, pp.383-387(1961).

MA82 P.Y.R. Ma, E.Y.S. Lee and M. Tsuchiya, "A task allocation model for distributed computing systems," *IEEE Trans. on Computers*, vol. C-31, no. 1, pp. 41-47, Jan. 1982.

MORG77 H.L. Morgan and K.D. Levin, "Optimal Program and Data Locations in Computer Networks," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 315-322, May 1977.

PAPA82 C.H. Papadimitriou and K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity, Prentice Hall, Inc., 1982.

PRIC81 C.C. Price, "The Assignment of Computational Tasks among Processors in a Distributed System," *Proc. Natl. Comput. Conf.*, May 1981, pp. 291-296.

RAO79 G.S. Rao, H.S. Stone and T.C. Hu, "Assignment of Tasks in a Distributed Processing System with Limited Memory," *IEEE Trans. on Computers*, vol. C-28, no. 4, pp. 291-299, Apr. 1979.

RAMA82 C.V. Ramamoorthy and B.W. Wah, "The Isomorphism of Simple File Allocation," *IEEE Trans. on Computers*, vol. C-32, no. 3, pp. 221-232, March 1983.

SAUE81 C.H. Sauer and K.M. Chandy, *Computer System Performance Modeling*. Prentice-Hall, Inc., 1981.

SCHO83 M. Scholl and L. Kleinrock, "On the M/G/1 Queue with Rest Period and Certain Service-Independent Queueing Disciplines," *Operations Research*, 31, pp.705-719(1983).

SHEN85 C.C. Shen and W.H. Tsai, "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion," *IEEE Trans. on Computers*, vol. C-34, no. 3, pp. 197-203, March 1985.

STON77 H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. on Software Eng.*, vol. SE-3, no. 1, pp. 85-93, Jan. 1977.

TITA85 Titan System, Inc., "Distributed Data Base Management (DDBM) Analysis," Semi-Annual Report, Contract No. DASG60-83-C-0080, CDRL No. A004, July 19, 1985.

APPENDIX A

RESPONSE TIME FOR AND-FORK TO AND-JOIN SUBGRAPH

Suppose the and-fork to and-join graph consists of n sequential threads as shown in Figure 2.4. Let us use y_i to denote the response time (a random variable) for the i^{th} thread, $\bar{y}_i$ and σ_i^2 its mean and variance respectively. Let Y be the response time for the control-flow subgraph. Due to the and-join function, the response time for this subgraph is the maximum of all threads response times y_i's. That is

$$Y = \max \{ y_1 , y_2 , \cdots y_n \} \tag{A.1}$$

Let the probability density function (pdf) and distribution function (PDF) for the random variable y_i be $f_i(y)$ and $F_i(y)$ respectively. Assume that all y_i's are independent, then we have

$$\begin{aligned} P(y) &= Prob[\, Y \leq y \,] \\ &= Prob[\, y_1 \leq y \,] \cdots Prob[\, y_n \leq y \,] \\ &= \prod_{i=1}^{n} F_i(y) \end{aligned} \tag{A.2}$$

Hence the m^{th} moment of the subgraph response time $\overline{y^m}$ can be computed by

$$\overline{y^m} = \int_0^\infty y^m \, dP(y) \tag{A.3}$$

A.1 RESPONSE TIME FOR A TWO-THREAD AND-TO-AND SUBGRAPH

If $F_i(y)$ is a general distribution function or n>2, the computation of the moments of subgraph response time from Eq.(A.3) will become very complicated. To overcome this difficulty, let us now consider a two-thread and-fork to and-join subgraph (i.e., $n = 2$). Based on the mean and variance ($\overline{y}_i$, σ_i^2) of response time for the i^{th} thread, we can approximate the pdf for the response time by either a branch Erlangian or a hyper-exponential distribution function. Therefore, if the coefficient of variation for the thread response time is less than 1, the pdf $f_i(y)$ for the i^{th} thread can be approximated by the following branch Erlangian function:

$$f_i(y) = p_i \, \mu_i e^{-\mu_i y} + (1-p_i) \frac{\mu_i^{\,n_i}}{(n_i - 1)!} \, y^{n_i - 1} \, e^{-\mu_i y} \tag{A.4}$$

where:

$$n_i = \left\lceil (\overline{y_i}/\sigma_i)^2 \right\rceil$$

$$p_i = \frac{2n_i(\sigma_i/\overline{y}_i)^2 + n_i - 2 - \sqrt{n_i^2 + 4 - 4n_i(\sigma_i/\overline{y}_i)^2}}{2[(\sigma_i/\overline{y}_i)^2 + 1\,](n_i - 1)}$$

$$\mu_i = \frac{n_i - p_i\,(n_i - 1)}{\overline{y}_i}$$

In case the coefficient is greater than 1, then $f_i(y)$ can be approximated by a hyper-exponential function as

$$f_i(y) = p_i\, \mu_{i1}\, e^{-\mu_{i1} y} + (1-p_i)\, \mu_{i2} e^{-\mu_{i2} y} \tag{A.5}$$

where:

$$p_i = \frac{(\sigma_i/\bar{y}_i)^2 + 1 - \sqrt{(\sigma_i/\bar{y}_i)^4 - 1}}{2[(\sigma_i/\bar{y}_i)^2 + 1\,]}$$

$$\mu_{i1} = \frac{2p_i}{\bar{y}_i}$$

$$\mu_{i2} = \frac{2(1-p_i)}{\bar{y}_i}$$

Since n=2, Eq.(A.2) becomes

$$P(y) = F_1(y) F_2(y) \tag{A.6}$$

Substituting P(y) from Eq.(A.6) in Eq.(A.3), the mean response time for subgraph can be computed as

$$\bar{y} = \int_0^{\infty} yF_1(y) f_2(y) dy + \int_0^{\infty} yF_2(y) f_1(y) dy \tag{A.7}$$

Suppose both $f_1(y)$ and $f_2(y)$ are branch Erlangian distribution functions as shown in Eq.(A.4). After some algebraic manipulation, we obtain the mean task response time for the two-thread and-fork to and-join graph. That is

$$\bar{y} = p_1 p_2 \mu_2 [\, \frac{1}{\mu_2^2} - \frac{1}{(\mu_1 + \mu_2)^2}\,]$$

$$+ p_1 q_2 (n_2)! \left[\frac{1}{\mu_2^{n_2+1}} - \frac{1}{(\mu_1+\mu_2)^{n_2+1}} \right]$$

$$+ p_2 q_1 \frac{(n_1-1)!}{(\mu_1+\mu_2)^{n_1}} \left[\frac{n_1}{\mu_1+\mu_2} + \frac{1}{\mu_2} \right]$$

$$+ q_1 q_2 \left[- \frac{1}{\mu_1} \frac{(n_1+n_2-1)!}{(\mu_1+\mu_2)^{n_1+n_2}} - \sum_{k=1}^{n_1-1} \frac{(n_1-1)\ldots(n_1-k)}{\mu_1^{k+1}} \frac{(n_1+n_2-k-1)!}{(\mu_1+\mu_2)^{n_1+n_2-k}} \right.$$

$$\left. + \frac{(n_1-1)!}{\mu_1^{n_1}} \frac{n_2!}{\mu_2^{n_2+1}} \right]$$

$$+ p_1 p_2 \mu_1 \left[\frac{1}{\mu_1^2} - \frac{1}{(\mu_1+\mu_2)^2} \right]$$

$$+ p_2 q_1 (n_1)! \left[\frac{1}{\mu_1^{n_1+1}} - \frac{1}{(\mu_1+\mu_2)^{n_1+1}} \right]$$

$$+ p_1 q_2 \frac{(n_2-1)!}{(\mu_1+\mu_2)^{n_2}} \left[\frac{n_2}{\mu_1+\mu_2} + \frac{1}{\mu_1} \right]$$

$$+ q_1 q_2 \left[- \frac{1}{\mu_2} \frac{(n_1+n_2-1)!}{(\mu_1+\mu_2)^{n_1+n_2}} - \sum_{k=1}^{n_2-1} \frac{(n_2-1)\ldots(n_2-k)}{\mu_2^{k+1}} \frac{(n_1+n_2-k-1)!}{(\mu_1+\mu_2)^{n_1+n_2-k}} \right.$$

$$\left. + \frac{(n_2-1)!}{\mu_2^{n_2}} \frac{n_1!}{\mu_1^{n_1+1}} \right] \qquad \text{(A.8)}$$

where:

$$q_1 = (1-p_1) \frac{\mu_1^{n_1}}{(n_1-1)!}$$

$$q_2 = (1-p_2)\,\frac{\mu_2^{n_2}}{(n_2-1)!}$$

Similarly, the second moment of the response time for the subgraph can be computed. By the same token, we can compute the moments of a two-thread subgraph's response time for the cases where the pdf's for thread response times are approximated by different combinations of the distribution functions shown in Eq.(A.4) and (A.5).

A.2 RESPONSE TIME FOR A N-THREAD AND-TO-AND SUBGRAPH

Now, we consider an and-fork to and-join subgraph which consists of n threads. Since the integrals for $\bar{y}$ (See Eq.(A.8)) and $\overline{y^2}$ may involve the incomplete Gamma function, the integrals are mathematically intractable if the subgraph has more than two threads (i.e., $n > 2$). However, the following shows a method to estimate the mean and second moment of the response time for the n-thread subgraph.

Due to the fork and join functions, a n-thread subgraph can be re-organized as shown in Figure A.1. The subgraph then contains another subgraph with two threads. These two threads are firstly aggregated into one. To aggregate all n threads, let us define u_i as the aggregated response time after aggregating the 1^{st} up to the i^{th} thread. We have

$$u_1 = y_1$$

$$u_{i+1} = \max\{u_i, y_{i+1}\} \tag{A.9}$$

for all $i = 1,2,\ldots,n-1$. Recall that

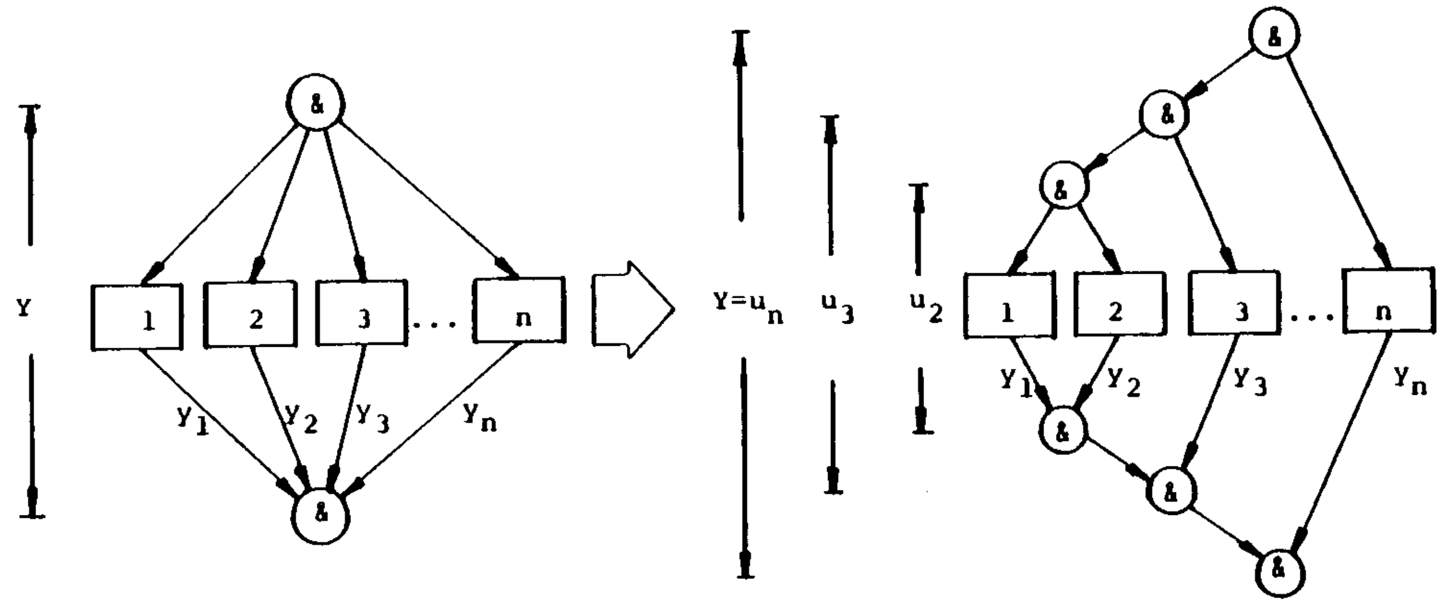


Figure A.1 Aggregating the N-Thread And-Fork to And-Join Subgraph

$$\begin{aligned} Y &= \max\{y_1, y_2, \cdots y_n\} \\ &= \max\{\max\{y_1, y_2\}, y_3, \cdots y_n\} \\ &= \max\{u_2, y_3, \cdots y_n\} \\ &\ldots\ldots \\ &= \max\{u_{n-1}, y_n\} \\ &= u_n \end{aligned}$$

As from Eq.(A.9), the mean and second moment of the aggregated response time u_{i+1}, denoted as $\overline{u_{i+1}}$ and $\overline{u_{i+1}^2}$, can be computed from the approximated pdf's for u_i and y_{i+1}. This suggests that we can further approximate u_{i+1}'s pdf by another branch Erlangian or hyper-exponential distribution function based on $\overline{u_{i+1}}$ and $\overline{u_{i+1}^2}$. Thus we can aggregate these two threads and yield the approximated pdf for the aggregated response time as a result. After aggregating these two threads, the number of threads in the subgraph is reduced by 1. Therefore, by repeating this two-thread aggregation for n-1 times, the first two moments of the response time for this n-thread and-fork to and-join subgraph, $\bar{y}$ and $\overline{y^2}$, can be computed.

APPENDIX B

MEAN MODULE RESPONSE TIMES WITH DEPENDENT INVOCATIONS

Consider a computer allocated with modules of v distinct control-flow subgraphs. Each subgraph consists of one or more sequential threads which may be invoked simultaneously (See Figure 2.8(A)). Assume that the i^{th} subgraph consists of b_i sequential threads, and the j^{th} sequential thread comprises $d_i(j)$ modules for $j=1,2,...b_i$. For a subgraph consists of a single sequential thread, then $b_i = 1$. Let $M_i(j;k)$ be the k^{th} module (starting from the entry module) of the j^{th} sequential thread for the i^{th} subgraph. In addition, let $P_i(j;k,t)$ be the probability of invoking $M_i(j;t)$ given that $M_i(j;k)$ is executed for $i=1,2,...v$, $j=1,2,...b_i$ and $1 \leq k \leq t \leq d_i(j)$. Thus if a subgraph does not contain any or-fork, $P_i(j;k,t) = 1$ for all k and t. For a subgraph containing a branch of an or-fork, the modules in this branch are not invoked if the control branches to the modules that do not reside on the same computer. Thus we have,

$$P_i(j;k,k) = 1 \tag{B.1}$$

$$\text{and} \quad P_i(j;k,k+a) = \prod_{t=k}^{k+a-1} P_i(j;t,t+1) \tag{B.2}$$

for $1 \leq k \leq k+a \leq d_i(j)$.

B.1 MEAN WAITING TIME FOR ENTRY MODULES

Let $W_i\ (j;k)$ be the mean waiting time for module $M_i\ (j;k)$. According to the assumptions 1a and 2a in Section 2.2.3, the mean waiting time for $M_i\ (1;1)$ (i.e., an entry module) for all $i=1,2,...v$ under the first-come-first-serve discipline is the average time to complete the current module execution and all the modules in the job queue on the computer when the invocation for $M_i\ (1;1)$ arrives. Thus we have

$$W_i\ (1;1) = W_o + \sum_{r=1}^{v} \sum_{s=1}^{b_r} \sum_{t=1}^{d_r(s)} \bar{n}_r(s;t)\, \bar{x}_r\ (s;t) \tag{B.3}$$

where:

$\bar{n}_r\ (s;t) =$ average number of invocations for $M_r\ (s;t)$ waiting in the job queue,

$\bar{x}_r\ (s;t) =$ average execution time for $M_r\ (s;t)$,

$W_o =$ mean residual module execution time

$$= \sum_{r=1}^{r} \sum_{s=1}^{b_r} \sum_{t=1}^{d_r(s)} \frac{1}{2} \lambda_r\ (s;t)\, \overline{x^2}_r\ (s;t),$$

$\overline{x^2}_r\ (s;t) =$ second moment of execution time for $M_r\ (s;t)$,

$\lambda_r\ (s;t) =$ invocation rate for $M_r\ (s;t)$.

Based on Little's result [LITT61] (i.e., $\bar{n}_r\ (s;t) = \lambda_r\ (s;t)\ W_r\ (s;t)$) and substituting the computer utilization of $M_r\ (s;t)$, $\rho_r\ (s;t) = \lambda_r\ (s;t)\, \bar{x}_r\ (s;t)$, into Eq.(B.3), we have

$$W_i\ (1;1) = W_o + \sum_{r=1}^{v} \sum_{s=1}^{b_r} \sum_{t=1}^{d_r(s)} \rho_r(s;t)\ W_r(s;t) \tag{B.4}$$

For simplicity in notation, we can order the thread index j such that the execution sequence for the bulk module invocations is to execute module $M_i\ (j;1)$ before module $M_i\ (s;1)$ for $j<s$. Thus the mean waiting time $W_i\ (j;1)$ is

$$W_i\ (j;1) = W_i\ (1;1) + \sum_{s=1}^{j-1} \bar{x}_i\ (s;1) \tag{B.5}$$

for $i=1,2,...v$, and $b_i \geq j \geq 2$.

B.2 MEAN WAITING TIME FOR NON-ENTRY MODULES

Let us consider the waiting time for the non-entry modules $M_i\ (j;k)$ (i.e., with $k \geq 2$). After completing its execution, a module invokes its succeeding module, if any, and places the invocation at the end of the job queue. Since these invocation arrivals are dependent and non-Poisson distributed, we need to keep track of the invocations generated from the modules residing on the local computer as well as the newly arrived module invocation from other computers. The waiting time for the non-entry modules can be divided into three components. The first component, $W1_i\ (j;k)$, is due to the executions of the succeeding modules invoked by the module invocations which are being executed or waiting in the job queue upon the arrival of the invocation for $M_i\ (j;1)$. The second component, $W2_i\ (j;k)$, is due to the waiting for the module executions invoked by the bulk module invocations (i.e., $M_i\ (s;1)$, $s = 1, \cdots b_i$ and $s \neq j$). The last component, $W3_i\ (j;k)$, is the waiting time due to the module invocations from other computers that arrive after the invocation for the entry module and their succeeding modules. Thus,

$$W1_i\ (j;k) = \sum_{r=1}^{v} \sum_{s=1}^{b_r} \sum_{t=1}^{d_r(s)-k+1} \bar{n}_r(s;t)\, P_r\ (s;t,t+k-1)\, \bar{x}_r\ (s;t+k-1)$$

$$+ \sum_{r=1}^{v} \sum_{s=1}^{b_r} \sum_{t=1}^{d_r(s)-k+1} \rho_r\ (s;t)\, P_r\ (s;t,t+k-1)\, \bar{x}_r(s;t+k-1) \qquad \text{(B.6)}$$

The first term of Eq.(B.6) is the total time for executing the succeeding modules invoked by $M_r\ (s;t)$ waiting in the job queue upon the arrival of the invocation for $M_i\ (j;1)$. Similarly, the second term is the execution times of the modules succeeding the module $M_r\ (s;t)$ which has probability $\rho_r\ (s;t)$ of being executed when the invocation for $M_i\ (j;1)$ arrives.

According to the definition of invocation probability $P_r\ (s;t,t+k-1)$,

$$\lambda_r\ (s;t+k-1) = \lambda_r\ (s;t)\, P_r\ (s;t,t+k-1) \qquad \text{(B.7)}$$

Applying Little's result and substituting Eq.(B.7) and $\rho_r\ (s;t) = \lambda_r\ (s;t)\, \bar{x}_r\ (s;t)$ into Eq.(B.6), after some algebraic manipulation, it yields

$$W1_i\ (j;k) = \sum_{r=1}^{v} \sum_{s=1}^{b_r} \sum_{t=1}^{d_r(s)-k+1} \rho_r\ (s;t+k-1)[W_r(s;t)+ \bar{x}_r(s;t)] \qquad \text{(B.8)}$$

The second component of $W_i\ (j;k)$ is

$$W2_i\ (j;k) = \sum_{s=1}^{j-1} P_i\ (s;1,k)\, \bar{x}_i(s;k) + \sum_{s=j+1}^{b_i} P_i\ (s;1,k-1)\, \bar{x}_i(s;k-1) \qquad \text{(B.9)}$$

The first term in Eq.(B.9) is the execution times for the modules succeeding the entry module(s) (M_i $(s;1)$ for $s<j$) that are executed before M_i $(j;1)$ in the bulk module invocation. The second term is the execution times of those modules succeeding the entry module(s) (M_i $(s;1)$ for $s>j$) executed after M_i $(j;1)$.

The third component of W_i $(j;k)$ is used to keep track of the new module invocations that arrive after the invocation for the entry module M_i $(j;1)$ and their succeeding modules subsequently generated from the newly arrived invocations. Thus, we have

$$W3_i\ (j;k) = [W_i\ (j;k-1) + \bar{x}_i\ (j;k-1)] \sum_{r=1}^{v} \sum_{s=1}^{b_r} \lambda_r(s;1)\, \bar{x}_r(s;1)$$

$$+[W_i(j;k-2)+\bar{x}_i(j;k-2)] \sum_{r=1}^{v} \sum_{\substack{s=1 \\ d_r(s)\geq 2}}^{b_r} \lambda_r(s;1)\, P_r(s;1,2)\, \bar{x}_r(s;2)+\cdots$$

$$+[W_i\ (j;1)+\bar{x}_i(j;1)] \sum_{r=1}^{v} \sum_{\substack{s=1 \\ d_r(s)\geq k-1}}^{b_i} \lambda_r(s;1)\, P_r(s;1,k-1)\, \bar{x}_r(s;k-1) \tag{B.10}$$

Due to the first-come-first-serve scheduling, those module invocations from other computers arriving during the response time (waiting plus execution) of M_i $(j;k-1)$ contribute a part of the waiting time for M_i $(j;k)$. Thus the first term of Eq.(B.10) is the total time for executing those module invocations arriving during the response time of M_i $(j;k-1)$. Likewise, the remaining terms of Eq.(B.10) are the times for executing those module invocations from the same computer where P_r $(s;1,t)$ represents the probability that M_r $(s;1)$ invokes M_r $(s;t)$. After substituting λ_r $(s;t)$ $= \lambda_r$ $(s;1)$ P_r $(s;1,t)$ and ρ_r $(s;t) = \lambda_r$ $(s;t)$ $\bar{x}_r$ $(s;t)$ into Eq.(B.10), and simplifying, we have

$$W3_i\ (j;k) = \sum_{t=1}^{k-1} \left\{ [W_i\ (j;t) + \bar{x}_i\ (j;t)] \sum_{r=1}^{v} \sum_{\substack{s=1 \\ d_r(s) \geq k-t}}^{b_r} \rho_r(s;k-t) \right\} \tag{B.11}$$

Therefore, the mean module waiting time for $M_i\ (j;k)$ is

$$W_i\ (j;k) = W1_i\ (j;k) + W2_i\ (j;k) + W3_i\ (j;k) \tag{B.12}$$

for $i=1,2,...v$, $j=1,2,...b_i$, and $d_i\ (j) \geq k \geq 2$.

The mean module waiting time for each module is expressed in Eq.(B.4), (B.5) or (B.12). They can be determined by solving this set of linear equations. The mean response time for each module is the sum of its mean waiting time and mean execution time.

APPENDIX C

MODULE RESPONSE TIMES FOR PRIORITY MODULE SCHEDULING

Consider that each computer schedules module execution on a predetermined module priority basis. To study the module response times, the computer is modeled as a non-preemptive head-of-line (HOL) priority queueing system. Recall that a module arrival may be a bulk arrival which contains several invocations of different modules. To find the first two moments of the waiting time (from arrival until execution starts) for a particular module k requires the knowledge of the L.T. of the waiting time. However, solving this queueing system is very complicated. In particular, we know of no general solution for the system with more than two priority queues [CHAU83]. Here we propose a new method using the notion of "rest period" [SCHO83] to find the L.T. of module waiting time.

Without loss of generality, let us assume each type of module has a distinct priority. Consider the waiting time of a particular module k. Suppose module k is contained (invoked) in bulk arrival of type i and its arrival rate is λ_k. Since module bulk arrivals are independent Poisson processes, all modules which are not contained in bulk i but with a higher priority than module k can be grouped together as from a single Poisson source with rate λ_h and the L.T. of execution time $B_h^*(s)$. Similar for those with a lower priority, the arrival rate is λ_l and $B_l^*(s)$ is its L.T. of execution time. Figure C.1 illustrates the arrival processes of this queueing system. In general, bulk i contains some modules of higher and lower priority than module k. The L.T.'s of module execution times of these higher and lower priorities modules are $B_a^*(s)$ and $B_b^*(s)$ respectively. To study the waiting time for module k, the equivalent queueing system (Figure C.1) has three types of customers: types p, k and q. Type p customers have the highest priority and type q customers have the lowest priority. Let $B_p^*(s)$ and $B_q^*(s)$ be the L.T.'s of execution times for type p and q modules respectively. We have

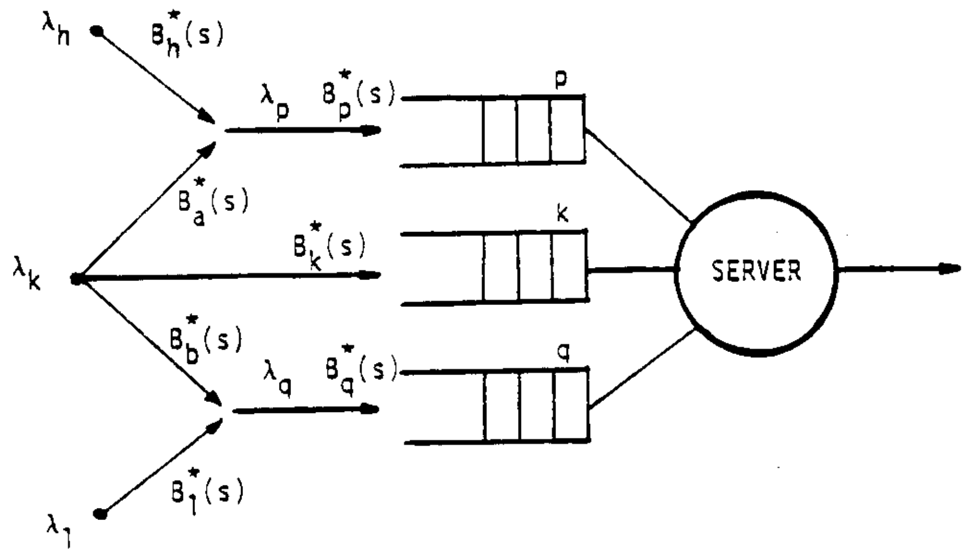
$$B_p^*(s) = \frac{\lambda_k}{\lambda_h + \lambda_k} B_a^*(s) + \frac{\lambda_h}{\lambda_h + \lambda_k} B_h^*(s) \tag{C.1}$$

and

$$B_q^*(s) = \frac{\lambda_k}{\lambda_l + \lambda_k} B_b^*(s) + \frac{\lambda_l}{\lambda_l + \lambda_k} B_l^*(s) \tag{C.2}$$

Assume that module k is contained in a bulk arrival; otherwise, the invocation arrivals are single arrivals and the module waiting time is readily solved in [CONW67]. Since subsequent arrivals with higher priority modules will get served before module k, it is appropriate to use the delay cycle analysis technique [KLEI76] to compute the L.T. of the waiting time. The computation consists of two steps:

(1) Suppose all arrival processes will be shut off (i.e., no more new arrivals) immediately after the bulk under consideration has arrived at the system. Find the waiting time of module k as the initial delay.



Priority(p) > Priority(k) > Priority(q)

$\lambda_p = \lambda_h + \lambda_k$

$\lambda_q = \lambda_l + \lambda_k$

Figure C.1 Arrival Processes and Service Times of the HOL Queueing System

(2) To consider the effect of subsequent arrivals which have higher priorities than module k, then the waiting time for module k is the delay cycle with initial delay computed in Step (1).

Let us define the *unfinished work* of a queue as the time required by the server to finish servicing all customers currently waiting in the queue. To proceed Step (1), let the unfinished work of queues p and k to be U_{pk}. In addition, we define a *sub-busy period* for U_{pk} to be the time interval from U_{pk} is increased from zero until U_{pk} reaches zero again (i.e., all customers are completely served).

It is apparent that the initial delay for module k is given by the sum of U_{pk}, the execution time of the higher priority modules invoked in the same bulk arrival i and the residual execution time of type q module if it is being serviced upon the bulk arrival. Since queues p and k have higher priority than q, the computer (server) will start servicing q module (if any) only when U_{pk} is zero. Consider the case that some module arrival at queue p and/or k which starts the sub-busy period U_{pk} finds a q module is being serviced. Due to the non-preemptive module execution, these arrivals have to wait until the completion of execution for the q module under service. This suggests that, to find the initial delay, the system may be viewed as a M/G/1 queue with rest period [SCHO83], in which the server will take a rest period with L.T. $B_q^*(s)$ (i.e., execution time for a q module) if $U_{pk} = 0$. The arrival rate and service distribution of this M/G/1 queue are given by the following equations:

$$\lambda = \lambda_h + \lambda_k \tag{C.3}$$

$$B^*(s) = \frac{\lambda_k}{\lambda} B_k^*(s) B_a^*(s) + \frac{\lambda_h}{\lambda} B_h^*(s) \tag{C.4}$$

Following the arguments in [SCHO83], the L.T. of the sum of U_{pk} and the residual execution time for a type q module is

$$\frac{(1-\rho_k-\rho_p)s}{s-\lambda+\lambda B^*(s)} \frac{1-B_q^*(s)}{s\bar{x}_q}$$

where

$\bar{x}_k, \bar{x}_p, \bar{x}_q$: respective mean execution time for types k, p and q modules,

$\rho_k = \lambda_k \bar{x}_k$ and $\rho_p = \lambda_p \bar{x}_p$.

Since the bulk arrival has a module of higher priority than k with execution time $B_a^*(s)$, the initial delay for module k is

$$I_k^*(s \mid \textit{sub-busy period starts when a q module in execution})$$

$$= \frac{(1-\rho_k-\rho_p)s}{s-\lambda+\lambda B^*(s)} \frac{1-B_q^*(s)}{s\bar{x}_q} B_a^*(s) \tag{C.5}$$

In case the computer is idle when an invocation arrival arrives at queues p and/or k, the computer will immediately start servicing the arrival. Therefore, the module bulks subsequently arriving during a sub-busy period for U_{pk} which is started with an idle system do not experience any delay due to servicing type q module. Hence the system becomes a regular M/G/1 queue with arrival rate and service time distribution given by Eq.(C.3) and (C.4). Then the L.T. of the initial delay for module k is given by

$$I_k^*(s \mid \textit{sub-busy period starts when server idle}\,) = \frac{(1-\rho_k-\rho_p)s}{s-\lambda+\lambda B^*(s)}\, B_a^*(s) \tag{C.6}$$

Let $\rho_q = \lambda_q \bar{x}_q$ and $\rho = \rho_q + \rho_k + \rho_p$. Due to the randomness of Poisson arrivals, the bulk arrival which starts the sub-busy period for U_{pk} will find the server either executing a q module (taking a rest) or idle with probability $1-\rho_k-\rho_p$. Out of this probability, ρ_q is the probability that the server is executing a q module, and $1-\rho$ is that of the server being idle. Thus, $\dfrac{\rho_q}{1-\rho_k-\rho_p}$ and $\dfrac{1-\rho}{1-\rho_k-\rho_p}$ are the respective probabilities that the sub-busy period for U_{pk} starts when a q module is being executed or the system idle. To uncondition the L.T.'s in Eq.(C.5) and (C.6) with these probabilities, we obtain the L.T. of the initial delay for module k as

$$I_k^*(s) = \frac{(1-\rho_k-\rho_p)s}{s-\lambda+\lambda B^*(s)}\,\frac{1-B_q^*(s)}{s\bar{x}_q}\, B_a^*(s)\,\frac{\rho_q}{1-\rho_k-\rho_p} + \frac{(1-\rho_k-\rho_p)s}{s-\lambda+\lambda B^*(s)}\, B_a^*(s)\,\frac{1-\rho}{1-\rho_k-\rho_p}$$

$$= \frac{s}{s-\lambda+\lambda B^*(s)}\, B_a^*(s)[\,1-\rho+\rho_q\,\frac{1-B_q^*(s)}{s\bar{x}_q}\,] \tag{C.7}$$

Let us proceed Step (2). Based on this initial delay, the L.T. $W_k^*(s)$ of the waiting time for module k is the delay cycle which includes executing all higher priority modules arriving during the waiting time. By the delay cycle analysis, we get

$$W_k^*(s) = I_k^*(\,s+\lambda p - \lambda_p G_p^*(s)\,) \tag{C.8}$$

where $G_p^*(s) = B_p^*(s + \lambda_p - \lambda_p G_p^*(s))$, $B_p^*(s)$ and $I_k^*(s)$ are given in Eq.(C.1) and Eq.(C.7). The first two moments of the waiting time for module k may be obtained by differentiating Eq.(C.8).